ADDRESSING CONNECTIVITY CHALLENGES FOR MOBILE COMPUTING AND COMMUNICATION

A Thesis
Presented to
The Academic Faculty

by

Cong Shi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
August 2014

ADDRESSING CONNECTIVITY CHALLENGES FOR MOBILE COMPUTING AND COMMUNICATION

Approved by:

Dr. Mostafa Ammar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ellen Zegura, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Kaustubh Joshi
AT&T Labs-Research
AT&T

Dr. Mayur Naik
School of Computer Science
Georgia Institute of Technology

Dr. Patrick Traynor
School of Computer Science
Georgia Institute of Technology

Date Approved: 2 May 2014

To my parents, for their endless love and support

To my wife and my little son

ACKNOWLEDGEMENTS

After spending ten years on research and six years in the PhD program at Georgia Tech, I finally arrive at the end of my long journey in obtaining my degree in computer science. It was the help and support from many people that kept me going through all the ups and downs over these years.

First and foremost, I would like to express my greatest gratitude to my thesis advisors, Prof. Mostafa Ammar and Prof. Ellen Zegura for their guidance and consistent support. It would not have been possible to complete this thesis without their patience, encouragement and guidance. They gave me the freedom in working the research topics I am interested in; they provided stimulating suggestions and candid opinions; they helped shape my research skills; they were always supportive whenever it was needed. It was really a great pleasure to work with them for the past six years.

I would also like to thank my committee members, Dr. Kaustubh Joshi, Prof. Mayur Naik, and Prof. Patrick Traynor for their long-term collaboration and help in my work. I learned a lot from each of them. Their insightful comments and suggestions have also significantly improved my work. I also owe many thanks to Dr. Rajesh Panta from AT&T and Dr. Haiyong Xie from Huawei for hosting me as a summer intern and working on interesting problems. I would also like to thank other professors in my research group, Prof. Jun (Jim) Xu, Prof. Nick Feamster and Prof. Constantinos Dovrolis for their suggestions and help.

I would like to thank all of my lab mates in the Networking and Telecommunications group who have given me a lot of help in my study and made my life at Georgia Tech easier. I would like to thank Shuang Hao, Yang Chen, Ahmed Mansy, Samantha Lo, AliReza Monfared, Pranesh Pandurangan, Karim Habak, Tongqing Qiu, Nan Hua for their friendship, help, and all the good times we spent together.

Last but not least, I am deeply indebted to my parents and my sister Yan for their

endless support and love. There are no words enough to thank my wife, Qiaoling, for her love, support and sacrifice in the long journey. I also thank our little boy, Andrew, for bringing all the happiness to our lives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Mobile devices are increasingly being relied on for computation intensive and/or communication intensive applications that go beyond simple connectivity and demand more complex processing. This has been made possible by two trends. First, mobile devices, such as smartphones and tablets, are increasingly capable devices with processing and storage capabilities that make significant step improvements with every generation. Second, many improved connectivity options (e.g., 3G, WiFi, Bluetooth) are also available to mobile devices.

In the rich computing and communication environment, it is promising but also challenging for mobile devices to take advantage of various available resources to improve the performance of mobile applications. First, with varying connectivity, remote computing resources are not always accessible to mobile devices in a predictable way. Second, given the uncertainty of connectivity and computing resources, their contention will become severe.

This thesis seeks to address the connectivity challenges for mobile computing and communication. We propose a set of techniques and systems that help mobile applications to better handle the varying network connectivity in the utilization of various computation and communication resources. This thesis makes the following contributions:

- We design and implement Serendipity to allow a mobile device to use other encountered, albeit intermittently, mobile devices to speedup the execution of parallel applications through carefully allocating computation tasks among intermittently connected mobile devices.

- We design and implement IC-Cloud to enable a group of mobile devices to efficiently use the cloud computing resources for computation offloading even when the connectivity is varying or intermittent.

- We design and implement COSMOS to provide scalable computation offloading service

to mobile devices at low cost by efficiently managing and allocating cloud computing resources.

- We design and implement CoAST to allow collaborative application-aware scheduling of mobile traffic to reduce the contention for bandwidth among communication-intensive applications without affecting their user experience.

CHAPTER I

INTRODUCTION

Recent years have seen a significant rise in the sophistication of mobile applications. Mobile devices are increasingly being relied on for a number of services that go beyond a single device's capability and require more complex processing. These include both communication-intensive services such as video streaming that delivers high-resolution videos from remote cloud to mobile users in real-time and computation-intensive services such as pattern recognition that aids in identifying snippets of audio or recognizing images whether locally captured or remotely acquired, reality augmentation that enhances our daily lives, and collaborative applications that enhance distributed decision making and planning and coordination. Some such applications are already in ubiquitous use today, others are still on the drawing boards and in lab prototypes awaiting the next generational change in device capability and connectivity

Mobile applications have become an indispensable part of everyday life. This has been made possible by two trends. First, truly portable mobile devices, such as smartphones and tablets, are increasingly capable devices with processing and storage capabilities that make significant step improvements with every generation. They allow communication-intensive applications to buffer and cache more contents on the client side and computation-intensive applications to do more complex processing on the end host.

A second trend that is directly relevant to this thesis is the availability of improved connectivity options (e.g., 3G, WiFi, Bluetooth) for mobile devices. They have enabled both communication-intensive applications and computation-intensive applications to transcend an individual device's capabilities and make use of various remote resource. Specifically, they have made the communication-intensive applications obtain higher aggregated bandwidth from various network interfaces [19, 51]. Additionally, they have also enabled computation-intensive applications makes use of remote computing resources (e.g., cloud computing

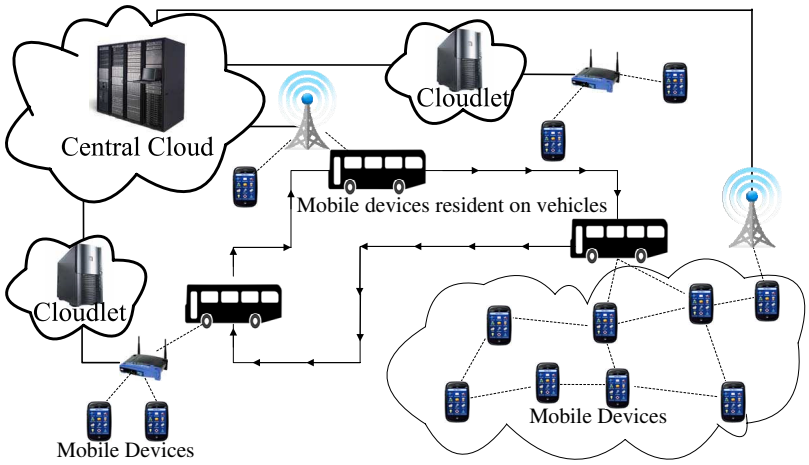


Figure 1: System components and network connectivity for mobile computing and communication.

resources) to offload the "heavy lifting" that may be required in some mobile applications to specially designated servers or server clusters [32, 33, 85].

A fundamental challenge to these mobile computing and communication applications is how to handle the *varying* (and sometimes even *intermittent*) connectivity of mobile devices. According to recent studies [20, 36, 70], mobile users typically experience intermittent connectivity to the Internet and highly variable access quality even when connectivity exists. For example, 3G access is only available 87% of the time even in a metropolis [20], while WiFi coverage is even more intermittent. We also make the important observation that a mobile device often encounters, albeit intermittently, many entities, including other mobile devices, capable of lending various resources, e.g., computing cycles, storage and cached contents. As a result if the applications are capable of handling varying and intermittent connectivity that may occur in the middle of computation and communication, one is not limited to using well-connected cloud and might be able to leverage this additional resources.

In this thesis we focus on a mobile computing and communication paradigm that uses various compute resources both mobile and not to enable its operation in the presence of intermittent connectivity. We posit that an ultimately successful system should have the flexibility to handle the connectivity challenge and use a mix of resources.

Mobile devices roam in a very rich computing and communication environment today,

as shown in Figure 1. For our purposes we classify system components of this environment in the following four categories.

- User-carried mobile devices: Today such devices are typically smartphones or tablets. They are portable and thus experience significant mobility. These devices are becoming increasingly powerful although they continue to be constrained relative to tethered devices. Additionally such devices typically have multiple communication interfaces as well as GPS and other sensing devices (such as cameras).

- Mobile computing resources attached to moving vehicles: It is increasingly possible today to piggyback computing resources on vehicles such as buses or taxicabs [93]. Such systems are not resource constrained since they can derive power from a vehicle's battery and as such can be quite useful in providing resources to user-carried constrained devices. These represent *mobile computing resources* that may have somewhat predictable mobility patterns.

- Infrastructure-based resources suitable for opportunistic use: These are similar to *cloudlets* [85] in the sense that they are pre-provisioned storage and computing resources that are accessible locally over a wireless access point. The main additional feature we allow in our work is the potential for user devices to intermittently connect with such systems.

- Central service-owned cloud resources: These are servers that are always equipped and ready to undertake a particular computation task. Compared with other system components, cloud is much more powerful and usually has more resources. However, its latency to mobile devices are also long.

In this environment, mobile nodes have access to multiple connectivity options, including widely-deployed cellular networks that have high coverage but low bandwidth, sparsely-deployed WiFi that has low coverage but high bandwidth, direct device-to-device communication through WiFi or Bluetooth. We are interested in supporting computation and communication in mobile user devices without constraining (or otherwise modifying) their

mobility patterns. While user devices will, over time, come in contact with other devices that can provide various resources (see classification above), such contact will be *intermittent* and of potentially indeterminate duration. Two nodes within communication range of each other experience a contact opportunity that lasts for as long as they can hear each other.

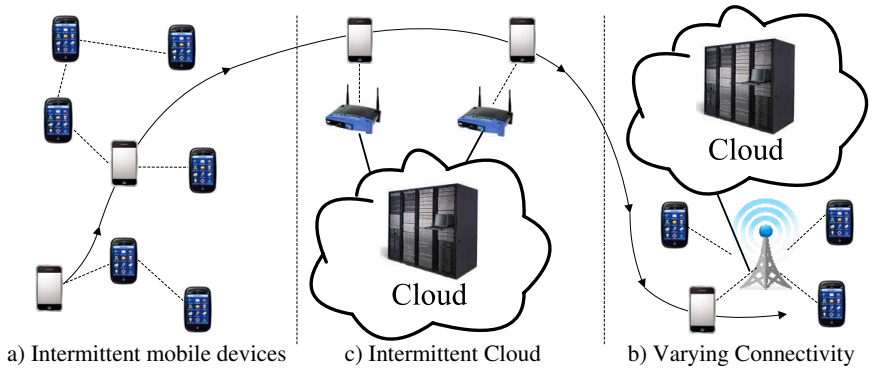


Figure 2: Scenarios on the spectrum of mobile computing and communication.

In this mobile computing and communication environment, we envision a spectrum of computing and communication contexts, some of which are shown in Figure 2.

- At one extreme is the mobile device cloud where a mobile device's contacts are only with other mobile devices, as shown in Figure 2(a). This is an extreme computational environment where the computation resources are mobile but less powerful while the connectivity is intermittent. We are interested in supporting computation-intensive services that aggregates the intermittently connected computing resources to handle complicated tasks. It represents the most challenging scenario that mobile applications may encounter. We summarize our work, Serendipity, on such environments in Chapter 3.

- Next on the spectrum is the cloud-computing context where a mobile device is intermittently connected to remote cloud resources maintained by a service provider with which it has an established relationship, as shown in Figure 2(b). We present our system, IC-Cloud, on supporting computation-intensive applications in this scenario in Chapter 4.

- In the intermittent-cloud scenario as shown in Figure 2(b), another important problem is how to efficiently manage the cloud resources for computation offloading. In this context, there exists a mismatch between how individual mobile devices demand computing resources and how cloud providers offer them: offloading requests from a mobile device usually require quick response, may be infrequent, and are subject to variable network connectivity, whereas cloud resources incur relatively long setup times, are leased for long time quanta, and are indifferent to network connectivity. We describe our work, COSMOS, which bridges this gap by providing computation offloading as a service to mobile devices in Chapter 5.

- Moving along the spectrum, we consider the case that multiple mobile devices connect to the remote cloud through a cellular network, as shown in Figure 2(c). In this context, mobile devices share a single link to access remote contents. It will cause severe contention for bandwidth among communication-intensive applications. We present our work, CoAST, in reducing the bandwidth contention for communication-intensive applications in Chapter 6.

This thesis seeks to address connectivity challenge in the above contexts. Specifically, this thesis work consists of the following components.

Serendipity. To address the issue of limited computing resources on mobile devices, this thesis work studies the *system design* approach in which other mobile devices are utilized to speedup the execution of computation-intensive mobile applications. Specifically, this thesis investigates the scenario where an initiator mobile device needs to run a computational task that exceeds the mobile devices ability and where portions of the task are amenable to remote execution. This thesis proposes the *Serendipity* system to decrease computation completion time through computation offloading among intermittently connected mobile devices. It leverages the fact that a mobile device within its intrinsic motion pattern makes frequent contact with other mobile devices that are capable of providing computing resources. Contact with these devices can be intermittent, limited in duration

when it occurs, and sometimes unpredictable. Serendipity divides a computational applications into multiple small computational tasks and allocates them among encountered mobile devices. A set of task allocation algorithms are developed for different scenarios to minimize the computation completion time by effectively using the available information.

IC-Cloud. To addresses the issue of using cloud computing resource to speedup the execution of computation-intensive applications, previous approaches [32, 33] rely on the identification of computation-intensive components and dynamically offload their execution to the cloud. There are two major challenges which are not considered in these approaches, including the varying or intermittent connectivity between mobile devices and the cloud and the contention for cloud computing resources. This thesis proposes the *IC-Cloud* system to enable computation offloading to the intermittently connected cloud. To handle the intermittent connectivity between mobile devices and the cloud, IC-Cloud uses a set of methods to predict the future connectivity based on historical information. In addition, to overcome the uncertainty in the prediction, it also uses a risk-control mechanism to limit the impact of prediction errors.

COSMOS. To bridge this gap between how individual mobile devices demand computing resources and how cloud providers offer them, this thesis investigates providing computation offloading as a service for mobile devices. There are three major challenges in designing such a system. First, the offloading requests require quick response and may not be very frequent. Second, the number of offloading requests changes over time. Third, due to the varying network connectivity, it is not always beneficial to offload computation to the cloud. This thesis proposes *COSMOS* to solve these problems. It efficiently manages cloud resources for offloading requests to both improve offloading performance seen by mobile devices and reduce the monetary cost per request to the provider. It also effectively allocates and schedules offloading requests to resolve the contention for cloud resources.

CoAST. To address the contention for bandwidth among communication-intensive applications, this thesis studies the scheduling of cellular traffic through the collaboration among

mobile applications. This thesis work makes two key insights derived from mobile traffic traces of a large US cellular provider. First, we observe that the mobile data traffic exhibits high burstiness over small time scales (few seconds). Thus, to ensure adequate quality of service at all times, it is important to reduce the instantaneous peak traffic, not just the aggregate traffic. Second, even applications like video streaming and mobile web browsing, can, in fact, tolerate small delays. Based on these insights, this thesis work proposes *CoAST* to reduce the peak traffic through the collaboration among mobile devices and network elements. To achieve this purpose, CoAST uses three key mechanisms: a protocol to allow mobile applications and providers to exchange traffic information, an incentive mechanism to incentivize mobile applications to collaboratively delay traffic at the right time, and mechanisms to delay application traffic.

This thesis work has been published in part in the following publications: [88], [89], [90], and [91].

The rest of the thesis is organized as follows. Chapter 2 discusses existing work related to the topics in this proposal. Chapter 3 investigates computation offloading among intermittently connect mobile devices. Chapter 4 describes a system to support computation offloading to an intermittently connected cloud. Chapter 5 describes a system that provides computation offloading as a service to mobile devices. Chapter 6 presents a system to enable collaborative application-aware scheduling of the cellular traffic. The contributions and future work of this thesis are summarized in Chapter 7.

CHAPTER II

RELATED WORK

This chapter provides an overview of the related work on the topics of mobile computing and network connectivity.

2.1 Mobile Computing

2.1.1 Computation offloading

The concept of cyber foraging [84], i.e., dynamically augmenting mobile devices with resource-rich infrastructure, was proposed more than a decade ago. Since then significant work has been done to augment the capacity of resource-constrained mobile devices using computation offloading [16–18, 47, 77].

Closer to our work, MAUI [33] enabled mobile applications to improve their performance and reduce the energy consumption through automated offloading. To profile the communication cost and computation gain, MAUI periodically measures the network bandwidth and uses the previous invocations to profile applications. Similarly, CloneCloud [32] can minimize either energy consumption or execution time of mobile applications through automatically identifying computation intensive methods and offloading those methods that achieve best performance. ThinkAir [65] enables scalable offloading of multiple applications with server-side support. All of these systems assume a stable environment where network connectivity and application execution time are easy to predict. In contrast, we target at the more challenging mobile environment where the Internet access is of highly variable quality and often intermittent.

Computation offloading is also very useful for improving the security of mobile devices as cryptographic functions are usually computation intensive. Green et al. [46] investigated how to maintain data privacy in offloading the costly decryption of ABE ciphertexts to servers. Kamara et al. [62] developed protocols to offload secure multiparty computation to the cloud in their Salus system. Carter et al. [28] proposed a protocol to offload garbled

circuits to the cloud for jointly evaluating functions while pretecting user privacy. Zonouz et al. [99] designed Secloud to perform resource-intensive security analysis for mobile devices in the cloud.

Other works, like COMET [45], enable the offloading of multi-threaded applications using distributed shared memory. ECOS [42] focuses on the data privacy in computation offloading. A detailed survey of cyber foraging can be found in [40].

2.1.2 Distributed Computing with Non-Dedicated Machines

Our work is related to systems that use non-dedicated machines with cycles that are donated and may disappear at any time. In this vein, our work takes some inspiration from the Condor system architecture [94]. Our work also resembles in part distributed computing environments that have well-connected networks but unreliable participation in the computation. Examples of these systems include BOINC [13], SETI@home [14], and folding@home [22], all leveraging the willingness of individuals to dedicate resources to a large computation problem. More recently, the Hyrax project envisions a somewhat similar capability to opportunistically use the resources of networked cellphones [72].

2.1.3 The Prediction of Program-Execution Time

Our work is also related to the studies on the prediction of program-execution time. Gupta et al. [48] used a variant of decision trees to predict execution-time ranges for database queries. Ganapathi et al. [41] used KCCA to predict time and resource consumption for database queries. These approaches either require manual efforts to identify good features, or require applications to high correlations between input size and execution time. Mantis [31] estimated the execution time of an entire application using executable program slices to obtain feature values at runtime. In contrast, we estimate the execution time of each offloadable function.

2.2 Network Connectivity

2.2.1 Delay Tolerant Network

Our work also leverages recent advances in the understanding of data transfer over intermittently-connected wireless networks (also known as disruption-tolerant networks or opportunistic networks). These networks have been studied extensively in a variety of settings, from military [73] to disasters [39] to the developing world [76]. These settings share the characteristic that fixed infrastructure is unavailable, highly unreliable, or expensive. Further, the communication links are subject to disruptions that mean network partitions are common.

Our work is also related to the efforts at developing useful applications over intermittently-connected mobile and wireless networks. Examples of this work include the work by Hanna et al. which develops mobile distributed information retrieval systems [52], and the work by Fall et al. on an architecture for disaster communications response [39] with a specific focus on situational awareness. In this latter work the authors propose an architecture that contains infrastructure-supported servers, mobile producer/consumer nodes and mobile field servers. Related, the Hastily Formed Networks (HFN) project [35] describes potential applications in disaster settings that match well with our vision requiring computation, including situational awareness, information sharing, planning and decision making.

2.2.2 Connectivity Characterization of Wireless and Mobile Network

Our work is related to the efforts at predicting network connectivity. BreadCrumbs [74] predicts the future locations of a mobile user by tracking her movements and creating a predictive model based on the observed data. Combined with a database of network connectivity of those locations, BreadCrumbs is able to predict future network connectivity. Deshpande et al. [37] proposed to predict the connectivity to WiFi from vehicles and use this information to improve vehicular WiFi access. These methods use energy-consuming GPS to obtain the location information. On contrast, we use energy-efficient methods to predict the connectivity properties that are critical to computation offloading.

Our work also leverages recent advances in the understanding of data transfer over intermittently-connected wireless networks (also known as disruption-tolerant networks or

opportunistic networks). These networks have been studied extensively in a variety of settings, from military [73] to disasters [39] to the developing world [76].

2.2.3 Traffic Reduction

Our work is related to the efforts at reducing the peak throughput over a link. Laoutaris et al. [66,67] proposed using diurnal variations in Internet traffic to transfer delay tolerant bulk data over the Internet at off-peak times. With peak pricing, these shifts can also reduce cost. Recently, Ha et al. [49] applied the idea of medium-to-long time scale shifting to cellular networks and proposed a time-dependent pricing mechanism, TUBE, to motivate mobile users to shift some cellular traffic sessions from peak time to off-peak times in exchange for lower prices. CoAST is fundamentally different from TUBE in many aspects. First, TUBE utilizes medium-to-long time-scale variation of the background traffic and thus are only suitable for applications that can tolerate substantial delays. Second, CoAST requires no involvement of mobile users, while TUBE requires user change their behavior. Third, CoAST also has a data plane to schedule traffic, while TUBE only focuses on the control plane. In addition, our approach is compatible with medium and long time scale traffic shifting: our work can reduce the short time scale peaks that will remain after other data is time shifted by hours or 10s of minutes.

In addition to reducing traffic peaks, time shifting mobile traffic can be beneficial in saving device energy. For example, one set of approaches [21, 87] utilize the observation that aggregation of traffic can reduce energy by avoiding the excessive energy-consuming RRC transitions and tail energy that occur in 3G and LTE networks [57, 80] when traffic is transferred in disjoint time periods. Studies show that mobile applications have sufficient periods of sparse traffic transfer to make these approaches useful for energy saving with relatively little traffic delay [78]. Another approach saves device energy by scheduling transfers when signal strength is strong [86], leveraging the observation that energy consumption per bit increases when signal strength degrades.

Our approach is not directly compatible with approaches that use scheduling to reduce

energy because both approaches operate on a similar time scale but with different objective functions. To use an energy saving scheduler with a peak reduction scheduler would require an integrated objective function that minimizes a combination of device energy and cost to use the shared link, while meeting deadlines. An adaptive approach may be most appropriate, where reducing cost is favored when device energy is plentiful, and reducing energy is favored when device energy is low. In its most simplistic form, a controller could simply switch from one scheduler to the other based on a device energy threshold.

CHAPTER III

SERENDIPITY: ENABLING REMOTE COMPUTING AMONG INTERMITTENTLY CONNECTED MOBILE DEVICES

3.1 Introduction

Mobile devices are increasingly being relied on for a number of services that go beyond simple connectivity and require more complex processing. These include pattern recognition to aid in identifying snippets of audio or recognizing images whether locally captured or remotely acquired, reality augmentation to enhance our daily lives, collaborative applications that enhance distributed decision making and planning and coordination, potentially in real-time. Additionally, there is potential for mobile devices to enable more potent "citizen science" applications that can help in a range of applications from understanding how ecosystems are responding to climate change[1] to gathering of real-time traffic information.[2]

Fortunately, a mobile device often encounters, possibly intermittently, many entities capable of lending it computational resources. This environment provides a spectrum of computational contexts for remote computation in a mobile environment. An ultimately successful system will need to have the flexibility to use a mix of the options on that spectrum. At one extreme of the spectrum is the use of standard cloud computing resources to offload the "heavy lifting" that may be required in some mobile applications to specially designated servers or server clusters. A related technique for remote processing of mobile applications proposes the use of *cloudlets* which provide software instantiated in real-time on nearby computing resources using virtual machine technology [85]. Likewise, MAUI [33] and CloneCloud [32] automatically apportion processing between a local device and a remote cloud resource. In this chapter we consider the other spectrum extreme, where a mobile device's contacts are only with other mobile devices, where both the computation initiator

[1] See http://blogs.kqed.org/climatewatch/2011/01/29/citizen-science-the-iphone-app/

[2] See http://www.crisscrossed.net/2009/08/31/citizen-scientist-how-mobile-phones-can-contribute-to-the-public-good/

and the remote computational resources are mobile, and where intermittent connectivity among these entities is the norm.

We investigate the basic scenario where an *initiator* mobile device needs to run a computational task that exceeds the mobile device's ability and where portions of the task are amenable to remote execution. We leverage the fact that a mobile device within its intrinsic motion pattern makes frequent contact with other mobile devices that are capable of providing computing resources. Contact with these devices can be intermittent, limited in duration when it occurs, and sometimes unpredictable. The goal of the mobile device is to use the available, potentially intermittently connected, computation resources in a manner that improves its computational experience, e.g., minimizing local power consumption and/or decreasing computation completion time. The challenge facing the initiator device is how to apportion the computational task into subtasks and how to allocate such tasks for remote processing by the devices it encounters.

The remainder of this chapter is organized as follows: we start with the discussion of the problem context and the design challenges in Section 3.2; we describe the design of a job model and the Serendipity system in Section 3.3; the task allocation algorithms are presented in Sections 3.4 and 3.5; we describe how to enable energy-aware computing in Section 3.6; we undertake an extensive evaluation of our system on Emulab in Section 3.7; the implementation and evaluation of Serendipity on mobile devices are presented in Section 3.8; We conclude this work in Section 3.9.

3.2 Problem Context and Design Challenges

Network Model: We focus on a network environment that is composed of a set of mobile nodes with computation and communication capabilities. The network connectivity is intermittent, leading to a frequently-partitioned network. Every node can execute computing tasks, the number of which is constrained by its resources, such as processor capability, memory, storage size, and available energy. The period of time during which two nodes are within communication range of each other is called a *contact*. During a contact nodes can transfer data to each other. Both the duration and the transfer bandwidth of a contact are

limited. There are some variants of the general network setting. For some mobile devices, a low-capacity control channel (e.g., over satellite link) is available for metadata sharing. In addition, in some special networks, such as networks with scheduled robotic vehicles or UAVs, the node mobility patterns are predictable and, thus, their future contacts are also predictable. All these variants are taken into consideration in our design.

Remote computing usually involves the execution of computationally complex jobs through the cooperation among a set of devices connected by a network. A major class of such jobs, supported by mainstream distributed computing platforms such as Condor [94], can be represented as a Directed Acyclic Graph (DAG). The vertices are programs and the directed links represent data flows between two programs. A traditional distributed computing platform maps the vertices to the devices and the links to the network so that all independent programs are executed in parallel and they transfer the output to their children. As a variant of such computing platforms, MAUI [33] and CloneCloud [32] have a simple network composed of a mobile device and the cloud.

Design Challenges: The intermittent connectivity among mobile devices poses three key challenges for remote computing. First, because the underlying connectivity is often unknown and variable, it is difficult to map computations onto nodes with an assurance that the required code and data can be delivered and the results are received in a timely fashion. This suggests a conservative approach to distributing computation so as to provide protection against future network disruptions. Second, given that the network bandwidth is intermittent, the network is more likely to be a bottleneck for the completion of the distributed computation. This suggests scheduling sequential computations on the same node so that the data available to start the next computation need not traverse the network. Third, when there is no control channel, the network cannot be relied upon to provide reachability to all nodes as needed for coordination and control. This suggests maintaining local control and developing mechanisms for loose coordination. Besides the intermittent connectivity, the limited available energy imposes another extra constraint on the remote computing among mobile devices.

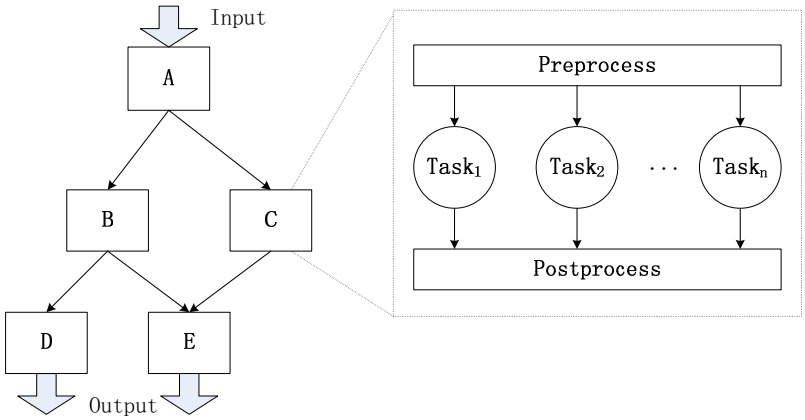


Figure 3: A job model for DTNs is a Directed Acyclic Graph (DAG), the vertices of which are PNP-blocks. Every PNP-block consists of a pre-process, a post-process and n parallel tasks.

3.3 Serendipity System Design

3.3.1 A Job Model for Serendipity

Our basic job component is called a *PNP-block*. As shown in Figure 3, a PNP-block is composed of a *pre-process* program, n parallel *task* programs and a *post-process* program. The pre-process program processes the input data (e.g., splitting the input into multiple segments) and passes them to the tasks. The workload of every task should be similar to each other to simplify the task allocation. The post-process program processes the output of all tasks; this includes collecting all the output and writing them into a single file.

The PNP-block design simplifies the data flow among tasks and, thus, reduces the impact of uncertainty on the job execution. All pre-process and post-process programs are executed on one initiator device, while parallel tasks are executed independently on other devices. The communication graph becomes a simple star graph. The data transfer delay can be minimized as the initiator device can simply choose nearby devices to execute tasks. In contrast, it is much more difficult for a complicated communication graph, such as the complete bipartite graph used in MapReduce [34], to achieve low delay among intermittently connected mobile devices because the optimization problem associated with mapping the general graph onto them is complex.

The single PNP-block job comprises an important class of distributed computing jobs often called embarrassingly parallel and useful in many applications, among which are

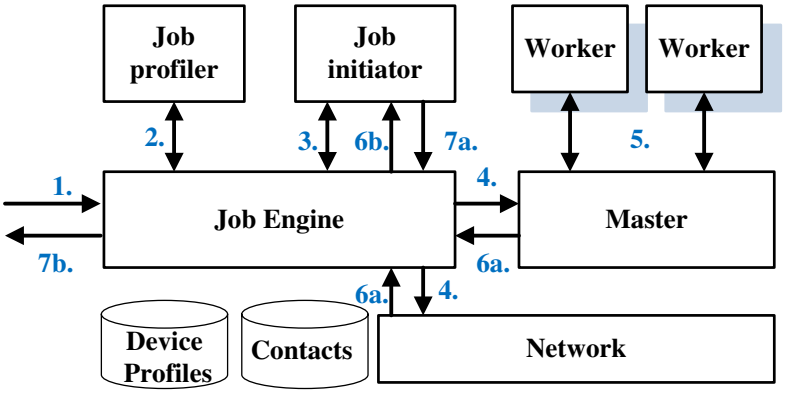


Figure 4: High-level Architecture of Serendipity. After receiving a job (1), the job engine constructs the job profile (2) and starts a job initiator, who will initiate a number of PNP-blocks and allocate their tasks (3). The job engine disseminates the tasks to either local or remote masters (4). After a worker finishes a task (5), the master sends back the results to the job initiator (6a, 6b), who may trigger new job PNP-blocks (3). After all results are collected, the job initiator returns the final results (7a, 7b) and stops.

SETI@home [14] and BOINC [13]. All jobs are graphically represented by a DAG of PNP-blocks, providing as much computational expressiveness as a regular DAG. For instance, the MapReduce model [34] can be implemented with two sequentially connected PNP-blocks, corresponding to the map phase and the reduce phase, respectively.

3.3.2 Serendipity System

Figure 4 shows the high-level architecture of Serendipity. A Serendipity node has a *job engine* process, a *master* process and several *worker* processes. The number of worker processes can be configured, for example, as the number of cores or processors of the node. Each node constructs its device profile and, then, shares and maintains the profiles of encountered nodes. A node's device profile includes its execution speed which is estimated by running synthetic benchmarks and its energy consumption model using techniques like PowerBooter [97]. These device profiles when combined with the jobs' execution profiles are used to estimate the jobs' execution time and energy consumption on every node, essential for task allocation. Serendipity also needs access to the contact database, if available, for better task allocation.

To submit a job, a user needs to provide a script specifying the job DAG, the programs and their execution profiles (e.g., CPU cycles) for all PNP-blocks and the input data to the job engine. Constructing accurate execution profiles of programs is a challenging problem

and out of the scope of this chapter. We simply follow the offline method used by both MAUI [33] and CloneCloud [32], i.e., running the programs multiple times with different input data.

The script is submitted to the *job profiler* for basic checking and constructing a complete job profile (i.e., tasks' execution time and energy consumption on every node) using its execution profiles and the device profiles. The generated job profile will be used to decide how to allocate its tasks among mobile devices.

If everything is correct, the job engine will launch a new *job initiator* responsible for the new job. It stores the job information in the local storage until the job completes. All PNP-blocks whose parents have completed will be launched by running their pre-process programs on a local worker and assigning a TTL (i.e., time-to-live), a priority and a worker to every task. The TTL specifies the time before which its results should be returned. If a task misses its TTL, it should be discarded, while a copy will be executed locally on the initiator's mobile device. The priority determines the relative importance of a job's different tasks. Section 3.5 will discuss how to assign the priorities.

Based on the consideration of task allocation and security, the assigned worker can be a single node, a set of candidate nodes, or a wildcard. In fact, only in the specific scenario that the future contacts are predictable while nodes have a control channel to timely coordinate the remote computing, the job initiator will use the global information to allocate tasks and assign a specific node for each task, which will be discussed in Section 3.4.1. Otherwise, the job initiator only specifies the set of candidate nodes it trusts and lets the job engine allocate the tasks. Finally, these tasks are sent to the job engine for dissemination.

The job engine is primarily responsible for disseminating tasks and scheduling the task execution for the local master. When two mobile nodes encounter, they will first exchange the metadata including their device profiles, their residual energy and a summary of their carried tasks. Using this information, the job engine will estimate whether it is better to disseminate a task to the encountered node than to execute it locally. Such a decision is based on the goal of reducing the job completion time (to be discussed in Section 3.4) or conserving the device energy (to be discussed in Section 3.6).

To schedule the task execution, the job engine first determines the job priority. Currently we use the first-in-first-serve policy. But it can be easily replaced by any arbitrary policy. For example, the job from a node that helps other nodes execute a lot of tasks is assigned a high priority. For the tasks of the same job, they are scheduled according to their task priorities.

The *master* is responsible for monitoring the task execution on workers. After receiving a task from the job engine, it starts a worker for it. When the task finishes, the output will be sent back to the job initiator using the underlying routing protocols like MaxProp [26]. If the task throws an exception during the execution, the master will report it to the job initiator who will terminate the job and report to the user.

In this chapter, we assume that all nodes are collaborative and trustworthy. However, there are also scenarios that some nodes are selfish (i.e., refusing to help other nodes) or even malicious (i.e., distorting the results). To motivate the selfish nodes, we can use some token-based incentive mechanism [69], making use of notional credit to pay off nodes for executing tasks. To protect the remote computing from malicious nodes, we can use reputation-based trust [27] in which nodes construct and share nodes' reputation information.

3.4 Task Allocation for PNP-blocks

One important goal of remote computing is to improve the performance of computationally complex jobs, especially when mobile nodes have enough energy. In this section, we will design efficient task allocation algorithms to minimize the job completion time. Specifically, since PNP-blocks are the basic blocks to allocate tasks, we will focus on the task allocation for PNP-blocks in various network settings. The problem of task scheduling for multi-processor systems [30, 56] is somewhat related to our task allocation problem. That work, however, does not deal with intermittent connectivity and cannot, therefore, be applied directly to our problem.

Figure 5 illustrates the timing and components of a PNP-block execution. Along the x-axis are the k remote nodes that will execute the parallel tasks of the block. Along the y-axis is a depiction of the time taken at each node to receive disseminated tasks from the initiator,

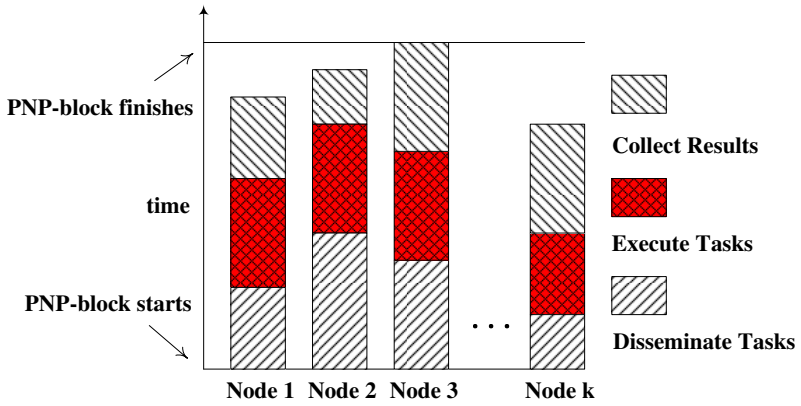


Figure 5: The PNP-block completion time is composed of a) the time to disseminate tasks, b) the time to execute tasks and c) the time to collect results, in addition to the time needed to execute pre-process and post-process programs.

execute those tasks, and provide the result collection back to the initiator. As illustrated, the time for each remote node to receive its disseminated tasks may vary, depending on the availability and quality of the network between the initiator and the remote node. When n tasks of a PNP-block are allocated to k nodes, each node will execute its assigned tasks sequentially, again taking a variable amount of time. After execution of all assigned tasks in the block, the node will send results back to the initiator, with time again being dependent on the network between the initiator and the remote node. Our goal for the task allocation is to reduce the completion time of the last task which equals to the PNP-block completion time.

We consider the design of task allocation algorithms in the context of three models with different contact knowledge and control channel availability assumptions.

3.4.1 Predictable Contacts with Control Channel

We first consider an ideal network setting where the future contacts can be accurately predicted, and a control channel is available for coordination. The performance in this type of scenarios represents the best possible performance of task allocation that is achievable among intermittently connected mobile devices. It is useful to identify the fundamental benefits and limits of Serendipity.

With future contact information a Dijkstra's routing algorithm for DTNs [60] can be used to compute the required data transfer time between any pair of nodes given its starting time. With the control channel the job initiator can obtain the time and number of tasks to

be executed on the target node with which to estimate the time to execute a task on that node. Therefore, given the starting time and the target node, the task completion time can be estimated.

Using this information, we propose a greedy task allocation algorithm, *WaterFilling*, that iteratively chooses the destination node for every task with the minimum task completion time (see Algorithm 1).

Algorithm 1 Water Filling

```
1: procedure WaterFilling(T, N)                          ▷ T is task set; N is node set.
2:     current ← currentTime();
3:     rsv ← getTaskReservationInfo();
4:     inputSize ← getTaskInputSize(T);
5:     outputsize ← estimateOutputSize(T);
6:     queue ← initPriorityQueue();
7:     for all n ∈ N do
8:         arrivalT ← dijkstra(this, n, current, inputSize);
9:         exeT ← estimateTaskExecutionTime(n,t);                              ▷ t ∈ T
10:        tfinishT ← taskFinishTime(rsv[n], arrivalT, exeT);
11:        completeT ← dijkstra( n, this, tfinishT, outputSize);
12:        queue.put({n, arrivalT, exeT, completeT});
13:    end for
14:    for all t ∈ T do
15:        {n, arrivalT, exeT, receiveT} ← queue.poll();
16:        updateReservation(rsv[n], t, inputSize, arrivalT, exeT);
17:        send(n, t);
18:        arrivalT ← dijkstra(this, n, current, inputSize);
19:        tfinishT ← taskFinishTime(rsv(n), arrivalT, exeT);
20:        completeT ← dijkstra( n, this, tfinishT, resultSize);
21:        queue.put({n, arrivalT, exeT, receiveT});
22:    end for
23:    reserveTaskTime(rsv);
24: end procedure
```

For every task, the algorithm first estimates its task dissemination time to every node. With the information of the tasks to be executed on the destination node and the estimated time to execute this task, it is able to estimate the time when this task will finish. Given that time point, the time when the output is sent back can also be computed. Among all the possible options, we choose the node that achieves the minimum task completion time to allocate the task. The allocation of the next task will take the current task into account and repeat the same process. Finally, the job initiator will reserve the task execution time on all related nodes, which will be shared with other job initiators for future task allocation.

3.4.2 Predictable Contacts without Control Channel

When mobile nodes have no control channels, it is impossible to reserve task execution time in advance. WaterFilling will cause contention for task execution among different jobs on popular nodes, prolonging the task execution time. To solve this problem, we propose an algorithm framework, Computing on Dissemination (CoD), to allocate tasks in an opportunistic way. The algorithm is shown in Algorithm 2.

Algorithm 2 Computing on Dissemination

```
1: procedure ENCOUNTER(n)                                   ▷ n is the encountered node.
2:     summary ← getSummary();
3:     send(n, summary);
4: end procedure
5: procedure GETSUMMARY
6:     compute ← getNodeComputingSummary();
7:     net ← getNetworkSummary();
8:     tasks ← getPendingTaskSummary();
9:     return {compute,net,tasks};
10: end procedure
11: procedure RECEIVESUMMARY(n, msg)          ▷ msg is the summary message of node n.
12:     updateNodes(msg.compute);
13:     updateNetwork(msg.net);
14:     toExchange ← exchangeTask(n, this.tasks, msg.tasks);
15:     isSent ← false;
16:     while n.isConnected() && !toExchange.isEmpty() do
17:         send(n, toExchange.poll());
18:         isSent ← true;
19:     end while
20:     if n.isConnected() && isSent == true then
21:         summary ← getSummary();
22:         send(n, summary);
23:     end if
24: end procedure
25: procedure RECEIVETASK(msg)                          ▷ msg contains exchanged tasks.
26:     addTasks(msg.tasks);
27: end procedure
```

The basic idea of CoD is that during the task dissemination process, every intermediate node can execute these tasks. Instead of explicitly assigning a destination node to every task, CoD opportunistically disseminates the tasks among those encountered nodes until all tasks finish. Every time two nodes encounter each other, they first exchange metadata about their status. Based on this information, they decide the set of tasks to exchange. When they move out of the communication range, they will keep the remaining tasks to

execute locally or exchange with other encountered nodes in the future.

The key function of this algorithm is the *exchangeTask* function of line 14 that decides which tasks to exchange. In this subsection we assume that future contact is still predictable. Therefore, the task completion time can be estimated when the task arrives at a node as discussed in last subsection. The intuition of CoD with predictable contacts (pCoD) is to locally minimize the task completion time of every task if possible. When a node receives the summary message from the encountered node, it first estimates the execution time of its carried tasks on the other node using the job profiles and the device profiles. For each task it carries, it estimates the task completion time (i.e., the time that its result is received by the initiator) of executing locally and that of executing on the other node by using the contact information. If the local task completion time is larger than the remote one, it sends the task to the encountered node. Every node conservatively makes the decision without considering the tasks the other node will send back.

3.4.3 Unpredictable Contacts

Finally we consider the worst case that future contacts cannot be accurately predicted. Our task allocation algorithm, CoD with unpredictable contacts (upCoD), is still based on CoD with the constraint that future contact information is unavailable. As shown in Figure 5, minimizing the time when the last task is sent back to the job initiator will reduce the PNP-block completion time. When the data transfer time is unpredictable, we envision that reducing the execution time of the last task will also help reduce PNP-block completion time. This is because the locality property of CoD indicates the existence of a short time-space path between the worker node and the job initiator node. Therefore, when two nodes encounter each other, upCoD tries to reduce the execution time of every task.

In reality, historical contact information is useful to roughly estimate the future contacts [26] and, thus, should be helpful to task exchange in CoD. Its performance is probably between upCoD and pCoD. We will investigate such possibility as part of our future work.

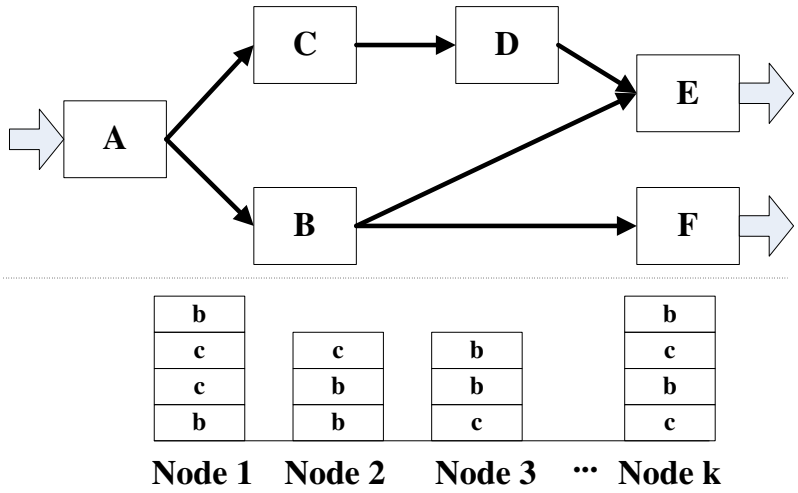


Figure 6: A job example where both PNP-block B and C are disseminated to Serendipity nodes after A completes. Their task positions in the nodes' task lists are shown blow the DAG.

3.5 PNP-block Scheduling

Our PNP-block design simplifies the task allocation so that every PNP-block is treated independently. However, it is still possible to further reduce the job completion time by assigning priorities to PNP-blocks since tasks from the same job are executed according to their priority assignment.

Let's consider a simple job DAG shown in Figure 6. PNP-blocks B and C are simultaneously allocated after A completes. Their tasks arrive at the destination nodes unordered. Given a network and a task allocation algorithm, the total time required for both B and C to finish remains almost the same. However, either B or C can have a shorter PNP-block finish time if any of them is given a higher priority over the other. This will be beneficial because their children PNP-block can start earlier.

Observation 1. *It is better to assign different priorities to the PNP-blocks of a job.*

In the example shown in Figure 6, PNP-block E can only start when both B and D finish. Thus, B and D are equivalently important to E. Meanwhile, there is a time gap between the execution time of C and that of D caused by the result collection of C and task dissemination of D. During that gap, the execution of other tasks (e.g., B) will not affect the PNP-block finish time of D. Therefore, if C is assigned a higher priority than B, the total time for both B and D to finish will be shorter.

Observation 2. *All parents of a PNP-block are equivalently important to it, while parents have higher priorities than their children.*

The next question arises when B and D are in the task list of the same node, which should have higher priority. We notice that both B and D are equivalent to E, while E and F are equivalent to the job. However, if B finishes earlier, F can start earlier. This is because F only relies on B.

Observation 3. *When two PNP-blocks have the same priority, the one with more children only depending on it should be assigned a higher priority.*

Algorithm 3 PNP-block Priority Assigning

```
1: procedure ASSIGNPRIORITY(J)                                  ▷ J is the job DAG
2:     while !J.allPNPblocksHavePriority() do
3:         for all s ∈ J do                                      ▷ s is a PNP-block
4:             if !s.haveChild() then
5:                 s.priority ← 0;
6:             else if s.allChildrenHavePriority() then
7:                 s.priority ← s.maxChildrenPriority()+1;
8:             end if
9:         end for
10:    end while
11:    for p = 0 → J.getMaxPriority() do
12:        PNPblocks ← J.getPNPblocksWithPriority(p);
13:        sort(PNPblocks);
14:        for i = 0 → PNPblocks.size()-1 do
15:            s ← PNPblocks.get(i);
16:            s.priority ← s.priority + i/PNPblocks.size();
17:        end for
18:    end for
19: end procedure
```

If there are still PNP-blocks with the same priority, we randomly assign some different priorities to them that keep their relative priorities with other PNP-blocks. Algorithm 3 shows our priority assigning algorithm. The sort method of line 13 is based on Observation 3.

3.6 Energy-Aware Computing

In the above two sections we focused on how to accelerate the job execution without any consideration of the energy consumption. Because of the limited energy available to some

kinds of mobile devices (e.g., smartphones), there are also scenarios when energy conservation is at least as important as execution performance, especially when the applications can tolerate delays. In this section, we describe how to support energy-aware computing with Serendipity.

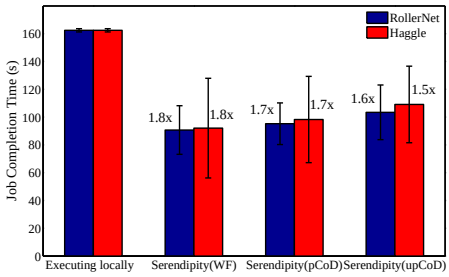
When a mobile device tries to off-load a task to another mobile device to save energy, the latter may have very limited energy, too. Meanwhile, if all nodes postpone the task execution forever, it definitely saves energy, but meaninglessly. Therefore, a reasonable objective of energy-aware computing among mobile devices makes all nodes last as long as possible while timely finishing the jobs, i.e., maximizing the lifetime of the first depleted node under the constraint that jobs complete before their deadline (i.e., TTL). Unfortunately, without information about the future jobs, it is impossible to solve this optimization problem.

An approximation to this ideal optimization is to greedily minimize a utility function when the job initiator allocates the tasks. Two factors should be considered in the utility functions, the energy consumption of all nodes involved in the remote computing of the task and the residual energy available to these nodes. A good utility function should consume less energy while avoiding nodes with small residual energy. We use a simple utility function that has been considered in energy-aware routing [29]:

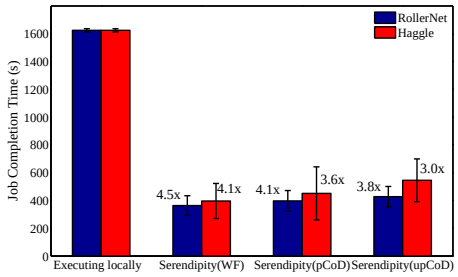
$$u(T) = \sum_{i \in N_T} \frac{e_{Ti}}{R_i} \tag{1}$$

where N_T is the set of nodes involved in the remote computing of task T, e_{Ti} is the energy consumption of node i for task T, and R_i is the residual energy of node i.

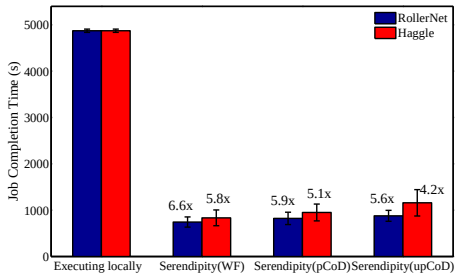
As discussed in Section 3.4 the task allocation algorithms, WaterFilling, pCoD and upCoD, try to optimize the job completion time. By replacing the time with the utility function $u(T)$, we can easily adapt these task allocation algorithms to be energy-aware. Specifically, the energy-aware WaterFilling algorithm iteratively chooses the destination node of every task with minimum $u(T)$ while satisfying the TTL constraint. When two nodes encounter, pCoD and upCoD will exchange a task if executing it on current node has higher utility than executing on the other node while satisfying the TTL constraint. If the future contacts are unpredictable, upCoD replaces TTL with the time that task is executed.



(a) 10 tasks



(b) 100 tasks



(c) 300 tasks

Figure 7: A comparison of Serendipity's performance benefits. The average job completion times with their 95% confidence intervals are plotted. We use two data traces, Haggle and RollerNet, to emulate the node contacts and three input sizes for each.

3.7 Evaluation

3.7.1 Experimental Setup

To evaluate Serendipity in various network settings, we have built a testbed on Emulab [96] to easily configure the experiment settings including the number of nodes, the node properties, etc. In our testbed, a Serendipity node running on an Emulab node has an emulation module to emulate the intermittent connectivity among nodes. Before an experiment starts, all nodes load the contact traces into their emulation modules. During the experiments, the emulation module will control the communication between its node and all other nodes according to the contact traces.

In the following experiments, we use two real-world contact traces, a 9-node trace collected in the Haggle project [59] and the RollerNet trace [95]. In the RollerNet trace, we select a subset of 11 friends (identified in the metadata of the trace) among the 62 nodes so that the number of nodes is comparable to the Haggle trace. The Haggle trace represents

the user contacts in a laboratory during a typical day, while RollerNet represents the contacts among a group of friends during the outdoor activity. These two traces demonstrate quite different contact properties. RollerNet has shorter contact intervals, while Haggle has longer contact durations.

We also use three mobility models to synthesize contact traces, namely the Levy Walk Model [81], the Random WayPoint Model (RWP) [82], and the Time-Variant Community Mobility Model (TVCM) [61]. We change various parameters to analyze their impact on Serendipity.

We implement a speech-to-text application based on Sphinx library [68] that translates audio to text. It will be used to evaluate the Emulab-based Serendipity. It is implemented as a single PNP-block job where the pre-process program divides a large audio file into multiple 2 Mb pieces, each of which is the task input.

To demonstrate how Serendipity can help the mobile computation initiator to speedup computing and conserve energy, we primarily compare the performance of executing applications on Serendipity with that of executing them locally on the initiator's mobile device. Previous remote-computing platforms (e.g., MAUI [33], CloneCloud [32], etc) don't work with intermittent connectivity and, thus, cannot be directly compared with Serendipity.

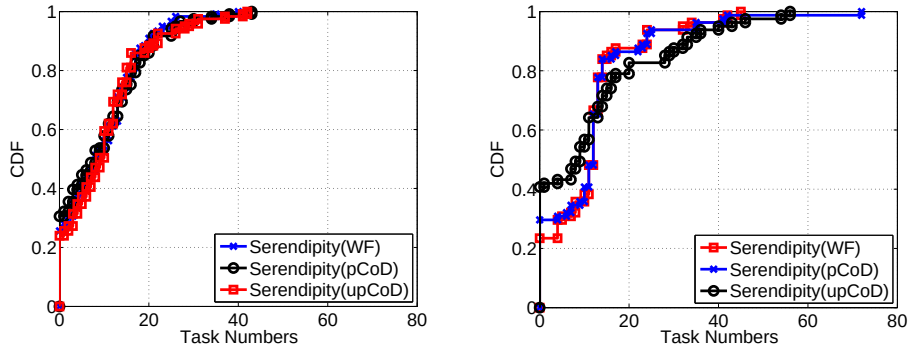
In all the following experiments every machine has a 600 MHz Pentium III processor and 256 MB memory, which is less powerful than mainstream PCs but closer to that of smart mobile devices. Every experiment is repeated 10 times with different seeds. The results reported correspond to the average values.

3.7.2 Serendipity's Performance Benefits

We initiate the experiments with the speech-to-text application using three workloads in three task allocation algorithms on both RollerNet and Haggle traces. The sizes of the audio files are 20 Mb, 200 Mb, and 600 Mb. As mentioned before, it is implemented as a single PNP-block job whose pre-process program divides the audio file into multiple 2 Mb pieces corresponding to 10, 100, and 300 tasks, respectively. The post-process program collects and combines the results. The baseline wireless bandwidth is set to 24 Mbps. We

also assume that all nodes have enough energy and want to reduce the job completion time.

Figure 7 demonstrates how Serendipity improves the performance compared with executing locally. We make the following observations. First, with the increase of the workload, Serendipity achieves greater benefits in improving application performance. When the audio file is 600 Mb, Serendipity can achieve as large as 6.6 and 5.8 time speedup. Considering the number of nodes (11 for RollerNet and 9 for Haggle), the system utilization is more than 60%. Moreover, the ratio of the confidence intervals to the average values also decreases with the workload, indicating all nodes can obtain similar performance benefits. Second, in all the experiments WaterFilling consistently performs better than pCoD which is better than upCoD. In the Haggle trace of Figure 7(c), WaterFilling achieves 5.8 time speedup while upCoD only achieves 4.2 time speedup. The results indicate that with more information Serendipity can perform better. Third, although Serendipity achieves similar average job completion times on both Haggle and RollerNet, their confidence intervals on Haggle are larger than those on RollerNet. This is because the Haggle trace has long contact interval and duration, resulting in the diversity of node density over the time.



(a) RollerNet (b) Haggle

Figure 8: The load distribution of Serendipity nodes when there are 100 tasks total, each of which takes 2 Mb input data.

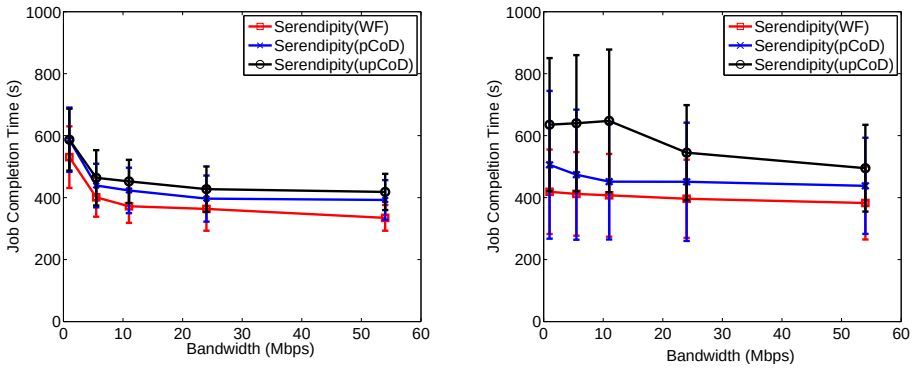
To further analyze the performance diversity, we plot the workload distribution on the Serendipity nodes of Figure 7(b) in Figure 8. In the RollerNet trace, all three task allocation algorithms have similar load distribution, i.e., about 25% nodes are allocated 0 tasks while about 10% of the nodes are allocated more than 20 tasks. In the Haggle trace, WaterFilling and pCoD have similar load distribution, while upCoD's distribution is quite different from them. The long contact intervals of the Haggle trace makes the blind task dissemination of

upCoD less efficient. In such an environment, the contact knowledge will be very useful to improve the Serendipity performance.

3.7.3 Impact of Network Environment

Next, we analyze the impact of the network environment on the performance of the three task allocation algorithms by changing the network settings from the base case.

Wireless Bandwidth: We first consider the effect of wireless bandwidth on the performance of Serendipity. The wireless bandwidth is set to be 1 Mbps, 5.5 Mbps, 11 Mbps, 24 Mbps, and 54 Mbps, which are typical values for wireless links. The audio file is 200 Mb, split into 100 tasks. We plot the job completion times of Serendipity with three task allocation algorithms in Figure 9.



(a) RollerNet (b) Haggle

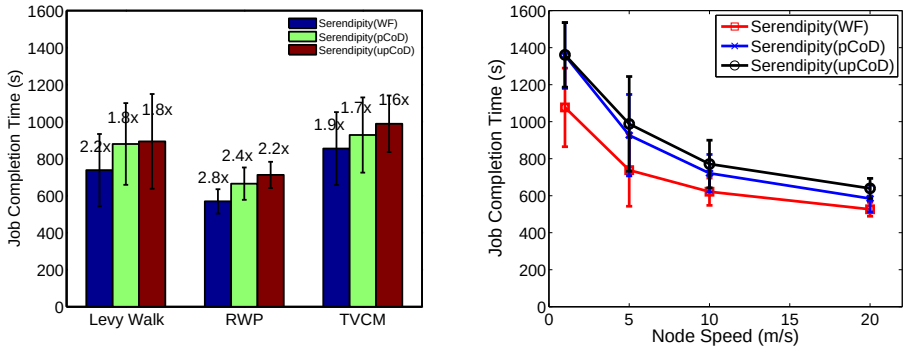
Figure 9: The impact of wireless bandwidth on the performance of Serendipity. The average job completion times are plotted when the bandwidth is 1, 5.5, 11, 24, and 54 Mb/s, respectively.

We observe the following phenomena. First, in RollerNet, all three task allocation algorithms accomplish similar performance. Because these nodes have frequent contacts with each other, using the locality heuristic (upCoD) is good enough to make use of the nearby computation resource for remote computing. Second, when the bandwidth reduces from 11 Mbps to 1 Mbps, the job completion time experiences a large increase. This is because RollerNet has many short contacts which cannot be used to disseminate tasks when the bandwidth is too small. Third, in the Haggle trace, the job completion time of upCoD increases from 545.0 seconds to 647.6 seconds when the bandwidth reduces from 24 Mbps to 11 Mbps. Meanwhile WaterFilling achieves consistently good performance in all

the experiments. This is because in the laboratory environment users are relatively stable and have longer contact durations. Thus, the primary factor affecting the Serendipity performance is the contact interval. On the other hand, since the contact distribution is more biased, only using locality is hard to find the global optimal task allocation.

Node Mobility: The above experiments demonstrate that contact traces impact the performance of Serendipity. To further analyze such impact, we use mobility models to generate the contact traces for 10 nodes. Specifically, we use Levy Walk Model [81], Random WayPoint Model (RWP) [82], and Time-Variant Community Mobility Model (TVCM) [61]. These models represent a wide range of mobility patterns. RWP is the simplest model and assumes unrestricted node movement. Levy Walk describes the human walk pattern verified by collected mobility traces. TVCM depicts human behavior in the presence of communities. The basic settings assume a 1 Km by 1 Km square activity area in which each node has a 100 m diameter circular communication range.

In this set of experiments we focus on the two most important aspects of node mobility, i.e., the mobility model and the node speed. The wireless bandwidth is set to 11 Mbps.



(a) Mobility Models (b) Node Speed

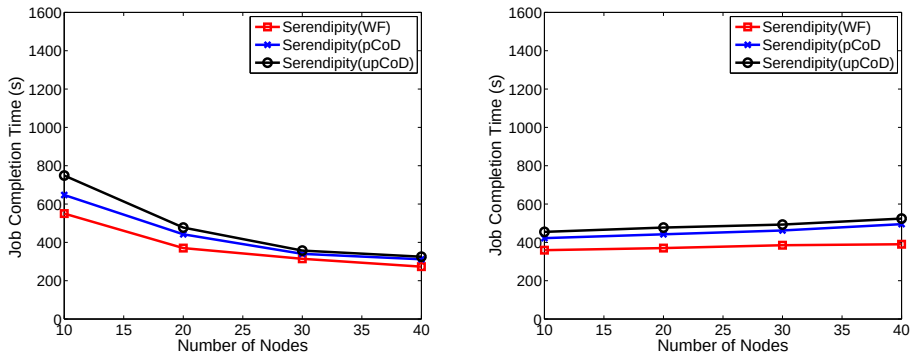
Figure 10: The impact of node mobility on Serendipity. We generate the contact traces for 10 nodes in a 1 km×1 km area. In (a) we set the node speed to be 5 m/s, while in (b) we use Levy Walk as the mobility model.

The results of this comparison are shown in Figure 10. Figure 10(a) shows that Serendipity has larger job completion time with all the mobility models than it had on Haggle and RollerNet traces. This is because their node densities are much sparser than Haggle and RollerNet traces. Thus it's harder for the job initiator to use other nodes' computation

resources. We also observe that Serendipity achieves the best performance when the RWP model is used. This is because RWP is the most diffusive [81] and, thus, results in more contact opportunities among nodes.

Node speed affects the contact frequencies and durations, which are critical to Serendipity. We vary the node speed from 1 m/s, i.e., human walking speed, to 20 m/s, i.e., vehicle speed. As shown in Figure 10(b), when the speed increases from 1 m/s to 10 m/s, the job completion times drastically decline, e.g., from 1077.1 seconds to 621.6 seconds for WaterFilling. This is because the increase of node speed significantly increases the contact opportunities and accelerates the task dissemination. When the speed further increases to 20 m/s, the job completion time is slightly reduced to 526.4 seconds for WaterFilling.

Number of Nodes: We finally examine how the quantity of available computation resources impacts Serendipity. To separate the effect of node density and resource quantity, we conduct two sets of experiments. In the first set, the active area is fixed, while in the second one, the active area changes proportionally with the number of nodes using the initial setting of 20 nodes in 1 km×1 km square area. Figure 11 shows the results where nodes follows RWP mobility model with wireless bandwidth at 2 Mbps.



(a) Fixed Active Area (b) Fixed Node Density

Figure 11: The impact of node numbers on the performance of Serendipity. We analyze the impact of both node number and node density by fixing the activity area and setting it proportional to the node numbers, respectively.

As shown in Figure 11(a), with the increase in the number of nodes in a fixed area, the job completion times of the three task allocation algorithms are reduced by more than 50%, from 550.0, 647.0, and 748.7 seconds to 273.0, 311.7, and 325.0 seconds for WaterFilling,

Table 1: A comparison of Serendipity's energy consumption. We report the number of jobs completed before at least one node depletes its battery and their average job completion time. Jobs arrive in a Poisson process with $\lambda = 0.005$ jobs per second.

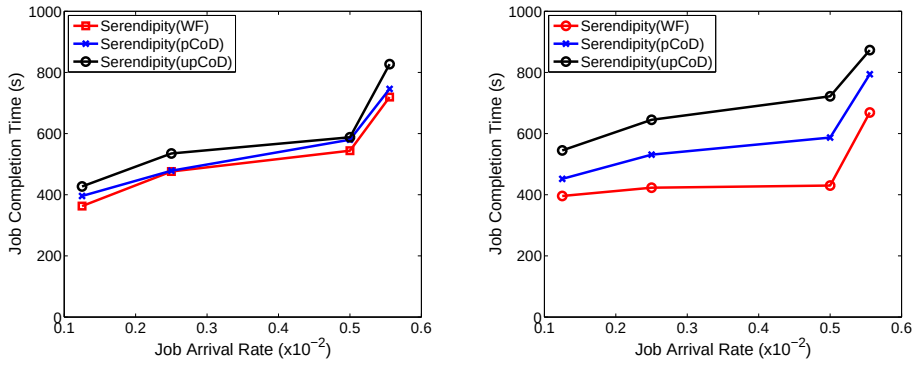
	Haggle				RollerNet			
	Energy Aware		Time Optimizing		Energy Aware		Time Optimizing	
	# Jobs	Time (s)	# Jobs	Time (s)	# Jobs	Time(s)	# Jobs	Time(s)
WF	17.0	2664.7	4.5	409.2	21.8	2823.9	2.5	496.2
pCoD	10.0	2162.4	3.0	435.3	16.8	2173.6	4.8	539.0
upCoD	9.3	2080.6	3.0	564.0	16.8	2082.6	3.5	562.7
Phone	N/A	N/A	1.3	1614.0	N/A	N/A	1.3	1614.0

pCoD and upCoD, respectively. Meanwhile, in Figure 11(b), the job completion times are almost constant despite the increase in node quantity.

3.7.4 The Impact of the Job Properties

Next we evaluate how the job properties affect the performance of Serendipity.

Multiple jobs: A more practical scenario involves nodes submitting multiple jobs simultaneously into Serendipity. These jobs will affect the performance of each other when their execution duration overlaps. In this set of experiments, nodes will randomly submit 100-task jobs into Serendipity. The arrival time of these jobs follows a Poisson distribution. We change the arrival rate, λ from 0.0013 (its system utilization is less than 20%) to 0.0056 (its system utilization is larger than 90%) jobs per second. Figure 12 shows the results on the RollerNet and Haggle traces.



(a) RollerNet (b) Haggle

Figure 12: Serendipity's performance with multiple jobs executed simultaneously. The job arrival time follows a Poisson distribution with varying arrival rates.

As expected, the job completion time increases with the job arrival rate. In both sets of

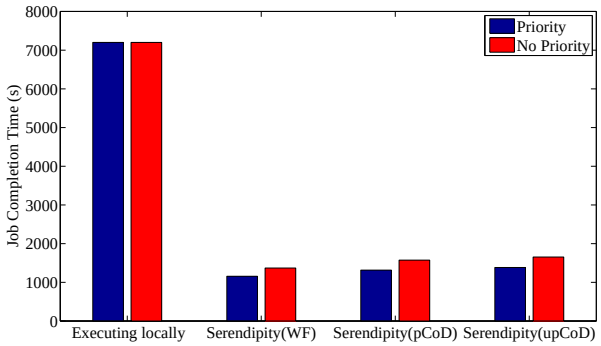


Figure 13: The importance of assigning priorities to PNP-blocks.

experiments, the job completion time gradually increases with the job arrival rate until 0.005 jobs per second and, then, drastically increases when the job arrival rate increase to 0.0056 jobs per second. According to queueing theory, with the system utilization approaching 1, the queueing delay is approaching infinity. However, even when the system utilization is larger than 90% (i.e., $\lambda = 0.0056$), the job completion times of Serendipity with various task allocation algorithms are still less than 54% of executing locally, showing the advantage of distributed computation.

DAG jobs: The above experiments show that Serendipity performs well for single PNP-block jobs. Since DAG jobs are executed iteratively for all dependent PNP-blocks while parallel for all independent PNP-blocks. The above experiment results also apply to DAG jobs. In this set of experiments we will evaluate how PNP-block scheduling algorithm further improves the performance of Serendipity.

We use the job structure shown in Figure 6, where the processing of one image impacts the processing of another. We use the PNP-blocks of speech-to-text application as the basic building blocks. PNP-block A has 0 tasks; B has 200 tasks; C has 50 tasks; D has 100 tasks; E has 100 tasks; F has 0 tasks. The performance difference between our algorithm and assigning equal priority to the PNP-blocks is shown in Figure 13.

Our priority assignment algorithm achieves the job completion time of 1155.8, 1315.8 and 1383.2 seconds for WaterFilling, pCoD, and upCoD, consistently outperforming that of 1369.2, 1573.4, and 1654.4 seconds when all PNP-blocks have the same priority. These experiments demonstrate the usefulness of priority assigning. Further evaluation of our

algorithm on diverse type of jobs will be part of our future work.

3.7.5 Energy Conservation

In this set of experiments, we demonstrate how Serendipity makes the entire system last longer by taking the energy consumption into consideration. We consider an energy critical scenario where node i has $E_i\%$ energy left, where E_i is randomly selected from [0, 20]. The energy consumption of task execution and communication is randomly selected from the measured values on mobile devices. The detailed measurement will be presented in the next section. In this set of experiments, nodes will randomly submit 100-task jobs into Serendipity. The arrival time of these jobs follows a Poisson distribution with $\lambda = 0.005$ jobs per second. We compare energy-aware Serendipity against "time-optimizing" Serendipity and executing jobs locally. The TTL of energy-aware Serendipity is set to twice the time of executing the job locally.

Table 1 shows the number of jobs completed before at least one node depletes its energy and the average job completion time of those completed jobs. We make the following observations. First, energy-aware Serendipity completes many more jobs than executing locally and using time-optimizing Serendipity. This is because energy-aware Serendipity balances the energy consumption of all the mobile devices through adaptively allocating more tasks to devices with more residual energy. In contrast, the time-optimizing Serendipity will quickly deplete the energy of some mobile devices by allocating many tasks to them. Second, through global optimization, energy-aware Serendipity with the WaterFilling allocation algorithm completes more jobs than those with pCoD and upCoD. Third, the job completion time of energy-aware Serendipity is much larger than that of time-optimizing Serendipity. There exists a tradeoff between energy consumption and performance. Finally, compared with executing locally, time-optimizing Serendipity both completes more jobs and has smaller job completion time. This is because statistically the few devices with limited residual energy will last longer by off-loading the computation to other devices.

3.8 Implementation

We implemented a prototype of Serendipity on the Android OS [2]. It comprises three parts: the Serendipity worker corresponding to the worker in Fig. 4, the Serendipity controller including all other components in Fig. 4 and a user library providing the key APIs for application development.

We currently use *WifiManager*'s hidden API, *setWifiApEnabled*, to achieve the ad hoc communication between two devices, i.e., one device acts as an AP while the other device connects to it as a client.

We use the Java reflection techniques to dynamically execute the tasks. Every task has to implement the function *execute* defined in the APIs. When the Serendipity worker executes a task, it executes this function.

The separation between the Serendipity worker and the Serendipity controller is based on access control. Android's security architecture defines many kinds of permission to various resources including network, GPS, sensors, etc. The Serendipity worker is implemented as a separate application with limited access permission to these resources, acting as a sandbox for the task execution. When the Serendipity controller receives a task to execute, it will start a Senredipity worker and get the results from it.

3.8.1 System Evaluation

To evaluate our system, we implemented two computationally complex applications, a face detection application, and a speech-to-text application. The face detection application takes a set of pictures and uses computer vision algorithms to identify all the faces in these pictures [54]. It is implemented as a single PNP-block job where the face detection in each picture is a task. The speech-to-text application takes an audio file and translates the speech into text using the Sphinx library [68]. It is also a single PNP-block job where the pre-process program divides a large audio file into multiple pieces, each of which is input to a separate task.

We tested Serendipity on a Samsung Galaxy Tab with a 1 GHz Cortex A8 processor and a Motorola ATRIX smartphone with a dual-core Tegra 2 processor, each at 1 GHz.

Both of them run the Android 2.3 OS. The face detection and speech-to-text applications are used for evaluation.

Table 2: The execution time of two applications on two devices.

	Input size (Mb)	Galaxy Tab (s)	ATRIX (s)
FaceDetection	2.2	17.9	7.2
Speech-to-text	3.0	40.3	18.8

We first executed the two applications locally on the two devices. As Motorola ATRIX smartphone has a dual-core processor, we split the input files into two parts of equal size and simultaneously executed the two tasks to fully utilize its processor. Table 2 shows their execution times. We also measured the TCP throughput between these two devices by sending 800 Mb data. We obtain 10.8 Mbps throughput on average when they are within 10 meters. In fact, they still achieve 5.9 Mbps throughput even when they are more than 30 meters away.

To assess the performance of Serendipity, we construct a simple network in which the two devices are consistently connected during the experiments. As expected, Serendipity speeds up more than 3 times than executing the applications on the Samsung Galaxy Tab.

To generate the energy consumption profiles of the two applications on these mobile devices, we repeatedly execute those applications starting with full battery until the batteries are depleted and count the number of iterations. Similarly, WiFi's energy profiles are obtained by continuously transferring data between them. Table 3 demonstrates the results.

Table 3: The energy consumption of mobile devices. The ratios of consumed energy to the total device energy capacity are reported.

	Input size (Mb)	Galaxy Tab	ATRIX
FaceDetection	2.2	4.14×10^{-4}	3.44×10^{-4}
Speech-to-text	3.0	9.32×10^{-4}	9.01×10^{-4}
WiFi	800	8.02×10^{-4}	2.04×10^{-3}

The energy required to transfer a task only accounts for 0.5 % (i.e., $\max(\frac{8.02\times2.2}{4.14\times800}, \frac{8.02\times3.0}{9.32\times800})$) and 1.6% (i.e., $\max(\frac{20.4\times2.2}{3.44\times800}, \frac{20.4\times3.0}{9.01\times800})$) of the energy required to execute the task on these devices, respectively. It indicates that Serendipity won't consume much extra

energy. Instead, by delegating tasks to devices with a lot of energy, it can significantly save the job initiator's energy.

We use an extreme example to show the gains of energy-aware Serendipity. Suppose the ATRIX phone has a lot of pictures for face detection. Assume it only has 5% energy left, and the Galaxy tablet has 50% energy left. Energy-aware Serendipity can detect about 1320 pictures before the ATRIX phone depletes its battery, while time-optimizing Serendipity can only detect about 203 pictures.

3.9 Summary

In this chapter we have developed and evaluated the Serendipity system that enables a mobile device to remotely access computational resources on other mobile devices it may encounter. The main challenge we addressed is how to model computational tasks and how to perform task allocation under varying assumptions about the connectivity environment. Through an emulation of the Serendipity system we have explored how such a system has the potential to improve computation speed as well as save energy for the initiating mobile device. We have also reported on a preliminary prototype of our system on Android platforms.

As mentioned previously we envision Serendipity as developed here to enable an extreme of a spectrum of remote computation possibilities that are available to mobile devices. In the next two chapters we will investigate how to utilize powerful cloud computing resources for computation offloading.

CHAPTER IV

IC-CLOUD: COMPUTATION OFFLOADING TO AN INTERMITTENTLY CONNECTED CLOUD

4.1 Introduction

The idea of offloading computation from mobile devices to remote servers to improve performance and reduce energy consumption has been around for more than a decade [16,17]. Its usefulness hinges on the ability to achieve computation speedups with small communication cost. In recent years, this idea has received more attention because of the significant rise in the sophistication of mobile computing applications and the availability of improved connectivity options for mobile devices. Some commercial applications such as Siri [9] have had the goal to provide sophisticated services to mobile users pervasively. Through dynamically identifying the offloadable tasks at runtime, recent work [32, 33, 45, 65] has aimed to generalize this approach to benefit more mobile applications without the burden of offloading logic.

Complicating the offloading function today is the fact that mobile users typically experience intermittent connectivity to the Internet and highly variable access quality even when connectivity exists. According to recent studies [20, 36, 70], 3G access is only available 87% of the time even in a metropolis, while WiFi coverage is even more intermittent. Figure 14 shows an example scenario where a mobile device is experiencing variable and intermittent connectivity. The uncertainties in connectivity make computation offloading challenging in two ways. First, it is hard to accurately estimate the communication cost and computation time, both of which are needed to make the offloading decision. Second, it requires computation-offloading systems to properly handle the uncertainty to avoid degrading performance.

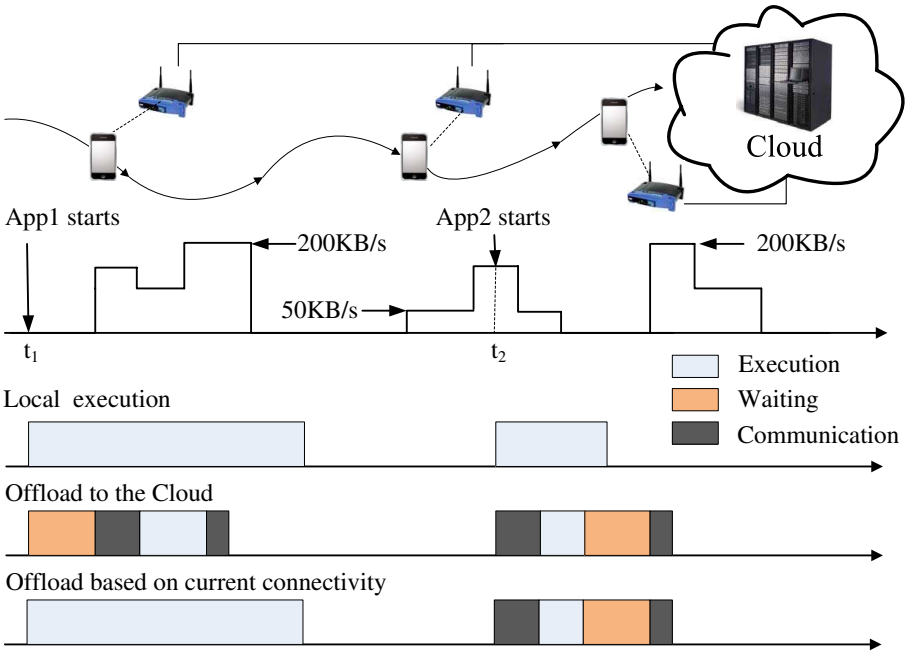


Figure 14: Simple examples showing the impact of intermittent connectivity on computation offloading.

Previous systems for computation offloading have often assumed stable network connectivity making them perform poorly when connectivity characteristics are variable and uncertain. Figure 14 provides some examples demonstrating how computation-offloading systems that do not explicitly handle intermittent or variable connectivity may degrade the application's performance. A mobile user connects to the cloud with varying access quality from time to time. She starts two computation-intensive applications at t_1 and t_2, respectively. Consider three simple strategies for computation offloading, i.e., *Local-execution*, *Offload-to-cloud* and *Offload-based-on-current-connectivity*. For App1, *Offload-to-cloud* achieves the best performance because App1 has long local execution time and, thus, may benefit from waiting for future connectivity to offload computation. Meanwhile, for App2, *Local-execution* achieves the best performance because the mobile device loses connectivity before receiving the results. *Offload-based-on-current-connectivity* has the worst performance for both applications. It should be noted that none of these simple strategies are able to always achieve good performance. Thus a robust solution must be able to adapt the strategy.

In this chapter, we propose IC-Cloud, a computation-offload-ing system that is designed

to handle all the above-mentioned challenges in the mobile environment. To achieve this, IC-Cloud uses two key techniques: lightweight connectivity prediction and prediction use in a risk-controlled manner to make offloading decisions. Our connectivity-prediction algorithm only uses the signal strength and user historical information to obtain a coarse-grained estimation of the network access quality. Acknowledging the uncertainties in these predictions, we propose a risk-control algorithm to reduce the impact of inaccurate predictions.

We have implemented a prototype of IC-Cloud on Android and tested the system using a Samsung Galaxy Tab equiped with both WiFi and 3G and an 8-core server for offloading. We modified three applications (i.e., face detection, voice recognition and chess) to use IC-Cloud for offloading. We conducted extensive experiments in three different mobile environment. In all these experiments, IC-Cloud helps improve the performance of mobile applications and reduce the energy consumption. It achieves 4.1x speedup and reduces energy consumption to 22% in some scenarios.

The rest of the chapter is organized as follows. Section 4.2 presents the overview of IC-Cloud's architecture. The design details of connectivity prediction and computation offloading are described in Section 4.3 and 4.4, respectively. The evaluation of IC-Cloud is provided in Section 4.5. Section 4.6 concludes the chapter.

4.2 IC-Cloud Architecture

IC-Cloud aims to achieve effective computation offloading in a mobile environment where Internet access to remote computation resources is of variable quality and even intermittent. Figure 35 shows the high-level architecture of IC-Cloud. On the mobile device, IC-Cloud consists of six major components: 1) a *connectivity predictor* that monitors the network states and maintains a database of the historical information of the network states; 2) a *connectivity manager* that handles data transfer between the mobile device and the cloud; 3) an *execution predictor* that collects program features from the applications and predicts the execution time of tasks considered for offloading; 4) a set of *application trackers* each of which monitors the offloaded tasks for an application and adjusts its strategy based on the connectivity over time; 5) an *offloading controller* that uses the information from the

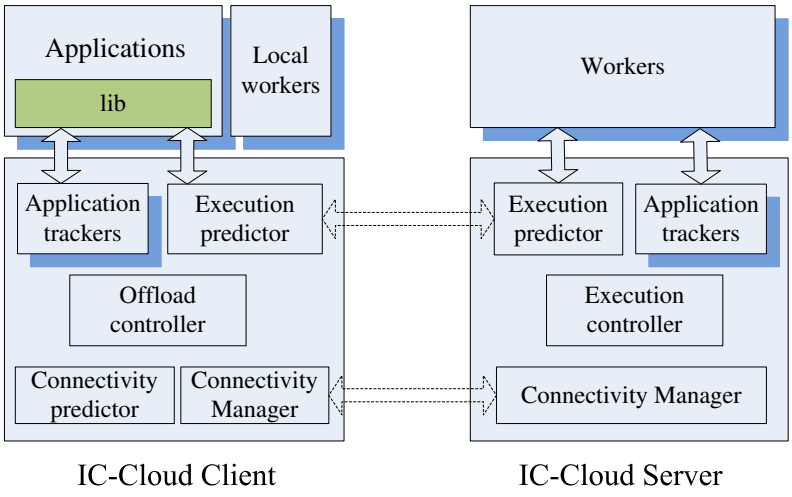


Figure 15: Overview of IC-Cloud system architecture.

connectivity predictor and the execution predictor to decide if a task should be offloaded to the cloud; 6) *local workers* that may execute some offloaded tasks in the case that they cannot return the results in time.

On the server, IC-Cloud consists of four major components: 1) an *execution predictor* that tracks the program features of the applications and sends them back to the mobile device; 2) a set of *application trackers* that monitor the execution of offloaded tasks; 3) an *execution controller* that controls the execution of the offloaded tasks; 4) a *connectivity manager* that communicates with the mobile devices.

Connectivity prediction is critical to the performance of IC-Cloud as the offloading decision relies on the prediction of future connectivity. There are two major concerns in its design: accuracy and energy-efficiency. Our main idea is to use the signal strength and the user's historical information to predict connectivity. Many studies [37, 74] have shown that it is sometimes possible to predict user mobility and, thus, their connectivity using the user's historical information. However, many of these prediction mechanisms are energy-consuming as they require GPS location to achieve accurate prediction. Instead of trying to obtain accurate connectivity prediction, IC-Cloud only uses the perceived signal strength to achieve coarse-grained prediction of the connectivity in an energy-efficient way and lets the offloading controller handle the uncertainties in the prediction. We provide further details

in Section 4.3.

Similar to MAUI [33] and ThinkAir [65], IC-Cloud provides a library to application developers and allows them to annotate all the tasks to be considered for offloading. IC-Cloud will also instrument these applications to collect features for all these offloadable tasks. Then IC-Cloud uses Mantis [31] to accurately predict their execution time based on these features.

Using the information from connectivity prediction and execution prediction, the offloading controller estimates the potential benefits to offload the computation. Due to the inherent uncertainties in user mobility and the connectivity prediction, the offloading controller will sometimes make wrong decisions. Therefore, it is essential to take the risk into account. In addition, different applications may tolerate different risks. For example, interactive applications (e.g., games) should be executed before their deadlines and are less tolerant to extra delays, while some background applications (e.g., virus scanning) can tolerate occasional extra delays. Therefore, IC-Cloud should allow applications to specify their tolerance to risks. The detailed design of the offloading controller is described in Section 4.4.

4.3 Connectivity Prediction

A fundamental challenge to the design of IC-Cloud is how to estimate the communication cost in the mobile environment. When offloading a piece of computation to the cloud, the communication cost consists of the time to transfer the data from the mobile device to the cloud and the time to return the result to the mobile device after execution. When the connectivity is intermittent, the cost may also include the time to wait for connectivity in either direction, making the estimation even more complicated.

A heuristic solution to this problem is to predict the Internet access quality and then use the predicted values to assess the communication cost. Developing a highly accurate prediction method will be beneficial to IC-Cloud but is out of the scope of this thesis. We expect a simple and energy-efficient method to have advantages over greater accuracy but

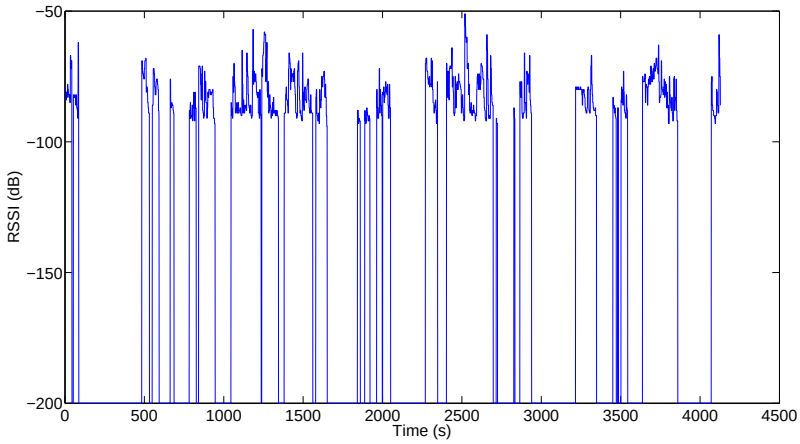


Figure 16: The WiFi signal strength measured on a campus shuttle. When the mobile device disconnects from WiFi, the signal strength is set to -200.

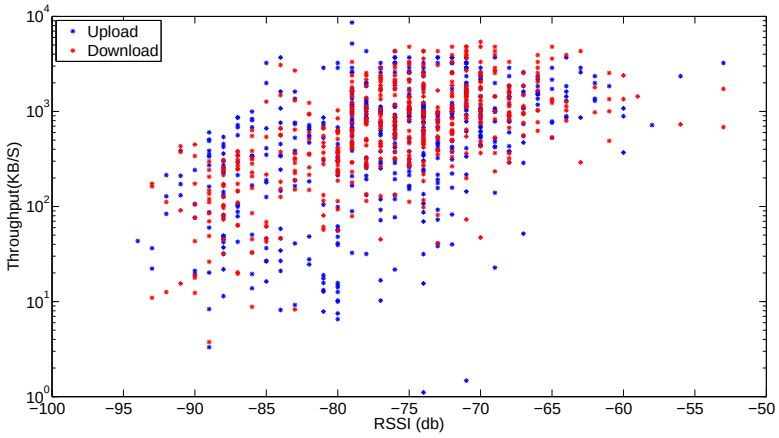


Figure 17: Measured throughput vs. WiFi signal strength.

also greater energy, which predicts future connectivity with user historical information. IC-Cloud maintains a database of the perceived signal strengths and the achieved throughput when using the network. Such information is easy to obtain and requires little energy to collect. This prediction method can be replaced by other more accurate methods. Our focus is on profiling the connectivity and assessing the communication cost for computation offloading.

To illustrate how dynamic Internet access quality impacts the communication cost of computation offloading, we plot the measured WiFi signal strength on a Georgia Tech campus shuttle in Figure 16 and the throughput to a server in our lab in Figure 17. The

figures demonstrate three key properties that impact the estimation of communication cost: intermittent connectivity, varying signal strength over time and uncertain throughput given the signal strength. The latter two properties are also very common for WiFi in the indoor environment and cellular 3G data access. We describe how to profile each of these three properties in the following three subsections.

4.3.1 Intermittent Connectivity

Intermittent connectivity primarily impacts the estimation of communication cost in the following two ways. First, if the mobile device disconnects from the server when a task is to be offloaded, IC-Cloud needs to decide if it should wait for the next connectivity to offload the task. If the local execution time is very long while the next connectivity will come soon, it may be beneficial to wait. Otherwise, it should start to execute the task immediately on the mobile device. Second, it is also possible that the mobile device will lose connectivity to the server after offloading the task. Therefore, before offloading the task IC-Cloud should estimate the possibility of disconnection before obtaining the result from the server. In addition, if it loses connectivity in the middle of offloading, it also needs to decide whether to wait for the result or restart a local worker for the task. To assess the communication cost under intermittent connectivity, we need to estimate the residual duration of current connectivity, if connected, or the start time of the next connectivity, if disconnected.

Let us use C to denote the duration of a contact during which the mobile device can always communicate with the server and use D to denote the duration of an inter-contact during which the mobile device totally loses connectivity to the server. Let us also use C_t to represent the duration of current contact until time t if it is connected at time t and use D_t to represent the duration of the inter-contact until time t. Finally, let $R_{C,t}$ and $R_{D,t}$ denote the residual duration of a contact and that of a inter-contact since time t, respectively. Therefore, the expected values and the standard deviations of the residual durations can be computed as follows:

$$E(R_{X,t}) = \int_{X_t}^{+\infty} \frac{(X - X_t)f(X)}{P(X \geq X_t)} dX \tag{2}$$

$$\sigma(R_{X,t}) = \sqrt{\int_{X_t}^{+\infty} \frac{(X - X_t)^2 f(X)}{P(X \geq X_t)} dX} \tag{3}$$

where X is either C or D.

There are also the cases where C_t or D_t are not available. For example, the user just restarted the mobile device. Let's consider the scenario that it is within a contact at time t. The possibility that it is within a contact of duration C is proportional to its duration, i.e., $\frac{Cf(C)}{E(C)}$. Since $R_{C,t}$ can be any value within $[0, C]$ under the condition that the contact duration is C, its expected value is $\frac{C}{2}$. Thus, the expected value and the standard deviation of $R_{C,t}$ are:

$$E(R_{C,t}) = E(\frac{Cf(C)}{E(C)} \times \frac{C}{2}) = \frac{E(C^2)}{2E(C)} \tag{4}$$

$$\begin{aligned} \sigma(R_{C,t}) &= \sqrt{\int_0^{+\infty} \frac{Cf(C)}{E(C)} \int_0^C \frac{(x - \frac{C}{2})^2}{C} dx dC} \\ &= \sqrt{\frac{E(C^3)}{12E(C)}} \end{aligned} \tag{5}$$

The results for $R_{D,t}$ are similar to the above two equations, i.e., replacing C with D in these equations.

4.3.2 Varying Signal Strength

IC-Cloud uses the signal strength as an indicator of Internet access quality because the wireless interface is usually the bottleneck of network performance in mobile environment. The communication cost of computation offloading includes both the time of sending data to the cloud and that of receiving the result from the cloud. The former uses the current signal strength for estimation, while the latter requires an estimate of the distribution of future signal strength and uses it for cost estimation. In this subsection, we describe how to obtain the distribution of future signal strength based on user historical information.

Our method is to use the current signal strength and the statistical distribution of user historical information to obtain that distribution. To demonstrate how current signal

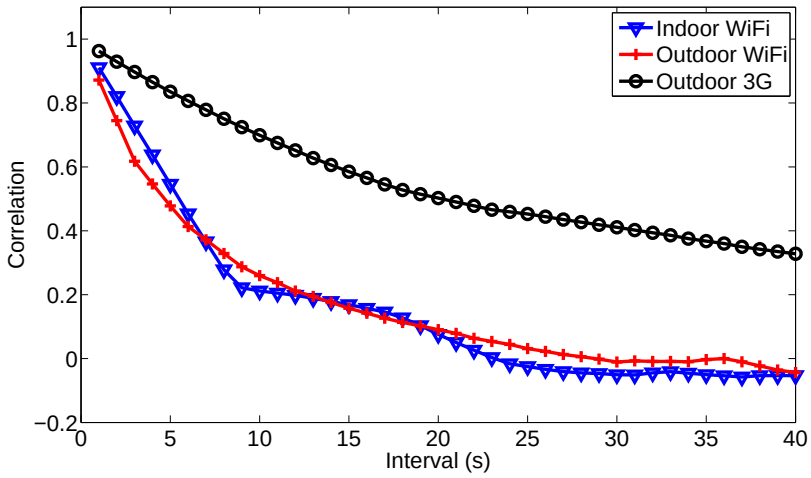


Figure 18: The correlation between current WiFi signal strength with future signal strength. The X-axis is the time difference.

strength can be used to obtain the distribution of future signal strength, we plot the correlation between the signal strength at time t and that at time $t + \Delta t$ for three different mobile environments in Figure 18. "Indoor WiFi" corresponds to the WiFi measurement conducted by a student randomly walking in a building with good WiFi coverage; "Outdoor WiFi" refers to WiFi measurement conducted on a Georgia Tech campus shuttle, the same as Figure 16; "Outdoor 3G" refers to 3G measurement conducted by a student on the commute between home and school. When Δt is small, the correlations in all these scenarios are high. However, the correlation of WiFi signal strength (both indoor and outdoor) quickly drops from about 0.9 to about 0.2 when Δt increases from 1 second to 10 seconds. Meanwhile the correlation of 3G signal strength is still about 0.4 when $\Delta t = 30$. If the correlation is larger than a threshold, we can obtain the distribution of future signal strength using the current signal strength.

Let us use $\langle x(t), x(t + \Delta t)\rangle$ to denote the pair of signal strengths at time t and $t + \Delta t$. We simply assume that $x(t)$ follows the normal distribution $N(u, \sigma^2)$ and use $\rho_{\Delta t}$ to denote the correlation between $x(t)$ and $x(t + \Delta t)$. In the implementation, u, σ^2 and $\rho_{\Delta t}$ are obtained from user historical information. Then, $\langle x(t), x(t+\Delta t)\rangle$ follows a bivariate normal

distribution:

$$N\left(\begin{pmatrix} u \\ u \end{pmatrix}, \begin{pmatrix} \sigma^2 & \rho_{\Delta t}\sigma^2 \\ \rho_{\Delta t}\sigma^2 & \sigma^2 \end{pmatrix}\right)$$

Using the conditional distribution of bivariate normal distribution [24], we can obtain the distribution of $x(t+\Delta t)$ given $x(t)$ as:

$$N(u + \rho_{\Delta t}(x(t) - u), (1 - \rho_{\Delta t}^2)\sigma^2) \tag{6}$$

It is noteworthy that when $\rho_{\Delta t}$ is small, the variance, $(1 - \rho_{\Delta t}^2)\sigma^2$, will be large, indicating inaccurate estimation. In addition, the value of $\rho_{\Delta t}$ will also be biased in the implementation because we use the historical information to approximate it. Therefore, in our implementation, when $\rho_{\Delta t}$ is smaller than a threshold (e.g., 0.4), IC-Cloud simply uses the overall statistical distribution of signal strength, namely, $N(u, \sigma^2)$, to describe the distribution.

4.3.3 Uncertain Throughput

To estimate the time of sending the data and receiving the result, IC-Cloud needs to predict its current throughput (for sending) as well as the future one (for receiving). There are three challenges in the prediction. First, since IC-Cloud uses the signal strength as the indicator of Internet access quality, it is hard to predict the throughput accurately. As shown in Figure 17, for any specific value of the signal strength, the measured throughput usually has high variance. Second, IC-Cloud uses historical information to obtain the relation between signal strength and throughput. It may be biased, especially when the data is sparse. Third, as discussed in the above subsection, we can only obtain a range of future signal strength, making it even harder to estimate the corresponding throughput.

To handle the uncertainties and make robust estimation, we divide the signal strength into several categories and generate a throughput distribution for each category using the historical information. Figure 19 shows an example using the measurement on the Georgia Tech campus shuttle. The throughput distributions of the various categories are quite different.

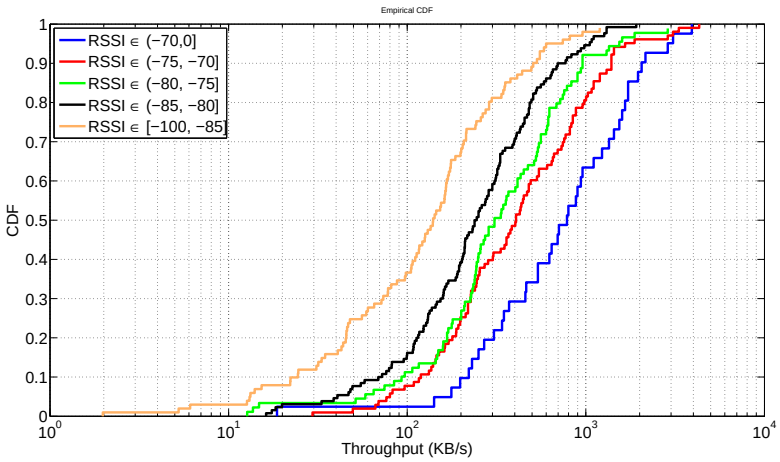


Figure 19: The distributions of the measured WiFi throughput on a Georgia Tech campus shuttle. They are divided into five categories based on the signal strength.

4.4 Computation Offloading

The offload controller of IC-Cloud uses the information from the connectivity and execution predictors to decide how to offload the computation-intensive tasks of mobile applications. Ideally, if future connectivity and execution time can be accurately predicted immediately after the mobile application starts, a global optimal solution [88] can be used to make the offloading decision. However, such global optimum is unavailable in the mobile environment investigated in this chapter because of two reasons. First, the dynamic Internet access quality makes it hard to accurately predict the communication costs for all the offloadable functions, especially for those to be called later. Second, some of the application features essential for the execution prediction of offloadable functions may be unavailable until these functions are about to be called.

Instead, the offload controller uses a greedy strategy to make the offloading decision. Every time when an offloadable function is called, the offload controller determines if it is beneficial to offload the function. Because of the uncertainty inherent in the mobile environment, the offloading decision takes risk into consideration. In case a bad decision has been made, it will also adjust its strategy with new information available. Meanwhile, for functions capable of executing concurrently, the offload controller will maximize their overall benefit. We describe these design details in the following subsections.

4.4.1 Offloading Gain

When an offloadable function is called at time t, the offload controller needs to determine if it is beneficial to offload this function to the cloud. Let us use T_{ws} to denote the time to wait for connectivity before sending the data, T_s for the time to send the data, T_c for the execution time in the cloud, T_{wr} for the time to wait for connectivity before receiving the result, T_r for the time to receive the result, and T_l for the local execution time on the mobile device. Let G represent the gain of offloading the function. Therefore,

$$G = T_{ws} + T_s + T_c + T_{wr} + T_r - T_l. \tag{7}$$

In Formula 7, T_c and T_l are independent of network connectivity and can be estimated using Mantis [31]. Meanwhile, T_{ws}, T_s, T_{wr} and T_r can be estimated using the information from the connectivity predictor.

Conceptually, when $G > 0$, it will be beneficial to offload the function. However, because of the uncertainties in the mobile environment, the offload controller can only obtain a distribution for G (i.e., $E(G)$ and $\sigma^2(G)$). Simply using $E(G)$ to make the offloading decision will introduce the risk of longer execution time and, thus, cause bad user experience. Therefore, controlling the risk in offloading is very important. We describe the risk-control mechanism of the offload controller in the next subsection.

4.4.1.1 The Computation of Offloading Gain

In this subsection, we describe how to compute the offloading gain G of a single function at time t. Depending on the connectivity at time t, T_{ws} and T_s have different distributions. In the case that the mobile device connects to the cloud at time t, $T_{ws} = 0$; $T_s = \frac{d_s}{b_u(t)}$, where d_s is the data size, and $b_u(t)$ is the upload bandwith at time t. d_s is available at time t, while $b_u(t)$ can be estimated using the current signal strength as described in Section 4.3.3. Otherwise, $T_{ws} = R_{D,t}$, which can be obtained according to Equation 2 and 3; $T_s = \frac{d_s}{b_u^*}$, where b_u^* is the overall upload bandwidth of the entire trace.

The value of T_{wr} depends on whether the mobile device still connects to the cloud when the cloud finishes execution at time $t + T_{ws} + T_s + T_c$. If connected, $T_{wr} = 0$. Otherwise,

$$T_{wr} = \begin{cases} D - R_{C,t} + T_s + T_c, & \text{if connected at time } t \\ D - C + T_s + T_c, & \text{otherwise} \end{cases} \quad (8)$$

where C and D are contact duration and inter-contact duration, respectively.

The value of T_r also depends on the connectivity at time $t' = t + T_{ws} + T_s + T_c$. If connected, $T_r = \frac{d_r}{b_d(t')}$, where d_r is the result size, and $b_d(t')$ is the download bandwidth. $b_d(t')$can be estimated according to Formula 6. Otherwise $T_r = \frac{d_r}{b_d^*}$, where b_d^* is the overall download bandwidth.

According to the above analysis, T_{wr} is directly related to T_s and T_c. T_r is indirectly related to T_s and T_c as $T_s + T_c$ may impact the distribution of signal strength which impacts T_r. However, this correlation is small and, thus, be ignored in the implementation for simplicity. Other variables are independent of each other. Therefore, the variance of offloading gain can be computed using

$$\begin{aligned} \sigma^2(G) &= \sigma^2(T_{ws}) + \sigma^2(T_s) + \sigma^2(T_c) + \sigma^2(T_{wr}) \\ &\quad + \sigma^2(T_r) + \sigma^2(T_l) + 2\sigma(T_s + T_c, T_{wr}) \end{aligned} \quad (9)$$

4.4.2 Risk Control

Our risk-controlled offloading is based on two key ideas. First, we use risk-adjusted return [23] in making the offloading decision so that the return and risk of offloading are simultaneously considered. Second, we re-evaluate the return and risk with new information available. The algorithm is shown in Algorithm 4.

When a computation task is initiated, the offloading controller evaluate its return and risk of offloading gain. $E(G)$ and $\sigma(G)$ are used as the return and risk, respectively. The detailed algorithms to compute them are described in the appendix. If the risk-adjusted return (i.e., $\frac{E(G)}{\sigma(G)}$) is larger than a threshold, the offloading controller offload the task to the cloud. In addition, it also listens to the connectivity status which has high impact on $E(G)$ and $\sigma(G)$. Once new connectivity information is updated, it re-evaluates the risk-adjusted return and adjust its decision accordingly.

Algorithm 4 Risk Controlled Offloading

```
1: procedure OFFLOADING(O_k)                          ▷ O_k is the computation task.
2:     if riskAdjustedOffloading(O_k) then
3:         offloadedTask ← O_k;
4:         registerReceiver(this,CONNECTIVITY);
5:     end if
6: end procedure
7: procedure RISKADJUSTEDOFFLOADING(O_k)
8:     gain ← getOffloadingGain(O_k);
9:     risk ← getOffloadingRisk(O_k);
10:    if gain/risk ≥ α then
11:        offload(O_k); return true;
12:    else
13:        localExecute(O_k)
14:        unregisterReceiver(this); return false;
15:    end if
16: end procedure
17: procedure ONRECEIVE(conn)                         ▷ conn is the connectivity status.
18:    riskAdjustedOffloading(offloadedTask);
19: end procedure
20: procedure RECEIVERESULT(O_k)
21:    offloadedTask ← NULL;
22:    unregisterReceiver(this);
23: end procedure
```

4.5 Evaluation

In this section, we evaluate how IC-Cloud improves the performance and energy consumption of various mobile applications in different types of mobile environments.

4.5.1 Methodology

We implemented our prototype of IC-Cloud (i.e., both IC-Cloud server and IC-Cloud client) on the Android OS. The IC-Cloud server runs on Android x86. We use a server with an 8-core 3.4GHz CPU, running VirtualBox 4.1.22, in our lab. The IC-Cloud client runs on a Samsung Galaxy Tab with a 1.0GHz Cortex A8 processor and equipped with both WiFi and 3G connections.

We evaluate IC-Cloud's benefits in three different mobile scenarios:

- *Indoor WiFi*: A student carrying a mobile device randomly walks in our department's building which has good WiFi coverage. WiFi is used for Internet access. In this mobile environment, the mobile user will experience varying signal strength as WiFi APs have limited communication range. However, intermittent connectivity is less

frequent.

- *Outdoor WiFi*: A student carrying a mobile device takes a shuttle running on the Georgia Tech campus. WiFi is used for Internet access. In this scenario, the mobile user will experience both varying signal strength and frequent intermittent connectivity.

- *Outdoor 3G*: A student is on his commute between home and school. The mobile device accesses the Internet through an EVDO network of a large US 3G carrier. Compared with WiFi, it has lower bandwith and longer delays but better coverage.

For each scenario, we first measure the network connectivity and construct a database for it. During the experiments, we use those databases as user historical information.

We modified three existing mobile applications to use IC-Cloud for computation offloading:

- FaceDetect is a face detection application that uses APIs in the Android SDK to detect all the faces in a given picture. We collected a data set of pictures containing faces from Google Image. During the experiments, each time we randomly choose a picture in the data set as input to the application.

- VoiceRecog is an Android port of the speech recognition program PocketSphinx [58]. For simplicity of experiments, we also modified the application to use audio files as input. We created a set of audio files with different lengths in advance. During the experiments, we randomly select a file as the input each time.

- DroidFish is an Android port of the Chess engine Stockfish [5] that allows users to set the strength of the AI to play with. Because computation requirement changes with the chosen AI player's strength, the user in our experiments randomly changes the strength before each move.

To evaluate the benefits of IC-Cloud, we use three offloading baselines:

- *LocalExe* executes all offloadable functions locally on the mobile devices. It provides a fundamental baseline to demonstrate the benefit of computation offloading.

- *Oracle* assumes accurate knowledge of all connectivity and execution profile information necessary to make the offloading decision. It represents the upper-bound of offloading benefits.

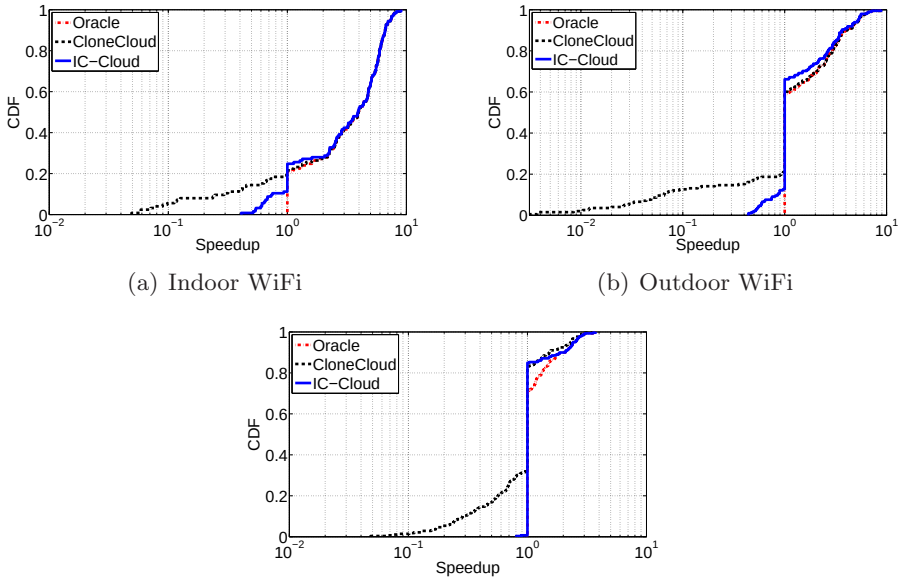
- *CloneCloud* is a basic offloading system in which each mobile device has a server in the cloud that is always on [32]. In addition, it assumes stable network connectivity.

The dynamic mobile environment makes the comparison very hard since each invocation of an offloadable function has different Internet access quality. To achieve fair comparison with those baselines, at runtime we force IC-Cloud to offload every offloadable function and record the information of network connectivity and application states. Then we replay these applications later for each baseline. We are aware that this method may introduce some errors but believe they are negligible.

The primary goal of IC-Cloud is to improve the performance of mobile applications. Therefore, we use *speedup*, i.e., $\frac{\text{Execution time}}{\text{Local execution time}}$, as the major metric for evaluation. When speedup is larger than 1, the application benefits from offloading. When speedup is less than 1, it spends more time for execution. We also measure the energy to analyze if IC-Cloud can also reduce energy consumption. We use PowerTutor [98], a power estimation tool for Android, to estimate the power consumption. We will primarily report the energy consumption of CPU and network interfaces because other background energy consumption (e.g., screen) is related to user settings.

4.5.2 The effect of connectivity scenarios

In the first set of experiments, we evaluate the performance of IC-Cloud in the three different mobile scenarios using the FaceDetect application. α (see Section 6.2) is set to 0.5 for IC-Cloud. Figure 20 shows the speedup distribution in those experiments. When the speedup is larger than 1, the system outperforms LocalExe. However, when the speedup is smaller than 1, the system causes longer execution time. For example, when speedup is 0.1, IC-Cloud takes 10 times the local execution time. In all these experiments, IC-Cloud performs well and achieves similar performance to Oracle. It also outperforms HistInfo by reducing the number of bad offloading decisions.



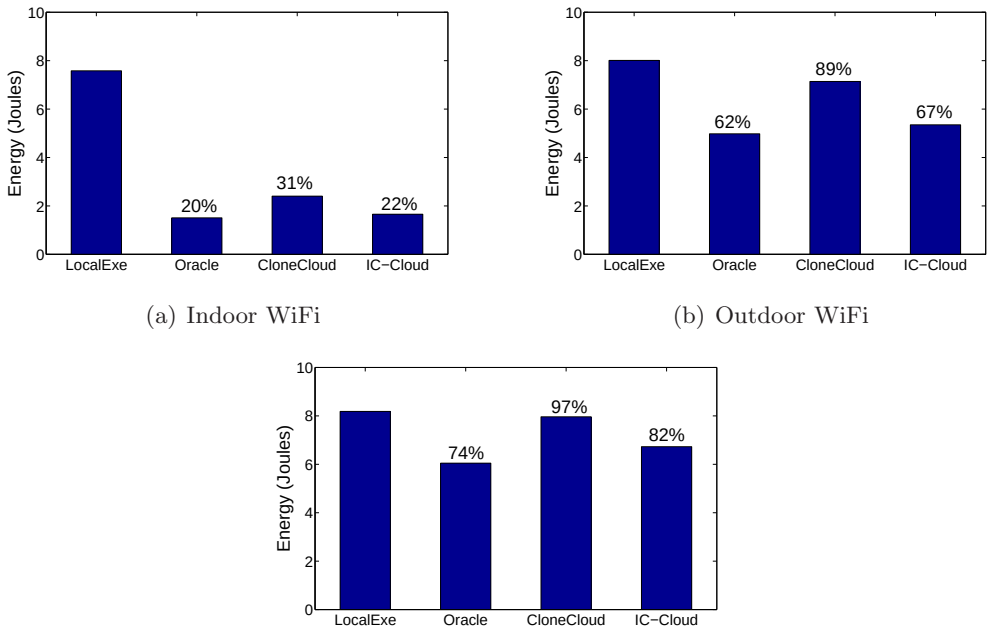
(a) Indoor WiFi

(b) Outdoor WiFi

(c) Outdoor 3G

Figure 20: A comparison of IC-Cloud's performance benefits using the FaceDetect application in different mobile scenarios. The speedups against local execution on the mobile devices are reported.

We also find some interesting phenomena in these experiments. First, in the scenario of Indoor WiFi where mobile users have good WiFi coverage, offloading computation to the cloud benefits the mobile applications in about 80% of the cases. However, there are still about 20% in which a simple method like HistInfo will increase the execution time as much as 20 times. IC-Cloud reduces the portion of negative cases to about 10% with the smallest speedup at approximately 0.4. In addition, it also enables 75% of the cases to benefit from offloading. IC-Cloud achieves 4.1x overall speedup in this scenario. Second, in the scenario of Outdoor WiFi where intermittent connectivity is common, at most 40% of the cases can benefit from offloading. HistInfo causes the bad offloading to take much more time to execute. In contrast, IC-Cloud still manages to control the smallest speedup in a similar range to Indoor WiFi scenarios. Third, in the scenario of Outdoor 3G, ideally at most 30% of the cases benefit from offloading, while the maximal speedup is only about 4. This is because the 3G has relatively smaller bandwidth and longer delays. In this scenario, IC-Cloud helps about 15% invocations of FaceDetect benefit from offloading. In addition, it only occasionally made some negative decision, while HistInfo causes more



(a) Indoor WiFi

(b) Outdoor WiFi

(c) Outdoor 3G

Figure 21: A comparison of IC-Cloud's energy consumption using the FaceDetect application in three different mobile environments. We primarily report the energy consumption of CPU and network interfaces.

than 30% invocations to take more time to execute.

To demonstrate how IC-Cloud can also save energy for mobile devices, we plot the average energy consumption for every scenario in Figure 21. IC-Cloud only consumes about 22%, 67% and 82% energy of local execution in those three scenarios, respectively. Its energy consumption is also very close to Oracle in all those scenarios. In addition, since IC-Cloud also reduces the execution time of these applications, it may also reduce the background energy consumption (e.g., screen). We also notice that although HistInfo made many bad offloading decisions in the scenario of Outdoor 3G, its average energy consumption is similar to LocalExe. This is because bad offloading decisions correspond to pictures with small local execution times. Although for each case it took much more energy to offload, the total extra energy is relatively small and is compensated by the energy saving of offloading other larger tasks in the experiments.

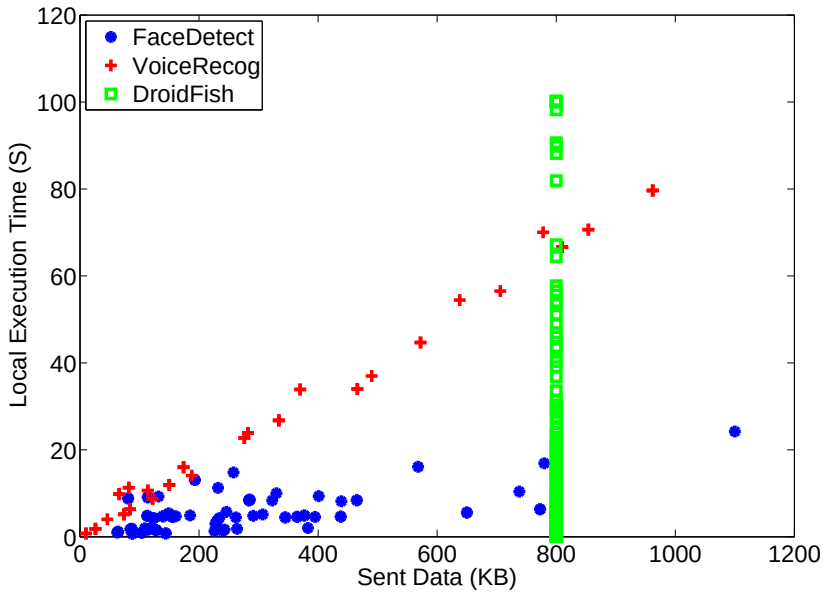


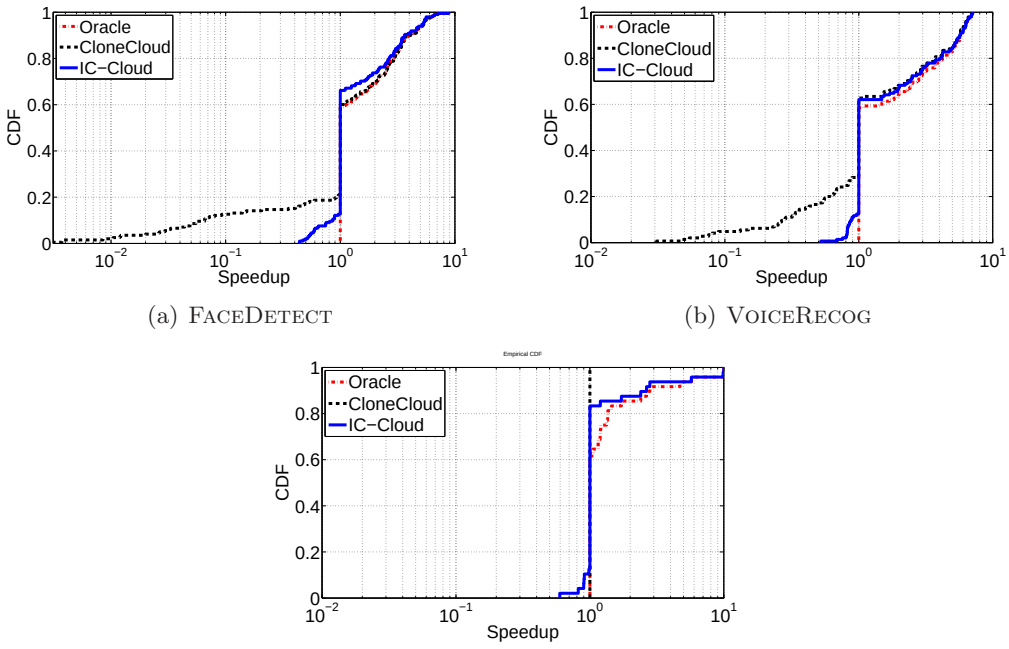
Figure 22: The local execution time vs. the size of uploaded data

4.5.3 Results for different applications

The gain from computation offloading is normally counterbalanced by the communication cost. Different applications usually have different execution times and different amount of data exchanged between the mobile device and the cloud. In this subsection, we evaluate IC-Cloud with different applications of different properties on computation and communication cost. Figure 22 plots the local execution time of the offloadable functions and the corresponding data to be sent to the cloud. We can see that local execution time of VoiceRecog is almost proportional to the data size, while DroidFish has constant data size.

To demonstrate how these application properties impact computation offloading, we compare the performance of different applications using IC-Cloud in the scenario of Outdoor WiFi. The results are plotted in Figure 23.

IC-Cloud performs well in all these experiments. In FaceDetect and VoiceRecog, IC-Cloud helps more than 35% of the function invocations benefit from offloading. Meanwhile, it limits the portion of bad decisions cases to be about 10% with small extra execution



(a) FaceDetect (b) VoiceRecog

(c) DroidFish

Figure 23: A comparison of IC-Cloud's performance with three different applications. All experiments are conducted in the scenario of Outdoor WiFi.

time. In contrast, HistInfo makes many more bad offloading decisions and causes these invocations to last much longer. Compared with FaceDetect app, HistInfo made about 30% invocations execute longer than LocalExe. This is because HistInfo's application prediction method can easily overestimate the local execution time and, thus, mistakenly decide to offload them. In contrast, our application prediction method helps IC-Cloud obtain accurate prediction and avoid the unnecessary risk in computation offloading.

The behavior of DroidFish is quite different from those of FaceDetect and VoiceRecog. Even Oracle can only help about 15% of those invocations achieve more than 2x speedup. This is because the data uploaded to the cloud is so large that if the computation gain is small it cannot compensate for the communication cost. As in our experiments only a small portion of invocations have long local execution time, the overall performance improvement is small. However, for those invocations with long local execution time, DroidFish can still benefit from offloading. We notice that HistInfo always chooses to execute locally because it underestimates the computation gain using previous invocations. IC-Cloud helps

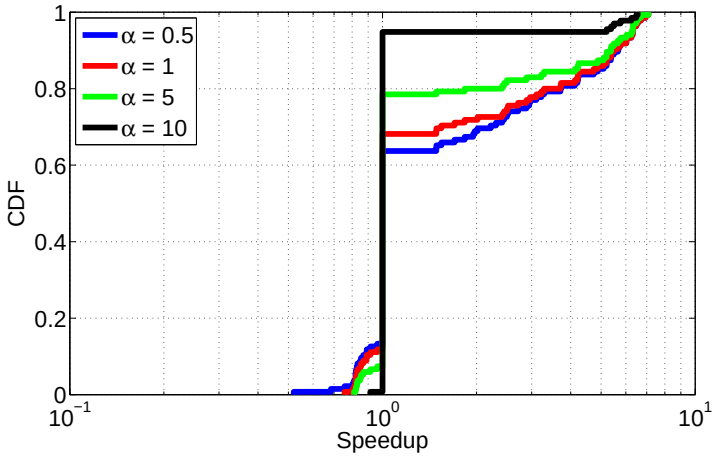


Figure 24: The tradeoff between risk and return. We use different values of α for VoiceRecog in the scenario of Outdoor WiFi.

about 20% of the invocations improve their performance and about 15% achieve 2x speedup. Compared with Oracle, IC-Cloud does not help computations that can only achieve small performance improvement because it tries to control the risk of offloading. As a result, only a small portion of invocations have longer execution time when using IC-Cloud.

4.5.3.1 The return-risk tradeoff

IC-Cloud enables applications to control the risk of offloading by setting the value of α. An application sensitive to extra delays can use large α value, while a small α value will result in higher expected return. To show how α impacts the return-risk tradeoff, we apply various values of α to the VoiceRecog application in the scenario of Outdoor WiFi. Figure 24 plots the results.

When the value of α increases from 0.5 to 10, the portion of invocations with speedup less than 1 decreases from about 10% to almost 0%. Meanwhile the portion of invocations that can benefit from offloading also drops from about 35% to 5%. It will be important to find a proper tradeoff between return and risk a question we relegate to future research.

4.6 Summary

In this chapter, we proposed IC-Cloud, a system for computation offloading in mobile environment where the Internet access to remote computation resources is of highly varying quality and often intermittent. IC-Cloud uses three key techniques to overcome the uncertainties in this environment, including lightweight connectivity prediction, lightweight

execution prediction and usage of these predictions in a risk controlled manner to make offloading decisions. We have implemented IC-Cloud on Android. Our evaluation explored a large space of possibilities by testing in three different mobile connectivity scenarios and three applications with differing computation and data I/O profiles. The experimental results show that IC-Cloud can enable effective computation offloading in a variety of mobile environments.

CHAPTER V

COSMOS: COMPUTATION OFFLOADING AS A SERVICE FOR MOBILE DEVICES

5.1 Introduction

There is great potential for boosting the performance of mobile devices by offloading computation-intensive parts of mobile applications to the cloud. Despite this potential, a key challenge in computation offloading lies in the mismatch between how individual mobile devices demand and access computing resources and how cloud providers offer them. Offloading requests from a mobile device require quick response and may not be very frequent. Therefore, the ideal computing resources suitable for computation offloading should be immediately available upon request and be quickly released after execution. In contrast, cloud computing resources have long setup time and are leased for long time quanta. For example, it takes about 27 seconds to start an Amazon EC2 VM instance. The time quantum for leasing an EC2 VM instance is one hour. If an instance is used for less than an hour, the user must still pay for one-hour usage. This mismatch can thus hamper offloading performance and/or incur high monetary cost.

Complicating this issue is the fact that mobile devices access cloud resources over wireless networks which have variable performance and/or high service cost. For example, 3G networks have relatively low bandwidth, causing long communication delays for computation offloading [33]. On the other hand, although WiFi networks have high bandwidth and are free to use in many cases, their coverage is limited, resulting in intermittent connectivity to the cloud and highly variable access quality even when connectivity exists [20, 36, 70].

In this chapter we propose COSMOS, a system that bridges the above-discussed gaps by providing computation offloading as a service. The key premise of COSMOS is that an intermediate service between a commercial cloud provider and mobile devices can make the properties of underlying computing and communication resources transparent to mobile

devices and can reorganize these resources in a cost-effective manner to satisfy offloading demands from mobile devices. The COSMOS system receives mobile user computation offload demands and allocates them to a shared set of compute resources it dynamically acquires (through leases) from a commercial cloud service provider. The goal of COSMOS is to provide the benefit of computation offloading to the mobile devices while at the same time minimizing the compute resource leasing cost.

Our goal is to develop a design for COSMOS, implement it and evaluate its performance. At the heart of COSMOS are two types of decisions: 1) whether a mobile device should offload a particular computation to COSMOS and 2) how COSMOS should manage the acquisition of compute resources from the commercial cloud provider.

We start by formulating an optimization problem whose solution can guide the required decision making. Because of its complexity, we are unable to provide a solution to this optimization. It does, however, lead us to the identification of three components of COSMOS decision making that we then explore individually. Specifically, we develop a set of novel techniques, including resource-management mechanisms that select resources suitable for computation offloading and adaptively maintain computing resources according to offloading requests, risk-control mechanisms that properly assess returns and risks in making offloading decisions, and task-allocation algorithms that properly allocate offloading tasks to the cloud resources with limited control overhead.

We have implemented COSMOS for Android and evaluated the system for offloading from a set of smartphones/tablets to Amazon EC2, across different applications in various types of mobile environments. We evaluate the performance of the system in several realistic mobile environments. To further explore the design space of COSMOS, we also conduct extensive trace-based simulation. In all these experiments, COSMOS achieves good offloading performance and significantly reduces the monetary cost compared with previous offloading systems like CloneCloud [32]. We find that COSMOS, configured with the right design choices, has significant potential in reducing the cost of providing cloud resources to mobile devices while at the same time enabling mobile computation speedup.

The rest of the chapter is organized as follows. Section 5.2 presents some background

materials and presents the problem statement and optimization formulation. Section 5.3 presents an overview of COSMOS's architecture incorporating the insight produced from the optimization formulation regarding the system decomposition. The design details are presented in Sections 5.4. COSMOS is evaluated in Section 5.5 and 5.6. We discuss related issues in Section 5.7. Section 5.8 concludes this chapter.

5.2 Background and Problem Statement

5.2.1 Background

Cloud computation resources are usually provided in the form of virtual machine (VM) instances. To use a VM instance, a user installs an OS on the VM and starts it up, both incurring delay. VM instances are leased based on a time quanta. e.g., Amazon EC2 uses a one hour lease granularity. If a VM instance is used for less than the time quanta, the user must still pay for usage. A cloud provider typically provides various types of VM instances with different properties and prices. Table 4 lists some properties and prices for three types of Amazon EC2 VM instances: *Standard On-Demand Small* instances (m1.small), *Standard On-Demand Medium* instances (m1.medium) and *High-CPU On-Demand Medium* instance (c1.medium). For some pricing models (e.g., *EC2 spot*), the leasing price may change over time.

Table 4: The characteristics of EC2 on-demand instances. The setup time is measured by starting and stopping each type of instances for 10 times. The average value and standard deviation are reported.

Instance	Cores	CPU(GHz)	Setup(second)	Price($/hr)
m1.small	1	1.7	26.5(5.5)	0.06
m1.medium	1	2.0	26.6(3.7)	0.12
c1.medium	2	2.5	26.7(8.4)	0.145

Note that the server component of offloaded mobile computation needs to run on a VM instance. This server component needs to be launched at the time the offloading request is made and terminated when the required computation is complete. The lifetime of the server component is typically much less than the lease quantum used by the cloud service provider. An important question we consider in our system design is how to ensure there is enough VM capacity available to handle the mobile computation load without needing to

always launch VM instances on-demand and incur long setup time.

5.2.2 Problem Statement

The basic idea of COSMOS is to achieve good offloading performance at low monetary cost by sharing cloud resources among mobile devices. Specifically, in this chapter our goal is to minimize the usage cost of cloud resources under the constraint that the speedup of using COSMOS against local execution is larger than $1-\delta$ of the maximal speedup that it can achieve using the same cloud service, where $\delta \in (0,1)$. The extension of COSMOS to support other optimization goals will be discussed in Section 5.7. A related but independent problem is how COSMOS charges mobile devices for computation offloading, which will also be discussed in Section 5.7.

Let's assume that the cloud can simultaneously run N VM instances. Let's use $\langle M_i, T_i \rangle$ to denote the usage of the i^{th} VM instance, where M_i is its type (see Table 4 for examples), and $T_i = \{\langle t_{ij}, t'_{ij} \rangle | \forall j, t_{ij} < t'_{ij}, t'_{ij} < t_{i(j+1)}\}$ represents all the leasing periods. We use t_{ij} and t'_{ij} to represent the start time and end time of the j^{th} leasing period, respectively. Let $\psi(M,T)$ be the leasing cost of $\langle M, T \rangle$, which is computed by multiplying the price with the total leasing time quanta.

Let's assume that there are K computation tasks generated by mobile users in the COSMOS system during the period of time of interest. Let $O_k = \langle t_k, c_k, d_{Ik}, d_{Ok} \rangle$ denote the k^{th} computation task, where t_k ($\forall k, t_{k-1} \leq t_k$) is the time the task initiated by the mobile user, c_k is the number of CPU cycles it requires, d_{Ik} is the size of its input data, and d_{Ok} is the size of its output data. Let $I(i,k)$ and $I_l(k)$ be indicator functions. If O_k is offloaded to the i^{th} VM instance, $I(i,k) = 1$. Otherwise, $I(i,k) = 0$. Similarly, if it is locally executed, $I_l(k) = 1$. Otherwise, $I_l(k) = 0$. Since O_k should be executed exactly once, $I_l(k) + \sum_i I(i,k) = 1$. Let $L(O_k)$ be the local execution time of O_k which can be estimated based on c_k and the CPU frequency of the mobile device [65]. Let $R_i(O_k)$ be the response time of offloading O_k to the i^{th} VM instance, i.e., the time elapsed from t_k to the time that the output is returned to the mobile device. It depends on the network delays of sending input data and receiving output data, the execution time on the VM instance and

the waiting time. Its formula will be shown in Section 5.4.2.

The maximal speedup of using COSMOS against local execution can be obtained by solving the following optimization problem:

$$\begin{aligned} \text{Max} \quad & \sum_{k=1}^{K} \frac{L(O_k)}{\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}} \\ \text{s.t.} \quad & I_l(k) + \sum_{i=1}^{N} I(i,k) = 1 \end{aligned} \tag{10}$$

Since for each k there is exactly one of the functions $I_l(k)$ and $I(i,k)$ that equals to 1, $\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}$ equals to the corresponding $L(O_k)$ or $R_i(O_k)$, representing the response time of task O_k in COSMOS. Thus, Eqn 10 is to maximize the speedup of all tasks. Let's use S_k^* to denote the corresponding maximal speedup achieved by task O_k. We have $\forall k, S_k^* \geq 1$. The goal of COSMOS can be formally formulated as:

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{N} \psi(M_i, T_i) \\ \text{s.t.} \quad & I_l(k) + \sum_{i=1}^{N} I(i,k) = 1 \\ & \frac{L(O_k)}{\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}} \geq (1-\delta)S_k^* \end{aligned} \tag{11}$$

where $\delta \in (0,1)$ can be arbitrarily chosen by the system. When $\delta \to 1$, all tasks will be executed locally while the total cost will approach 0. When $\delta \to 0$, the speedups approach to the optimal values while the total cost will be high.

COSMOS is to find the values of $\langle M_i, T_i \rangle$, $I_l(k)$, and $I(i,k)$ that optimize Eqn 11. This is a challenging problem especially because we have no information regarding future computation tasks. Our approach to solve this problem is to break it down into three sub-problems and address each of them separately:

- **Cloud resource management**: This is the problem of determining the number and type of VM instances to lease over time, i.e., $\langle M_i, T_i \rangle$. It has two major goals. First, there should always be enough VM instances to ensure high offloading speedup. Second, the cost of leasing VM instances should be minimized.

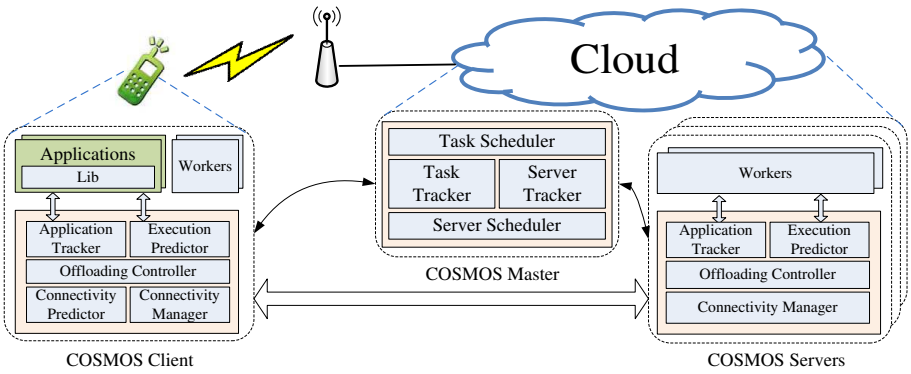


Figure 25: Architecture of the COSMOS system.

- **Offloading decision**: This is the problem of deciding if a mobile device offloads a computation task, i.e., deciding whether to set $I_l(k)$ to 0 or 1. The challenge comes from the uncertainties of network connectivity, program execution, and resource contention. A wrong offloading decision will both waste cloud resources and result in lower speedup. It is very important to properly handle the uncertainties.

- **Task allocation**: This is the problem of choosing a VM instance among all available instances if we decide to offload a computation task, i.e., deciding whether to set $I(i, k)$ to 0 or 1. The decision is made independently by each mobile device. In addition, the decisions for previous computation tasks will impact the decision of the current one because of resource sharing. Therefore, task allocation should be designed as a distributed mechanism.

In the following sections, we will present the design of COSMOS and its mechanisms to solve these problems.

5.3 COSMOS System

Figure 25 provides a high-level overview of the COSMOS system. It consists of three components: a COSMOS Master running on a VM instance that manages cloud resources and exchanges information with mobile devices; a set of active COSMOS Servers each of which runs on a VM instance and executes offloaded tasks; and a COSMOS Client on each mobile device that monitors application execution and network connectivity and makes offloading decisions.

The COSMOS Master is the central component for cloud resource management. It periodically collects information of computation tasks from COSMOS Clients through *task tracker* and the workloads of COSMOS Servers through *server tracker*. Using this information, its *server scheduler* decides the number and type of active COSMOS Servers over time. Note that when a COSMOS Server is turned on/off, its corresponding VM instance is also turned on/off. The detailed mechanism will be described in Section 5.4.1.

An active COSMOS Server is responsible for executing offloaded computation tasks. Each COSMOS Server has one *task queue* and multiple *worker*s the number of which equals to its number of CPU cores. Computation tasks are executed on a first-come-first-serve basis. The COSMOS Server also estimates and provides the workload information upon request by predicting the execution time of all tasks in the queue through the *execution predictor*. We use Mantis [31], a state-of-the-art predictor for mobile applications.

A COSMOS Client tracks all applications running on its mobile device, makes offloading decisions for them, and allocates tasks to COSMOS Servers. When a mobile application starts, an *application tracker* monitors the application execution and identifies its compute-intensive tasks in the same way as MAUI [33] and CloneCloud [32]. When it reaches the entry point of such a task, the *offloading controller* obtains the computation speedup from the execution predictor and the communication delay from the *connectivity predictor* (e.g., BreadCrumbs [74]) and decides if it should offload the task based on them. The design details of this offloading decision will be described in Section 5.4.2. If it decides to offload, the COSMOS Client allocates the task to an active COSMOS Server, which will be described in Section 5.4.3. Finally the COSMOS Client offloads the task and waits to receive the result. If the COSMOS Client cannot obtain the results before a deadline, it executes the task on a local worker.

The COSMOS system is based on a generic architecture that enables the efficient sharing of cloud computation resources in the presence of intermittent connectivities. The key mechanisms (i.e., cloud resource management, offloading decision, and task allocation) are designed as independent modules so that they can be easily replaced to achieve various objectives of the system operators. In the next section, we will describe the design details

of the key mechanisms to achieve the objective presented in Section 5.2. There are many alternatives for each mechanisms that can be used in the COSMOS system. We will pick some basic alternative mechanisms to compare with our current design in the evaluation.

5.4 Design Details

5.4.1 Cloud Resource Management

The cloud resource management has two major mechanisms: how to select the type of VM instance (i.e., M_i) and when to start and stop COSMOS Servers (i.e., T_i).

5.4.1.1 Resource Selection

COSMOS strives to minimize the cost per offloading request under the constraint that the offloading speedup is large enough. Recall that our goal is to achieve speedup of at least $1 - \delta$ of the maximally possible. Therefore, the resource selection algorithm selects the least cost VM instance whose CPU frequency is larger than $1 - \delta$ of the most powerful VM instance. The algorithm is shown in Algorithm 5.

Algorithm 5 Resource Selection

```
1: procedure RESOURCESELECTION({I},δ)                    ▷ I is a instance type.
2:     maxFreq ← 0; minCost ← +∞ ;
3:     for I in {I} do
4:         if maxFreq < I.f then
5:             maxFreq ← I.f;
6:         end if
7:     end for
8:     for I in {I} do
9:         if I.f ≥ (1 − δ)maxFreq && I.p/(I.c×I.f) < minCost  then
10:            maxCost ← I.p/(I.c×I.f);
11:            selected ← I;
12:        end if
13:    end forreturn selected;
14: end procedure
```

In Algorithm 5, c, f, and p denote the number of processor cores, the CPU frequency and the price of a VM instance, respectively. If all cores are 100% utilized during a time quanta τ, the VM instance totally executes $cf\tau$ CPU cycles. The cost of each CPU cycle is $\frac{p}{cf}$. We use this value as the cost for comparison in Algorithm 5.

5.4.1.2 Server Scheduling

Server scheduling is the key mechanism to balance the usage cost of VM instances and the offloading performance. Its basic operations are as follows: The COSMOS Master periodically collects the number of offloading requests and the workloads of COSMOS Servers (every 30 seconds in our implementation). When the workloads are too large, it turns on new COSMOS Servers. When a time quantum of a COSMOS Server is to expire, it turns off the COSMOS Server if the remaining COSMOS Servers are enough to handle the offloading requests. The algorithm is shown in Algorithm 6.

Algorithm 6 Server Scheduling

```
1: procedure SERVERSCHEDULING(λ,μ)   ▷ λ and μ are reported arrival rate and service time of
   offloading requests in the last round.
2:     λ_s ← getMaxArrivalRatePerServer(μ, μ/(1−δ));
3:     n ← ⌈λ/λ_s⌉;
4:     if n > activeServers.size() then
5:         turnOnServers(n - activeServers.size());
6:     else
7:         for s ∈ activeServers do
8:             if s.quantumExpiring() then
9:                 pendingServers.add(s);
10:            end if
11:        end for
12:        turnOffServers();
13:    end if
14: end procedure
15: procedure TURNONSERVERS(n)                          ▷ n is the number of server
16:    for i = 1:n do
17:        if pendingServers.isEmpty() then
18:            activeServers.add(turnOnAServer())
19:        else
20:            activeServers.add(pendingServers.remove(0));
21:        end if
22:    end for
23: end procedure
24: procedure TURNOFFSERVERS
25:    for s ∈ pendingServers do
26:        if s.quantumaExpired() then
27:            turnOff(s);
28:        end if
29:    end for
30: end procedure
```

Every round the COSMOS Master computes the current arrival rate and service time of offloaded computation tasks and uses them to estimate the minimal number of COSMOS

Servers to achieve the desired offloading performance. If the number of active COSMOS Servers is smaller than this value, it turns on new COSMOS Servers. A COSMOS Server whose current time quantum expires will be turned off only if the number of remaining COSMOS Servers are larger than the minimal value for several rounds. Otherwise, its lease will be renewed for another quantum.

A key function of Algorithm 6 is how to estimate the maximal arrival rate (i.e., λ_s) that a COSMOS Server can handle, as shown in Line 2. Since a COSMOS Server serves offloading requests in a first-come-first-serve manner, it can be modeled as a G/G/c system. Based on the required speedup (i.e., $1 - \delta$ of the maximal speedup), we obtain that the maximal response time of the system should be smaller than $\frac{\mu}{1-\delta}$. According to Kingman's formula [63], we can obtain the value of λ_s.

5.4.2 Offloading Decision

The offloading controller uses the information from the connectivity and execution predictors to estimate the potential benefits of the offloading service. Ideally, if future connectivity and execution time can be accurately predicted immediately after the mobile application starts, the offloading controller can make the global optimal offloading decision. However, such global optimum is unavailable in reality.

Instead, the offloading controller uses a greedy strategy to make the offloading decision. Every time an offloadable task is initiated, the offloading controller determines if it is beneficial to offload it. Because of the uncertainties inherent in the mobile environment, the offloading decision takes risk into consideration. In case a bad decision has been made, it will also adjust its strategy with new information available.

We use the risk-controlled offloading developed for IC-Cloud to make the offloading decision. The details can be found in Section 4.4.

5.4.3 Task Allocation

When a COSMOS Client is to offload a task, it must decide which COSMOS Server should execute the task. We consider three heuristic methods. The first method is that the COSMOS Master maintains a global queue to directly accept offloading requests and allocates

tasks when a COSMOS server has idle cores. Although it should have high server utilization, the network connecting the COSMOS Master may become a bottleneck. The second method is that the COSMOS Client queries the workloads of a set of COSMOS servers and randomly chooses one with low workload to allocate the new task. Although tasks are directly sent to COSMOS Servers in this method, it will cause huge control traffic. In addition, it will cause extra waiting time. The third method is that the COSMOS Master provides each COSMOS Client a set of active COSMOS Servers and informs it the average workloads of all COSMOS Servers. Each COSMOS Client randomly chooses a server among them to offload the task. This method has minimal control overhead. As the resource-management mechanism ensures that the workloads of COSMOS Servers are low, it should also have good performance. Thus, COSMOS uses the third method for task allocation.

5.5 System Implementation and Evaluation

In this section, we evaluate our prototype implementation of COSMOS in various mobile environments.

5.5.1 Implementation

We implemented the COSMOS Server on Android x86 [3]. To run COSMOS Servers on Amazon EC2, we use an Android-x86 AMI [64] to create EC2 instances. Since Android-x86 is a 32-bit OS, three types of EC2 instances can be used for COSMOS Servers, as listed in Table 4. Based on our resource selection algorithm, *High-CPU On-Demand Medium* instances are used to run COSMOS Servers.

The COSMOS Master runs on an EC2 instance running Ubuntu 12.04. It uses the Amazon EC2 API tools [1] to start and stop EC2 VM instances on which COSMOS Servers run.

A COSMOS Client runs on an Android device equipped with both WiFi and 3G connections. It uses the Java reflection techniques to enable the offloading of computation tasks.

We modified three existing Android applications to use COSMOS, including:

FaceDetect is a face detection application that uses APIs in the Android SDK. We collected a data set of pictures containing faces from Google Image.

VoiceRecog is an Android port of the speech recognition program PocketSphinx [58]. For simplicity of experiments, we also modified it to use audio files as input.

DroidFish is an Android port of the Chess engine Stockfish [5] that allows users to set the strength of the AI logic.

5.5.2 Experimental Setup

In the system evaluation, we consider a stable WiFi environment in which a student carrying a mobile device sits in his lab. We measure the network connectivity and construct a database for the connectivity predictor.

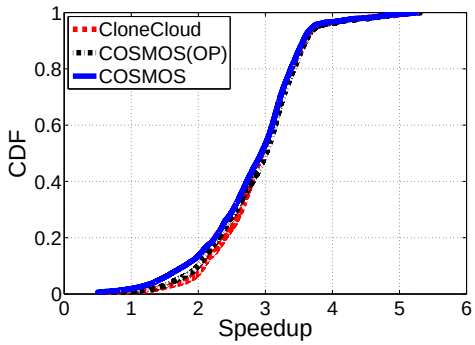
We compare COSMOS with two baseline systems:

- **CloneCloud** is a basic offloading system in which each mobile device has a server in the cloud that is always on [32]. In addition, it assumes stable network connectivity.
- **COSMOS(OP)** is a variant of COSMOS with a simple strategy for resource management, i.e., the number of active COSMOS Servers are over-provisioned for the peak requests. We assume the number of peak requests is accurately estimated in advance.

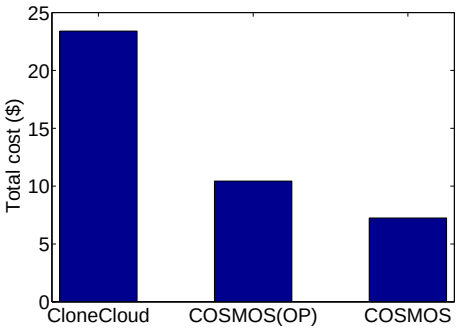
We use two metrics to evaluate COSMOS: *speedup* and *cost*. The values of speedup are different for different mobile devices. For fair comparison, we use a Samsung Galaxy Tab running Android 2.3 to obtain the local execution time in calculating speedup. The prices for EC2 on-demand instances are used to compute the cost.

5.5.3 Experiment Results

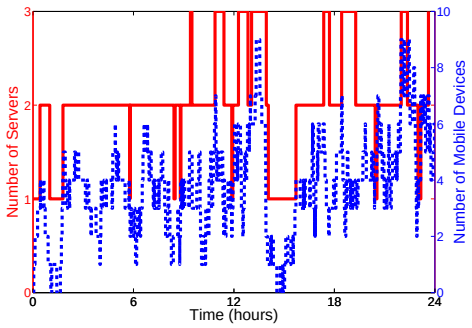
In this set of experiments, we use 10 Android devices to conduct a one-day experiment for each of the systems: CloneCloud, COSMOS(OP), and COSMOS. During an experiment, each device randomly becomes active or idle from time to time. Their durations follow an exponential distribution, with average values of 0.5 hour and 1 hour, respectively. When



(a) Speedup



(b) Cost



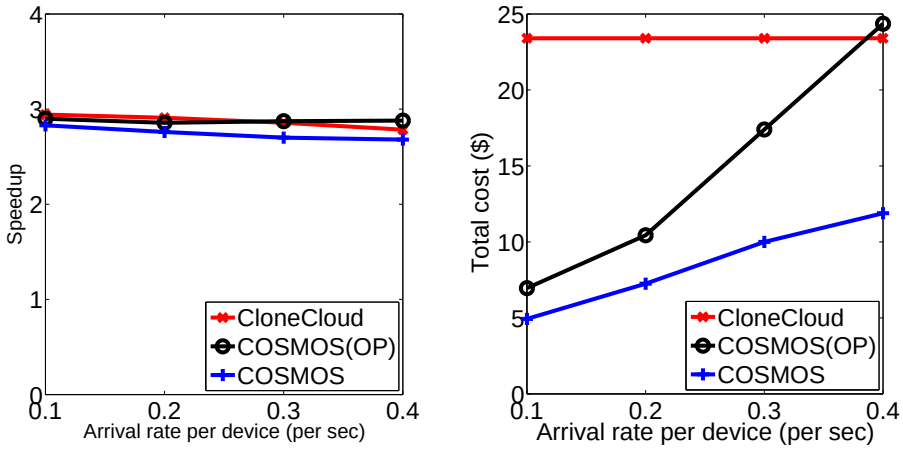
(c) Number of active servers in COSMOS

Figure 26: The performance of COSMOS on Amazon EC2 for FaceDetect in one day. There are 10 Android devices which randomly become active or idle. When they are active, they randomly execute the FaceDetect application.

a device is active, it randomly starts the FaceDetect application following a Poisson distribution with arrival rate of 0.2. The same random seed is used for all three experiments.

The experiment results are reported in Figure 26. As shown in Figure 26(a), all three systems achieve similar speedups, i.e., 2.91X, 2.86X and 2.76X on average for CloneCloud, COSMOS(OP), and COSMOS, respectively. Meanwhile, the total cost of COSMOS ($7.25) is significantly lower than that of CloneCloud ($23.4) and COSMOS(OP) ($10.44), as shown in Figure 26(b).

To demonstrate how COSMOS reduces its cost, we plot the number of active devices (the dotted blue line) and that of active COSMOS Servers (the red line) in Figure 26(c) . COSMOS adaptively changes the number of active COSMOS Servers according to the



(a) Speedup (b) Cost

Figure 27: Impact of the arrival rate on COSMOS performance.

arrival rate of offloading requests. Therefore, COSMOS is able to reduce its cost by turning off some COSMOS Servers when the number of offloading requests is low. In contrast, COSMOS(OP) spends 44% more money to keep 3 COSMOS Servers active the whole day, whereas CloneCloud spends 223% more money than COSMOS with only 5.4% extra speedup.

5.6 Trace Based Simulation

In this section, we use trace-based simulation to extensively evaluate the properties of COSMOS and how its components impact its performance.

5.6.1 The impact of request rate

In this subsection, we analyze the impact of offload request intensity on the performance of COSMOS.

In the first set of experiments, we conduct simulation-based experiments using information logged in the experiments of Section 5.5.3. We vary the arrival rate of offloading requests from 0.1 to 0.4 per second and keep other settings unchanged.

Figure 27 plots the experiment results. In all experiments, COSMOS achieves the lowest cost and high speedup. We also make the following observations. First, both COSMOS and COSMOS(OP) have lower cost with lower arrival rate, while CloneCloud has constant cost. This indicates the importance of cloud resource sharing in reducing the cost. Second, when

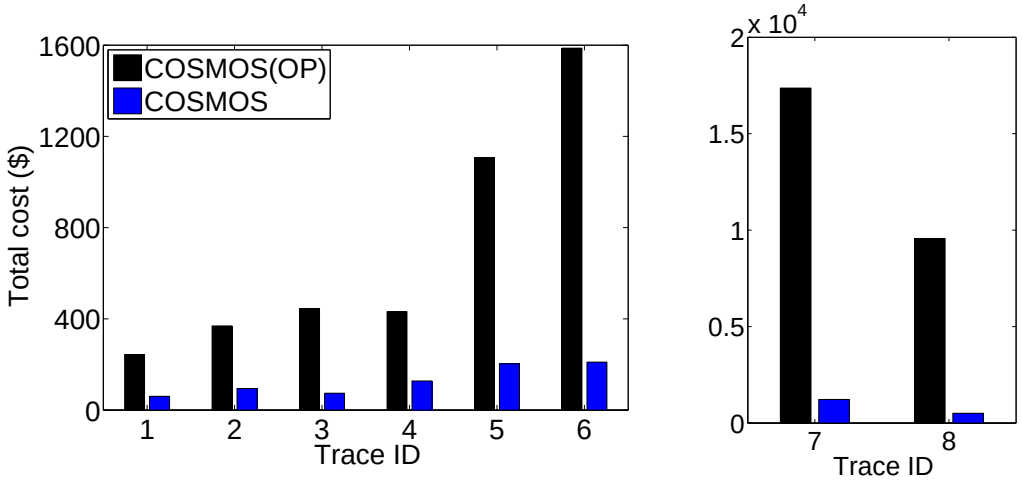


Figure 28: Offloading cost for various real-world access traces.

the arrival rate is very high (e.g., 0.4), COSMOS(OP) has higher speedup (i.e., 2.87X) than CloneCloud (i.e., 2.78X). Its cost is slightly higher than CloneCloud. Third, the speedup of COSMOS is similar with those of CloneCloud and COSMOS(OP) in all experiments.

5.6.2 Scalability

Next, we evaluate COSMOS using real-world access traces [10]. The data set consists of 8 access traces each of which is composed of access requests in 2 days. We use these timestamps of access requests as the start time of mobile applications on various mobile devices. We evaluate the performance of COSMOS through simulation. The average number of requests from the same user is very low in the traces, indicating extremely high cost of CloneCloud. Therefore, we only compare COSMOS with COSMOS(OP). The costs and speedups for the FaceDetect application on various traces are plotted in Figure 28 and 29, respectively.

COSMOS yields slightly smaller speedups but at significantly lower cost than COSMOS(OP) on all traces. Specifically, COSMOS(OP) pays 13.2 times more money than COSMOS on trace 7, while its speedup (i.e., 2.87X) is only slightly higher than that of COSMOS (i.e., 2.7X). COSMOS is able to reduce the cost by an order of magnitude while still achieving 2.7X speedup. The results of this set of experiments demonstrate that COSMOS is able to provide computation offloading with high performance at very low cost.

We also conducted experiments for our other two mobile applications, as well as a mix

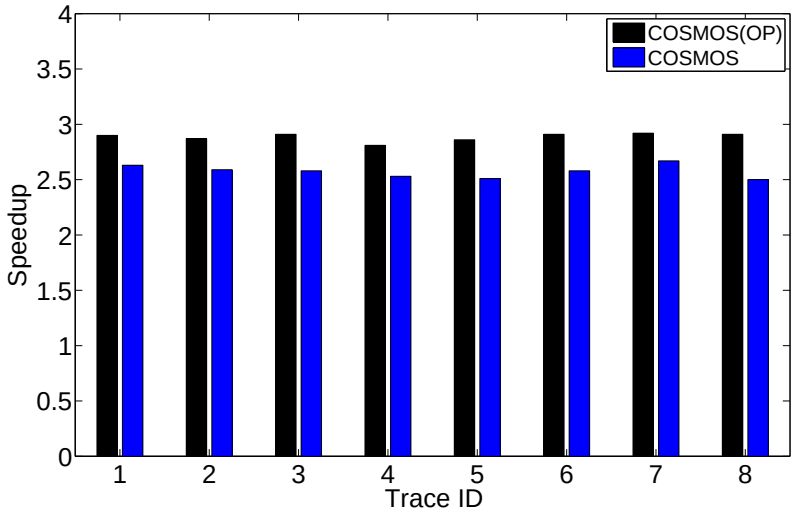


Figure 29: Offloading speedup for various real-world access traces.

of all three applications. The results are similar and we omit them for brevity.

5.7 Discussion

In this section we disscuss some important issues in extending COSMOS and the possible solutions.

Energy minimization: In this chapter COSMOS is to speed up the execution of mobile applications. There are also scenarios in which minimizing energy consumption is more important than speeding up the execution.Extending COSMOS to minimize energy consumption requires two major changes. First, the offloading controller should make the offloading decision based on energy consumption. It should delay computation offloading until the network connectivity is good. Second, the cloud resources can be used in a more efficient way. Instead of immediately executing each offloaded task, COSMOS can wait until enough tasks are aggregated.

Pricing model: There are several possible methods for COSMOS to charge the usage of its computation-offloading services. First, users pay monthly service fees and can use COSMOS as frequent as they want. Second, an offloaded task could be charged according to its execution time. Third, mobile devices could bid for computation-offloading. When the number of offloading requests is small, COSMOS could charge at a lower price to attract

more requests and thereby avoid wasting cloud resources.

Server implementation: In our current implementation, COSMOS servers run on Android-x86 and rely on java reflection techniques to enable computation offloading. By doing this, it requires no modification to the original mobile applications and simplifies the implementation. However, its execution speed will be a little slower than native code in the cloud. In performance-sensitive scenarios, the application developers can implement a c++ version for each mobile application and run it on COSMOS servers to further improve the performance.

5.8 Summary

In this chapter, we proposed COSMOS, a system that provides computation offloading as a service to resolve the mismatch between how individual mobile devices demand computing resources and how cloud providers offer them. COSMOS solves two key challenges to achieve this goal, including how to manage and share the cloud resources and how to handle the variable connectivity in making offloading decision. We have implemented COSMOS and conducted an extensive evaluation. The experimental results show that COSMOS enables effective computation offloading at low cost.

CHAPTER VI

COAST: COLLABORATIVE APPLICATION-AWARE SCHEDULING OF LAST-MILE CELLULAR TRAFFIC

6.1 Introduction

The rapid proliferation of smartphones, tablets, and mobile applications has led to a tremendous increase in mobile data traffic in the last few years. For example, the mobile data traffic on major US based mobile carriers has increased by more than 20,000% in five years [15]. Furthermore, according to forecasts by major equipment manufacturers, this trend is likely to continue in future with 78% compound annual growth rate [4]. In contrast, the capacity of cellular networks, especially the wireless spectrum, has not increased proportionally. The efficient management of mobile traffic is, therefore, critical for cellular network operators.

Various solutions have been proposed to manage the mismatch between the ever-increasing traffic demand and finite wireless spectrum. These solutions can be broadly classified into two categories—adding more network resources to increase the overall capacity (i.e. increasing supply), or managing user demands and behavior to reduce the load on the network (i.e. controlling demand). Examples of the first category include the use of small cells for augmenting the capacity of traditional macro cells, adding WiFi hotspots to offload cellular traffic to WiFi, using portable base stations (e.g. Cells On Wheels or COWs) to meet high traffic demands in event venues where large numbers of users gather for some time periods, etc. Examples of the second category include congestion pricing, off-peak delivery, network-aware throttling, etc. These approaches reduce the aggregate traffic in busy periods by either shifting the parts of traffic that can tolerate some delay to off-peak hours (e.g. backup, synchronization, cloud offload, etc.) [50], or causing the user to use the network less frequently.

These existing approaches have some fundamental limitations. Shifting traffic to off-peak hours can cause degradation of quality of service experienced by the end users. The

vast majority of mobile data traffic, including video streaming and mobile web browsing, cannot be shifted to off-peak hours because of latency requirements. Although additional network resources increase the overall capacity, they also incur significant costs. Furthermore, solutions like small cells can be deployed only gradually because of the detail radio engineering trials required for the correct positioning and deployment of such infrastructure.

In this chapter, we take a fundamentally different approach to tackle this problem—delaying mobile traffic like video streaming and mobile web browsing that are not traditionally thought to be delay tolerant. This is based on two key insights derived from mobile traffic traces of a large US cellular provider. First, we observe that the mobile data traffic exhibits high burstiness over small time scales (tens of seconds). Thus, to ensure adequate quality of service at all times, it is important to reduce the instantaneous peak traffic, not just the aggregate traffic. Second, even applications like video streaming and mobile web browsing can, in fact, tolerate small delays. For example, a video streaming client can tolerate delays of tens of seconds as long as its playback buffer is not empty. Mobile web browsers can delay downloading the contents that are not currently displayed on the screen. These two insights suggest that if the right user traffic (from the set of all current user traffic in the cell) is delayed at the right time for the right time duration, it is possible to reduce the peak traffic in a cell without affecting the user experience on any mobile device. This requires both device-level information (e.g. tolerable delay values at the given time instant) and cell-level information (e.g. the total traffic demand in the cell at the given time instant). Thus, an efficient interaction mechanism between mobile devices and cellular infrastructure is necessary to enable collaboration between them to make proper decisions about delaying the user traffic.

Using these insights, we present the design, implementation, and extensive evaluation of a novel system called CoAST (**Co**llaborative **A**pplication-Aware **S**cheduling of Last Mile Cellular **T**raffic). CoAST provides an interface to enable an efficient collaboration between mobile devices and the network element to which they are connected (e.g. a base station). The interface is simple and flexible, allowing dynamic policies and protocols to be built on top of it, according to the requirements and capabilities of individual mobile applications.

CoAST also provides an incentive mechanism for mobile applications to delay their traffic and the actual mechanisms to delay the application traffic.

In addition to benefits to the mobile network operator, CoAST also improves the application performance for the end user. By delaying traffic of users with enough playback buffer contents, CoAST can aggressively fill the starving buffers of other users, thereby reducing their buffering delays.

In summary, CoAST makes following novel contributions:

1) Rather than reducing the aggregate busy hour traffic, CoAST reduces the instantaneous peak load in a cell, without compromising the quality of service experienced by the end users.

2) CoAST can handle traffic which is not traditionally thought to be delay tolerant.

3) CoAST provides a simple and flexible interface for mobile devices and the cellular network to exchange various information to enable them to make proper decisions about delaying user traffic.

We implement a prototype of CoAST on the Android platform for two sample application categories, streaming (e.g., YouTube) and web browsing, which are the top two generators of cellular network traffic, accounting for nearly 70% of global cellular traffic [83]. We evaluate our implementation on a per-cell basis using emulation based on real YouTube and web browsing traces obtained from a major US cellular provider. Our results show that CoAST reduces traffic peaks by an average of 30-50%, or conversely, increases the capacity of a cell by 20% without compromising the quality of the end user experience. In fact, we show that CoAST achieves a better quality of service in terms of reduced buffering delay compared to the cell that does not use CoAST. Our experiments also show that the control plane overhead introduced by CoAST is negligible.

The rest of the chapter is organized as follows. In Section 6.2, we describe the background of this work and present an overview of CoAST design. We motivate the CoAST design with an analysis on cellular traffic in Section 6.3. The design details of CoAST is described in Section 6.4. The deployment and implementation of CoAST system is discussed in Section 6.5. The system evaluation of CoAST prototype and the trace-driven evaluation

of CoAST are presented in Section 6.6 and 6.7, respectively. We discuss the related issues in Section 6.8. Section 6.9 summarizes the related work. We conclude this chapter in Section 6.10.

6.2 Background and Design Overview

In this section, we describe the background of this work and present a high-level overview of CoAST design.

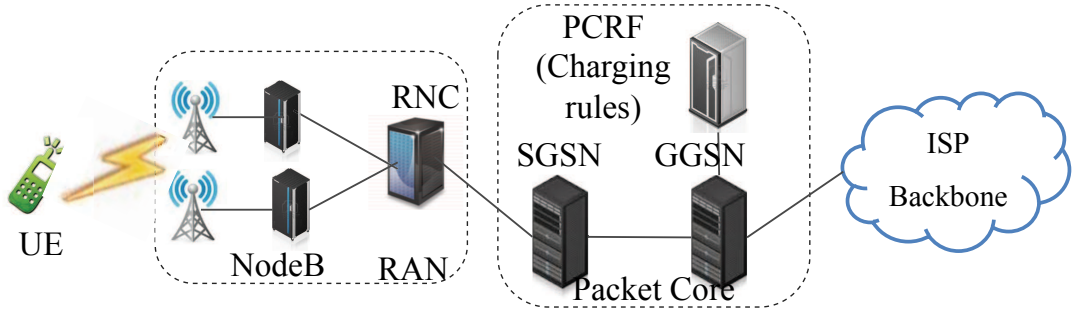


Figure 30: Architecture of UMTS data network.

6.2.1 Background of Cellular Networks

In this subsection, we first describe the basics of cellular architecture and then provide information about the data set used in our evaluations.

Figure 30 shows the key components of a typical UMTS data network. It consists of 2 major components: the Radio Access Network (RAN) and the Core Network (CN) (or Packet Core). The mobile device, called User Equipment (UE) in UMTS terminology, is connected to one or several cell sectors in RAN. A physical base station (called NodeB in 3G and eNodeB in LTE) can have multiple cell sectors, which provide radio resources to UEs for wireless communications. Cellular data traffic from several NodeBs are then passed to the Radio Network Controller (RNC), which manages handovers, and scheduling of wireless resources among the NodeBs under its control. The RNCs connect to Serving GPRS Support Nodes (SGSNs) at the core network. The SGSNs are connected to the external networks, such as the Internet, via Gateway GPRS Support Nodes (GGSNs). When a UE connects to the network, it establishes a Packet Data Protocol (PDP) context which facilitates tunneling of IP traffic from the UE to the peering GGSN using GPRS

Tunneling Protocol (GTP) (see [55] for details of the UMTS network).

We evaluate CoAST using real-world cellular traffic data from a tier-1 cellular network carrier. All device and user identifiers are anonymized for our analysis. The first data set is collected from the link between SGSN and GGSN in the core network. It contains information about IP flows carried in PDP contexts for a 3% random sample of devices collected every minute, e.g. start and end time stamps, per flow traffic volume, application identifier, etc. This data set is used to collect information about the characteristic of video streaming traffic. To gain more fine grained information about the actual traffic volume in a given cell sector, we use another dataset collected every 2 seconds at RNCs in the RAN network.

6.2.2 Design Overview

CoAST aims to reduce the peak-to-average ratio of the cellular last-hop traffic, and consequently improve application performance for the end user. It is not designed for persistently congested networks whose average traffic is close to the capacity. Such networks need to be upgraded to increase the capacity. We focus primarily on *downstream* as the upstream traffic intensity is not as significant. (Note that CoAST can apply equally to upstream traffic as well. But we do not claim it is the contribution of this chapter.) We accomplish this by delaying downstream traffic at times when the link is experiencing heavy load. In order to avoid degrading the quality of service experienced by the end user, the acceptable delays may range from a few to tens of seconds depending on the applications. As will be shown in the next section such relatively short traffic delays are enough to produce the desired effect of reducing traffic peaks.

One key insight of CoAST is that only mobile applications know the delay constraints of their traffic while the eNodeB has the aggregated traffic information. Thus, solutions that rely solely on the eNodeB or on the mobile devices do not have enough information to simultaneously achieve both high utilization and good quality of service. Collaboration between the eNodeB and mobile devices is required to determine if and by how much the downstream data should be delayed. The scheme, described in detail in Section 6.4, enables

the users to optimize their download rates while minimizing the cost of data download and maximizing the quality of service.

It is important to note that the pricing incentive used in our scheme is, strictly speaking, internal to the system and is used to facilitate decision making regarding whether to delay a certain chunk of the download traffic. It is not meant to be exposed directly and as-is to the user, nor is it meant to translate directly into billing. It will be important, however, to incentivize users to deploy this system in their devices. So we expect that there will be some correlation between billing and pricing practices of operators to be influenced by the usage of this system. However, the exact approach here is beyond the scope of our technical discussion.

Although a wide user participation increases the effectiveness of CoAST, it is not necessary that all users in a cell deploy CoAST—only enough number of UEs that can make a difference in traffic peaks is needed. Also note that users who do not deploy the system will not necessarily have any advantage over those who use the system since the user experience is preserved for those using it. So there is no individual incentive to "cheat" the system from this perspective. On the contrary, users will be interested to participate if operator pricing does somehow reflect the usage of the system. CoAST also monitors the behaviors of participating devices to identify potential cheating, which will be discussed in Section 6.8.

6.3 Feasibility and Benefits of Delaying Mobile Traffic

In this section we present an analysis of the real-world traffic traces of a large US cellular carrier to provide motivation about the feasibility and potential benefits of delaying mobile traffic for short time durations.

6.3.1 Short-Term Burstiness of Mobile Traffic

First, we analyze the traffic distribution of a large number of cells (13522 NodeBs) using a large dataset of cellular traffic collected at several RNCs to demonstrate that the mobile data traffic of a typical cell demonstrates high variations over short time scales (e.g. 30 seconds). This dataset provides per cell as well as per UE cellular data records. We focus on the downlink traffic because it is significantly larger than the uplink traffic in cellular

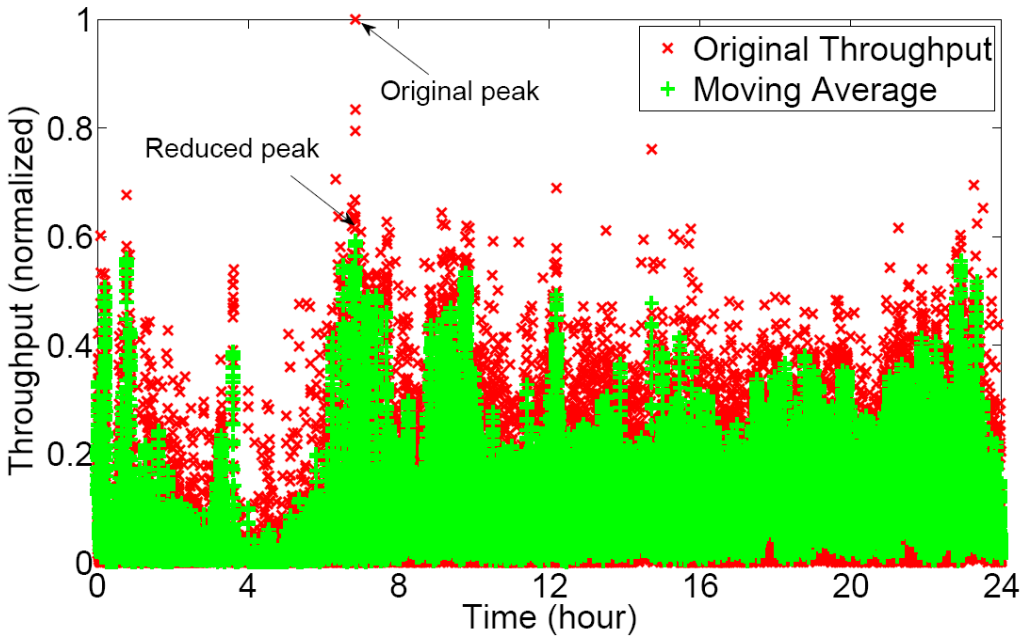


Figure 31: The downlink traffic and its 20 second moving average in a cell in one typical day. The throughput is normalized with the maximal capacity used. The difference between the original traffic and its moving average demonstrates its high short-term variations.

networks [38, 71].

Figure 31 plots the downlink throughput along with its 20 second moving average in a typical cell in one day. The traffic is clearly very bursty with large short-term variations. To ensure adequate quality of service at all times, the cellular network provider needs to over-provision the network resources based on the peak traffic demand. Therefore, the burstiness of mobile traffic makes the resources underutilized during most periods of time.

A heuristic idea to better utilize the cellular resources is to delay a portion of mobile traffic for a short period of time to reduce the peak throughput over time. As a simple approximation, we use the moving average in a short period of time to demonstrate the potential benefits of delaying mobile traffic. Figure 31 shows that the peak value of the 20 second moving average is only about 60% of the original peak throughput. This implies that delaying a portion of mobile traffic by 20 seconds or less can result in a significant reduction in the peak throughput demand in the cell. It will lead to two major benefits. First, the reduction in the peak throughput allows the network to support more users and mobile traffic without upgrading the infrastructure. In some cases, it also means that the cellular

network may be able to support better quality services. For example, it may be feasible to support a better quality video streaming (say HD video with better resolution) if there is sufficient reduction in the overall load on the cell. Second, this also reduces congestion and helps improve the performance of mobile applications that are sensitive to delays (e.g., VoIP).

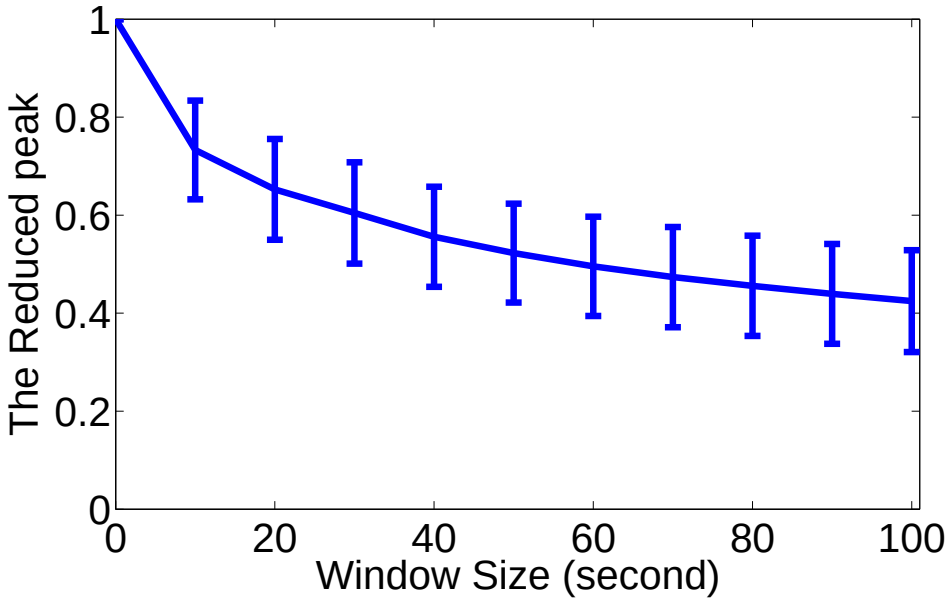


Figure 32: The reduction of the peak downlink throughput for moving averages computed over different time periods. The average and standard deviation are plotted.

To further quantify the relationship between the extra delay and the reduction on the peak throughput, we compare the reduced peak throughput achieved by moving averages computed over different time intervals for the top 200 heavy-loaded cells. The results are plotted in Figure 32. When the window size increases from 1s to 30s, the reduced peak throughput quickly decreases from 100% to 60.5% of the original peak on average. As the window size further increases to 100s, the peak is gradually reduced to 42.5% of the original peak. This figure implies that most benefits in terms of reduction of peak throughput can be obtained by delaying traffic for short time durations (e.g., 30 seconds), with diminishing returns for larger delay intervals.

6.3.2 Delay Tolerance of Mobile Applications

The real-world cell traffic traces indicate that delaying mobile traffic for a few seconds can reduce the peak load in the cell significantly. The next natural question is: Can real-world mobile applications tolerate such delays without affecting the quality of service experienced by the end-users? To investigate this, we consider following major traffic classes:

6.3.2.1 Streaming

Streaming applications (e.g., video streaming, audio streaming) account for around 34% of the total mobile traffic [83]. As they usually buffer some data, they can tolerate small delays which are equivalent to the current buffer occupancy of their playback buffer.

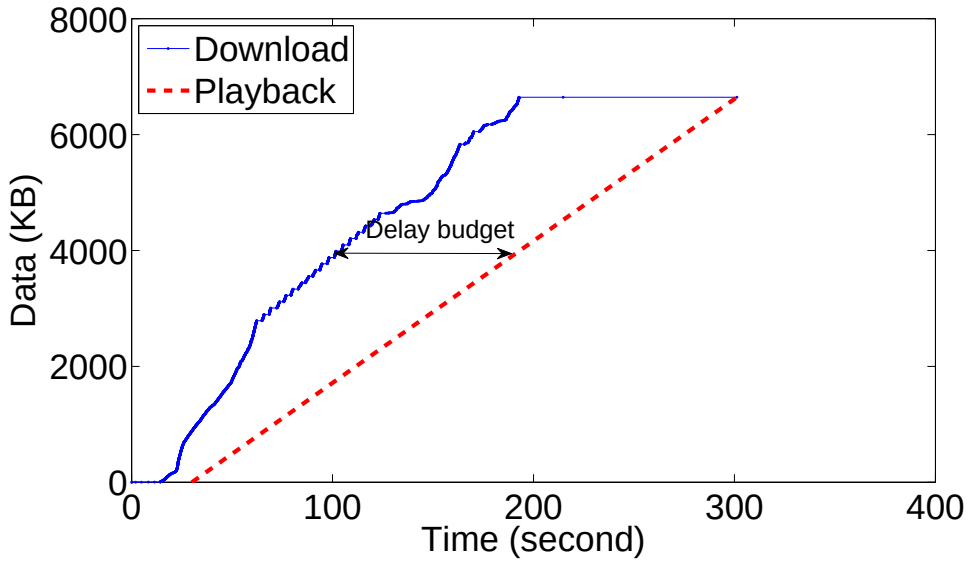


Figure 33: The download and playback progress of a Youtube video on an Android smartphone using a cellular network. The difference between download and playback represents the delay it can tolerate.

To investigate delay tolerance of streaming applications over cellular networks, we play a Youtube video on an Android smartphone and record the cellular traffic using Wireshark. Figure 33 compares the number of downloaded bytes and the actual playback progress during the experiment. Initially, the Youtube client aggressively buffers the video contents. But once the client has sufficient contents to play for some time, it slows the download to avoid downloading unnecessary contents in case the user does not watch the complete video. The difference between the actual download and the playback progress at any given

time is the amount of delay the Youtube client can tolerate at that time without affecting the user experience (i.e., pausing the video).

We also see from Figure 33 that depending on the size of the buffered data, the tolerable delay varies from a few seconds to more than one hundred seconds in this example. In addition, user operations like "back" and "forward" will also impact the delay that the video client can tolerate at the specific time.

To accurately predict the future traffic and delay that the video client can tolerate, video size, bitrate, buffered data, and playback progress are required. Specifically, we can predict the amount of data to download based on the video size and buffered data. Meanwhile, the bitrate, buffered data and playback progress can be used to estimate the delay that it can tolerate. Fortunately, all of them are available to the video client. Video size and bitrate can be obtained from the video metadata, while buffered data and playback progress are internal states of the video client. In contrast, delaying the streaming traffic arbitrarily, say from the network side without taking real time input from the application, may affect the user experience.

6.3.2.2 Web Browsing

Web browsing applications, which are also one of the top generators of cellular mobile traffic [83], are generally not regarded as being delay-tolerant. Due to their small size, mobile devices like smartphones and tablets can display only a small portion of a webpage (e.g., texts, images, and other multimedia contents) at any given time. Thus when a user browses a web page, only contents that are shown on the screen need to be downloaded immediately, while off-screen contents can be downloaded a little bit later without impacting the user experience. In other words, off-screen contents can be treated as being delay-tolerant. In fact, some websites (e.g., Huffington Post [6]) already support progressive download and display of the web pages. When web pages contain many multimedia contents (e.g., images), the amount of traffic that can be delayed will be significant.

To identify the delay tolerance of web browsing, we analyze the on-screen and off-screen contents of the top 500 Alexa websites [12] in various categories. We only treat

off-screen images as off-screen contents and the rest components as on-screen contents. We use PhantomJS [8] to download and render the web pages. For every website, we only download and analyze its home page. We set the user agent of all HTTP requests to that of the Android web browser and let those websites decide whether to return the mobile version or the full version. The screen size is set to 480×800, a typical setting for smartphones. For each web page, we identify all the off-screen images and treat them as off-screen content, while all other contents are treated as on-screen content.

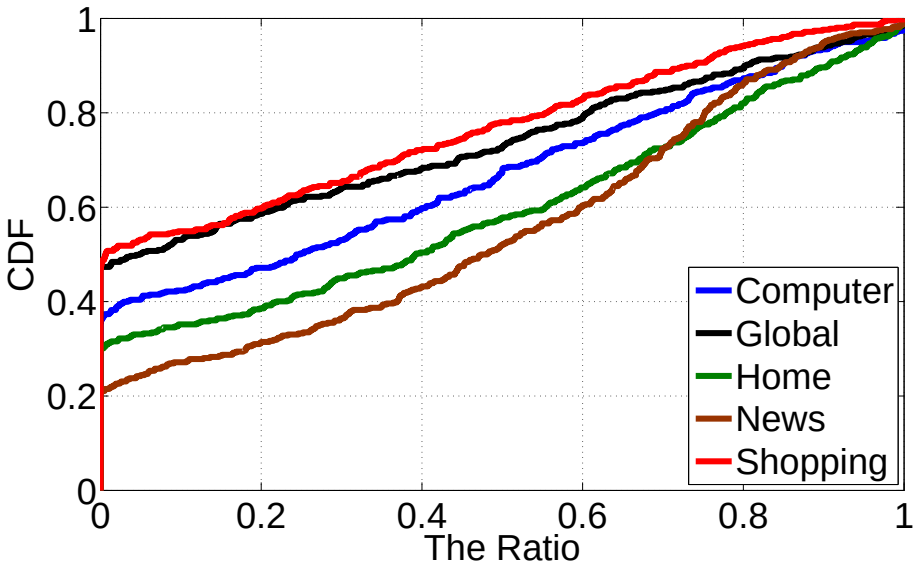


Figure 34: The ratio of off-screen content to the total content on the homepages of the top 500 websites. We also select 4 categories and plot the top 500 websites in these categories.

Figure 34 shows the ratio of the size of the off-screen contents to that of the total contents on the homepage of the top 500 Alexa websites [12]. The category "Global" represents the overall top 500 websites. We also select and plot 4 categories among 16 special categories listed by Alexa. The distributions of other categories are between that of "News" and that of "Shopping". Generally a significant portion of the web content is off-screen and can tolerate short extra delays. For categories like "News", more than 50% of those websites have more than 50% of the contents that can tolerate extra delay. In addition, since the homepages of the websites that we analyzed usually contain less content than other pages, our estimation of the potential benefits as shown in Figure 34 is likely very conservative.

Like streaming applications, we also notice that only the web browser knows which

images are off-screen and can tolerate short delay, especially when the user scrolls the web page. Specifically, by parsing the HTML file, the web browser can identify the images that are not shown on the screen. In addition, using the height and width attributes of the corresponding "img" tags, it can estimate their sizes.

6.4 CoAST Design

CoAST enables collaboration among mobile devices for scheduling their mobile traffic, when necessary and feasible, to reduce the peak traffic load on a cell. The basic idea is to use dynamic pricing to motivate the mobile applications to proactively shift their traffic in small time scales (up to 30 seconds) while still satisfying their delay constraints. To realize this idea, CoAST uses three major mechanisms: a protocol to allow mobile applications and their associated cell to exchange traffic information, an incentive mechanism to incentivize mobile applications to collaboratively delay traffic at the right time for the right time duration, and a mechanism to enable applications to delay their traffic. The first two mechanisms are incorporated into the *control plane* of CoAST, while the last one is realized in its *data plane*. However, it should be noted that all CoAST mechanisms are data plane functions from 3GPP protocol perspective, i.e. CoAST does not change the 3GPP protocol itself.

6.4.1 Design Principles

Minimal modification to mobile applications: For ease of deployment, CoAST requires no modification on the server side, but only small changes on the client side because only mobile applications know the delay constraints of their traffic. CoAST modifies the underlying socket API implementation to allow client applications to specify the tolerable delay information via these socket calls in a transparent manner. The actual value of the tolerable delay at a given time instant depends on the application itself. In this chapter, we describe how playback buffer size can be used to figure out the value of tolerable delay for video streaming applications. For other applications, mobile developers may use existing instrumentation systems [53] to determine the tolerable delay, thereby reducing the development efforts.

Privacy preservation: To reduce the peak load on the cell, CoAST relies on collaboration and sharing of traffic information among all mobile devices in that cell. A malicious mobile device may participate CoAST and collect the traffic information of other mobile devices to infer their application usage. To prevent privacy leak under such attack, the control plane is divided into UE proxies on mobile devices and a market proxy on the eNodeB. Each mobile device shares its aggregated traffic demand only with the market proxy and obtains the prices for downloading data only from the market proxy, avoiding the direct sharing of traffic demands among participating devices. With this design, it will be impossible for malicious mobile device to obtain accurate traffic information of mobile applications running on other mobile devices.

Control of demand through pricing: To motivate mobile applications to delay their traffic when necessary, CoAST uses dynamic "prices" to charge mobile traffic at different time instants. The prices used by CoAST can be $ per bit as used in [50]. More generally, it can also be treated as the discount ratio on the accounted traffic. For example, when the price is 0.8, 1 Mb mobile traffic can be accounted as 0.8 Mb. The latter case is compatible with the usage-based pricing model used by most cellular providers nowadays. In this chapter, we don't specify the pricing model used by the cellular providers. We treat the price as the ratio of accounted traffic to the transferred traffic.

Tackling system abuse: CoAST requires UEs to report traffic demands to the market proxy, which sets the prices based on demand information from all UEs in the cell. Malicious UEs may attempt to report fake information to abuse the system. To prevent such cheating behavior, CoAST records and compares the reported traffic demands and the real traffic to identify suspicious behavior (please see Section 6.8 for a detail discussion).

6.4.2 Overview of CoAST Operation

Figure 35 provides a high-level overview of the CoAST architecture. It consists of two major components: a *market proxy* that resides on a cell-level network element like eNodeB and a *UE proxy* on each mobile device. The market proxy collects the traffic demands for some future time window from all mobile devices in a cell. These are used by the market proxy as

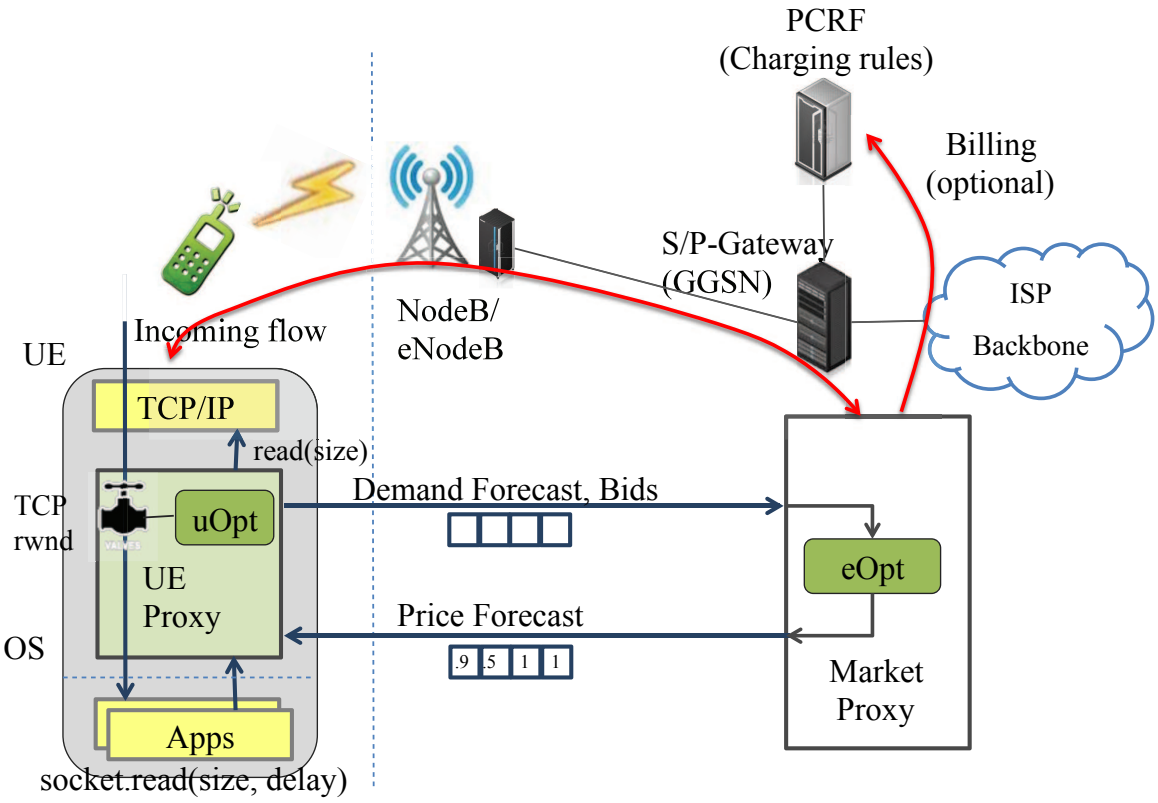


Figure 35: CoAST Architecture.

parameters of an optimization problem (eOpt) which determines new future prices for each UE (for the next time window) with the goal of minimizing the traffic peak in the cell. The traffic demands and prices may also be reported to the LTE network for accounting and billing purpose. On the user side, applications determine their traffic demands in a manner that satisfies their delay constraints and sends this information to its UE proxy through the user library. The UE proxy uses these demand information as inputs to an optimization problem (uOpt) that attempts to minimize the cost of satisfying these demands based on prices obtained from the market proxy. CoAST also includes a mechanism by which applications can control their traffic according to the outputs of uOpt.

6.4.3 Control Plane

CoAST ensures that the traffic demand does not exceed the network capacity at any time without affecting the user experience of mobile applications. Let's denote the capacity of a cell sector by c, the throughput at time t of the i^{th} flow under the control of CoAST by $th_i(t)$, and the total throughput of all other flows by $th_b(t)$. CoAST's goal can be expressed

as optimizing an objective function of the difference between the throughput and capacity over time, as follows:

$$\min \ \Phi(\{\sum_{i=1}^{n} th_i(t) + th_b(t) - c\}) \tag{12}$$

where Φ can be any meaningful utility function. For simplicity of description, we omit the constraints in Eqn 12 and will present them in Eqn 15.

One design challenge is the fact that CoAST has neither control nor accurate information about the background traffic in the cell. By background traffic, we mean all mobile traffic generated by applications that do not use CoAST. Given the burstiness of mobile traffic as shown in Figure 31, it's also hard to accurately predict the background traffic using historical information. To solve this problem, CoAST optimizes the maximal throughput of the flows under its control over time, as follows:

$$\min \ \max_{t} \sum_{i=1}^{n} th_i(t) \tag{13}$$

Therefore, no information of the background traffic is required. Let $th_b^* = \max_t th_b(t)$. A pleasant property of this objective function is that it is equivalent to minimizing $\sum_{i=1}^{n} th_i(t) + th_b^*$, which will be proved in Section 6.4.5. When enough numbers of UEs use CoAST, the peak throughput of total traffic will be minimized.

A centralized solution to Formula 13 would require the market proxy and the UE proxies to share information of all flows, resulting in a lot of control overhead. Moreover, it violates our privacy preservation design goal. Thus, a distributed control protocol that exchanges limited information is required. To solve this problem, we use the dual decomposition method [25] to decompose the original problem into a master problem and multiple independent sub-problems. The master problem (eOpt) is solved by the market proxy, while the independent sub-problems (uOpt) are solved by the UE proxies. We describe this control protocol next.

At a high level, the control plane functions as follows: The market proxy periodically computes the projected prices for mobile traffic for some time window in the future. These prices are calculated based on the traffic demand collected from all connected UEs. The prices are broadcasted to all connected UEs. Each UE proxy uses the price information from

the market proxy and the demand information from its mobile applications to schedule the mobile traffic such that the demand of each application is satisfied and the overall cost is minimized. The UE proxy then sends back the traffic demands that it calculates for some future time window to the market proxy, and the process is repeated.

6.4.3.1 Market Proxy Operations

The market proxy operates in time slots with length τ (e.g., 1 second). The time slot that time t belongs to is denoted as $\lfloor t \rfloor_\tau = \lfloor \frac{t-t_0}{\tau} \rfloor$, where t_0 is the start time. Let $p(\lfloor t \rfloor_\tau)$ represent the price for the downlink traffic through a cell sector during time slot $\lfloor t \rfloor_\tau$. Let $th_e(\lfloor t \rfloor_\tau)$ represent the downlink traffic demand of UE e during time slot $\lfloor t \rfloor_\tau$.

In every δ seconds (e.g., 0.1), the market proxy receives a vector of traffic demand for the next κ (e.g., 30) time slots, i.e., $\overline{th_e} = \{th_e(\lfloor t \rfloor_\tau), th_e(\lfloor t \rfloor_\tau + 1), \ldots, th_e(\lfloor t \rfloor_\tau + \kappa)\}$, where t is the current time. Then it updates the price vector , i.e., $\overline{p} = \{p(\lfloor t \rfloor_\tau), \ldots, p(\lfloor t \rfloor_\tau + \kappa)\}$, as follows:

$$\overline{p'} = [\overline{p} + \beta(\sum_e \overline{th_e} - \alpha)]_H \tag{14}$$

where $\alpha = \max_t \sum_e th_e(t)$, $[\overline{p}]_H$ represents the projection of $\overline{p}$ onto the hyperplane $H = \{p(t) | \sum_t p(t) = 1, p(t) \geq 0\}$, and β is the step length. The value of β is set to ensure that $\forall t, |p'(t) - p(t)| < p(t)/10$ in our implementation.

The market proxy then broadcasts the prices to all UEs in the cell.

6.4.3.2 UE Proxy Operations

The UE proxy collects the information about traffic demand per slot (τ) for κ slots into the future from all mobile applications and periodically receives the future price information from the market proxy. It generates the future traffic demand, $\overline{th_e}$, and sends it to the market proxy.

Each mobile application reports its delay constraints for the next κ time slots to its UE proxy. Let's use $\langle D, t \rangle$, i.e., a tuple of the amount of data D and its deadline t, to express the delay constraint that data D should be downloaded before deadline t. Therefore, for a specific flow i, its delay constraints can be expressed as a set of tuples, i.e.,

$\{\langle D_{i,0}, t_{i,0}\rangle, \langle D_{i,1}, t_{i,1}\rangle, \ldots, \langle D_{i,n}, t_{i,n}\rangle\}$, where $t_{i,0} < t_{i,1} < \cdots < t_{i,n}$. Let's define function $d_i(t) = \sum_{j=0}^{T} D_{i,j}$, where $t_{i,T} \leq t \leq t_{i,T+1}$. It represents the amount of data required to be transferred before time t.

When a UE proxy receives the price information from the market proxy, it tries to minimize the cost of transferring data under the constraint that the delay constraints of all flows are satisfied. Specifically, at time t^* the UE proxy solves the following optimization function:

$$\begin{aligned} \min \quad & \sum_i \sum_{t \in T} th_i(t) \times p(t) \\ \text{s.t.} \quad & \sum_i th_i(t) \leq b_d, \forall t \in T \\ & \sum_i \sum_{t \leq t'} th_i(t) \times \tau \geq d_i(t), \forall t' \in T \end{aligned} \tag{15}$$

where $th_i(t)$ is the throughput of flow i at time t, b_d is the bandwidth, and $T = \{\lfloor t^* \rfloor_\tau, \ldots, \lfloor t^* \rfloor_\tau + \kappa\}$.

By solving the above optimization function we obtain the desired throughput of all downlink flows over time. $th_i(\lfloor t^* \rfloor_\tau)$ corresponds to the bandwidth allocated to flow i in the current time slot. The UE proxy will send this value back to the mobile applications to control their traffic in the data plane accordingly as described in Section 6.4.4. It should be noted that those constraints may not be satisfied, i.e., the available bandwidth may be smaller than the traffic demand. Under such scenario, CoAST will allow mobile applications to transfer data as fast as possible and let users decide if they want to stop some applications.

The UE proxy will send $\{th_e(t) | th_e(t) = \sum_i th_i(t), t \in T\}$ to the market proxy. The market proxy collects such traffic demands from all UEs in the cell and updates the prices in the next round.

6.4.4 Data Plane

The primary functionality of the CoAST data plane is to control the downlink traffic based on the throughput cap assigned by the control plane. As most of the mobile applications we are considering use TCP and we don't want to modify the server, we focus on controlling the TCP traffic from the receiver side. It is also important to note that while the control

plane operates with a window of projected demands and prices, the data plane only controls the traffic for the current slot.

In TCP the amount of data that the sender can send within an RTT is limited by $\min\{cwnd, rwnd\}$ where $cwnd$ is the congestion window size, and $rwnd$ is the receiver's window size advertised in the acknowledgement packets. When cwnd is larger than rwnd, rwnd will determine the throughput of a TCP flow. To control the downlink traffic from the receiver side, we set an upper-bound on rwnd as:

$$\text{DL\_CAP} = \max\{\text{throughput} \times \text{RTT}, \text{MSS}\} \tag{16}$$

where *throughput* is the target throughput assigned by the control plane. When throughput is very small, DL_CA*P* is set to MSS to avoid totally blocking the flow. In our implementation, we add a new socket option, DL_CAP, to allow applications to dynamically specify the upper-bound on the advertised receiver's window size at runtime.

It's noteworthy that the receiver will obtain the required downlink throughput after one RTT since the sender receives the new advertised rwnd after RTT/2 and then spends RTT/2 to deliver the new packets to the receiver. When RTT is large (e.g., 1 second), the receiver should use the estimated future throughput to set DL_CAP.

6.4.5 CoAST Performance Guarantee

The primary goal of CoAST is to schedule the last-mile traffic to ensure that the total traffic demand does not exceed the network capacity. For scalability, CoAST is designed as a distributed system that only schedules the traffic under its control. A natural question is whether the CoAST design meets its goal. Here we present a theoretical analysis to answer this question.

CoAST uses an iterative control protocol between a market proxy and a set of UE proxies to schedule the traffic. This control protocol allows CoAST to minimize the maximal throughput of flows under its control over time. Formally, we have the following theorem:

Theorem 1. *With the interaction interval, δ, approaching 0, CoAST approaches the optimization goal defined in Formula 13.*

Proof. The goal of the market proxy is to minimize the maximal throughput on a cell sector over time, as shown in Formula 13. To decompose this problem and derive a distributed protocol, we rewrite the objective of the market proxy as follows:

$$\min \quad \alpha \tag{17}$$

$$\text{s.t.} \quad \forall t, \sum_i th_i(t) \leq \alpha \tag{18}$$

Introducing a dual variable $p(t) > 0$ (i.e., price for downlink traffic through the cell sector at time slot t) for each constraint of (18), we define the Lagrange dual function

$$L(\{p(t)\}) = \min \alpha + \sum_t p(t)(\sum_i th_i(t) - \alpha) \tag{19}$$

To make $L(\{p(t)\})$ finite, the coefficient of α should be 0:

$$\sum_t p(t) = 1 \tag{20}$$

Then, we simplify $L(\{p(t)\})$ to

$$L(\{p(t)\}) = \min \sum_t p(t) \sum_i th_i(t) \tag{21}$$

$$= \min \sum_i \sum_t p(t) th_i(t) \tag{22}$$

The original problem can be decomposed into independent problems for each mobile applications. If we aggregate the throughput by UEs, we obtain

$$L(\{p(t)\}) = \min \sum_e \sum_{i \in e} \sum_t p(t) th_i(t) \tag{23}$$

$$= \sum_e \min \sum_{i \in e} \sum_t p(t) th_i(t) \tag{24}$$

Therefore, the original problem is decomposed into independent sub-problems for each UE. The objective of each UE is to select $th_e(t)$ among all feasible values so that $\sum_{i \in e} \sum_t p(t) th_i(t)$ is minimized. The derived objective of each UE is exactly the same as (15).

The market proxy, running the master program of the decomposition method, gets feedbacks (i.e., $th_e(t)$) from UE proxies and updates $\{p(t)\}$ using a projected sub-gradient method as described in (14). Therefore, the objective of Formula (13) is equivalent to the control protocol in Section 6.4.3. □

As CoAST minimizes the maximal throughput of flows under its control, it also reduces the maximal throughput of all flows including those background traffic. Formally, we have the following theorem:

Theorem 2. *Assume* $\sum_{i=1}^{n} th_i(t)$ *and* $th_b(t)$ *are independent random variables. With* $t \to +\infty$*, the expected value of the peak throughput obtained by using CoAST approaches* $\max_t \sum_{i=1}^{n} th_i(t) + \max_t th_b(t)$.

Proof. Let $x_t = \sum_{i=1}^{n} th_i(t)$ have a probability density function $f(x_t)$ and a cumulative probability distribution function of $F(x_t)$. Let $y_t = th_b(t_t)$ have a probability density function $g(y_t)$ and a cumulative probability distribution function of $G(y_t)$. Let x_{max} and y_{max} be the lowest value of x_t and y_t such that $F(x_t) = 1$ and $G(y_t) = 1$, respectively. Since the number of flows associated with a cell is limited, x_{max} and y_{max} are finite. Let p be the convolution of f and g, and P be its cumulative probability distribution function. Then the sum $z_t = x_t + y_t$ is a random variable with the density function $p(z_t)$ and the cumulative probability distribution function of $P(z_t)$. Since x_t and y_t are independent, $P(z_t \leq x_{max} + y_{max}) = 1$.

Let $z^* = \max_t z_t$ where $t \in \{1, 2, \ldots, T\}$. The probability density function of z^*, $h(z^*)$ is

$$h(z^*) = T[P(z^*)]^{T-1} p(z^*) \tag{25}$$

The expected value of z^* is

$$\begin{aligned} E(z^*) &= \int_{-\infty}^{+\infty} z^* T[P(z^*)]^{T-1} p(z^*) dz^* && (26) \\ &= \int_0^1 P^{-1}(z^{*\frac{1}{T}}) dz^* && (27) \end{aligned}$$

Therefore, when $T \to +\infty$, we have

$$\lim_{T \to +\infty} E(z^*) = \int_0^1 P^{-1}(1) dz^* = x_{max} + y_{max} \tag{28}$$

Since CoAST minimizes x_{max} with y_{max} unchanged, it is equivalent to minimizing $\max_t \sum_i th_i(t) + y_{max}$ when $T \to +\infty$. □

Thus, when the number of UEs that uses CoAST is large enough, CoAST can significantly reduce the overall peak traffic.

6.5 Deployment and Implementation

In this section, we discuss various ways in which CoAST can be deployed on a real network and describe our prototype implementation.

6.5.1 Deployment

CoAST proposes two new functions to be added to the cellular network: a UE proxy for each mobile device and a market proxy for each cell sector. The UE proxy interacts with mobile applications running on the UE to collect their delay constraints and allocates bandwidth to them. The market proxy aggregates demand information from all the UEs in a cell and sets the prices. UE and market proxies communicate with each other to exchange demand and pricing information. The network elements on which these functions are deployed determines both the information available to CoAST and the changes needed to the network. We consider three deployment options and their merits.

Clean Slate: The market proxy for a cell and the UE proxies for all UEs in the cell are hosted in the cell's eNodeB. Mobile applications directly communicate their requirements to their UE proxy through extensions to the 3GPP RAN control plane (via mobile OS APIs to expose these extensions). The UE proxy directly controls bandwidth allocations through the eNodeB scheduler. Because the market proxy also resides on the eNodeB, it has full access to cell utilization information when setting prices and no latency between the UE and market proxy. The market proxy can also compare reported traffic and real traffic to detect price manipulation. While this represents the cleanest and most functional design,

it requires changes to the 3GPP protocol to exchange demand and price information, to eNodeBs, and to the mobile OS, and thus may be difficult to deploy.

Incremental: The UE proxy is deployed on the mobile device as a daemon process, while the market proxy is deployed by the network provider as a new network element in the packet core. The UE proxy interacts with mobile applications through user library calls for collecting delay constraints and allocating bandwidth (via a controlled socket abstraction). Interaction between the UE and market proxy for exchanging demand and price information occurs using the normal cellular network data plane, through a well known UDP port and a special destination IP address that points to the market proxy for the current cell. This configuration imposes higher latency between the UE and market proxies, but because the market proxy is deployed by the network provider, it can be made as low as possible. Also, the market proxy may be given access to real-time traffic information through a private interface to the eNodeB as well as the provider's charging and data metering systems [75]. Thus, it provides most of the benefits of the clean slate design while being easier to deploy - 3GPP protocols or eNodeBs do not have to be extensively modified on the provider side, while the UE proxy can be implemented as a user space library without modifying the mobile OS on the UE side.

Over-the-top: The UE proxy is deployed as a library on the UE just as in the incremental design, but the market proxy is deployed as a third-party service on an external cloud server, possibly even on an application-by-application basis. E.g., a large video streaming provider may have its own market proxy that serves to smooth only its own traffic within a cell and improve the performance for its own users. In this design, when a mobile device connects a cell, its UE proxy uses the cell ID to find the IP address of the corresponding market proxy through standard DNS mechanisms. UE and market proxies exchange information through UDP as in the incremental design, but the third party nature of the market proxy precludes the market proxy from basing its pricing decisions based on real-time traffic information (e.g., it cannot lower prices during periods of low utilization to incentivize more aggressive transfers) or from actually providing real monetary incentives to users. However, the design requires no modification to any cellular network elements, or to the mobile OS,

and thus is the easiest to deploy.

Due to its ease of deployment, we chose the over-the-top design for our prototype. However, the goal of this chapter is to evaluate the feasibility of CoAST and quantify its benefits and costs instead of advocating a particular deployment choice. Our evaluation will hold irrespective of the design chosen.

6.5.2 Prototype Implementation

We implemented market proxy on a Linux system with a dual-core 2.53 GHz CPU and 4GB of RAM. The market proxy divides time into 1-second slots and dynamically sets the prices for future 30 time slots. Every 100ms the market proxy collects the traffic demands for the future 30 time slots from all mobile devices that connect to it. Then it updates the prices based on the aggregated traffic demand and sends the new prices to all the mobile devices.

Ideally the market proxy should reside on a cell-level network element like the eNodeB in a 4G LTE network or the RNC in a 3G network where it can also monitor the traffic over the cell. However, because we have no access to such network elements, we have to run the market proxy on a remote server whose RTT to the mobile devices through a 4G LTE network is 57ms on average. Since the RTT between the market proxy and UE proxies is much smaller than their interaction interval (i.e., 100ms), the functionality of the control plane will not be affected.

We implemented the UE proxy on Android 4.1. It is implemented as an Android application collecting delay information from mobile applications and the prices from the market proxy and assigning throughput caps to the mobile traffic. To dynamically control the downlink traffic from the receiver side, we also patched the Android kernel to add a new socket option, DL_CAP, to limit the maximal throughput of the TCP flows at runtime.

6.5.2.1 Communication and Computation Overhead

The traffic demands and prices are the only data exchanged between the UE proxy and the market proxy. In the current implementation, we use 2 bytes for the traffic demand and 1 byte for the price in each time slot. Therefore, each UE proxy uploads 60 bytes and downloads 30 bytes from the market proxy in every round. Since they exchange information

every 100ms, the control overhead in this case is 600 Bytes/s in the upload direction traffic and 300 Bytes/s in the download direction in the worst case. The overhead is much lower in reality because of two reasons. First, the UE proxy will exchange information with the market proxy only if some communication-intensive applications (e.g., video streaming) are running. Second, compared with the high traffic volume of those target applications, the extra communication overhead of CoAST is negligible. As the experiment in Section 6.6.2 will show, CoAST only leads to 0.07% extra traffic.

The computation overhead is also very low. The market proxy updates the prices by solving a projection problem. Its computational complexity is $O(\kappa)$ where κ is the number of time slots (30 in our implementation). The UE proxy needs to solve an optimization problem that minimizes the cost under the delay constraints. As we discretize time into time slots, the maximal number of delay constraints for a flow is $O(\kappa)$. The optimization problem can be solved by gradually finding the minimal cost for each constraint. Therefore, its computational complexity is $O(f \times \kappa^2 \times log(\kappa))$, where f is the number of concurrent flows. On an old Motorola ATRIX smartphone with Android 2.3, it takes less than 1ms for the UE proxy to solve the problem for 100 concurrent flows.

6.6 System Evaluation

In this section, we evaluate the CoAST prototype as described in Section 6.5.2. We will demonstrate how CoAST improves the performance of mobile applications under various LTE network congestion states.

6.6.1 Experimental Setup

Our testbed is composed of 5 Linux workstations and 4 Android smartphones with 4G LTE capability. The first workstation acts as the market proxy. Its average RTT to these smartphones through the LTE network is 57ms. The other 4 workstations act as the remote mobile application servers, each of which serves one smartphone. The Linux traffic control tool *tc* is used on application servers to control their available bandwidth. We make sure that all these smartphones connect to the same cell during the experiments by checking the cell id of their connected cell. All experiments are conducted in a residential area around

midnight to reduce the impact of other cellular users.

CoAST is designed to improve the performance of mobile applications when the LTE network is congested. We use following mobile streaming strategies to create different levels of network congestion and streaming behavior. The evaluation for other types of mobile applications (e.g. web browsing) will be presented in Section 6.7. We find that the streaming strategies impact the distribution of mobile traffic and how they interact with each other in CoAST.

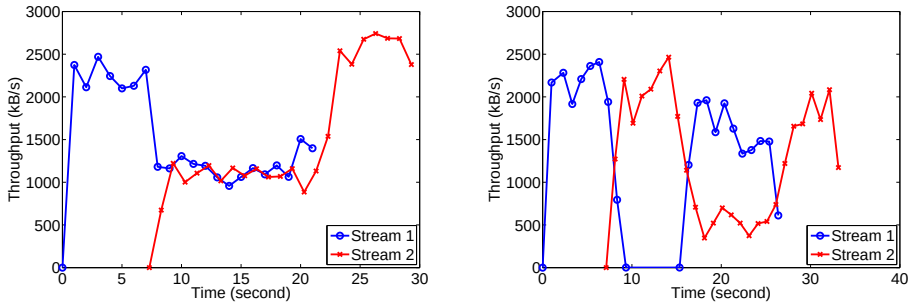
- **All-at-once:** strategy aggressively transfers data from the server to the client as fast as possible. This strategy is usually used by some audio streaming [79] and video streaming applications on some specific phones [92].
- **Pacing:** strategy controls the download throughput at the "steady state" after initial buffering. This strategy is widely used in video streaming services such as Youtube, Hulu, and Netflix [43].
- **Bundling:** strategy is proposed to reduce the energy consumption of video streaming applications [92]. It divides the entire stream into several large chunks and aggressively transfers each chunk periodically.

We use two metrics to evaluate the impact of CoAST on mobile users:

- **Buffering time:** It is a direct measure of the user experience for streaming applications. We consider both initial buffering period and rebuffering period in the steady state.
- **Energy consumption:** The energy consumption of video streaming is primarily caused by the device screen in the "on" state and the LTE network interface. For fair comparison, we use the LTE energy model that calculates energy consumption based on traffic traces [57].

6.6.2 The Reduction of Buffering Time in Video Streaming

We first demonstrate the basic mechanism through which CoAST enables collaborative traffic scheduling via a simple two-device video streaming experiment. Let the two devices



(a) Native LTE network (b) CoAST

Figure 36: The interaction between two video streams

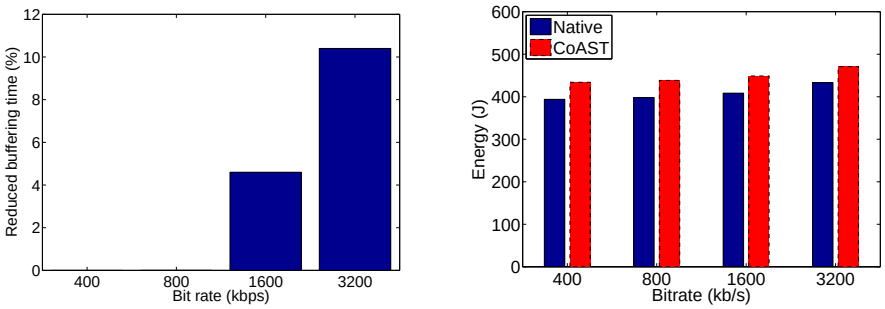
start streaming two HD videos at time 0s and 7s, respectively. Both of them use the all-at-once strategy. The LTE cell is the only bottleneck of both streams.

Figure 36 plots the throughputs of two streams for the native LTE network and CoAST cases. In the native LTE case, stream1 is unaware of the traffic demand of stream2 and, thus, continues downloading data from its server even after stream2 starts. Therefore, the initial buffering period for stream2 is twice as long as that of stream1. In CoAST, when stream2 starts at 7sec with an empty buffer, it tries to download aggressively, thus causing the market proxy to increase prices due to the increased demand. Because stream1 has a relatively full buffer, it is less willing to pay the increased price than stream2, and thus delays its traffic. After some time, when stream2 has buffered enough data for playback, it reduces its willingness to pay higher prices, and thus begins delaying its data more. In the meantime, stream1 has already played back a portion of data in its buffer and becomes more aggressive to meet its delay constraints. Due to this cooperative inter-play between the UEs — the UE with full buffer deferring its download in favor of the UE with empty buffer— the buffering time of stream2 is reduced by 50% while that of stream1 is not affected.

By exchanging traffic demands and prices between the UE proxies and the market proxy, CoAST incurs a total of 47.5 kB extra traffic in the experiment. Compared with the total download traffic (i.e., 64 MB), CoAST only incurs a 0.07% communication overhead.

6.6.3 The Impact of Network Congestion

Next we evaluate the impact of traffic demand on CoAST performance by varying the stream bitrate from 400 kb/s (i.e., average Youtube bitrate [44]) to 3200 kb/s while keeping



(a) The reduced buffering time (b) Energy consumption

Figure 37: The impact of taffic demand on CoAST performance

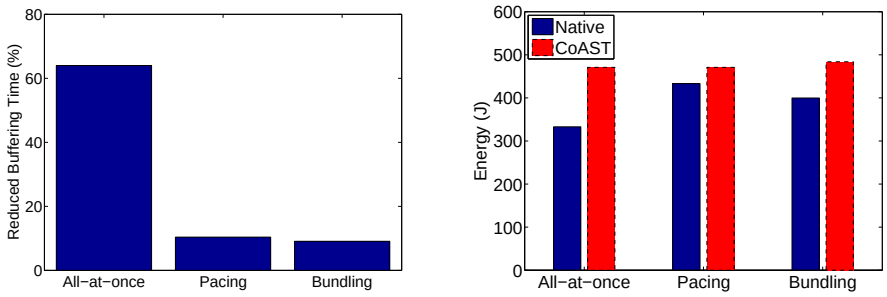
the number of streams unchanged. We randomly start 4 streams within 30 seconds. The length of each stream is 3.33 minutes, i.e., the median Youtube video length [44]. The upload bandwidth of servers are unlimited, i.e., the LTE cell is the only bottleneck. Each experiment is repeated 3 times with different random seeds. The average values are reported.

Figure 37 plots the reduced buffering time and average energy consumption of the smartphones. It is clear that the benefits of CoAST start increasing with the increase in traffic demand (i.e. bitrate in the experiments). When the bitrate is 3200 kb/s, CoAST is able to reduce buffering time by more than 10% on average. It is exactly the design goal of CoAST, i.e., to reduce the impact of increased mobile traffic on user experience.

Figure 37(b) also shows that CoAST increases the energy consumption by less than 8% in all the experiments. More importantly, with the increase in traffic demand, the extra energy consumption is even smaller (e.g. 7% for 3200 kb/s case). The increase in energy consumption is caused by longer streaming time in CoAST. Note that energy consumption is based on the assumption that there is no background traffic. Otherwise, CoAST will incur even smaller energy consumption overhead.

6.6.4 The Impact of Streaming Strategies

Finally we evaluate the impact of streaming strategy on the performance of CoAST. The stream bitrate is 3200 kb/s. Other parameters are the same as previous experiments. The experiment results are shown in Figure 38. When the all-at-once strategy is used, CoAST reduces buffering time by 66%. However, it also increases the energy consumption significantly if the entire video is downloaded. Compared to the pacing strategy, CoAST



(a) The reduced buffering time (b) Energy consumption

Figure 38: The impact of streaming strategies on CoAST performance

incurs similar buffering time and slightly more energy consumption in the bundling strategy.

6.7 Trace-Driven Evaluation

In this section, we demonstrate the potential benefits of CoAST by using trace-driven evaluations of real-world cellular traffic traces from a large US mobile network operator.

6.7.1 Experimental Setup

We implement CoAST on a packet-level simulator, ns-3 [7], which supports the simulation of LTE networks. The network is composed of an eNodeB, a remote server and multiple mobile devices. The market proxy is installed on the eNodeB, while the application servers are installed on the remote server. All mobile devices connect to the same eNodeB during the experiments. We use the default parameters for all the experiments.

Traffic traces: We identify the top 100 heavily-loaded cell sectors in our cellular traffic dataset and evaluate how CoAST reduces their peak throughputs within 1 day. For each cell, we generate the flows for Youtube video streaming and web browsing as follows.

- **YouTube video streaming:** We identify all flows from Youtube servers to the mobile devices in the cellular traffic dataset. But only flows whose size and duration are large enough (i.e., size $>$ 100 kB and duration $>$ 10 s) are treated as streaming flows. Since we don't have video information (e.g., bitrate, video length) in our dataset, we use the empirical models from [44] to generate the video profile for each flow. The pacing strategy is used to control the stream.

- **Web browsing:** We first identify all web traffic using port number 80. If two flows between the same source and destination start within 5 seconds, they are considered as belonging to the same browsing event. Using this method, we obtain a set of browsing events with their start time instants. Since the dataset doesn't contain the identity of the web page, we randomly pick one of the top 500 Alexa websites for each browsing event.

Metrics: We compare CoAST against the native network using the following metrics:

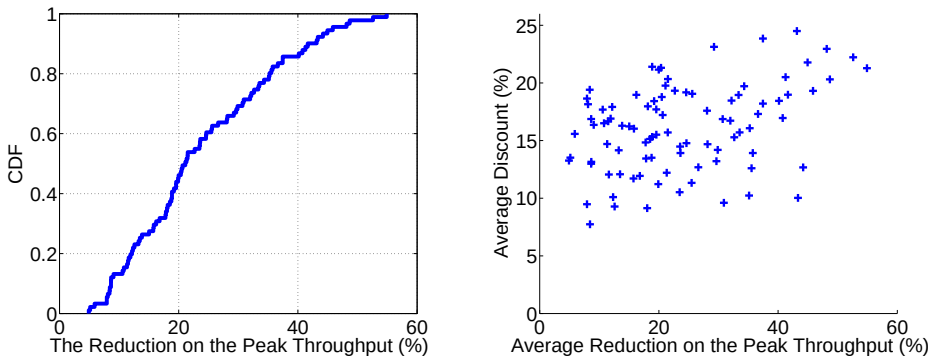
- **Peak reduction**: This is the most important metric because the goal of CoAST is to reduce the peak cell throughput. It is defined as $\frac{Peak_{native}-Peak_{CoAST}}{Peak_{native}}$.
- **Discount**: A key promise of CoAST is that mobile users will also benefit from CoAST and, thus, be willing to use it. We use $\frac{D_t-D_a}{D_t}$ to denote the user benefit, where D_t is the amount of transferred data, D_a is the amount of accounted data.
- **Overhead**: We also analyze if CoAST causes any overhead, including energy consumption and RRC signaling overhead [55].

We acknowledge that our trace-based evaluation has some limitations. First, there is no system feedback. By scheduling the traffic better, CoAST may cause mobile users to use more data and, thus, increase the peak traffic. However, our experiments cannot capture this phenomenon. Second, the user behavior in using applications is not available. For video streaming, the user may skip ahead or pause the video. For web browsing, the user may quickly scroll down in the webpage. These user behaviors impact the delay constraints of the corresponding traffic and, thus, affect the performance of CoAST.

6.7.2 Experimental Results

6.7.2.1 Video Streaming Experiments

We first perform simulation-based video streaming experiments using 100 heavy-loaded cells for 1 day. In all experiments we only consider Youtube video streams. As shown in Figure 39, CoAST successfully reduces the peak throughput for all cells. We also observe the following



(a) The Peak Reduction (b) Discount

Figure 39: The performance of video streaming supported by CoAST on various sectors.

phenomena. First, the peak reduction ranges from 5% to 55% for different cells, as shown in Figure 39(a). More than 30% cells reduce their peak by 30%. The high variation of the peak reduction among different cells is probably caused by diverse distribution of Youtube video streams among various cells. Cells with more video streams are able to achieve better performance. Second, the average discount obtained by the mobile users correlates with the peak reduction on the cells, as shown in Figure 39(b). Finally, none of the videos is paused during the experiments.

Next we explore how the reduced peaks actually help increase the capacity of cellular networks by examining the impact of increasing user demand on application level performance with and without CoAST. Specifically, we increase the users/spectrum ratio and measure how long the video streams pause waiting for more data. However, rather than increasing the number of users in a cell artificially, we increase the ratio by reducing the effective bandwidth (spectrum) available to a cell by a fraction $\alpha \in [0, 1)$, i.e., Capacity = $(1 - \alpha)\times$ MaxCapacity. The average values of video pause time over 100 cells are reported for different values of α in Figure 40. We see that CoAST results in very little application performance degradation while supporting a capacity increase of up to 20% ($\alpha = 0.2$). Also, for the same value of application performance degradation, CoAST can support more users per unit of available spectrum.

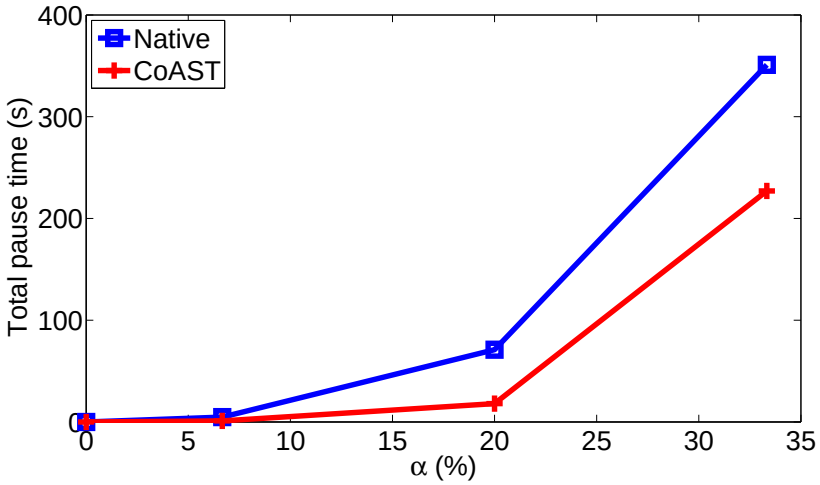
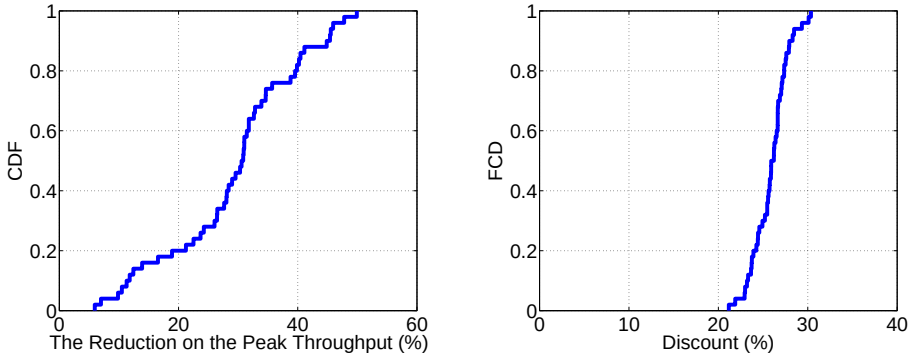


Figure 40: Comparison of video pause time for different number of users with and without CoAST.



(a) The Peak Reduction (b) Discount

Figure 41: The performance of web browsing supported by CoAST on various sectors.

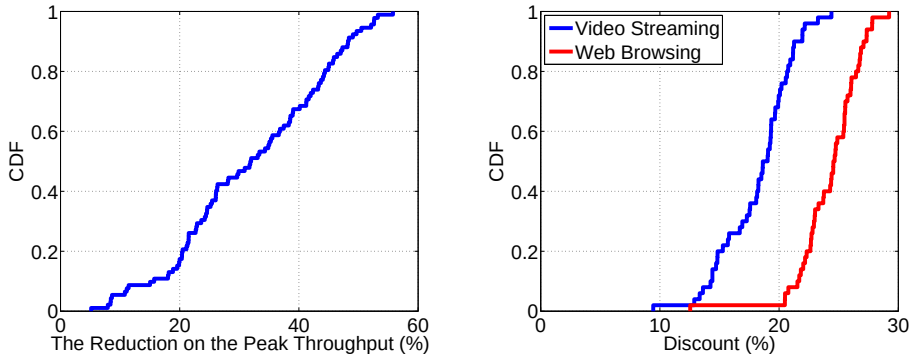
6.7.2.2 Experiments on Web Browsing

Web browsing is another important application that can benefit from CoAST because of its delay tolerance. Unlike video streaming, web browsing flows are relatively small. In this subsection, we analyze the performance of CoAST-based web browser.

First, we consider the simple scenario that all cellular traffic are web traffic. For each browsing event, we randomly choose one of the top 500 Alexa websites in the "news" category. We report the results for the 100 heavy-load cells in Figure 41.

As expected, CoAST-based web browser reduces the peak throughput for all cells. Like in video streaming experiments, the peak reduction varies significantly among cells, ranging

from 6% to 50%. However, different from video streaming, CoAST-based web browser helps more than 55% cells achieve more than 30% peak reduction. In addition, as shown in Figure 41(b), the average discount obtained by the mobile users ranges from 22% to 31%, which is more than video streaming case. This is because web browsing flows are usually very small and are able to take advantage of the variation in price.



(a) The Peak Reduction (b) Discount

Figure 42: The performance of both video streaming and web browsing supported by CoAST.

Next, we evaluate the scenario in which both video streaming and web browsing use CoAST to schedule their traffic. Video streaming and web browsing are the most important mobile applications, accounting for more than 70% mobile traffic. The experimental results are plotted in Figure 42. In this more realistic scenario, CoAST helps all cells to reduce their peak throughputs. We also plot the discount obtained by video streaming and web browsing in Figure 42(b). Web browsing is still able to obtain higher discount than video streaming.

In Section 6, we noticed that CoAST slightly increases the energy consumption in some scenarios. This is because by delaying mobile traffic, mobile devices need to keep the network interface active for longer duration, resulting in extra energy consumption. To analyze the overhead of CoAST, we assume that the mobile device is initially in the RRC_IDLE state, and there is no other traffic on the mobile device. We use the RRC state model [57] to analyze the overhead. For both video streaming and web browsing, the mobile devices always stay at the RRC_CONNECTED state when using these applications. Thus, CoAST

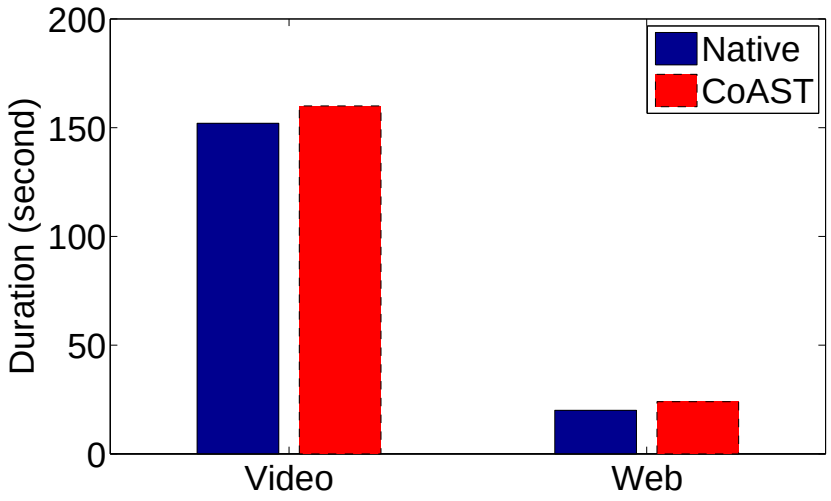


Figure 43: The time duration at RRC_CONNECTED state for video streaming and web browsing.

does not introduce extra RRC state transitions. However, CoAST does keep cellular interface alive for slightly longer duration and, thus, consumes a little more energy. Figure 43 shows that in CoAST, the mobile devices stay slightly longer in the RRC_CONNECTED state than the native case.

6.7.2.3 The Impact of Partial Deployment

CoAST reduces the peak throughput by rescheduling the traffic of mobile devices under its control. The number of participating mobile devices will impact the CoAST performance. In this subsection, we evaluate the performance of CoAST when only a portion of mobile devices support it.

In this set of experiments, we randomly select r% mobile devices to support CoAST, where r varies from 50 to 100. The video streaming application is used in the evaluation. Each experiment is repeated 10 times with different random seeds. We report the average value of all the 100 cells.

The experimental results are shown in Figure 44. It's clear that the participation rate has significant impact on the CoAST performance. When r reduces from 100 to 50, the average peak reduction decrease from 27% to 8%. We also observe that the slope of the curve decreases with the increase of the r value. This is because the peak value of the background

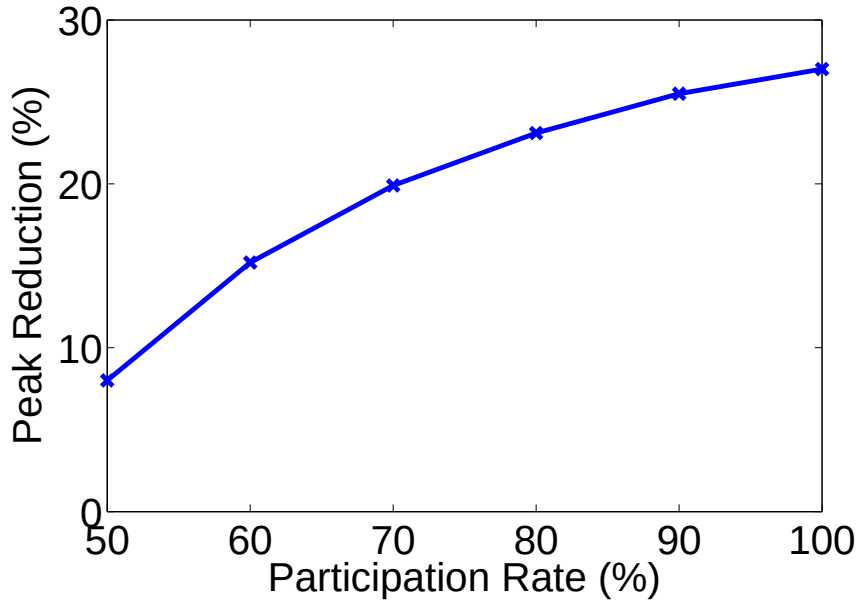


Figure 44: The impact of partial deployment on CoAST performance.

traffic will be higher when the participation rate is low. Thus, according to Theorem 2, the peak reduction achieved by CoAST will be much lower. This set of experiments indicate the importance of increasing the participation rate in CoAST.

To analyze whether partial deployment will impact the user experience of early adopters or non-adopters, we compare the buffering time of streaming applications in the above experiments and that in LTE networks. The buffering time of each stream in both scenarios are the same in all experiments. This is because CoAST tries to satisfy the delay constraints of all adopters. When the network is not very congested, their delay constraints can be easily satisfied. In contrast, when the network is persistently congested which is uncommon in our dataset, adopters behave the same as non-adopters, i.e., downloading as fast as possible without rescheduling their traffic.

Therefore, the partial deployment will primarily impact the peak reduction with little impact on the user experience of mobile applications.

6.7.2.4 The Impact of Design Choices

In this subsection, we evaluate the impact of CoAST parameters and network factors on the performance of CoAST.

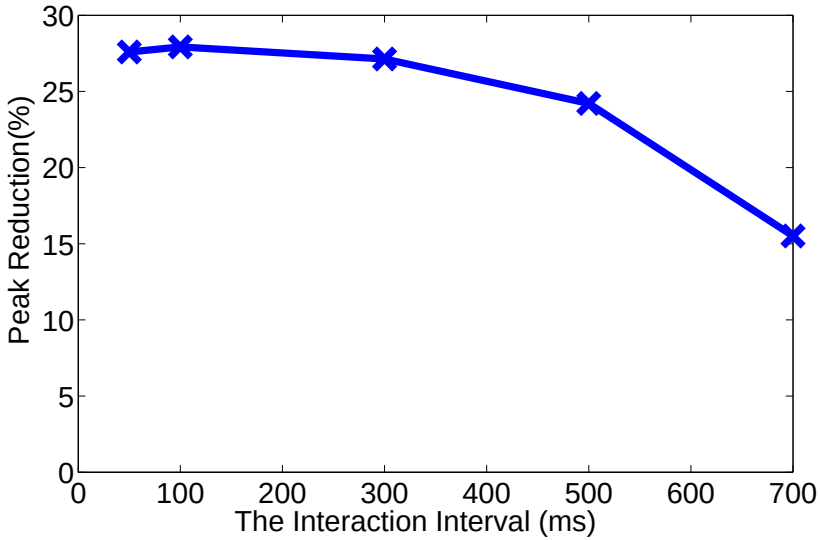


Figure 45: The impact of interaction frequency between UEs and the market proxy.

In the first set of experiments, we analyze the impact of the interaction frequency between the UE and Market proxies. We change the interaction interval (i.e., δ) from 50 ms to 700 ms, while other parameters are kept unchanged. The video streaming application is used in the evaluation. We report the average value of all the cells.

The experimental results are shown in Figure 45. When the interaction interval is less than 100 ms, CoAST achieves similar performance in terms of peak reduction. As the interval increases from 100 ms to 500 ms, the average peak reduction slightly drops from 27% to 24%. When it further increases to 700 ms, the obtained peak reduction is quickly reduced to 15%. This is because the control plane of CoAST utilizes an iterative protocol that gradually optimizes its performance in each iteration. When interaction interval is too large, it is hard for the system to converge to the optimal solution. On the other hand, increase in interaction frequency results in more communication overhead. From our experiments, we find that 100 ms is a good choice as it achieves good tradeoff between performance and overhead.

In the second set of experiments, we analyze the impact of the distance between the UE and market proxies. We vary the RTT between them from 10 ms to 90 ms. The results are reported in Figure 46. We observe that the RTT between UE proxy and market proxy has very small impact on peak reduction. This indicates that CoAST can still achieve

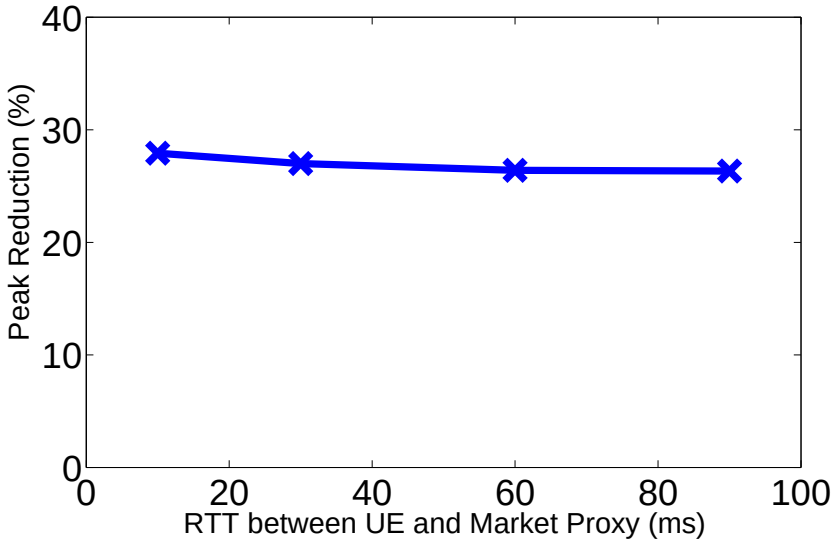


Figure 46: The impact of the delay between UEs and the market proxy. The interaction interval is 100 ms.

good performance even if it is not possible to deploy the market proxy in cellular network elements close to the end user devices, like eNodeB in LTE or NodeB in 3G networks.

6.8 Discussion

In this section, we discuss how the mechanisms proposed by CoAST interact with other existing and proposed mechanisms in the RAN.

3GPP quality of service: The LTE specification provides a QoS model based on Quality Class Indicators (QCI) [11]. A UE may create bearers with one of up to 9 QCI classes, each with a different priority (diffserv), packet delay budget, loss rate, and bitrate guarantee. Many cellular providers today reserve QCI only for managed services (e.g., IMS), and do not expose QCI classes to third party applications. Because CoAST is an over-the-top protocol that can operate without any support from 3GPP infrastructure, it is applicable even on networks which do not expose QCI. Furthermore, QCI provides a static and inflexible partitioning of applications into a small number of priorities. It is not sufficient for scenarios presented by both our examples — streaming and web browsing — in which the *same* application requires different QoS and delay tolerance at different times, depending on the context. Furthermore, any non-collaborative mechanism cannot coordinate behavior across multiple UEs the way that CoAST does.

Congestion aware pricing/control: Access to real-time congestion information at the eNodeBs is not exposed through 3GPP standardized interfaces. Therefore, to remain independent of vendors-specific implementations, CoAST does not assume any access to the eNodeB, including information about whether there is cell congestion or not. Instead, it tries to continuously minimize the peaks across the applications whose traffic is managed through its APIs, independently of other background traffic. If real-time congestion information could be made available, it could easily be used to trigger when CoAST optimization mechanisms kick-in and provide improved fidelity and price control to the market proxy's demand estimation step. We leave this extension to future work.

RNC/eNodeB schedulers: The UMTS RNC or LTE eNodeB have a scheduler that allocates scrambling codes (variable sized slices of the spectrum) to UEs every 2ms based on their demands, QCI, channel noise, and overall cell congestion. Beyond differentiation using QCI, this scheduler is *application context agnostic*. CoAST does not interfere with this scheduler because it operates at a much coarser granularity (hundreds of msec). CoAST adds an additional application aware layer of control on the top, and helps the RAN scheduler by reducing demand peaks themselves, thus reducing the need for the RAN scheduler to allocate less than what UEs demand. However, the RAN scheduler can have an impact on applications that use CoAST because it may restrict the bandwidth available to a UE (because of noise or congestion), and thus decrease the amount of time an application's data can be delayed. To solve this problem, CoAST should take the radio link condition into consideration. We leave it for future work since it requires deeper integration with the eNodeB scheduler.

Energy vs. congestion tradeoffs: Several proposals have been made in the literature to help mobile devices save energy by batching mobile traffic into short concentrated bursts and reducing the time UE radios spend in the active state, e.g., [57, 80]. It would seem that CoAST proposes the opposite philosophy—spread out traffic to minimize congestion. However, in reality, these two mechanisms are relatively complementary. CoAST can just as easily work with applications where data transfers occur in bursts—e.g., web browsing. All CoAST advocates is that when there is contention, priority be given to the traffic on which

users are waiting as opposed to applications that are just filling up their buffers. While this can increase the amount of time needed to transfer an application's background data, as Figure 43 shows, the increase is not substantial. An interesting future use of CoAST is for applications to increase their price based on how much available battery they have, thus prioritizing their transfers over everyone else.

Potential for price manipulation: Because CoAST's mechanisms do not force users to transfer any data after they have indicated their demand forecast, it is possible that malicious users may increase the price others have to pay by falsely forecasting high demands. While it is not easy to detect one-off instances of such behavior (a user's demand may legitimately have changed), it is easy to detect systematic abuse over a period of time by statistically comparing forecasts to actual data transfers using cellular providers existing data tracking mechanisms. Abusers may then have their bids ignored in the future, thus effectively removing them from the set of CoAST managed devices. With incremental deployment, it is possible that some legitimate applications don't support CoAST, resulting in the discrepancy between reported traffic and the real traffic. To solve this problem, the UE proxy needs to report the flow information of those adopters, while the CoAST will only monitor those flows.

Impact of handover: Finally, we briefly describe CoAST's interactions with mobility mechanisms, i.e., handover. UEs detect when a handover takes place by querying their baseband chip for the current cell id. They inform the market proxy for every handover, which then simply assigns the UE's projected demand to the new cell and recomputes the demand for both the old and new cells. Because we expect each market proxy to cover a relatively large area (we expect one market proxy per P-GW), the number of inter-proxy handoffs are few and are handled by the UE simply reconnecting to the new market proxy.

6.9 Summary

In this chapter, we present a new approach to improving the capacity of cellular network cells through application-aware collaborative microscheduling and delaying of traffic. Our implementation, CoAST, supports applications such as streaming and web-browsing that

are not normally considered to be delay tolerant, but yet account for over 70% of cellular network traffic today. Our extensive evaluation demonstrates shows the approach's potential by showing that it can reduce traffic peaks by up to 50%, and increase the capacity of cells to serve such workloads by up to 20% without any degradation to user experience.

CHAPTER VII

SUMMARY OF CONTRIBUTIONS AND FUTURE WORK

This thesis work studies addressing connectivity challenges for mobile computing and communication. The contributions of the thesis work can be summarized as follows:

- **Serendipity.** This thesis work presents the design and implementation of Serendipity system to enable remote computing among intermittently connected mobile devices. This work designs computation models for this purpose. This work designs algorithms for task allocation for various network settings. This work also designs allocation algorithms for energy-aware computing.

- **IC-Cloud.** This thesis work presents the design and implementation of IC-Cloud system to support computation offloading with intermittent connectivity. This work designs mechanisms to handle intermittent connectivity, including connectivity prediction mechanism and risk control mechanism.

- **COSMOS.** This thesis work presents the design and implementation of COSMOS system to provide computation offloading as a service to mobile devices. This work formulates it as an optimization problem whose solution guides the required decision making. This work designs resource-management mechanisms that select resources suitable for computation offloading and adaptively maintain computing resources according to offloading requests and task-allocation algorithms that properly allocate offloading tasks to the cloud resources with limited control overhead.

- **CoAST.** This thesis work presents the design and implementation of CoAST to support collaborative application-aware scheduling of cellular traffic. This work designs a protocol to allow mobile applications and providers to exchange traffic information. This work will design an incentive mechanism to incentivize mobile applications to

collaboratively delay traffic at the right time. This work also designs mechanisms to delay application traffic.

7.1 Future work

The work done in this thesis can be extended in several directions. We describe some of the potential future work below.

- **Computation offloading to cloudlets**: Mobile devices are in intermittent contact with multiple (stationary) cloudlet resources over time. This scenario adds the multiple compute resource dimension to the problem. This environment presents additional concerns beyond the problem of partitioning for remote execution. Among the problems is how to provide for continued execution if a mobile device loses contact with a cloudlet before it completes the processing of an allocated task. Depending on the contact model there is the possibility that the mobile device will meet the same cloudlet again, in which case results can be obtained at this subsequent meeting. Alternatively, we can consider a "hand-off" process where data and computation are migrated among cloudlets over the Internet in anticipation of future contacts with the initiator.

- **Computation offloading in cellular networks**: There are scenarios that mobile devices requires to access computation offloading service through the cellular networks. However, previously proposed systems are not very suitable for cellular networks because of two reasons. First, the communication latency of cellular network is very high. Second, computation offloading will cause extra overhead on the already congested cellular networks. How to make computation offloading beneficial in cellular networks is very important. A potential solution to this problem is to incorporate clouds in the cellular networks and offload the computation-intensive tasks to the in-network cloud. It has three major benefits. First, the latency between end-device to the cloud will be low. Second, the traffic over the backbone network will not be increased by offloading. Third, the cellular networks can increase their revenues by providing computation offloading service.

- **Computation offloading in a hybrid environment**: We envision that the environment in which a user-carried mobile device operates will be a hybrid of the individual scenarios we previously discussed. The goal of our work in general is to allow the device to leverage the combination of resource types available in its environment from which it derives the most benefit while adhering to the constraints of the environment (such as willingness of other devices to contribute resources). There are two types of such environments. The first is where multiple options for remote computation are available contemporaneously. In this case the question is how to optimally use the mixture of available resource types to maximum benefit. The second type is where the environment can change over time. For example the initiator device can encounter a cloudlet for some time and then be in an environment with a number of neighbor user-carried devices some time later. In this case, the environment needs to be monitored and strategies for adaptation of the computation off-loading need to be adopted.

- **Cloud-based storage system**: Nowadays users usually have multiple mobile devices (e.g., smart phone, tablet, laptop) and access similar contents (e.g., music) on different devices in various environments. For example, when a user is traveling, she tends to use smart phone to listen to the music; when she is at home, she probably prefers to play the same music on laptop; when she is to sleep, she may play the music on tablet. A challenging problem in this scenario is how to manage the contents among all these mobile devices. A simple solution is to store the contents in the Cloud (e.g., iCloud, Dropbox) and directly download the required contents to the mobile devices in use at run time. However, this simple method introduces a lot of network traffic, a severe burden on the network infrastructure (especially for cellular networks). A better solution is to build a generic cloud-based storage system that can makes contents easily accessible to users through different devices with low overhead on the network infrastructure. It is composed of mobile devices, personal server (e.g., desktop at home) and cloud storage system. It will store various contents at different storage resources according to user usage habit and the access patterns to various contents.

In the study of the future work, attention should be paid to the development trends of techniques that impact the usefulness and performance of mobile computing and communication.

- **The ratio of cloud power to device power**: Computation offloading relies on the powerful cloud computation resources to speed up the execution of computation-intensive functions. As shown in [88], with the increase of the ratio of the cloud power to device power, computation offloading systems can achieve higher performance. Recent year we have seen the continuous expansion of clouds. Clouds become more and more powerful and easier to be accessed. Meanwhile, although mobile devices are also becoming more and more powerful, they are fundamentally limited by their small sizes. It is highly possible that the ratio of cloud power to device power will continuously increase in future. In addition, with more clouds geographically distributed, the latency between mobile devices and the clouds will be reduced.

- **The battery life time**: One major goal of computation offloading is to reduce the energy consumption of mobile devices. Although the battery power availability continues to improve, it is still very constrained and will continue to be a major bottleneck of mobile devices. Techniques that reduce the energy consumption will remain important to mobile computing. Therefore, more efforts may be required to optimize computation offloading techniques to reduce the energy consumption.

- **The wireless networks**: The wireless networks impact computation offloading in two ways. First, the wireless speed determines the communication cost of computation offloading. Second, the network coverage determines if computation offloading can be used. Although the wireless speed increases slowly, the network coverage (e.g., WiFi and cellular networks) continues increasing. This trend will make computation offloading ubiquitously accessible to mobile devices and achieve more consistent performance at different locations.

- **Future applications**: The development of mobile applications will also determine the usefulness of computation offloading. As more and more sophisticated applications

are required by mobile user for various advanced services, there are a lot of pressure for application developers to improve the application performance. As an important mobile computing technique, computation offloading will play an important role. In addition, as computation offloading becomes available to application developers, they may also consider how to better use this technique to build more powerful applications.

In summary, computation offloading is a very promising and useful technique for mobile computing. Efforts are required to continuously address emerging challenges according to the development trends of related techniques.

REFERENCES

[1] "Amazon ec2 api tools." `http://aws.amazon.com/developertools/351`.

[2] "Android open source project." http://source.android.com.

[3] "Android-x86." `http://www.android-x86.org/`.

[4] "Cisco systems. cisco visual networking index: Forecast and methodology, 2011-2016, may 30 2012." `http://tinyurl.com/VNI2012`.

[5] "Droidfish." `http://web.comhem.se/petero2home/droidfish`.

[6] "Huffington post." `http://www.huffingtonpost.com`.

[7] "ns-3." http://www.nsnam.org/.

[8] "PhantomJS: Headless WebKit with JavaScript API." http://phantomjs.org/.

[9] "Siri." `http://www.apple.com/ios/siri`.

[10] "National laboratory for applied network research. anonymized access logs." `ftp://ftp.ircache.net/Traces/`, 2007.

[11] "3GPP TS 25.214: 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Physical layer procedures (FDD) (Release 8)," 2010.

[12] Alexa, "Top 500 website." http://www.alexa.com/.

[13] Anderson, D. P., "BOINC: A system for public-resource computing and storage," in *IEEE/ACM GRID*, 2004.

[14] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D., "SETI@home: an experiment in public-resource computing," *Commun. ACM*, 2002.

[15] AT&T, "2011 annual report," 2011. http://bit.ly/Lf0goQ.

[16] Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H.-I., "The case for cyber foraging," in *ACM SIGOPS European workshop*, 2002.

[17] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J., "Simplifying cyber foraging for mobile devices," in *ACM MobiSys*, 2007.

[18] Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T., "Tactics-based remote execution for mobile computing," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, (New York, NY, USA), pp. 273–286, ACM, 2003.

[19] Balasubramanian, A., Mahajan, R., and Venkataramani, A., "Augmenting mobile 3g using wifi," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, (New York, NY, USA), pp. 209–222, ACM, 2010.

[20] Balasubramanian, A., Mahajan, R., and Venkataramani, A., "Augmenting mobile 3g using wifi," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, (New York, NY, USA), pp. 209–222, ACM, 2010.

[21] Balasubramanian, N., Balasubramanian, A., and Venkataramani, A., "Energy consumption in mobile phones: a measurement study and implications for network applications," in *ACM IMC*, 2009.

[22] Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., and Pande, V. S., "Folding@home: Lessons from eight years of volunteer distributed computing," in *IEEE IPDPS*, 2009.

[23] Berk, J. and DeMarzo, P., *Corporate finance*. Addison-Wesley, 2007.

[24] Bertsekas, D. and Tsitsiklis, J., *Introduction to probability*, vol. 1. Athena Scientific Nashua, NH, 2002.

[25] Boyd, S., Xiao, L., Mutapcic, A., and Mattingley, J., "Notes on decomposition methods." `http://www.stanford.edu/class/ee364b/notes/decomposition_notes.pdf`.

[26] Burgess, J., Gallagher, B., Jensen, D., and Levine, B. N., "Maxprop: Routing for vehicle-based disruption-tolerant networks," in *IEEE INFOCOM*, 2006.

[27] Buttyán, L. and Hubaux, J.-P., "Enforcing service availability in mobile ad-hoc wans," in *ACM MobiHoc*, 2000.

[28] Carter, H., Mood, B., Traynor, P., and Butler, K., "Secure outsourced garbled circuit evaluation for mobile devices.," in *USENIX Security Symposium*, 2013.

[29] Chang, J.-H. and Tassiulas, L., "Energy conserving routing in wireless ad-hoc networks," in *IEEE INFOCOM*, 2000.

[30] Chekuri, C., Goel, A., Khanna, S., and Kumar, A., "Multi-processor scheduling to minimize flow time with ε resource augmentation," in *ACM STOC*, 2004.

[31] Chun, B., Huang, L., Lee, S., Maniatis, P., and Naik, M., "Mantis: Predicting system performance through program analysis and modeling," *Computing Research Repository (CoRR)*, 2010.

[32] Chun, B., Ihm, S., Maniatis, P., Naik, M., and Patti, A., "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, pp. 301–314, 2011.

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

[34] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[35] Denning, P. J., "Hastily formed networks," *Commun. ACM*, vol. 49, pp. 15–20, April 2006.

[36] Deshpande, P., Hou, X., and Das, S. R., "Performance comparison of 3g and metro-scale wifi for vehicular network access," in *Proc. ACM IMC*, 2010.

[37] Deshpande, P., Kashyap, A., Sung, C., and Das, S. R., "Predictive methods for improved vehicular wifi access," in *Proceedings of the 7th international conference on Mobile systems, applications, and services*, MobiSys '09, (New York, NY, USA), pp. 263–276, ACM, 2009.

[38] Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S., and Estrin, D., "A first look at traffic on smartphones," in *Proceedings of the 10th annual conference on Internet measurement*, pp. 281–287, ACM, 2010.

[39] Fall, K., Iannaccone, G., Kannan, J., Silveira, F., and Taft, N., "A disruption-tolerant architecture for secure and efficient disaster response communications," in *ISCRAM*, 2010.

[40] Flinn, J., "Cyber foraging: Bridging mobile and cloud computing," *Synthesis Lectures on Mobile and Pervasive Computing*, vol. 7, no. 2, pp. 1–103, 2012.

[41] Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., and Patterson, D., "Predicting multiple metrics for queries: Better decisions enabled by machine learning," pp. 592–603, 2009.

[42] Gember, A., Dragga, C., and Akella, A., "Ecos: leveraging software-defined networks to support mobile application offloading," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, (New York, NY, USA), pp. 199–210, ACM, 2012.

[43] Ghobadi, M., Cheng, Y., Jain, A., and Mathis, M., "Trickle: rate limiting youtube video streaming," in *USENIX ATC*, 2012.

[44] Gill, P., Arlitt, M., Li, Z., and Mahanti, A., "Youtube traffic characterization: a view from the edge," in *ACM IMC*, 2007.

[45] Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M., and Chen, X., "Comet: code offload by migrating execution transparently," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 93–106, USENIX Association, 2012.

[46] Green, M., Hohenberger, S., and Waters, B., "Outsourcing the decryption of abe ciphertexts.," in *USENIX Security Symposium*, p. 3, 2011.

[47] Gu, X., Nahrstedt, K., Messer, A., Greenberg, I., and Milojicic, D., "Adaptive offloading inference for delivering applications in pervasive computing environments," in *IEEE PERCOM*, 2003.

[48] Gupta, C., Mehta, A., and Dayal, U., "PQR: Predicting query execution times for autonomous workload management," pp. 13–22, 2008.

[49] Ha, S., Sen, S., Joe-Wong, C., Im, Y., and Chiang, M., "Tube: time-dependent pricing for mobile data," in *ACM SIGCOMM*, 2012.

[50] Ha, S., Sen, S., Joe-Wong, C., Im, Y., and Chiang, M., "Tube: time-dependent pricing for mobile data," in *ACM SIGCOMM*, 2012.

[51] Han, B., Hui, P., Kumar, V. A., Marathe, M. V., Pei, G., and Srinivasan, A., "Cellular traffic offloading through opportunistic communications: a case study," in *Proceedings of the 5th ACM workshop on Challenged networks*, CHANTS '10, (New York, NY, USA), pp. 31–38, ACM, 2010.

[52] Hanna, K. M., Levine, B. N., and Manmatha, R., "Mobile Distributed Information Retrieval For Highly Partitioned Networks," in *IEEE ICNP*, pp. 38–47, Nov 2003.

[53] Hao, S., Li, D., Halfond, W. G., and Govindan, R., "Sif: A selective instrumentation framework for mobile applications," in *ACM MobiSys*, 2013.

[54] Hjelms, E. and Lowb, B. K., "Face detection: A survey," *Elsevier Computer Vision and Image Understanding*, September 2001.

[55] Holma, H. and Toskala, A., *Hsdpa/Hsupa For Umts*. Wiley Online Library, 2006.

[56] Hou, E. S., Ansari, N., and Ren, H., "A genetic algorithm for multiprocessor scheduling," in *IEEE IPDPS*, 1994.

[57] Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "A close examination of performance and power characteristics of 4g lte networks," in *ACM MobiSys*, 2012.

[58] Huggins-Daines, D., Kumar, M., Chan, A., Black, A., Ravishankar, M., and Rudnicky, A., "Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 1, pp. I–I, IEEE, 2006.

[59] Hui, P., Scott, J., Crowcroft, J., and Diot, C., "Haggle: a networking architecture designed around mobile users," in *WONS*, 2006.

[60] Jain, S., Fall, K., and Patra, R., "Routing in a delay tolerant network," *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 145–158, August 2004.

[61] jen Hsu, W., Thrasyvoulos Spyropoulos, a. K. P., and Helmy, A., "Modeling time-variant user mobility in wireless mobile networks," in *INFOCOM*, 2007.

[62] Kamara, S., Mohassel, P., and Riva, B., "Salus: A system for server-aided secure function evaluation," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 797–808, ACM, 2012.

[63] Kingman, J., "The single server queue in heavy traffic," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 57, pp. 902–904, Cambridge Univ Press, 1961.

[64] Kosta, S., Perta, V., Stefa, J., Hui, P., and Mei, A., "Clone2Clone (C2C): Peer to Peer Networking of Smartphones on the Cloud," in *5th USENIX Workshop on Hot Topics in Cloud Computing*, (San Jose, CA), jun 2013.

[65] Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X., "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *IEEE Infocom*, 2012.

[66] Laoutaris, N., Sirivianos, M., Yang, X., and Rodriguez, P., "Inter-datacenter bulk transfers with netstitcher," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 74–85, Aug. 2011.

[67] Laoutaris, N., Smaragdakis, G., Rodriguez, P., and Sundaram, R., "Delay tolerant bulk data transfers on the internet," in *ACM SIGMETRICS*, 2009.

[68] Lee, K.-F., Hon, H.-W., and Reddy, R., "An overview of the SPHINX speech recognition system," *IEEE Transaction on Acoustics, Speech and Signal Processing*, 1990.

[69] Lu, R., Lin, X., Zhu, H., Shen, X., and Preiss, B., "Pi: A practical incentive protocol for delay tolerant networks," *IEEE Transactions on Wireless Communications*, April 2010.

[70] Mahajan, R., Zahorjan, J., and Zill, B., "Understanding wifi-based connectivity from moving vehicles," in *ACM IMC*, 2007.

[71] Maier, G., Schneider, F., and Feldmann, A., "A first look at mobile hand-held device traffic," in *Passive and Active Measurement*, pp. 161–170, Springer, 2010.

[72] Marinelli, E., "Hyrax: Cloud computing on mobile devices using mapreduce," Master's thesis, Computer Science Dept., CMU, September 2009.

[73] Marshall, P., "DARPA progress towards affordable, dense, and content focused tactical edge networks," in *IEEE MILCOM*, 2008.

[74] Nicholson, A. J. and Noble, B. D., "Breadcrumbs: forecasting mobile connectivity," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, MobiCom '08, (New York, NY, USA), pp. 46–57, ACM, 2008.

[75] Peng, C., Tu, G.-h., Li, C.-y., and Lu, S., "Can we pay for what we get in 3g data access?," in *ACM MobiCom*, pp. 113–124, 2012.

[76] Pentland, A. S., Fletcher, R., and Hasson, A., "DakNet: Rethinking connectivity in developing nations," *Computer*, 2004.

[77] Porras, J., Riva, O., and Kristensen, M., "Dynamic resource management and cyber foraging," *Middleware for Network Eccentric and Mobile Applications*, vol. 1, p. 349, 2009.

[78] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O., "Profiling resource usage for mobile applications: a cross-layer approach," in *ACM MobiSys*, 2011.

[79] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O., "Profiling resource usage for mobile applications: A cross-layer approach," in *ACM MobiSys*, 2011.

[80] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Characterizing radio resource allocation for 3g networks," in *ACM IMC*, 2010.

[81] Rhee, I., Shin, M., Hong, S., Lee, K., and Chong, S., "On the levy-walk nature of human mobility," in *INFOCOM*, 2008.

[82] Saha, A. K. and Johnson, D. B., "Modeling mobility for vehicular ad-hoc networks," in *Proceedings of the 1st ACM international workshop on Vehicular ad hoc networks*, VANET '04, 2004.

[83] Sandvine, "Global internet phenomena report." `http://www.sandvine.com/news/global_broadband_trends.asp`, 2012.

[84] Satyanarayanan, M., "Pervasive computing: Vision and challenges," *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, 2001.

[85] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N., "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, 2009.

[86] Schulman, A., Navda, V., Ramjee, R., Spring, N., Deshpande, P., Grunewald, C., Jain, K., and Padmanabhan, V. N., "Bartendr: a practical approach to energy-aware cellular data scheduling," in *ACM MobiCom*, 2010.

[87] Sharma, A., Navda, V., Ramjee, R., Padmanabhan, V. N., and Belding, E. M., "Cool-tether: energy efficient on-the-fly wifi hot-spots using mobile phones," in *ACM CoNEXT*, 2009.

[88] Shi, C., Ammar, M. H., Zegura, E. W., and Naik, M., "Computing in cirrus clouds: the challenge of intermittent connectivity," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, MCC '12, (New York, NY, USA), pp. 23–28, ACM, 2012.

[89] Shi, C., Habak, K., Pandurangan, P., Ammar, M., Naik, M., and Zegura, E., "COSMOS: Computation offloading as a service for mobile devices," under submission.

[90] Shi, C., Joshi, K., Panta, R., Ammar, M., and Zegura, E., "CoAST: Collaborative application-aware scheduling of last-mile cellular traffic," in *Proceedings of the Twelfth International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[91] Shi, C., Lakafosis, V., Ammar, M., and Zegura, E., "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *ACM MobiHoc*, 2012.

[92] Siekkinen, M., Hoque, M. A., Nurminen, J. K., and Aalto, M., "Streaming over 3g and lte: How to save smartphone energy in radio access network-friendly way," in *Proceedings of the 5th Workshop on Mobile Video*, 2013.

[93] Soroush, H., Banerjee, N., Balasubramanian, A., Corner, M. D., Levine, B. N., and Lynn, B., "DOME: A Diverse Outdoor Mobile Testbed," in *ACM HotPlanet*, 2009.

[94] Thain, D., Tannenbaum, T., and Livny, M., "Distributed computing in practice: the condor experience," *Concurr. Comput. : Pract. Exper.*, 2005.

[95] Tournoux, P. U., Leguay, J., Benbadis, F., Conan, V., de Amorim, M. D., and Whitbeck, J., "The accordion phenomenon: Analysis, characterization, and impact on dtn routing," in *Proc. IEEE INFOCOM*, 2009.

[96] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A., "An integrated experimental environment for distributed systems and networks," in *USENIX OSDI*, 2002.

[97] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *IEEE/ACM/IFIP CODES/ISSS*, 2010.

[98] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[99] Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., and Sanders, W., "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers & Security*, vol. 37, pp. 215–227, 2013.