SAPIENZA
UNIVERSITÀ DI ROMA

MOBILITY MODELS, MOBILE CODE OFFLOADING, AND P2P NETWORKS OF SMARTPHONES ON THE CLOUD

by

Sokol Kosta

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the Degree of
DOCTOR OF PHYLOSOPHY in **COMPUTER SCIENCE**
at the
SAPIENZA UNIVERSITY OF ROME

February 2013

Thesis Committee

Prof. Alessandro Mei (First Member)
Department of Computer Science
Sapienza University of Rome, Italy

Prof. Luigi Vincenzo Mancini (Second Member)
Department of Computer Science
Sapienza University of Rome, Italy

Prof. Chiara Petrioli (Third Member)
Department of Computer Science
Sapienza University of Rome, Italy

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Prof. Alessandro Mei Thesis Advisor

Approved for the University Committee on Graduate Studies.

External Reviewers

Prof. Jon Crowcroft
Faculty of Computer Science and Technology
Universiy of Cambridge
Cambridge, England, UK

Prof. Marcelo Dias de Amorim
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
Paris, France

Dedikuar Ledies me shume dashuri.

To Ledia.

Acknowledgements

First, and most of all I want to thank my advisor, Prof. Alessandro Mei, who offered his continuous advice and encouragement throughout the course of this thesis. These three years have been very important for my life, and I consider myself very lucky to have been guided by Prof. Alessandro. I thank him for the systematic guidance and great effort he put into training me in the scientific field. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make research fun for me.

A special thank goes to my friend and collaborator, Julinda Stefa, for always believing in me. She was the one that pushed me in the research area, and I am very happy I followed her advice. It is a pleasure to work with Julinda. She is always eager to get the results and she has brilliant ideas on solving problems. Julinda is the most reliable person I have known and worked with, and she taught me to be the same. Thank you Julku!

A sincere thank-you goes to the members of my internal thesis committee, Prof. Luigi Vincenzo Mancini and Prof.ssa Chiara Petrioli, and to my external reviewers, Prof. Jon Crowcroft, and Prof. Marcelo Dias de Amorim for the patience to go through all my thesis and for the valuable comments and advices.

I want to thank Dr. Pan Hui of T-Labs, Berlin, for the amazing time I had during the visit in Berlin. Pan taught me many things, how to work very hard during the week, and still have fun and drink on the weekend. A big thank-you goes also to the other guys working in Berlin, for their hospitality and their friendship.

I owe a lot to my family, who have always encouraged and have always been supportive during my academic and personal life. A big thank goes to my grandma Kristina who loves me very much, to my parents who are so proud of me, and to my sister Matilda and my brother Malvis for their good wishes.

Three years in a department can be boring or fun. It happened that I made many friends here and I had an amazing time during this journey. A special thanks goes to my friends Julinda Stefa, Blerina Sinaimeri (thank you Shleka), Marco Barbera, Dora Spenza, Claudiu Perta, Ornela Dardha, Alessandro Cammarano, Angelo Capossele, Antonio Davoli, and Andrea Moro.

I want to thank my old and new friends: Denis Kondi, Fjordi Memaj, Oltion Doda, Anila Hoxha, Erinda Bezhani, Klajdi Tako, Artila Vavako, Mamica Burda, Erion Janaqi, Eraldo Gjonca, Erion Nuri, Kledi Memaj, Ilir Beqiri, Taulant Mellaraj, Blerta Lipo, Iva Qesja, Adelona Salaj, Arsela Prelaj, Dorjan Kosova, Emiliano Pigini, Klajda Deliu, Davide Sammartino, Aida Hoxha, Diego Villecco, Irma Gjini, Ervin Mertiri, Blerina Zeza, and many others for the amazing experiences we had these years.

Finally, I want to thank my love, Enkeleda Kertalli, for always being there for me, deadline after deadline. She is the most special person I've ever known, and I can consider myself the luckiest and happy person for being with her. Ledia has been supportive during difficult times, trying to cheer me up and raise my confidence. She is the perfect companion to study with, to walk with, to talk with, to travel with, and to live with.
Thank you for making my life so beautiful!

Contents

Introduction

It was just a few years ago when I bought my first smartphone. And now, (almost) all of my friends possess at least one of these powerful devices. International Data Corporation (IDC) reports that smartphone sales showed strong growth worldwide in 2011, with 491.4 million units sold – up to 61.3 percent from 2010 [1]. Furthermore, IDC predicts that 686 million smartphones will be sold in 2012, 38.4 percent of all handsets shipped [2]. Silently, we are becoming part of a big mobile smartphone network, and it is amazing how the perception of the world is changing thanks to these small devices. If many years ago the birth of Internet enabled the possibility to be online, smartphones nowadays allow to be online *all the time*. Today we use smartphones to do many of the tasks we used to do on desktops, and many new ones. We browse the Internet, watch videos, upload data on social networks, use online banking, find our way by using GPS and online maps, and communicate in revolutionary ways. Along with these benefits, these fancy and exciting devices brought many challenges to the research area of mobile and distributed systems.

One of the first problems that captured our attention was the study of the network that potentially could be created by interconnecting all the smartphones together. Typically, these devices are able to communicate with each other in short distances by using communication technologies such as Bluetooth or WiFi. The network paradigm that rises from this intermittent communication, also known as Pocket Switched Network (PSN) or Opportunistic Network ([10, 11]), is seen as a key technology to provide innovative services to the users without the need of any fixed infrastructure. In PSNs nodes are short range communicating devices carried by humans. Wireless communication links are created and dropped

[1] http://www.idc.com/getdoc.jsp?containerId=prUS23299912
[2] http://www.idc.com/getdoc.jsp?containerId=prUS23523812

in time, depending on the physical distance of the device holders. From one side, social relations among humans yield recurrent movement patterns that help researchers design and build protocols that efficiently deliver messages to destinations ([12, 13, 14] among others). The complexity of these social relations, from the other side, makes it difficult to build simple mobility models, that in an efficient way, generate large synthetic mobility traces that look real. Traces that would be very valuable in protocol validation and that would replace the limited experimentally gathered data available so far. Traces that would serve as a common benchmark to researchers worldwide on which to validate existing and yet to be designed protocols.

With this in mind we start our study with re-designing SWIM [15], an already existing mobility model shown to generate traces with similar properties of that of existing real ones. We make SWIM able to easily generate large (small)-scale scenarios, starting from known small (large)-scale ones. To the best of our knowledge, this is the first such study. In addition, we study the social aspects of SWIM-generated traces. We show how to SWIM-generate a scenario in which a specific community structure of nodes is required. Finally, exploiting the scaling properties of SWIM, we present the first analysis of the scaling capabilities of several forwarding protocols such as Epidemic [16], Delegation [13], Spray&Wait [14], and BUBBLE [12]. The first results of these works appeared in [1], and, at the time of writing, [2] is accepted with minor revision.

Next, we take into account the fact that in PSNs cannot be assumed full cooperation and fairness among nodes. Selfish behavior of individuals has to be considered, since it is an inherent aspect of humans, the device holders (see [17], [18]). We design a market-based mathematical framework that enables heterogeneous mobile users in an opportunistic mobile network to compromise optimally and efficiently on their QoS [3] demands. The goal of the framework is to satisfy each user with its achieved (lesser) QoS, and at the same time maximize the social welfare of users in the network. We base our study on the consideration that, in practice, users are generally tolerant on accepting lesser QoS guarantees than what they demand, with the degree of tolerance varying from user to user. This study is described in details in Chapter 2 of this dissertation, and is included in [3].

[3] In general, QoS could be parameters such as response time, number of computations per unit time, allocated bandwidth, etc.

Along the way toward our study of the smartphone-world, there was the big advent of mobile cloud computing—smartphones getting help from cloud-enabled services. Many researchers started believing that the cloud could help solving a crucial problem regarding smartphones: improve battery life. New generation apps are becoming very complex — gaming, navigation, video editing, augmented reality, speech recognition, etc., — which require considerable amount of power and energy, and as a result, smartphones suffer short battery lifetime. Unfortunately, as a consequence, mobile users have to continually upgrade their hardware to keep pace with increasing performance requirements but still experience battery problems. Many recent works have focused on building frameworks that enable mobile computation offloading to software clones of smartphones on the cloud (see [19, 20] among others), as well as to backup systems for data and applications stored in our devices [21, 22, 23]. However, none of these address dynamic and scalability features of execution on the cloud. These are very important problems, since users may request different computational power or backup space based on their workload and deadline for tasks.

Considering this and advancing on previous works, we design, build, and implement the ThinkAir framework, which focuses on the elasticity and scalability of the server side and enhances the power of mobile cloud computing by parallelizing method execution using multiple Virtual Machine (VM) images. We evaluate the system using a range of benchmarks starting from simple micro-benchmarks to more complex applications. First, we show that the execution time and energy consumption decrease *two* orders of magnitude for the *N*-queens puzzle and *one* order of magnitude for a face detection and a virus scan application, using cloud offloading. We then show that a parallelizable application can invoke multiple VMs to execute in the cloud in a seamless and on-demand manner such as to achieve greater reduction on execution time and energy consumption. Finally, we use a memory-hungry image combiner tool to demonstrate that applications can dynamically request VMs with more computational power in order to meet their computational requirements. The details of the ThinkAir framework and its evaluation are described in Chapter 4, and are included in [6, 5].

Later on, we push the smartphone-cloud paradigm to a further level: We develop Clone2Clone (C2C), a distributed platform for cloud clones of smartphones. Along the

way toward C2C, we study the performance of device-clones hosted in various virtualization environments in both private (local servers) and public (Amazon EC2) clouds. We build the first Amazon Customized Image (AMI) for Android-OS—a key tool to get reliable performance measures of mobile cloud systems—and show how it boosts up performance of Android images on the Amazon cloud service. We then design, build, and implement Clone2Clone, which associates a software clone on the cloud to every smartphone and interconnects the clones in a p2p fashion exploiting the networking service within the cloud. On top of C2C we build CloneDoc, a secure real-time collaboration system for smartphone users. We measure the performance of CloneDoc on a testbed of 16 Android smartphones and clones hosted on both private and public cloud services and show that C2C makes it possible to implement distributed execution of advanced p2p services in a network of mobile smartphones. The design and implementation of the Clone2Clone platform is included in [7], recently submitted to an international conference.

We believe that Clone2Clone not only enables the execution of p2p applications in a network of smartphones, but it can also serve as a tool to solve critical security problems. In particular, we consider the problem of computing an efficient patching strategy to stop worm spreading between smartphones. We assume that the worm infects the devices and spreads by using bluetooth connections, emails, or any other form of communication used by the smartphones. The C2C network is used to compute the best strategy to patch the smartphones in such a way that the number of devices to patch is low (to reduce the load on the cellular infrastructure) and that the worm is stopped quickly. We consider two well defined worms, one spreading between the devices and one attacking the cloud before moving to the real smartphones. We describe CloudShield [8], a suite of protocols running on the peer-to-peer network of clones; and show by experiments with two different datasets (Facebook and LiveJournal) that CloudShield outperforms state-of-the-art worm-containment mechanisms for mobile wireless networks. This work is done in collaboration with Marco Valerio Barbera, PhD colleague in the same department, who contributed mainly in the implementation and testing of the malware spreading and patching strategies on the different datasets.

The communication between the real devices and the cloud, necessary for mobile computation offloading and smartphone data backup, does certainly not come for free. To

the best of our knowledge, none of the works related to mobile cloud computing explicitly studies the actual overhead in terms of bandwidth and energy to achieve full backup of both data/applications of a smartphone, as well as to keep, on the cloud, up-to-date clones of smartphones for mobile computation offload purposes. In the last work during my PhD—again, in collaboration with Marco Valerio Barbera—we studied the feasibility of both mobile computation offloading and mobile software/data backup in real-life scenarios. This joint work resulted in a recent publication [9] but is not included in this thesis. As in C2C, we assume an architecture where each real device is associated to a software clone on the cloud. We define two types of clones: The *off-clone*, whose purpose is to support computation offloading, and the *back-clone*, which comes to use when a restore of user's data and apps is needed. We measure the bandwidth and energy consumption incurred in the real device as a result of the synchronization with the off-clone or the back-clone. The evaluation is performed through an experiment with 11 Android smartphones and an equal number of clones running on Amazon EC2. We study the data communication overhead that is necessary to achieve different levels of synchronization (once every 5min, 30min, 1h, etc.) between devices and clones in both the off-clone and back-clone case, and report on the costs in terms of energy incurred by each of these synchronization frequencies as well as by the respective communication overhead. My contribution in this work is focused mainly on the experimental setup, deployment, and data collection.

Chapter 1

SWIM

Pocket Switched Networks (PSN), networks of mobile humans carrying short-range communication devices such as smartphones, PDAs, or lap-tops, have received significant attention from the research community during the last few years. The complexity of these networks derives mostly from the difficulty of predicting human mobility. Much research has been dedicated to the study of real life experimental data traces [24, 10, 11, 25, 26, 27] so as to compute statistical properties of human mobility and therefore of PSNs. These works have mostly focused on inter-contacts (time intervals between two consecutive contacts of the same couple of nodes), contact-duration, and contact number distributions among node pairs, and have confirmed the complexity and the unpredictability of human mobility. Another large flow of works have been dedicated to uncovering structural properties of PSNs such as the presence of social-based community sub-structures [28, 12, 26] and to using these properties to design efficient message forwarding [12, 29]. Additionally, in [30] the authors discuss on the limits of experiments based on logging contacts and show how to infer plausible mobility patterns from them.

Also have a large number of works been presented on designing models for human mobility [31, 32, 33, 34, 35, 36, 37, 15, 38]. Most of these works validate their models with real life data traces available on-line and unfortunately not very large.

In this chapter we extend small world in motion (SWIM [15, 1]), an existing mobility model that generates small worlds of mobile humans. The authors in [15] show that the

model is able to generate traces with similar statistical properties (distribution of inter-contact times, contact number and contact durations among couples) of that of real traces. Furthermore, it is proven mathematically that the SWIM-generated traces show the power-law exponential dichotomy of inter-contact times that has been observed in the real-life experiments. SWIM is very simple to implement, is easily tuned by setting just a few parameters, and very efficient in simulations. The mobility pattern of the nodes is based on a simple intuition on human mobility: People go more often to places not very far from their home and where they can meet a lot of other people. As the first extension to the model in this chapter, we show that by implementing this simple rule, SWIM is able to raise social behavior among nodes, a fundamental ingredient of human mobility in real life. Then, we validate the model using four different real traces and compare the distributions of inter-contact times, contact durations and number of contacts between nodes, showing that synthetic data that SWIM generates match very well each of the four real scenarios simulated.

The new features of SWIM introduced in this chapter are as follows:

- generate traces with the same social community structure to well-known, small-scale experimental traces;
- validate correctly sophisticated protocols based on the social structure of the network such as BUBBLE [12] (as well as Delegation [13], Epidemic [16], and Spray&Wait [14]);
- easily generate large (small)-scale scenarios, starting from known small (large)-scale ones.

This last feature of SWIM allows us to address the fundamental problem of generating *large scale* synthetic social mobile networks that can be used to assess the performance of forwarding protocols. We SWIM-generate larger versions of well-known real life experiments on human mobility in two different ways—larger number of nodes and same network area (the *Manhattan model*), and larger number of nodes and same density (the *Phoenix model*)—and then use these traces to validate the aforementioned forwarding protocols. SWIM is able to extrapolate key properties of human mobility and can be used to

understand how protocols scale to larger and larger networks. To the best of our knowledge, this is the first mobility model that addresses this issue and this is the first work that can show reliable performance evaluation of well-known forwarding protocols on large scale networks.

The rest of the chapter is organized as follows: Section 1.1 briefly reports on current work in the field; in Section 1.2 we discuss the fundamental requirements of a good mobility model; in Sections 1.2.1–1.2.3 we describe the way SWIM operates and mathematically prove the presence of exponentially distributed tail of the inter-contact times in SWIM, whereas Section 1.2.5 describes the methodology used to make it able to scale up. In Section 1.3 we show experimentally the good matching between statistical properties of SWIM and the real-traces, present the experiments related to the enlarged SWIM-scaling scenarios, and show how remarkably similarly Epidemic [16], Delegation [13], Spray&Wait [14], and BUBBLE [12] perform on both the real and synthetic SWIM-generated traces. In Section 1.4 we show for the first time how these protocols perform on the enlarged scenarios, and give insights on their scaling properties. Section 1.5 shows how to customize SWIM to generate networks with known community sub-structure. We lastly conclude with Section 1.6.

The results presented in this chapter appear in [15, 1, 2].

1.1 Modeling Human Mobility

The problem of designing a mobility model for human mobility is felt as an important one in the community and in the literature. In the last few years there have been a considerable number of papers on this topic. The work in [39] is one of the first to argue heterogeneous movement of nodes and to present a mobility model where nodes target a few concentration destination points in the area.

More recently, the model presented in [33] generates movement traces using a model which is similar to a random walk, except that the flight lengths and the pause times in destinations are generated based on Levy Walks—with power law distribution. In the past, Levy Walks have been shown to approximate well the movements of animals. The model produces inter-contact time distributions similar to real world traces. However, since every

node moves independently, the model does not generate any social structure in the network. In [31, 32], the authors present a mobility model based on social network theory which takes in input a social network and discuss the community patterns and groups distribution in geographical terms. They validate their synthetic data with real traces and show a good matching.

The work in [34] presents a new mobility model for clustered networks. Moreover, a closed-form expression for the stationary distribution of node position is given. The model captures the phenomenon of emerging clusters, observed in real partitioned networks, and correlation between the spatial speed distribution and the cluster formation. In [35], the authors present a mobility model that simulates the everyday life of people that go to their workplaces in the morning, spend their day at work and go back to their homes at evenings. Each one of these scenarios is a simulation per se. The synthetic data they generate match well the distribution of inter-contact time and contact durations of real traces. In [36] the authors proposed the SLAW mobility model, which is a modification of the Levy-walk based model, where the human waypoints are modeled as fractals. The model matches well the inter-contact times distribution of the real traces, and predicts quite accurately performance of simple forwarding protocols. Yet, no results are presented in terms of contact duration and contact number distributions and in the structure in communities of the resulting network, and the model seems to be hard to be used in theoretical analysis.

The work of Barabasi et al. [40] studies the trajectory of a very large (100,000) number of anonymized mobile phone users whose position is tracked for a six-months period. They observe that human trajectories show a high degree of temporal and spatial regularity, each individual being characterized by a time independent characteristic travel distance and a significant probability to return to a few highly frequented locations. They also show that the probability density function of individual travel distances are heavy tailed and also are different for different groups of users and similar inside each group. Furthermore, they plot the frequency of visiting different locations and show that it is well approximated by a power law. All these observations are in contrast with the random trajectories predicted by Levy flight and random walk models, and support the intuition behind SWIM. Also the authors of [37] are inspired by the work of Barabasi et al. They point out the following three rules of human mobility: a) Nodes move more frequently and visit more locations if

they have many friends; b) users tend to visit a few locations where they spend the majority of their time; c) users prefer shorter paths to longer ones. With these rules in mind, they propose HCMM, an improvement of their previous work in [31, 32]. They also include evaluation of temporal properties, in terms of inter-contact times, of the traces generated by their model. In [41] the authors propose a mobility model that aims to reproduce real world mobility traces, trying to capture group movements present in real life mobility. The model is validated against real-world traces of vehicular networks, and the performance of the ADV and DSDV routing protocols is compared on both real and synthetic traces.

More recent works such as [42, 43] present other models for human mobility that are simple, and match well statistical properties of traces. However, these models have not been shown nor to generate community sub-structure such as those of real scenarios, neither to accurately validate protocols. Lastly, to the best of our knowledge no mobility model has been shown to have the capability to scale to larger scenarios in a consistent way.

1.2 Small World in Motion

The complexity of inter-personal relationships and the multitude of hobbies/interests that people have in a life that becomes more and more hectic make human mobility all but easy to model. In our vision, a model should be simple, easy to implement, and able to extrapolate key properties of human mobility. We can't underestimate the importance of having a *simple* model. A simple model is easier to understand, can be useful to distill the fundamental ingredients of human mobility, can be easier to implement, easier to tune (just one or few parameters), and can be useful to support theoretical work. We are also looking for a model that generates traces with the same statistical properties that real traces have. Statistical distribution of inter-contact time and number of contacts, among others, are useful to characterize the behavior of a mobile network. A model that generates traces with statistical properties that are far from those of real traces is probably useless. Simultaneously, a good model should also be able to generate similar social behavior among nodes to that of real-life. However, it is important to keep in mind that matching statistical properties is not our final goal. It can even be misleading—if in the quest for matching a large number of statistical indicators we design a complicated model that is hard to use and understand, we

are not doing a good job. It is much more important that the model is accurate in predicting the performance of network protocols on real networks. If a protocol performs well (or bad) in the model, it should also perform well (or bad) in the real network. As accurately as possible.

Lastly, we're looking for a model that, starting from a small (large) well-known scenario, can generate *large (small) scale* versions of it. A model that we can trust and use to assess the performance of forwarding protocols on networks whose size far exceeds (or is way below) the size of any available real experiment.

None of the mobility models in the literature meets *all* of these properties. The random way-point mobility model is simple, but its traces do not look real. Some of the other protocols we reviewed in the related work section can indeed produce traces that has good statistical properties, at least with respect to some of the statistics, but are far from being simple. And, as far as we know, no model has been shown to predict real world performance of community based protocols accurately, and no model has been validated on larger scenarios (larger than known real traces) in a consistent way.

1.2.1 The Intuition

According to studies by the Temple University, Phi, USA[1], the 5 topmost factors that impact peoples' choice when reallocating are safety, costs, good (high level) schools, convenience to shopping, proximity to work, proximity to family. While it is difficult to re-interpret safety and costs in terms of a mobility model where simplicity is the main requirement, the other factors suggest that people do consider proximity and popularity (high level of schools, good shopping, for example) when making decisions about mobility. People tradeoff these two basic elements in everyday mobility as well—the best supermarket/school or the most popular restaurant that are also not far from home, for example. It is unlikely (though not impossible) that we go to a location that is far from our place and that is not so popular, or interesting. Not only that, usually there are just a few places where a person spends a long period of time (for example home and work office or school), whereas there are lots of places where she stays less, like for example post office,

[1] http://americashometown.blogspot.it/2005/12/why-people-choose-to-live-where-they.html

bank, cafeteria, etc. So, supported by the studies in [33, 44], we expect that the wait-time follows a bounded power-law distribution. These are the two basic intuitions SWIM is built upon. Of course, trade-offs humans face in their everyday life are usually much more complicated, and there are plenty of unknown factors that influence mobility. However, we will see that simple rules—trading-off proximity and popularity, and distribution of waiting time—are enough to get a mobility model with a number of desirable properties and an excellent capability of predicting the performance of forwarding protocols. These simple rules our model is based upon are enough to make typical properties of real traces emerge, just naturally.

1.2.2 The Model in Details

In SWIM, to each node is assigned a so called *home*—a randomly and uniformly chosen point over the network area. The domain is continuous, so we divide the network area into many small contiguous squared cells that represent possible destinations. The size of the cells depends on the transmitting range r of the nodes–the cell diagonal equals r; this way, nodes that are in the same cell at the same time are able to communicate. Each node can thus easily build a map of the network area. That said, every node independently assigns to every destination cell a *weight* that grows with the popularity of the place and decreases with the distance from the node's home. The node chooses its destination cell randomly and proportionally with its weight. The exact destination point (remind that the network area is continuous) is taken uniformly at random over the cell's area.

More specifically, let A be one of the nodes and h_A its home. Let C be one of the possible destination cells. We denote with $seen(C)$ the number of nodes that node A encountered in C the last time it reached C. This number is 0 at the beginning of the simulation and it is updated each time node A reaches a destination in cell C. The weight that node A assigns to cell C is as follows:

$$w(C) = \alpha \cdot distance(h_A, C) + (1 - \alpha) \cdot seen(A, C). \tag{1.2.1}$$

Constant $\alpha \in [0;1]$ tradeoffs distance from home and popularity. The larger α, the more a node will tend to go to places near its home and to meet neighbors. Conversely, the

smaller α, the more a node will tend to go to "popular" places and to meet large "crowds of nodes". Of course, there is no "correct" scenario. Both are correct, they simply model different social structures.

Let h_A, x, and x_j be respectively node's A home-point, and the center of cells C and C_j. Let also r be the nodes' radius and d be the nodes' density in the network area (computed as a function of r and the total number of nodes). The *seen* and the *distance* functions of Equation 1.2.1 are defined as follows:

$$seen(A,C) = \frac{1 + \frac{1}{d} TSeen(A,C)}{max_j\{1 + \frac{1}{d} TSeen(A,C_j)\}} \tag{1.2.2}$$

where $TSeen(A,C)$ and $TSeen(A,C_j)$ denote the number of nodes A has encountered during all its visits respectively in C and C_j, and,

$$distance(A,C) = \frac{\frac{1}{(1+\frac{1}{r}||h_A - x||)^2}}{max_j\{\frac{1}{(1+\frac{1}{r}||h_A - x_j||)^2}\}} \tag{1.2.3}$$

As can be noticed from Equation 1.2.2, node density plays a crucial role in a given cell's popularity. Indeed, a given density value has the same impact on popularity, regardless of network area. As well, the *seen* function that we propose depends on the total number of encounters a node has seen during all the visits in a cell. This tend to build a stable mobility pattern over time: After an initial set-up period, nodes tend to belong to a static set of communities.

The $distance(A,C)$ function (Equation 1.2.3) depends on the communication range r of the nodes. The model is built so that r determines the number of possible cells. Thus it directly impacts the network area map for nodes. It is easy to see that the *distance* function of Equation 1.2.3 *does* scale with the scaling of network area.

After a destination is chosen, a node moves towards it following a straight line and with a constant speed that is proportional to the distance between the starting point and the destination. In particular, that means that nodes finish each leg of their movements in constant time. This can seem quite an oversimplification, however, it is useful and also not far from reality. Useful to simplify the model; not far from reality since we are used to

move slowly (maybe walking) when the destination is nearby, faster when it is farther, and extremely fast (maybe by car) when the destination is far-off. When reaching destination the node decides how long to remain there by using a bounded (also known as truncated) power law. As discussed above, this is a key observation coming from real experiments.

1.2.3 Power Law and Exponential Decay Dichotomy

In a recent work [25], it is observed that the distribution of inter-contact time in real life experiments shows a so called dichotomy: First a power law until a certain point in time, then an exponential cut-off. In [27], the authors suggest that the cut-off is due to the bounded domain where nodes move. In SWIM, inter-contact time distribution shows exactly the same dichotomy. More than that, our experiments show that, if the model is properly tuned, the distribution is strikingly similar to that of real life experiments.

We show here, with a mathematically rigorous proof, that the distribution of inter-contact time of nodes in SWIM has an exponential tail (cut-off). Later, we will see experimentally that the same distribution has indeed a head distributed as a power law. Note that the proof has to cope with a difficulty due to the social nature of SWIM—every decision taken in SWIM by a node *does not* depend only on its own previous decisions, but also on other nodes' decisions. Where a node goes affects where it will choose to go in the future, and it also affect where other nodes will chose to go in the future. So, in SWIM there are no renewal intervals and nodes never "forget" their past.

In the following, we will consider two nodes A and B. Let $A(t)$, $t \geq 0$, be the position of node A at time t. Similarly, $B(t)$ is the position of node B at time t. We assume that at time 0 the two nodes are leaving visibility after meeting. That is, $||A(0) - B(0)|| = r$, $||A(t) - B(t)|| < r$ for $t \in 0^-$, and $||A(t) - B(t)|| > r$ for $t \in 0^+$. Here, $||\cdot||$ is the euclidean distance in the square. The inter-contact time of nodes A and B is defined as:

$$T_I = \inf_{t>0}\{t : ||A(t) - B(t)|| \leq r\}$$

Observation 1.2.1. *For all nodes A and for all cells C, the distance function distance*(A,C) *returns at least* $\mu > 0$.

Theorem 1.2.1. *If* $\alpha > 0$, the tail *of the inter-contact time distribution between nodes A*

and B in SWIM has an exponential decay.

Proof. To prove the presence of the exponential cut-off, we will show that there exists constant $c > 0$ such that

$$\mathbb{P}\{T_I > t\} \leq e^{-ct}$$

for all sufficiently large t. Let $t_i = i\lambda$, $i = 1, 2, \ldots$, be a sequence of times. Constant λ is large enough that each node has to make a way point decision in the interval between t_i and t_{i+1} and that each node has enough time to finish a leg. Recall that this is of course possible since waiting time at way points is bounded above and since nodes complete each leg of movement in constant time. The idea is to take snapshots of nodes A and B and see whether they see each other at each snapshot. However, in the following, we also need that at least one of the two nodes is not moving at each snapshot. So, let

$$\delta_i = \min\{\delta \geq 0 : \text{either } A \text{ or } B \text{ is at a way point at time } t_i + \delta\}.$$

Clearly, $t_i + \delta_i < t_{i+1}$, for all $i = 1, 2, \ldots$.

We take the sequence of snapshots $\{t_i + \delta_i\}_{i>0}$. Let $\varepsilon_i = \{||A(t_i + \delta_i) - B(t_i + \delta_i)|| > r\}$ be the event that nodes A and B are not in visibility range at time $t_i + \delta_i$. We have that

$$\mathbb{P}\{T_I > t\} \leq \mathbb{P}\left\{\bigcap_{i=1}^{\lfloor t/\lambda \rfloor - 1} \varepsilon_i\right\} = \prod_{i=1}^{\lfloor t/\lambda \rfloor - 1} \mathbb{P}\{\varepsilon_i | \varepsilon_{i-1} \cdots \varepsilon_1\}.$$

Consider $\mathbb{P}\{\varepsilon_i | \varepsilon_{i-1} \cdots \varepsilon_1\}$. At time $t_i + \delta_i$, at least one of the two nodes is at a way point, by definition of δ_i. Say node A, without loss of generality. Assume that node B is in cell C (either moving or at a way point). During its last way point decision, node A has chosen cell C as its next way point with probability at least $\alpha\mu > 0$, thanks to Observation 1.2.1. If this is the case, the two nodes A and B are now in visibility. Note that the decision has been made after the previous snapshot, and that it is not independent of previous decisions taken by node A, and it is not even independent of previous decisions taken by node B (since the social nature of decisions in SWIM). Nonetheless, with probability at least $\alpha\mu$ the two

nodes are now in visibility. Therefore,

$$\mathbb{P}\{\varepsilon_i|\varepsilon_{i-1}\cdots\varepsilon_1\} \leq 1-\alpha\mu.$$

So,

$$\begin{aligned}\mathbb{P}\{T_I > t\} &\leq \mathbb{P}\left\{\bigcap_{i=1}^{\lfloor t/\lambda\rfloor-1}\varepsilon_i\right\} = \prod_{i=1}^{\lfloor t/\lambda\rfloor-1}\mathbb{P}\{\varepsilon_i|\varepsilon_{i-1}\cdots\varepsilon_1\} \\ &\leq (1-\alpha\mu)^{\lfloor t/\lambda\rfloor-1} \sim e^{-ct},\end{aligned}$$

for sufficiently large t. □

1.2.4 The Simulation Environment

In order to evaluate SWIM, we built a discrete event simulator of the model (see the website for SWIM [45]). The simulator takes as input:

- n: the number of nodes in the network;
- r: the transmitting radius of the nodes;
- the simulation time in seconds;
- coefficient α that appears in Equation 1.2.1;
- the distribution of the waiting time at destination.

The output of the simulator is a text file containing records on each main event occurrence. The main events of the system and the related outputs are:

- *Meet* event: When two nodes are in range with each other. The output line contains the ids of the two nodes involved and the time of occurrence.
- *Depart* event: When two nodes that were in range of each other are not anymore. The output line contains the ids of the two nodes involved and the time of occurrence.
- *Start* event: When a node leaves its current location and starts moving towards destination. The output line contains the id of the location, the id of the node and the time of occurrence.

- *Finish* event: When a node reaches its destination. The output line contains the id of the destination, the id of the node and the time of occurrence.

During the simulation each node A keeps a vector $TSeen(A,C)$ updated. So, in every moment A is able to compute the value $seen(A,C_i)$ for all the cells C_i. Note that the nodes do not necessarily agree on what is the popularity of each cell. Indeed, usually they don't since nodes visit cells at different times. As mentioned earlier, it is not necessary to keep in memory the whole vector, without changing the qualitative behavior of the mobile system. However, the four real scenarios we will consider next are not large enough to cause any real memory problem. Vector $TSeen(A,C)$ is updated at each *Finish* and *Start* event, and is not changed during movements.

Lastly, note that people do not trade-off proximity for popularity in the same way. Take a salesman for example—he moves frequently from one town to another, or from one building to another. Surely, he has a different mobility pattern compared to a high-school teacher, that tends to move in a more repetitive way. SWIM is able to simulate these scenarios too, simply by setting different α values to different nodes. Nonetheless, the scenarios we simulate in this work involve only people with similar jobs/interests (students or conference attendees), so here we set α to be the same for all nodes.

1.2.5 Generating large scenarios with SWIM

Obtaining large and trustworthy synthetic mobility traces is both important and challenging. It is important in order to assess networking protocols on data sets larger than those available today and thus check their scalability; it is challenging since it is not clear how a large mobility trace should look like by looking just at the few available and small real world data sets. Here we propose a methodology. To the best of our knowledge this is the first work where this issue is considered.

To generate mobility traces with SWIM, we choose the parameters and let the model generate traces as long as we need. In the literature, it is customary to choose the parameters in such a way that the mobility pattern is similar, in some precise statistical sense, to a real data set. For example, the data set collected during the Infocom conference in 2006 (in the experimental section of this chapter, we show how to do it for this real data set among

others). In this way, we can build a model that looks like Infocom 2006 with $n = 78$ nodes and density ρ (tuned with a large set of experiments). This is already a very useful thing to do, we are now able to generate traces that are much longer than the three days of the conference in a sound way.

Here we consider the problem of generating traces for the same scenario in which the number of nodes is $N > n$. If the basic assumption of SWIM is correct (people trade-off popularity of places and vicinity with a parameter α), it is enough to replace the number of nodes n in the original model with N. We can also assume that transmission range does not change with the number of the nodes. The only issue, which is not obvious indeed, is how density $\rho(N)$ changes as N grows and, consequently, how the area of the network changes as N grows.

Actually, it is impossible to give an answer to this problem. It is like predicting the future growth of a mobile community. Nonetheless, it seems reasonable to bound the possible future of a growing community by using two extremes that we define in this work: The Phoenix model and the Manhattan model. In the *Phoenix model*, $\rho_P(N) = \rho$ for all $N > n$ (recall that ρ is the density that has experimentally been shown to be appropriate for the scenario when the number of nodes is n). Speaking in metaphorical terms, this is the case when a town grows in size without creating denser agglomerates and just covering a larger geographical area. In the *Manhattan model*, $\rho_M(N) = N\rho/n$ for all $N > n$. In this model, as the network grows more people populate the same geographical area. The place is just much more crowded, and that means that every node meet many more other nodes in the same unit of time and that people mix more (it is more common to meet people that are not in your circle of friends).

When assessing the performance of networking protocols, a fundamental property to check is scalability. This is one of the contributions of this work, showing how some of the protocols that are the state of the art perform on large networks—larger than any real data. Thanks to SWIM, we are able to show the performance under the Phoenix and the Manhattan models. If a protocol shows good performance in larger and larger networks under both models, then we can have some confidence that the model has good scalability. Clearly, a more comprehensive experiment can consider a class of density functions ρ such that $\rho_P(N) \leq \rho(N) \leq \rho_M(N)$ and thus understand under what conditions of scalability the

protocol has good performance.

1.3 Experimental Results

In order to show the accuracy of SWIM in simulating real life scenarios, we will compare SWIM with four traces gathered during experiments done with real devices carried by people. We will refer to the real traces as *Cambridge*, *Infocom 05*, *Infocom 06*, and *Dartmouth*. Characteristics of these data sets such as inter-contact times and contact distribution have been observed in several previous works [10, 46, 11].

- In *Cambridge* [12, 47] the authors used Intel iMotes to collect the data. The iMotes were distributed to two groups of students (Year1 and Year2) of the University of Cambridge and were programmed to log contacts of all visible mobile devices. Also, a number of stationary nodes were deployed in various locations around the city of Cambridge UK. The data of the stationary iMotes will not be used in this work. The number of mobile devices used is 36 (plus 18 stationary devices). This data set covers 11 days.

- In *Infocom 05* [12, 48] the same devices as in *Cambridge* were distributed to students attending the Infocom 2005 student workshop. Participants belong to different social communities (depending on their country of origin, research topic, etc.)The number of devices is 41. This experiment covers approximately 3 days.

- In *Infocom 06* [12, 48] the scenario was similar to Infocom 05 except that the scale is larger, with 78 participants. Participants were selected so that 34 out of 78 form 4 subgroups by academic affiliation: ParisA with 10 participants, ParisB with 4 participants, Lausanne 5 participants, and, Barcelona 15 participants. In addition, 20 long range iMotes were deployed at several places in the conference site to act as access points. However, the data from these fixed nodes is not used in this work.

- *Dartmouth* [49] includes SNMP logs from the access points (Smartphones and Laptops) across the Dartmouth College campus from April 2001 to June 2004. To generate user-to-user contacts from the data-set, we follow the popular consideration in

Dataset	Cambridge	Infocom 05	Infocom 06	Dartmouth
Device	iMote	iMote	iMote	SPh, laptops
Network type	Bluetooth	Bluetooth	Bluetooth	AP
Duration (days)	11	3	3	60
Devices number	36	41	78	1146

Table 1.1: The three experimental data sets.

the literature that devices associated to the same AP at the same time are assumed to be in contact [11]. We consider activities from the 5th of January to the 6th of March 2004, corresponding to a 2-month period during which the academic campus life is reasonably consistent.

Further details on the real traces are shown in Table 1.1.

1.3.1 Tuning SWIM

Each parameter in SWIM has an impact on the outcome of the simulation. Table 1.2 shows, in details, the parameters we have used to tune SWIM when simulating each of the real scenarios considered. Here we explain the tuning methodology we used.

Number of nodes, area, and radius

First, the SWIM simulation area is fixed, 1×1. Parameters such as the number of nodes and the node radius are quite easy to set. They are taken from the real setting. In Infocom 06, for example, the number of nodes is set to 78. We then set the radius to 0.04, as an approximate proportion between bluetooth range and an estimation of the conference area. In the case of Dartmouth, being a campus, we set the radius to 0.013. Clearly, this is not an automatic process. However, according to our experiments the radius influences the number of contacts. In particular, "random" contacts that happen when the nodes move from one point of interest to another, as opposed to contacts that happen at destination points. Larger radius means higher number of (random) contacts. Most importantly, we

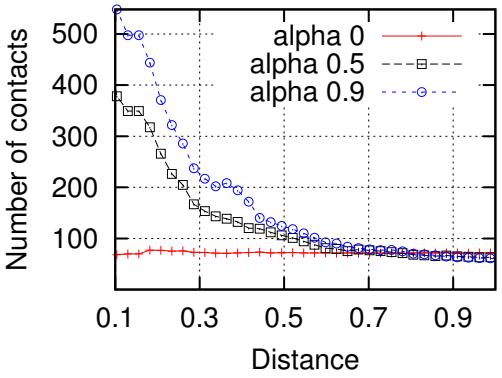


Figure 1.1: How does the number of contacts with other nodes depend on the mutual home-point distance for various values of α.

have observed that these parameters follow intuition very precisely, and that small differences in the parameters create small differences in the traces, thus allowing to tune the model systematically.

Waiting time distribution

The parameters of the waiting time comes directly from the traces in an automatic way. In Cambridge, Infocom 05 and Infocom 06 the head distribution of inter contact times has a slope of 1.35; whereas in Dartmouth, it is 1.65. Accordingly, we set the slope of the wait time distribution to be exactly the one observed from the real trace. Similarly for the cut-off—it is set to match the length of the power-law head of the inter contact time distribution of the respective real scenario. The values are: 24h for Cambridge, 12h for both Infocom scenarios, and, 11.5 days (277h) for the Dartmouth scenario.

Parameter α: Local Small Restaurant or VIP Bar?

To understand how parameter α influences the results we setup the following experiment: We simulate a 100 node network by keeping all parameters fixed but α, which is set to three different values: 0, 0.5, and, 0.9. Then, we compute the number of contacts of a randomly chosen node A with other nodes in the network as a function of the distance of their homes. We plot the results in Figure 1.1. The result confirms our intuition: The bigger α, the more

frequently nodes meet their neighbors. This is due to the fact that for high values of α nodes tend to restrict their movement to cells nearby their home. With lower α the phenomenon is attenuated and for $\alpha = 0$ the meeting rate does not depend at all on the distance of the homes (from Figure 1.1 the trend of the meeting probability when this distance varies is almost uniform).

Playing with the α parameter, and only this one, it is possible to boost social aspects of a well-known test scenario, or to boost geographical aspects of the same scenario. In Cambridge, the nodes are freshmen and sophomores at the Cambridge university. It is reasonable to think that freshmen meet freshmen and that sophomores meet sophomores more frequently. Indeed, we have seen that $\alpha = .8$ (giving more weight to geographical aspects) works well. In conferences, participants typically meet people that share affiliation or research interests. However, there are occasions that favor social mixing (e.g. social events, coffee brakes, etc.). This is why, experimentally, a smaller $\alpha = .7$ proved to work best. Dartmouth is different—the AP based contacts make so that people that go to the same place meet even though they might not share much. Nonetheless, students with same interests (e.g. taking the same classes) still tend to meet more often between them than with other students. In this case, which has higher mixing, the best α has been .6. Clearly, setting parameter α is not an automatic process. It is thus important to observe that in SWIM the results are always consistent with intuition, and that the number of parameters that have to be set in a non-automatic way is very limited.

1.3.2 SWIM vs Reality: Statistical properties

Here we present experimental results comparing statistical properties of the real scenarios with respect to SWIM. The parameters are shown in Table 1.2. We will call the four synthetic versions of Cambridge, Infocom 05, Infocom 06, and Dartmouth respectively SWIM 36, SWIM 41, SWIM 78, and SWIM 1146, where the number refers to the number of nodes in the scenario. It is particularly interesting that we might as well have got the (almost) exact parameters for SWIM 78 (the synthetic version of Infocom 06) by scaling SWIM 41 (the synthetic version of Infocom 05) according to the Manhattan model (constant area, higher density). Indeed, we can conjecture that the two real scenarios run in an area of

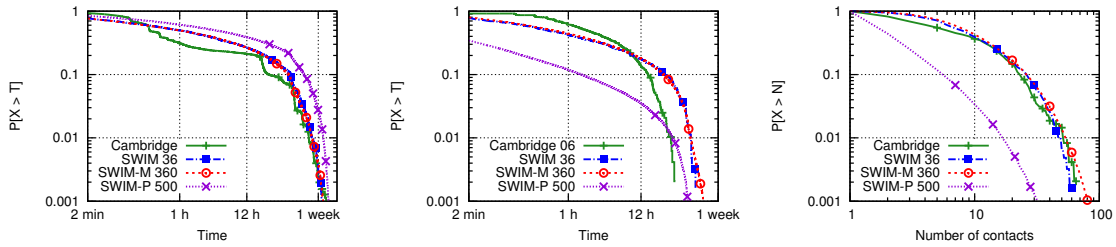
Scenario	Cambridge	Infocom 05	Infocom 06	Dartmouth
Radius	.05	.04	.04	.013
Duration (days)	11	3	3	60
Number of devices	36	41	78	1146
Value of α	.8	.7	.7	.6
Waiting time slope	1.35	1.35	1.35	1.65
Waiting time bound	24h	12h	12h	277h

Table 1.2: Tuning parameters

approximately the same size, with roughly double density since the number of devices distributed is roughly the double. This simple fact is a good support to our methodology.

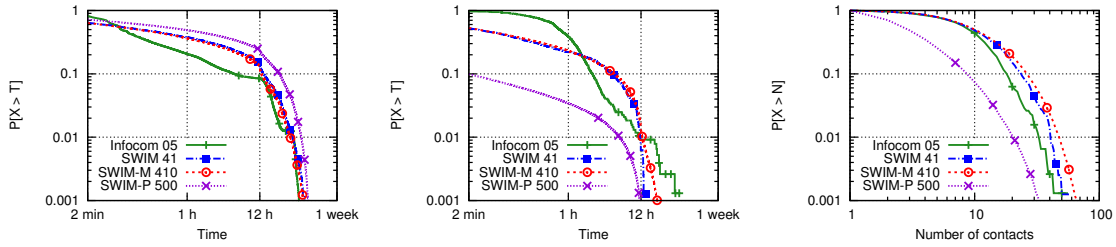
For each of the experiments we consider the following metrics: Inter-contact time CCD function, contact distribution per pair of nodes, and number of contacts per pair of nodes. The inter-contact time distribution is important in mobile networking since it characterizes the frequency with which information can be transferred between nodes. It has been studied for real traces in a large number of previous papers [10, 11, 46, 27, 25, 31, 50]. The distributions of contact durations and contact frequency per node-pairs are also important. Indeed they represent a way to measure relationship between people. As also discussed in [28, 51, 12], it's natural to think that if two people spend time together and meet frequently then they are familiar to each other. Familiarity is important in detecting communities, which may help improve significantly the design and performance of forwarding protocols in mobile environments [12].

In Figure 1.2, we show the results for Cambridge and for SWIM 36 (the synthetic version of Cambridge). Moreover, we have considered SWIM-M 360, that is a larger version of Cambridge with ten times the number of nodes according to the Manhattan model, and SWIM-P 500 a version with 500 nodes according to the Phoenix model. Similarly, Figures 1.3 and 1.4 show the results for Infocom 05, Infocom 06, their synthetic versions, and the larger scenarios built according to the Manhattan and the Phoenix models. In Figure 1.5 we show the results for Dartmouth. As the figures suggest, SWIM yields synthetic



(a) Inter-contacts (b) Contact duration per node pairs (c) Contact number per node pairs

Figure 1.2: SWIM and Cambridge. Cambridge is the real scenario, SWIM 36 is the synthetic version of Cambridge, SWIM-P 500 is Cambridge with 500 nodes according to the Phoenix model, and SWIM-M 360 is Cambridge with 360 nodes according to the Manhattan model.

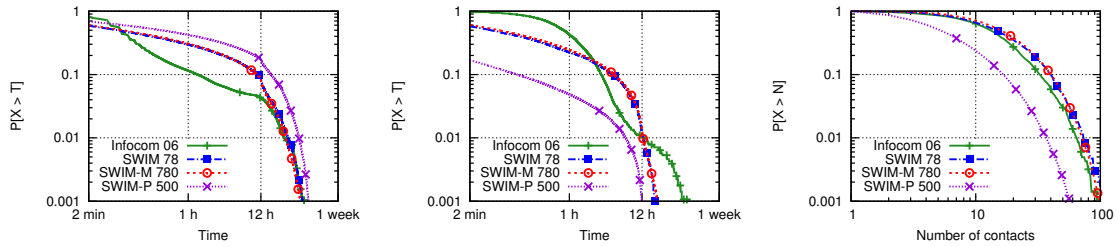


(a) Inter-contacts (b) Contact duration per node pairs (c) Contact number per node pairs

Figure 1.3: SWIM and Infocom 05. Inf 05 is the real scenario, SWIM 41 is the synthetic version of Infocom 05, SWIM-P 500 is Infocom 05 with 500 nodes according to the Phoenix model, and SWIM-M 410 is Infocom 05 with 410 nodes according to the Manhattan model.

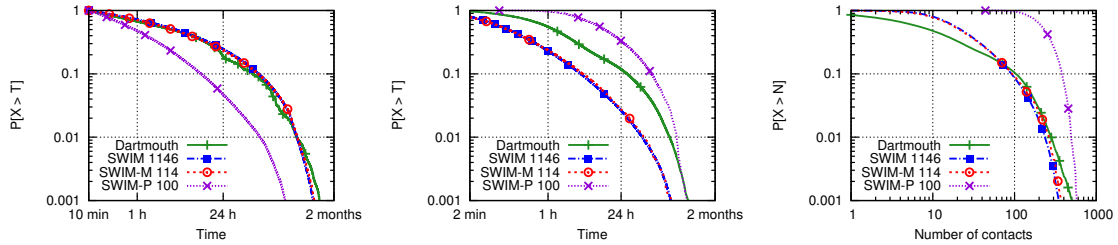
traces with statistical properties that are similar to the real ones. To strengthen this claim we show, in Table 1.3, the Jensen-Shannon divergence [52] between a given distribution in one of the real scenarios and its synthetic alter ego, for all the distributions considered. The Jensen-Shannon divergence measures the similarity of two probability distributions and takes values in $[0; 1]$, higher values mean higher divergence. We note that all values are low, which confirm what we observed from the figures. Note that the same choice of parameters gets good results for all the metrics under consideration at the same time.

In the figures we also show the behavior of the Phoenix (constant density) and Manhattan (constant area) models. Let us firstly discuss the Phoenix model: If we consider two *arbitrary* nodes, it is more likely that they meet less frequently as the number of nodes grows



(a) Inter-contacts (b) Contact duration per node pairs (c) Contact number per node pairs

Figure 1.4: SWIM and Infocom 06. Inf 06 is the real scenario, SWIM 78 is the synthetic version of Infocom 06, SWIM-P 500 is Infocom 06 with 500 nodes according to the Phoenix model, and SWIM-M 780 is Infocom 06 with 780 nodes according to the Manhattan model.



(a) Inter-contacts (b) Contact duration per node pairs (c) Contact number per node pairs

Figure 1.5: SWIM and Dartmouth. Dartmouth is the real scenario, SWIM-1146 is the synthetic version of Dartmouth, SWIM-P 100 is Dartmouth with 100 nodes according to the Phoenix model, and SWIM-M 114 is Dartmouth with 114 nodes according to the Manhattan model.

(and so the area). As a consequence, the inter-contact time should decay slower, while the contact-duration and the number of contacts should decay faster. Intuition is fully confirmed by the experimental results (see Figures 1.2(a)—1.4(a) for the inter-contact times distribution and Figures 1.2(b)–1.4(b) and Figures 1.2(c)–1.4(c) for the contact duration and the contact-number distributions). In the same figures, we can see that the Manhattan model is different. Since the area is the same when the number of nodes grows, the distribution of inter-contact time, contact duration, and number of contacts between any arbitrary pair of nodes should not change. It is just a more crowded world. This is also completely supported by our results.

For the Dartmouth case we down-scale: We scale to obtain smaller networks—this

Trace	Inter-contacts	Contacts Duration	Contact Number
Cambridge	.058	0.15	0.004
Infocom 05	.062	0.21	0.005
Infocom 06	.049	0.18	0.0114
Dartmouth	.028	0.11	0.073

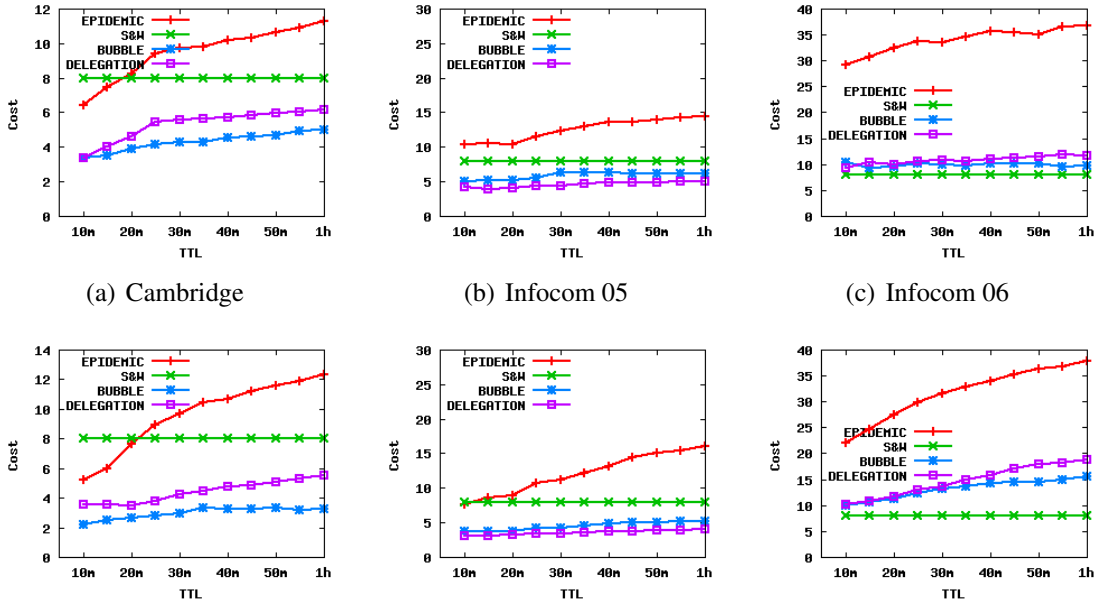
Table 1.3: Jensen-Shannon divergence between distributions of the real and SWIM traces.

trace is large enough (1146 nodes) to make it possible. For the Manhattan case we keep the area constant and lower the nodes number (so to lower the density), whereas, for the Phoenix case we lower the number of nodes yet keeping the density constant (so we lower the area). As in the up-scaling case (getting enlarged traces), from the graphics we observe that, for the Manhattan scaling (lower density, constant area), the distributions are preserved. Whereas, for the Phoenix scaling (constant density, smaller area), the effect in the distributions is exactly the opposite of that of the up-scaling: The smaller area makes so that nodes couple meet more frequently, and for longer times if averaged with all the nodes in the network. So, inter-contact times decay faster, while the contact duration and the number of contacts decay slower (see Figures 1.5(a), 1.5(b), and 1.5(c) for respectively the inter-contact times, contact duration and the contact-number distributions.

1.3.3 Protocol performance

Now, we get to a fundamental aspect for every model. We want to show that SWIM is good to predict the performance of forwarding protocols. We describe the experimental results of SWIM and four forwarding protocols for DTNs: Epidemic Forwarding [16], Delegation Forwarding[13], Spray&Wait [14], and BUBBLE [12]. In the experiments, we use exactly the same tuning used in the previous section. That is, the parameters input to SWIM are not "optimized" for each of the forwarding protocols, they are just the same that has been used to fit real traces with synthetic traces.

For the evaluation we use the same assumptions and the same way of generating traffic



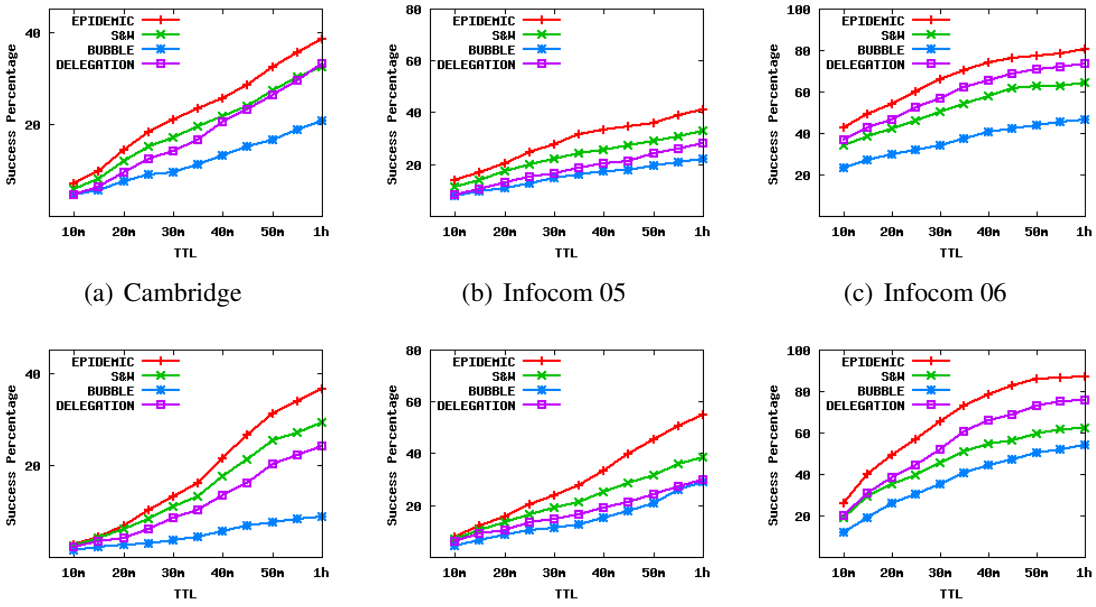
(a) Cambridge (b) Infocom 05 (c) Infocom 06

(d) SWIM-Cambridge (e) SWIM-Infocom 05 (f) SWIM-Infocom 06

Figure 1.6: Average cost of forwarding protocols. Cambridge, Infocom 05 (06) are the results on the real-traces. SWIM-Cambridge, SWIM-Infocom 05 (06) are the results on the simulated traces.

to be routed as in [13]. For each trace and forwarding protocol a set of messages is generated with sources and destinations chosen uniformly at random, and generation times form a Poisson process averaging one message every 4 seconds. The nodes are assumed to have infinite buffers and carry all message replicas they receive until the end of the simulation—this is in accordance with the literature on these protocols. The comparison is done in terms of *success percentage* (rate of messages delivered to destination) and *cost* (average number of replicas per delivered message) as a function of message TTL (time to leave). Message traffic follows a uniform traffic pattern (source-destination distributed uniformly at random among network nodes). As in [13], we isolated 3-hour periods for each data trace (real and synthetic) for our study. Each simulation runs therefore 3 hours. To avoid end-effects no messages were generated in the last hour of each trace.

Figures 1.6–1.7 show how the forwarding protocols perform in both real and synthetic traces, generated with SWIM. As can be seen, the performance of *all* protocols in the small-scale scenarios can be accurately predicted by running the protocols on the synthetic traces.



(a) Cambridge (b) Infocom 05 (c) Infocom 06

(d) SWIM-Cambridge (e) SWIM-Infocom 05 (f) SWIM-Infocom 06

Figure 1.7: Average success percentage of forwarding protocols. Cambridge, Infocom 05 (06) are the results on the real-traces. SWIM-Cambridge, SWIM-Infocom 05 (06) are the results on the simulated traces.

What's most, the trend of the protocols is the same in both synthetic and real-scenarios–the ones that perform better in the real world do so also in the SWIM-generated one. This support the claim that SWIM is an excellent model for protocol validation. In particular, this is also true for complex forwarding protocols such as BUBBLE, that depend on the structure of the network in social communities. In addition, in Table 1.4 we show the average performance difference of each protocol in SWIM compared to its performance on the respective real trace, for the Infocom 06 scenario. Note that the difference is small in both terms of cost and success percentage. This similar trend can be observed also for the other datasets. Most importantly, this is not due to a customized tuning that has been optimized for these forwarding protocols, it is just the same output that SWIM has generated with the tuning of the previous section. This can be important methodologically: To tune SWIM on a particular scenario, you can concentrate on a few well known and important statistical properties like inter-contact time, number of contacts, and duration of contacts. Then, you can have a good confidence that the model is properly tuned and usable

Protocol	Cost	Success
Epidemic forwarding	.08	.10
Delegation forwarding	.24	.11
BUBBLE Rap	.24	.15
Spray & Wait	0	.12

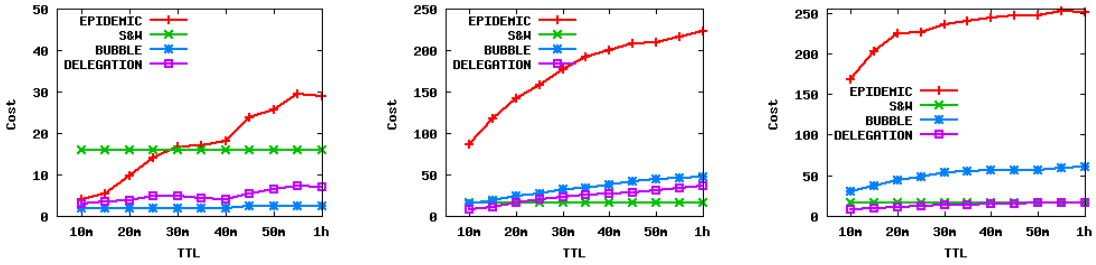
Table 1.4: Performance difference of protocols in percentage: SWIM vs Infocom 06.

to get meaningful estimation of the performance of a forwarding protocol.

Finally, to compare SWIM to the well-known RWP model, we setup the following experiment: we simulate with RWP one of the real scenarios considered in the work–mainly, the Infocom 06 scenario–and we run on the RWP trace and on the SWIM trace Delegation Forwarding and Random Forwarding (when A and B meet, B is decided to be a relay of a message depending on the result of a coin toss). Whereas Delegation performs highly better than Random forwarding on the SWIM trace, there is no distinction between the performance of the two protocols on the RWP trace. This is because in RWP the nodes' movement is memoryless, and it does not follow any social rule. So, the fact that a given node has seen the destination soon or not does not give any information on what will happen in the future. This is why using Delegation, a social-based forwarding strategy, rather than a random strategy to forward messages does not make any difference. Conversely, in SWIM the movement is social based–nodes tend to regularly go to cells nearby their home-points, and where they have met in the past many other nodes. Thus, social based strategies (such as Delegation, in this case), perform particularly better with respect to random strategies.

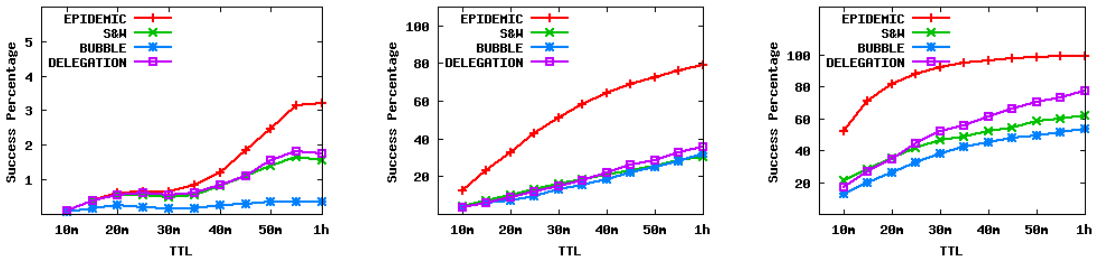
1.4 Scaling capabilities of forwarding protocols

When designing a networking protocol, scalability is a most desired property. SWIM can be used to address this important question: How do well-known forwarding protocols perform in large-scale social mobile networks? To give an accurate answer to this question, we validate the previously-considered forwarding protocols on large-scale SWIM-generated traces. The experimental setting is the same of the last section, whereas the Spray&Wait's



(a) Cambridge with SWIM-P (b) Infocom 05 with SWIM-P (c) Infocom 06 with SWIM-P

Figure 1.8: Average cost of forwarding protocols on enlarged Phoenix scenarios (constant density, larger area)



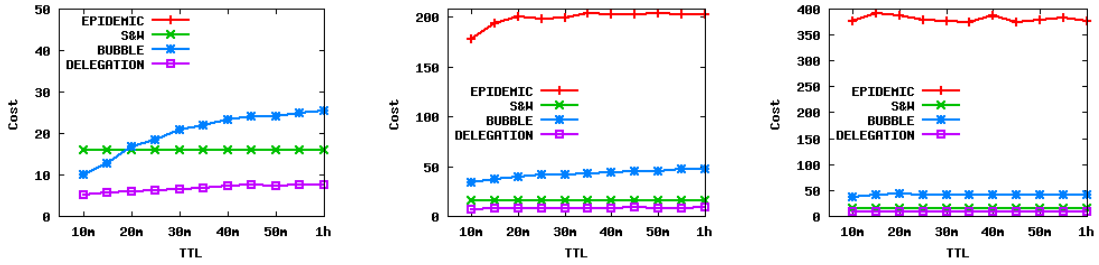
(a) Cambridge with SWIM-P (b) Infocom 05 with SWIM-P (c) Infocom 06 with SWIM-P

Figure 1.9: Average success percentage of forwarding protocols on enlarged Phoenix scenarios (constant density, larger area)

limit on message copies differs from scenario to scenario, and is set following the suggestions of the authors in [14]. Again, we study the *success percentage* and *cost* for various TTL (time to leave). The results are presented in Figures 1.8–1.11. Here are our observations:

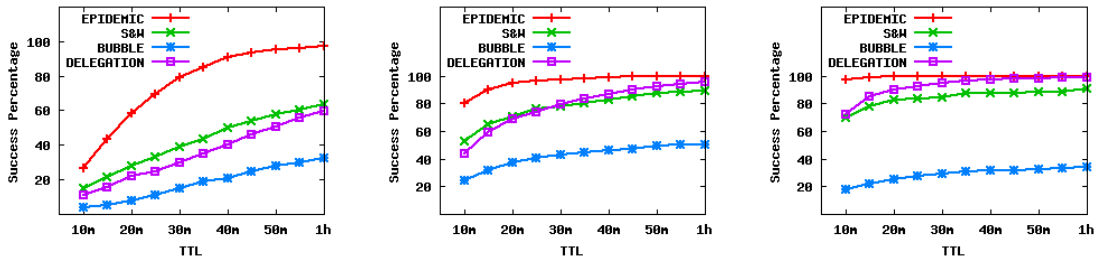
Scaling with the Phoenix model: When the number of nodes grows, the cost in terms of number of replicas is much higher, whereas the delivery rate drops considerably (compare Figure 1.6 with 1.8 for the cost and Figure 1.7 with 1.9 for the delivery rate). This is because when the network is enlarged by keeping the density constant, more hops are required to deliver a message (increasing the cost), and simultaneously, the network area is much larger, which makes it more difficult to get a message to destination.

Scaling with the Manhattan model: The cost again is much higher for all protocols but so is the delivery ratio (compare Figure 1.6 with 1.10 for the cost and Figure 1.7 with



(a) Cambridge with SWIM-M (b) Infocom 05 with SWIM-M (c) Infocom 06 with SWIM-M

Figure 1.10: Average cost of forwarding protocols on enlarged Manhattan scenarios (higher density, constant area)



(a) Cambridge with SWIM-M (b) Infocom 05 with SWIM-M (c) Infocom 06 with SWIM-M

Figure 1.11: Average success percentage of forwarding protocols on enlarged Manhattan scenarios (higher density, constant area)

1.11 for the delivery rate). This scaling method yields much denser networks, so the many contacts help all protocols to deliver messages quickly. Nonetheless, this also makes more probable that a high number of replicas are generated in the network, so, the cost is increased.

It is worthy to notice that these effects are attenuated for BUBBLE, Delegation, and Spray&Wait, which adopt more sophisticated rules to keep the cost reasonably low. Also, Delegation Forwarding and Spray&Wait seem to offer the best trade-off. They are not always the best when the network is small, but they show a good behavior when the network size grows compared both to Epidemic and to BUBBLE.

Overall, the experiments show that the quest for a scalable forwarding protocol for pocket switched network is still largely an open issue. Most probably, the techniques used in these protocols are excellent tools that can be used for larger and larger networks as well,

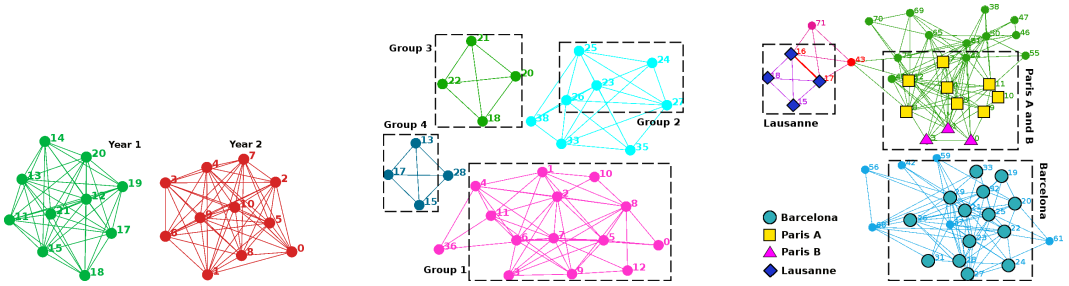
but it seems that some new additional idea is needed to keep cost in terms of messages low enough and success rate reasonably high.

1.5 Ad-hoc communities with SWIM

Many works have studied the communities that appear in traces of social mobile networks obtained from real experiments. To detect community sub-structures, the k-clique algorithm is widely used [53, 10, 11, 28]. The algorithm determines as belonging to the same community a union of adjacent cliques of k nodes sharing $k-1$ nodes [53]. In particular, this algorithm has been used in two of the scenarios we consider in this work—Infocom 06, and Cambridge [28]. The authors, which are also the ones who set up the experiments, have gathered information on the social relations of the participants. After detecting communities from the traces, they observe that the social relationships in real life have a good match with the ones uncovered from the traces by the k-clique algorithm.

In the Cambridge scenario, they detect two main communities of 11 members each that correspond to the students of the first and the second year. In the Infocom 06 scenario they observe that most of the participants with the same academic affiliation (ParisA, ParisB, Lausanne and Barcelona) do belong to the same community detected by the k-clique algorithm. Unlike the first two traces, Infocom 05 only contains partial information on the participants: There are four groups of respectively 10, 6, 4, 4 members each. It is not known how node IDs are mapped to participants, thus, which node is member of which group.

The next step of our study is to SWIM-generate these scenarios in an ad-hoc manner, such that a given desired social structure is observed at the end of the simulation. Let us start with Cambridge 05. There are 36 students involved, grouped by academic year in two groups: Year1 and Year2. As we mentioned, in the real trace there are two clear communities of 11 students. To each community we assign a "center point" in the network area: $p_1 = (.05; .05)$ and $p_2 = (.95; .95)$ (for respectively groups Year1 and Year2). The members of each group is given a home point obtained by perturbing the center point of their community with a Gaussian distribution of standard deviation of 0.01. The remaining 14 nodes are assigned a home point obtained with a uniform distribution over the network



(a) Simulated Cambridge. (b) Simulated Infocom 05. (c) Simulated Infocom 06.

Figure 1.12: Communities detected in the synthetic traces.

area.

In Infocom 06 there are four communities (ParisA, ParisB, Lausanne and Barcelona) of respectively 10, 4, 5, and, 15 members each. Therefore, to simulate this scenario we divide 34 nodes in 4 groups of as much members as in the real case. For each group we assign a central point as follows: $p_1 = (.01; .01)$ for ParisA, $p_2 = (.013; .013)$ for ParisB, $p_3 = (.95; .01)$ for Lausanne, and, $p_4 = (.5; .95)$ for Barcelona. Note that the members of the two Paris groups are initially placed close, in order to simulate social connection among them. The members of each group is given a home point obtained by perturbing the respective center point with a Gaussian distribution of standard deviation of 0.01. The remaining nodes are assigned home-points chosen uniformly and randomly over the network area.

Unlike the Cambridge and the Infocom 06 scenario, in Infocom 05 we have no exact information on the social relationships among participants. We have however information on the initial affiliation of some of the members (given by the authors of the experiment). So, in this case, we obtain the community information from the trace itself. We first run the k-clique algorithm on the Infocom 05 trace. The communities that we detect are consistent with the information we have on the experiment–4 communities of respectively 10, 6, 4 and 4 members each. This is not surprising. The k-clique algorithm is indeed one of the most used in the area to uncover social sub-structures from real-traces that reflect well the social relationships in real-life. Then, to simulate this scenario we feed the simulator with the information extrapolated from the k-clique communities uncovered in the real trace. We divide our nodes in 4 groups of as much members as in the real case. For each group we assign a central point as follows: $p_1 = (.95; .95)$ for group 1, $p_2 = (95; .05)$ for group

2, $p_3 = (.05; .95)$ for group 3, $p_4 = (.05; .05)$ for group 4. The members of each group is given a home point obtained by perturbing the respective center point with a Gaussian distribution of standard deviation of 0.01. The remaining nodes are assigned home-points chosen uniformly and randomly over the network area.

The rest of the simulation parameters are set as described in Table 1.2. In particular, the choice of α is done based on the grade of relationship people have in the scenarios (conferences vs university): .8 and .7 for Cambridge and the two Infocom scenarios, respectively. Also, the choice of the waiting time bound is done based on the real traces inter-contact time distribution's head. In the Cambridge case it follows a power law for 24 hours, whereas in both Infocom scenarios for 12 hours.

In Figure 1.12 we show the communities detected from the synthetic traces. As can be seen, in each simulated scenario the structure in communities reflects very well the real scenario: Nodes whose affiliation was emulated by assigning adjacent home-points result being members of the same community detected after the simulation. This means that SWIM preserves initial "social relationships" among nodes in the same way as a real social mobile network does and that it can be used to recreate traces with known community structures.

1.6 Conclusions

In this chapter we have presented SWIM, a mobility model that we can use to generate small mobile worlds. SWIM is very simple and it generates synthetic traces with excellent statistical properties. More than that, SWIM can predict extremely well the performance of forwarding protocols, even the most sophisticated ones that base their mechanisms on the structure in communities of the network.

We have also shown how we can get larger networks with SWIM in a sound way. We have used this capability to perform the first experimental analysis of the scaling properties of several of the best forwarding protocols in the literature.

Chapter 2

Settling for Less - A QoS Compromise Mechanism For Opportunistic Mobile Networks

The power and increasing prevalence of smartphones in combination with current research on opportunistic mobile networking have (1) increased the range of applications that could be supported on an opportunistic mobile network and (2) given birth to new fields of research such as mobile crowd computing [54] that are geared towards large–scale distributed computations. An opportunistic network is created between mobile phones using local peer-to-peer connections. The nodes in such a network are mobile phones carried by human users on the move, and a link between two phones represent the fact that the corresponding phone users are within each other's wireless communication range. Opportunistic networks are usually intermittently connected and are characterized by social-based mobility and heterogeneous contact rate. Their basic principle of operation is based on the store-and-forward strategy [55].

Keeping in mind the fact that opportunistic networks in the near future will primarily comprise of smartphones as nodes, and would be geared towards servicing numerous applications of varied QoS demands, the opportunistic network research community today still face three basic hurdles to achieving good performance on most applications. User mobility is one such hurdle. In a relatively sparse network, user mobility might lead to network

disconnectivity at times, which in turn increases response time of a user application. The second hurdle is the uncertainty in the quality of the wireless transmission channel. Effects like fading, shadowing, and interference might result in data packets being lost during transmission or being transmitted at low speeds. Finally, individual user selfishness is a psychological hurdle which users in an opportunistic network face. A mobile user would be unwilling to forward packets for someone it does not know due to (1) individual security concerns and (2) it unnecessarily expending battery power and computation resources for an application it has no relation with. Under the above mentioned hurdles, it is not guaranteed that user QoS[1] demands could be satisfied to a certain degree at all times let alone guaranteeing complete user satisfaction. However, in practice, users are generally tolerant on accepting lesser QoS guarantees than what they demand, with the degree of tolerance varying from user to user. The latter fact has been taken into account in some sense in traditional opportunistic networks research, where the primary goal was to make sure that users can somehow get the information through data relaying without thinking of QoS. On the other hand, an opportunistic mobile network of the near future needs to focus on the user tolerance of QoS degradation in order to justify it handling varied applications of different QoS demands.

In this short chapter we present a market based mathematical framework that enables heterogeneous mobile users in an opportunistic mobile network to compromise optimally and efficiently on their QoS demands in a manner such that each user is satisfied with its achieved (lesser) QoS, and at the same time the social welfare of users in the network is maximized. Our market based framework is practically implementable and is based on the concept of parameterized supply function bidding in traditional microeconomics theory [56, 57]. The contribution made in this abstract is important because (1) the hurdles related to opportunistic mobile networks mentioned in the previous paragraph are not easy to get rid of in a practical sense, and as a result mobile users have to compromise with lesser QoS than they would have ideally liked (2) the mobile users would love to make sure that they can comprise in an optimal and efficient manner, given uncertain network conditions, and (3) In an opportunistic mobile network, the network conditions vary from time to time,

[1] In general, QoS could be parameters such as response time, number of computations per unit time, allocated bandwidth, etc.

and it may not be possible to conjure up network resources on demand to meet user QoS choices; thus there is the need of an efficient technique that matches user demand to supply rather than the other way round. Supply function bidding is one such technique specifically suited for this purpose. In the rest of the abstract we use the terms 'mobile user' and 'user' interchangeably.

The results presented in this chapter appear in [3, 4].

2.1 System Model

We consider a mobile network system of N_t mobile users in a time slot t. Each time slot t lies within a total time period T[2] and is of the form $[t-1,t]$. Within each time slot the total number of mobile users is assumed to be constant. We assume that the system is geared towards executing distributed computation tasks, in addition to regular data forwarding as in an opportunistic network. Each user in a time slot could either be (a) someone initiating a computation task, (b) someone doing computations for a task at hand, (c) someone just relaying information, or (d) someone doing all of (a), (b), and (c). In every time slot both the task initiators and task executors register with a central market agency. The agency could either be the one who develops the framework for efficient and optimal large–scale distributed computation or a third party. The agency has two functions in every time slot: (1) to accept user QoS demands and supply functions (QoS compromise functions) from task initiators and (2) to assess the aggregate service capacity of the task executors and enable market clearing, i.e., ensure aggregate user QoS compromise equals aggregate service capacity deficit. We also assume that the agency is connected to the mobile users via a control channel for signaling purposes (e.g., via a 3G connection).

2.1.1 The Basic Idea in a Nutshell

In every time slot the task initiators 'supply' (via an *iterative bidding process* [57] between themselves and the central agency) their supply functions to the central agency. A supply function is a measure of the amount of QoS a user is willing to compromise in return for

[2]For example, the period T could be a single day.

a certain amount of benefit the agency would provide to the mobile user for making the compromise[3]. The agency estimates[4] the deficit in the aggregate service capacity (if there is any) that prevents the network from servicing ideal user QoS demands, and chooses a common benefit value that clears the market. This benefit value is passed on to all the task initiators in the time slot who in turn settle for the corresponding compromise level based on their compromise (supply) functions. In this abstract we consider two ways in which mobile users could choose their supply functions: (1) it chooses an optimal function in a 'price taking' (competitive) [58] market[5] of task initiators and (2) it chooses an optimal function in an 'price anticipating' (oligopolistic) [58] market of task initiators.

2.1.2 QoS Compromise Function

Let $c_{it}(k_{it}, b_t)$ be the QoS compromise function for user i in time slot t. We parameterize user i's compromise function in each time t as follows.

$$c_{it}(k_{it}, b_t) = k_{it} b_t, \forall i \in N_t^{init} \subseteq N_t, \tag{2.1.1}$$

where N_t^{init} consists of those users in time slot t who initiate the execution of a task. The function $c_{it}(\cdot)$ is the 'supply' function for user i and gives the amount of QoS it is committed to compromise. In this abstract we treat QoS as the reciprocal of response time of an application initiated by a user. For example, if a user expects to ideally achieve a response time of 2 time units, its QoS metric would have a value of $\frac{1}{2}$. However, it could compromise[6] say a response time of 3 additional seconds in which case its achieved QoS is $\frac{1}{5}$. Thus, it makes a compromise of $\frac{3}{10}$ QoS units.
$k_{it} \geq 0$ is the supply function profile [57] for user i in time slot t. It is a scalar quantity that determines the supply function of a user in time slot t, and is known to the central agency.

[3]The benefit provided also incentives users to compromise. Benefits could be in the form of reduction of prices charged for service.

[4]Message passing between the agency and task executors could be one way of estimating service capacity deficits.

[5]We note here that the market is jointly run by the central agency and the task initiators.

[6]To decide on its amount of compromise, a user, among other factors, may account for the delay due to computation information percolating through relay nodes before it reaches the intended recipient. In this abstract we do not explicitly model the role that relay nodes have on the optimal supply function of a user.

b_t is the benefit that the central agency provides to all the task initiators.

2.1.3 Clearing the Market

The central agency clears the market in every time slot by solving the following equation:

$$\sum_i c_{it}(k_{it}, b_t) = \sum_i k_{it} b_t = d_t, \tag{2.1.2}$$

where d_t is the aggregate service capacity deficit in time slot t. Solving the latter equation we get the value of b_t as

$$b_t(\overrightarrow{k_t}) = \frac{d_t}{\sum_i k_{it}}, \forall t, \tag{2.1.3}$$

where $\overrightarrow{k_t} = (k_{1t}, k_{2t}, \ldots\ldots, k_{|N_t^{init}|})$ is the vector of support profiles for the task initiators in time slot t.

2.2 Competitive Market Analysis

We consider a competitive market of task initiators where the latter are 'benefit' taking. Given a benefit value b_t in time slot t, each user i maximizes its profit according to the following optimization problem:

$$argmax_{k_{it}} b_t c_{it}(k_{it}, b_t) - C_{it}(c_{it}(k_{it}, b_t))$$

where $C_{it}(c_{it}(\cdot))$ is the disutility or cost incurred by user i in time slot t when it compromises $c_{it}(\cdot)$ QoS units. We assume that $C_{it}(\cdot)$ is continuous, increasing, and strictly convex with $C_{it}(0) = 0$.

In every time slot t, a competitive (Walrasian) equilibrium amongst the task initiators and the central agency is defined as a tuple $(k_{it}^{eq})_{i \in N_t^{init}}, b_t^{eq}$ that satisfies the following conditions:

$$(C'_{it}(c_{it}(k_{it}^{eq}, b_t^{eq})) - b_t^{eq})(b_t - b_t^{eq}) \geq 0, \forall b_t \geq 0 \tag{2.2.1}$$

$$\sum_i c_{it}(k_{it}^{eq}, b_t^{eq}) = d_t \tag{2.2.2}$$

Theorem 2.2.1. *There exists a competitive equilibrium in the market of task initiators in every time slot, t that maximizes the following:*

$$argmax_{c_{it}} \sum_i -C_{it}(c_{it})$$

subject to $\sum c_{it} = d_t$.

Proof Sketch. We get the optimality conditions of the optimization problem in Theorem 2.2.1 from equations (2.2.1) and (2.2.2). The uniqueness of the optimal solution, i.e., the equilibrium solution, follows from the fact that the optimization problem and its dual are strictly convex.

Theorem Implications. The equilibrium solution maximizes the social welfare, i.e., minimizes the sum of the disutility of the task initiators, via the optimization problem in the theorem. Thus in every time slot, the central agency is able to clear the market by enabling optimal user QoS compromises as well as by ensuring social welfare.

The Iterative Bidding Process. We provide a distributed iterative bidding scheme based on the dual gradient algorithm in [59] that achieves the market equilibrium in each time slot t.

At the j-th iteration in time slot t, we execute the following steps:

1. Upon receiving benefit $b_t(j)$ announced by the central agency, task initiator i updates its supply function profile, $k_{it}(j)$ as

$$k_{it}(j) = \left\{ \frac{(C'_{it})^{-1}(b_t(j))}{b_t(j)} \right\}^+, \tag{2.2.3}$$

   and supplies it to the central agency. Here '$+$' denotes the projection onto $\mathtt{R}^+$, the set of non–negative real numbers.

2. The central agency updates its benefit according to the following equation

$$b_t(j+1) = \left[b_t(j) - \rho \left(\sum_i k_{it}(j) b_t(j) - d_t \right) \right]^+, \tag{2.2.4}$$

   and announces the new benefit to the task initiators.

The above distributed bidding process converges for small enough values of step size, ρ [59].

2.3 Oligopolistic Market Analysis

We consider an oligopolistic market of task initiators where the latter are 'benefit' anticipating. The initiators are strategic in the sense that they know that benefit b_t in each time slot t is computed according to equation (2.1.3) and as a result choose their supply function profile in a manner so as to maximize their net utility functions. The net utility function for each user i in time slot t is represented by $U_{it}(k_{it}, k_{(-i)t})$ and is given as:

$$U_{it}(k_{it}, k_{(-i)t}) = b_t c_{it}(k_{it}, b_t(\overrightarrow{k_t})) - C_{it}(c_{it}(k_{it}, b_t(\overrightarrow{k_t}))), \tag{2.3.1}$$

where $k_{(-i)t} = (k_{1t}, \ldots, k_{(i-1)t}, k_{(i+1)t}, \ldots, k_{|N_t^{init}|})$ is the vector of supply function profile of users other than i. Each user participates in a non–cooperative game of selecting k_{it}'s, with other task initiators in time slot t, in order to maximize its net utility function. The intersection of the best responses of all the task initiators results in a Nash equilibrium [58].

Lemma 2.3.1. *If $(\overrightarrow{k_t^{neq}})$ is a Nash equilibrium of the non–cooperative game at time slot t, then (1) $\sum_{j\neq i} k_{jt}^{neq} > 0$ for any $i \in N_t^{init}$, (2) $k_{it}^{neq} < K_{(-i)t}^{neq}$ for any $i \in N_t^{init}$, and (3) No Nash equilibrium exists when $|N_t^{init}| = 2$, where $K_{(-i)t}^{neq} = \sum_{j\neq i} k_{jt}^{neq}$*

Lemma Implications. At Nash equilibrium in every time slot, each task initiator compromises at most $\frac{d_t}{2}$ amount of QoS units, and at least two task initiators are necessary to reach a Nash equilibrium.

Theorem 2.3.1. *There exists a Nash equilibrium, $(\overrightarrow{k_t^{neq}})$, in the market of task initiators in every time slot t that maximizes the following:*

$$argmax_{0\leq c_{it}<\frac{d_t}{2}} \sum_i -H_{it}(c_{it})$$

subject to $\sum_i c_{it} = d_{it}$, *where*

$$H_{it}(c_{it}) = \left(1 + \frac{c_{it}}{d_t - 2c_{it}}\right) C_{it}(c_{it}) - \int_0^{c_{it}} \frac{d_t}{(d_t - 2x_{it})^2} C_{it}(x_{it}) dx_{it}.$$

Proof Sketch. The uniqueness of the Nash equilibrium (optimal) solution follows from the fact that the optimization problem and its dual are strictly convex.

Theorem Implications. The equilibrium solution *maximizes the social welfare*, i.e., minimizes the sum of the disutility of the task initiators, via the optimization problem in the theorem. Thus in every time slot, the central agency is able to clear the market by enabling optimal user QoS compromises, reaching a unique Nash equilibria, as well as ensuring social welfare.

Proposition 2.3.1. *The Nash equilibrium benefit* b_t^{neq} *is bounded within a factor* $(1 + \frac{r_t}{d_t - 2r_t})$ *of* b_t^{eq}, *the Walrasian equilibrium benefit where* $r_t = \max_i (H'_{it})^{-1}(Y_t)$ *and* $Y_t = \max_i H'_{it}(\frac{d_t}{|N_t^{init}|})$.

Proposition 2.3.1 is an important result and implies that the benefit anticipating and mutually competitive nature of task initiators in an oligopoly market leads to the Nash equilibrium benefit being bounded by the Walrasian equilibrium benefit as Walrasian markets are benefit taking.

The Iterative Bidding Process. At the j-th iteration in time slot t, we execute the following steps:

1. Upon receiving benefit $b_t(j)$ announced by the central agency, task initiator i updates its supply function profile, $k_{it}(j)$ as

$$k_{it}(j) = \left\{ \frac{(H'_{it})^{-1}(b_t(j)}{b_t(j)} \right\}^+,$$

   and supplies it to the central agency.

2. The central agency updates its benefit according to the following equation:

$$b_t(j+1) = \left[b_t(j) - \rho \left(\sum_i k_{it}(j) b_t(j) - d_t \right) \right]^+,$$

and announces the new benefit to the task initiators.

2.4 Conclusions

In this short chapter we studied ways to optimally and efficiently entail user QoS compromises in opportunistic mobile networks via market mechanisms. As part of future work, we plan to conduct a simulation study of our proposed theory, and extend our theory to include different forms of parameterized supply functions.

Chapter 3

Introduction to Mobile Cloud Computing

With the advent of cloud computing many companies embraced the computational power and storage capability offered by this technology. Google, Amazon, and Microsoft are only some of the many companies that nowadays are pushing their products on the cloud. We, the private users, are also being pushed toward the cloud in a seemly way. Many of the programs we use everyday — Google Docs, Picasa, iTunes, etc., — are supported by some remote entities. Recently, also many smartphone applications started benefitting from the power of the cloud. Take for example Siri, Google Voice, Shazam, and etc.; all these apps execute the heavy tasks on their remote counterparts, wait for the result to come back on the smartphone, and continue the execution. Indeed, many researchers believe that cloud computing is an excellent candidate to help reduce battery consumption of smartphones, as well as to backup smartphone user's data. Many recent works have focused on building frameworks that enable mobile computation offloading to software clones of smartphones on the cloud, as well as to backup systems for data and applications stored in our devices.

Continuing our research in the smartphone world, we focus now on the field of mobile cloud computing. In this brief chapter we give an overview of the existing works in literature regarding smartphone code offloading and smartphone backup on the cloud. Then, in the chapters that follow we will extend the list of existing works specifically for each problem treated.

3.1 Computation offloading on the cloud

The basic idea of dynamically switching between (constrained) local and (plentiful) remote resources, often referred as cyber-foraging, has shed light on many research work [60, 61, 62, 63, 64, 65, 66, 67]. These approaches augment the capability of resource-constrained devices by offloading computing tasks to nearby computing resources, or *surrogates*. Mobile computation offloading and data backup involve communication between the real device and the cloud. This communication does not certainly come for free, in terms of both energy consumption (utilization of network interfaces to send the data) and bandwidth [68]. Many works consider the trade-off between the energy spent to offload specific application modules and the energy saved thanks to the cloud [20, 69, 19]. Also do exist analytical models that aim to predict approximately these costs in the case of mobile computation offloading [70, 71]. Several approaches have been proposed to predict resource consumption of a computing task or method. Narayanan *et al.* [72] use historical application logging data to predict the fidelity of an application, which decides its resource consumption although they only consider selected aspects of device hardware and application inputs. Gurun *et al.* [73] extend the Network Weather Service (NWS) toolkit in grid computing to predict offloading but give less consideration to local device and application profiles.

MAUI [20] describes a system based on the Microsoft .NET framework that enables energy-aware offloading of mobile code to infrastructure. Its main aim is to optimize energy consumption of a mobile device, by estimating and trading off the energy consumed by local processing *vs.* transmission of code and data for remote execution. Although it has been found that optimizing for energy consumption often also leads to performance improvement, the decision process in MAUI only considers relatively coarse-grained information, compared with the complex characteristics of mobile environment.

More recently, CloneCloud [19] uses a process-based offloading methodology: The binary of the application is partitioned and an off-line analysis decides which binary pieces are to be migrated to the cloud. Experiments with clones hosted on private cloud (local servers) show up to 20x energy saving with applications such as virus scanning, image search, and behavioral profiling.

Cloudlets [74, 75] proposes the use of nearby resource-rich computers, to which a

smartphone connects over a wireless LAN, with the argument against the use of the cloud due to higher latency and lower bandwidth available when connecting. In essence, Cloudlets makes the use of smartphone simply as a thin-client to access local resources, rather than using the smartphone's capabilities directly and offloading only when required.

Paranoid Android [69] uses QEMU to run replica Android images in the cloud to enable multiple exploit and attack detection techniques to run simultaneously with minimal impact on phone performance and battery life.

Later on, the authors in [76] present a system that allows users to create customized virtual images of their smartphones on the cloud. Their prototype makes applications running on the cloud appear like local applications on the physical device—the user interacts with the virtual smartphone remotely and seamlessly. The virtual device, after executing the commands, sends back to the real device the data output and the screen updates. Experimental results on private cloud (local servers) show that this system is particularly adequate for computation-intensive applications.

Virtual Smartphone [77] uses Android x86 port to execute Android images in the cloud efficiently on VMWare ESXi virtualization platform, although it does not provide any programmer support for utilizing this facility.

3.2 Using the cloud for backup

Computation offloading is not the only thing the cloud comes to use: Arguing on the importance of data that nowadays users store on their mobile devices, the authors in [21] develop a remote control system for lost handsets that aims to protect personal information of users. Nonetheless, the remote control system has to be triggered so that to lock/unlock real device's functionalities/access to data, and eventually backup the data on an online server. If the thief abruptly cancels the data by formatting the SD card and re-installing the OS, the user will never be able to get her data again. So, backup/restore systems that regularly send user's data to remote servers for backup are very valuable in this context. The system proposed in [22], besides from backup/restore, also allows for sharing information in smartphones among groups of people. The authors test their system in terms of time needed (on the phone) to backup three different data types: SMS, calendar events,

and contacts. In [23] the authors argue that not only contacts and emails—synced by e.g. Google sync on Android OS—but also application settings, game scores etc., are important to users. With this in mind they build ASIMS, a tool that has the goal of providing a better application settings integration and management scheme for Android mobiles. ASIMS is based on SQLite, it stores other applications' settings and syncs them to the Internet. Its interface makes it possible for other applications to store settings in one common place and for users to select which applications they want to sync.

The work in [70] presents energy models that trade-off the energy consumption on the mobile device versus the energy needed to send the data to the cloud, to encrypt it, and to manage other operations related to this process. The authors of [68] discuss on the main factors that affect the energy consumption of mobile apps in cloud computing, and deem that such factors are workload, data communication patterns, and usage of WLAN and 3G. In a more recent work [71] the authors present an analytical study to find the optimal execution policy. This is identified by optimally configuring the clock frequency in the mobile device to minimize the energy used for computation and by optimally scheduling the data rate over a stochastic wireless channel to minimize the energy for data transmission. By formulating these issues as a constrained optimization problem they obtain close-formed solutions which give analytical insight in finding the optimal offloading decision.

Chapter 4

ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading

As we saw in the previous chapter, considerable research work have proposed solutions to address the issues of computational power and battery lifetime by offloading computing tasks in the cloud. Prominent among those are the MAUI [20] and the CloneCloud [19] projects. MAUI provides method level code offloading based on the .NET framework. However, MAUI does not address the scaling of execution in cloud. CloneCloud tries to extrapolate the binary pieces of a given process whose execution on the cloud would make the overall process execution faster. It determines these pieces with an offline static analysis of different running conditions of the process binary on both a target smartphone and the cloud. The output of such analysis is then used to build a database of pre-computed partitions of the binary; this is used to determine which parts should be migrated on the cloud. However, this approach only considers limited input/environmental conditions in the offline pre-processing and needs to be bootstrapped for every new application built.

In this chapter, we present *ThinkAir*[1], a new mobile cloud computing framework which

[1]This work is a collaboration with Dr. Pan Hui of Deutsche Telekom Labs Berlin, Andrius Aucinas of University of Cambridge, Richard Mortier of University of Nottingham, and Xinwen Zhang of Huawei Technologies. The collaboration started while Sokol was doing an internship at the Deutsche Telekom Labs under the supervision of Dr. Pan Hui.

takes the best of the two worlds. ThinkAir addresses MAUI's lack of scalability by creating virtual machines (VMs) of a complete smartphone system on the cloud, and removes the restrictions on applications/inputs/environmental conditions that CloneCloud induces by adopting an online method-level offloading. mt focuses on the elasticity and scalability of the cloud and enhances the power of mobile cloud computing by parallelizing method execution using multiple virtual machine (VM) images.

ThinkAir (1) provides an efficient way to perform on-demand resource allocation, and (2) exploits parallelism by dynamically creating, resuming, and destroying VMs in the cloud when needed. To the best of our knowledge, this is the first framework to address these two aspects in mobile clouds. Supporting on-demand resource allocation is critical as mobile users request different computational power based on their workloads and deadlines for tasks, and hence the cloud provider has to dynamically adjust and allocate its resources to satisfy customer expectations. Existing work does not provide any mechanism to support on-demand resource allocation, which is an absolute necessity given the variety of applications that can be run on smartphones, in addition to the high variance of CPU and memory resources these applications could demand. The problem of exploiting parallelism becomes important because mobile applications require increasing amounts of processing power, and parallelization reduces the execution time and energy consumption of these applications with significant margins when compared to non-parallel executions of the same.

ThinkAir achieves all the above mentioned goals by providing a novel execution offloading infrastructure and rich resource consumption profilers to make efficient and effective code migration possible; it further provides library and compiler support to make it easy for developers to exploit the framework with minimal modification of existing code, and a VM manager and parallel processing module in cloud to manage smartphone VMs as well as automatically split and distribute tasks to multiple VMs.

We show that the execution time and energy consumption decrease two orders of magnitude for a *N*-queens puzzle application and one order of magnitude for a face detection and a virus scan application. We then show that a parallelizable application can invoke multiple VMs to execute in the cloud in a seamless and on-demand manner such as to achieve

greater reduction on execution time and energy consumption. We finally use a memory-hungry image combiner tool to demonstrate that applications can dynamically request VMs with more computational power in order to meet their computational requirements.

We now continue outlining the ThinkAir architecture (Section 4.1) and then describe the three main components of ThinkAir in more detail: the execution environment (Section 4.2), the application server (Section 4.3), and the profilers (Section 4.4). We then evaluate the performance of benchmark applications with ThinkAir (Section 4.5), discuss design limits and future plans (Section 4.6), and conclude the chapter (Section 4.7).

The results presented in this chapter appear in [6, 5].

4.1 Design Goals and Architecture

The design of ThinkAir is based on some assumptions which we believe are already, or soon will become, true: (1) Mobile broadband connectivity and speeds continue to increase, enabling access to cloud resources with relatively low Round Trip Times (RTTs) and high bandwidths; (2) As mobile device capabilities increase, so do the demands placed upon them by developers, making the cloud an attractive means to provide the necessary resources; and (3) Cloud computing continues to develop, supplying resources to users at low cost and on-demand. We reflect these assumptions in ThinkAir through four key design objectives.

(*i*) *Dynamic adaptation to changing environment*. As one of the main characteristics of mobile computing environment is rapid change, ThinkAir framework should adapt quickly and efficiently as conditions change to achieve high performance as well as to avoid interfering with the correct execution of original software when connectivity is lost.

(*ii*) *Ease of use for developers*. By providing a simple interface for developers, ThinkAir eliminates the risk of misusing the framework and accidentally hurting performance instead of improving it, and allows less skilled and novice developers to use it and increase competition, which is one of the main driving forces in today's mobile application market.

(*iii*) *Performance improvement through cloud computing*. As the main focus of ThinkAir, we aim to improve both computational performance and power efficiency of mobile devices

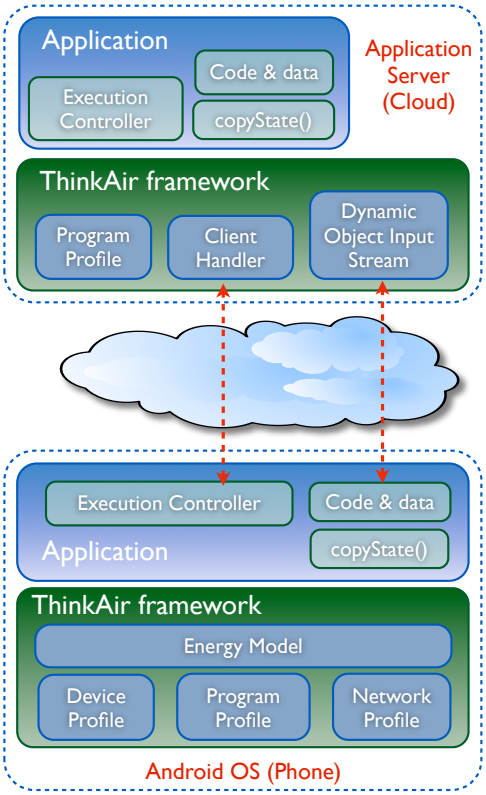


Figure 4.1: Overview of the ThinkAir framework.

by bridging smartphones to the cloud. If this bridge becomes ubiquitous, it serves as a stepping stone towards more sophisticated software.

(*iv*) *Dynamic scaling of computational power.* To satisfy the customer's performance requirements for commercial grade service, ThinkAir explores the possibility of dynamically scaling the computational power at the server side as well as parallelizing execution where possible for optimal performance.

The ThinkAir framework consists of three major components: the execution environment (Section 4.2), the application server (Section 4.3) and the profilers (Section 4.4).

We will now give details of each component of the framework, depicted in Figure 4.1.

4.2 Compilation and Execution

In this section we describe in detail the process by which a developer writes code to make use of ThinkAir, covering the programmer API and the compiler, followed by the execution flow.

4.2.1 Programmer API

Since the execution environment is accessed indirectly by the developer, ThinkAir provides a simple library that, coupled with the compiler support, makes the programmer's job very straightforward: any method to be considered for offloading is annotated with `@Remote`.

This simple step provides enough information to enable the ThinkAir code generator to be executed against the modified code. This takes the source file and generates necessary *remoteable* method wrappers and utility functions, making it ready for use with the framework - method invocation is done via the *ExecutionController*, which detects if a given method is a candidate for offloading and handles all the associated profiling, decision making and communication with the application server *without* the developer needing to be aware of the details.

4.2.2 Compiler

A key part of the ThinkAir framework, the compiler comes in two parts: the Remoteable Code Generator and the Customized Native Development Kit (NDK). The Remoteable Code Generator is a tool that translates the annotated code as described above. Most current mobile platforms provide support for execution of native code for the performance-critical parts of applications, but cloud execution tends to be on x86 hosts, while most smartphone devices are ARM-based, therefore the Customized NDK exists to provide native code support on the cloud.

4.2.3 Execution Controller

The Execution Controller drives the execution of remoteable methods. It decides whether to offload execution of a particular method, or to allow it to continue locally on the phone. The

decision depends on data collected about the current environment as well as that learned from past executions.

When a method is encountered for the first time, it is unknown to the Execution Controller and so the decision is based only on environmental parameters such as network quality: for example, if the connection is of type WiFi, and the quality of connectivity is good, the controller is likely to offload the method. At the same time, the profilers start collecting data. On a low quality connection, however, the method is likely to be executed locally.

If and when the method is encountered subsequently, the decision on where to execute it is based on the method's past invocations, i.e., previous execution time and energy consumed in different scenarios, as well as the current environmental parameters. Additionally, the user can also set a policy according to their needs. We currently define four such policies, combining execution time, energy conservation and cost:

- *Execution time*. Historical execution times are used in conjunction with environmental parameters to prioritize fast execution when offloading, i.e. offloading only if execution time will improve (reduce) no matter the impact on energy consumption.

- *Energy*. Past data on consumed energy is used in conjunction with environmental parameters to prioritize energy conservation when offloading, i.e., offloading only if energy consumption is expected to improve (reduce) no matter the expected impact on performance.

- *Execution time and energy*. Combining the previous two choices, the framework tries to optimize for both fast execution and energy conservation, i.e., offloading only if both the execution time and energy consumption are expected to improve.

- *Execution time, energy and cost*. Using commercial cloud services also implies cost - you pay for as much as you use, therefore offloading decision based on execution time and energy could also be adjusted according to how much a user is prepared to pay for the retained CPU time and battery power.

Clearly more sophisticated policies could be expressed; discovering policies that work well, meeting user desires and expectations is the subject of future work.

4.2.4 Execution flow

The result of the above compilation process is that, flow of control is handed over to the Execution Controller when a remoteable method is called, as depicted in Figure 4.2.

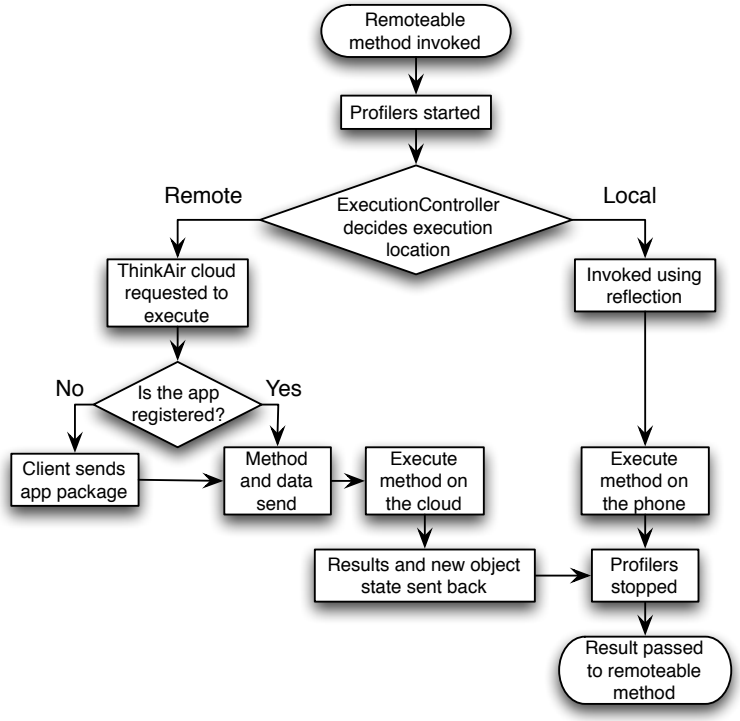


Figure 4.2: Flow execution from calling a method to getting the result.

On the phone, the Execution Controller first starts the profilers to provide data for future invocations. It then decides whether this invocation of the method should be offloaded or not. If not, then the execution continues normally on the phone. If it is, Java reflection is used to do so and the calling object is sent to the application server in the cloud; the phone then waits for results, and any modified local state, to be returned. If the connection fails for any reason during remote execution, the framework falls back to local execution, discarding any data collected by the profiler. At the same time, the Execution Controller initiates asynchronous reconnection to the server. If an exception is thrown during the remote execution of the method, it is passed back in the results and re-thrown on the phone, so as not to change the original flow of control.

In the cloud, the Application Server manages clients that wish to connect to the cloud, which is illustrated in the next section.

4.3 Application Server

The ThinkAir Application Server manages the cloud side of offloaded code and is deliberately kept lightweight so that it can be easily replicated. It is started automatically when the remote Android OS is booted, and consists of three main parts, described below: a client handler, the cloud infrastructure, and an automatic parallelization component.

4.3.1 Client Handler

The Client Handler executes the ThinkAir communication protocol, managing connections from clients, receiving and executing offloaded code, and returning results.

To manage client connections, the Client Handler registers when a new application, i.e., a new instance of the ThinkAir Execution Controller, connects. If the client application is unknown to the application server, the Client Handler retrieves the application from the client, and loads any required class definitions and native libraries. It also responds to application-level *ping* messages sent by the Execution Controller to measure connection latency.

Following the initial connection set up, the server waits to receive execution requests from the client. A request consists of necessary data: containing object, requested method, parameter types, parameters themselves, and a possible request for extra computational power. If there is no request for more computational power, the Client Handler proceeds much as the client would: the remoteable method is called using Java reflection and the result, or exception if thrown, is sent back. There are some special cases regarding exception handling in ThinkAir, however. For example, if the exception is an *OutOfMemoryError*, the Client Handler does not send it to the client directly; instead, it dynamically resumes a more powerful clone (a VM), delegates the task to it, waits for the result and sends it back to the client. Similarly, if the client explicitly asks for more computational power, the Client Handler resumes a more powerful clone and delegates the task to it. In the case that

Resource	basic	main	large	×2 large	×4 large	×8 large
CPUs	1	1	1	2	4	8
Memory (MB)	200	512	1024	1024	1024	1024
Heap size (MB)	32	100	100	100	100	100

Table 4.1: Different configurations of VMs.

the client asks for more clones to execute its task in parallel, the Client Handler resumes the needed clones, distributes the task among them, collects and sends results back to the client. Along with the return value, the Client Handler also sends profiling data for future offloading decisions made by the Execution Controller at the client side.

4.3.2 Cloud Infrastructure

To make the cloud infrastructure easily maintainable and to keep the execution environment homogeneous, e.g., w.r.t. the Android-specific Java bytecode format, we use a virtualization environment allowing the system to be deployed where needed, whether on a private or commercial cloud. There are many suitable virtualization platforms available, e.g., Xen [78], QEMU [79], and Oracle's VirtualBox. In our evaluation we run the Android x86 port [2] on VirtualBox [3]. To reduce its memory and storage demand, we build a customized version of Android x86, leaving out unnecessary components such as the user interface and built-in standard applications.

Our system has 6 types of VMs with different configurations of CPU and memory to choose from, which are shown in Table 4.1. The VM manager can automatically scale the computational power of the VMs and allocate more than one VM for a task depending on user requirements. The default setting for computation is only one VM with 1 CPU, 512MB memory, and 100MB heap size, which clones the data and applications of the phone and we call it the *primary server*. The primary server is always online, waiting for the phone to connect to it. The second type of VMs can be of any configuration shown in Table 4.1, which in general does not clone the data and applications of a specific phone and can be

[2] http://android-x86.org/
[3] http://www.virtualbox.org/

allocated to any user on demand - we call them the *secondary servers*. The secondary servers can be in any of these three states: *powered-off*, *paused*, or *running*. When a VM is in powered-off state, it is not allocated any resources. The VM in paused state is allocated the configured amount of memory, but does not consume any CPU cycles. In the running state the VM is allocated the configured amount of memory and also makes use of CPU.

The Client Handler, which is in charge of the connection between the client (phone) and the cloud, runs in the main server. The Client Handler is also in charge of the dynamic control of the number of running secondary servers. For example, if too many secondary VMs are running, it can decide to power-off or pause some of them that are not executing any task. Utilizing different states of the VMs has the benefit of controlling the allocated resources dynamically, but it also has the drawback of introducing latency by resuming, starting, and synchronizing among the VMs. From our experiments, we observe that the average time to resume one VM from the paused state is around 300ms. When the number of VMs to be resumed simultaneously is high (seven in our case), the resume time for some of the VMs can be up to 6 or 7 seconds because of the instant overhead introduced in the cloud. We are working on finding the best approach for removing this simultaneity and staying in the limit of 1s for total resume time. When a VM is in powered-off state, it takes on average 32s to start it, which is very high to use for methods that run in the order of seconds. However, there are tasks that take hours to execute on the phone (e.g., virus scanning), for which it is still reasonable to spend 32s for starting the new VMs. A user may have different QoS requirements (e.g. completion time) for different tasks at different times, therefore the VM manager needs to dynamically allocate the number of VMs to achieve the user expectations.

To make tests consistent, in our environment all the VMs run on the same physical server which is a large multicore system with ample memory to avoid effects of CPU or memory congestion.

4.3.3 Automatic Parallelization

Parallel execution can be exploited much more efficiently on the cloud than on a smartphone, either using multiprocessor support or splitting the work among multiple VMs. Our

approach for parallelization considers intervals of input values. It is particularly useful in two main types of computationally expensive algorithms:

- Recursive algorithms or ones that can be solved using *Divide-and-Conquer* method. They are often based on constructing a solution when iterating over a range of values of a particular variable, allowing sub-solution computation to be parallelized (e.g. the classic example of 8-queens puzzle, which we present in 4.5.2).

- Algorithms using a lot of data. For example, a face recognition application requires comparison of a particular face with a large database of pre-analyzed faces, which can be done on a distributed database on the cloud much more easily than just on a phone. Again, this allows to split computations based on the intervals of data to be analyzed on each clone.

We provide results of such parallelization in the evaluation section (§4.5).

4.4 Profiling

Profilers are a critical part of the ThinkAir framework: the more accurate and lightweight they are, the more correct offloading decisions can be made, and the lower overhead is introduced. The profiler subsystem is highly modular so that it is straightforward to add new profilers. The current implementation of ThinkAir includes three profilers (hardware, software, and network), which collect variant data and feed into the energy estimation model.

For efficiency we use Android *intents* to keep track of important environmental parameters which do not depend on program execution. Specifically, we register listeners with the system to track battery levels, data connectivity presence, and connection types (WiFi, cellular, etc.) and subtypes (GPRS, UMTS, etc.). This ensures that the framework does not spend extra time and energy polling the state of these factors.

4.4.1 Hardware Profiler

The Hardware Profiler feeds hardware state information into the energy estimation model, which is considered when an offloading decision is made. In particular, CPU and screen have to be monitored [4]. We also monitor the WiFi and 3G interfaces. The following states are monitored by the Hardware Profiler in current ThinkAir framework.

- *CPU*. The CPU can be *idle* or have a utilization from 1–100% as well as different frequencies;
- *Screen*. The LCD screen has a brightness level between 0–255;
- *WiFi*. The power state of WiFi interface is either *low* or *high* (Figure 4.3);
- *3G*. The 3G radio can be either *idle*, or in use with a *shared* or *dedicated* channel (Figure 4.4).

4.4.2 Software Profiler

The Software Profiler tracks a large number of parameters concerning program execution. After starting executing a remoteable method, whether locally or remotely, the Software Profiler uses the standard Android Debug API to record the following information.

- Overall execution time of the method;
- Thread CPU time of the method, to discount the affect of pre-emption by another process;
- Number of instructions executed [5];
- Number of method calls;
- Thread memory allocation size;
- Garbage Collector invocation count, both for the current thread and globally.

[4] We leave the screen always ON during the experiments, since simply turning it OFF during offloading would be too intrusive to users.

[5] This requires an adaptation of the distributed kernel due to what we believe is a bug in the OS using cascading profilers leading to inconsistent results and program crashes.

4.4.3 Network Profiler

This is probably the most complex profiler as it takes into account many different sets of parameters, by combining both intent and instrumentation-based profiling. The former allows us to track the network state so that we can e.g., easily initiate re-estimation of some of the parameters such as RTT on network status change. The latter involves measuring the network RTT as well as the amount of data that ThinkAir sends/receives in a time interval, which are used to estimate the *perceived network bandwidth*. This includes the overheads of serialization during transmission, allowing more accurate offloading decisions to be taken.

In addition, the Network Profiler tracks several other parameters for the WiFi and 3G interfaces, including the number of packets transmitted and received per second, uplink channel rate and uplink data rate for the WiFi interface, and receiving and transmitting data rate for the 3G interface. These measurements enable better estimation of the current network performance being achieved.

4.4.4 Energy Estimation Model

A key parameter for offloading policies in ThinkAir is the effect on energy consumption. This requires dynamically estimating the energy consumed by methods during execution. We take inspiration from the recent PowerTutor [80] model which accounts for power consumption of CPU, LCD screen, GPS, WiFi, 3G, and audio interfaces on HTC Dream and HTC Magic phones. PowerTutor indicates that the variation of estimated power on different types of phones is very high, and presents a detailed model for the HTC Dream phone which is used in our experiments. We modify the original PowerTutor model to accommodate the fact that certain components such as GPS and audio have to operate locally and cannot be migrated to the cloud (Table 4.2).

By measuring the power consumption of the phone under different cross products of the extreme power states, PowerTutor model further indicates that the maximum error is 6.27% if the individual components are measured independently. This suggests that the sum of independent component-specific power estimates is sufficient to estimate overall system power consumption.

Model	$(\beta_{uh} \times freq_h + \beta_{ul} \times freq_l) \times util + \beta_{CPU} \times CPU_{on}$ $+ \beta_{Wifi_l} \times Wifi_l + \beta_{Wifi_h} \times Wifi_h$ $+ \beta_{3G_{idle}} \times 3G_{idle} + \beta_{3G_{FACH}} \times 3G_{FACH}$ $+ \beta_{3G_{DCH}} \times 3G_{DCH} + \beta_{br} \times brightness$		
Category	System variable	Range	Power coefficient
CPU	util	$1-100$	β_{uh} : 4.32 β_{ul} : 3.42
	freq_l, freq_h	$0,1$	n.a.
	CPU_{on}	$0,1$	β_{CPU} : 121.46
WiFi	npackets, R_{data}	$0-\infty$	n.a.
	$\text{R}_{channel}$	$1-54$	β_{cr}
	Wifi_l	$0,1$	β_{Wifi_l} : 20
	Wifi_h	$0,1$	$\beta_{Wifi_h} :\approx 710$
Cellular	data_rate	$0-\infty$	n.a.
	downlink_queue	$0-\infty$	n.a.
	uplink_queue	$0-\infty$	n.a.
	3G_{idle}	$0,1$	$\beta_{3G_{idle}}$: 10
	3G_{FACH}	$0,1$	$\beta_{3G_{FACH}}$: 401
	3G_{DCH}	$0,1$	$\beta_{3G_{DCH}}$: 570
LCD	brightness	$0-255$	β_{br} : 2.40

Table 4.2: Modified PowerTutor model for the HTC Dream Phone, dropping accounting for GPS and audio energy consumption.

Using this approach we devise a method with only minor deviations from the results obtained by PowerTutor. We implement this energy estimation model inside the ThinkAir Energy Profiler and use it to dynamically estimate the energy consumption of each running method.

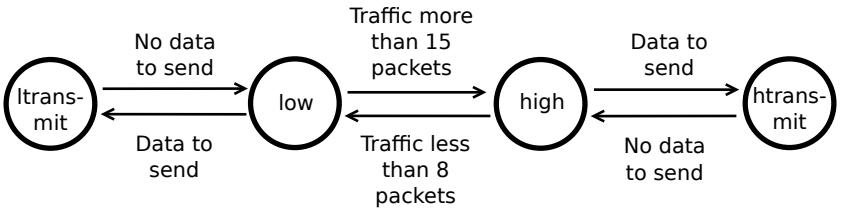


Figure 4.3: WiFi interface power states.

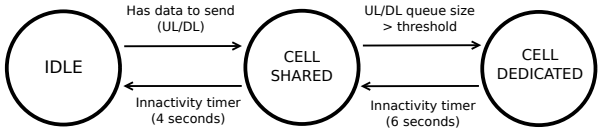


Figure 4.4: 3G interface power states.

4.5 Evaluation

We evaluate ThinkAir using two sets of experiments. The first is adapted from the Great Computer Language Shootout [6], which was originally used to perform a simple comparison of Java *vs.* C++ performance, and therefore serves as a simple set of benchmarks comparing local *vs.* remote execution. The second set of experiments uses four applications for a more realistic evaluation: an instance of the *N*-queens problem, a face detection program, a virus scanning application, and an image merging application.

To evaluate, we define the *boundary input value* (BIV) as the minimum value of the input parameter for which offloading would give a benefit. We use the *execution time policy* throughout, so for example when running *Fibonacci(n)* under the execution time profile, we find a BIV of 18 when the phone connects to the cloud through WiFi, i.e., the execution of *Fibonacci(n)* is faster when offloaded for $n \geq 18$ (cf. Table 4.3). We run the experiments under four different connectivity scenarios as follows.

- *Phone*. Everything is executed on the phone;
- *WiFi-Local*. The phone directly connects to a WiFi router attached to the cloud server via WiFi link;

[6] http://kano.net/javabench/

- *WiFi-Internet*. The phone connects to the cloud server using a normal WiFi access point via the Internet;
- *3G*. The phone is connected to the cloud using 3G data network.

Every result is obtained by running the program 20 times for each scenario and averaging; there is a pause of 30 seconds between two consecutive executions. The typical RTT of the 3G network that we use for the experiments is around 100ms and that for the WiFi-Local is around 5ms. In order to test the performance of ThinkAir with different quality of WiFi connection, we use both a very good dedicated residential WiFi connection (RTT 50ms) and a commercial WiFi hotspot shared by multiple users (RTT 200ms), which the device may encounter on the move, for the WiFi-Internet setting. We do not find any significant difference for these two cases, and hence we simplify them to a single case except for the full application evaluations.

4.5.1 Micro-benchmarks

Originally used for a simple Java *vs*. C++ comparison, each of these benchmarks depends only on a single input parameter, allowing for easier analysis. Our results are shown in Table 4.3. We find that, especially for operations where little data needs to be transmitted, network latency clearly affects the boundary value, hence the difference between boundary values in the case of WiFi and 3G network connectivity. This effect is also noted by Cloudlets [74]. We also include computational complexity of the core parts of the different benchmarks to show that, with growing input values, ThinkAir becomes more efficient. Note that there are large constant factors hidden by the O notation, hence the different BIVs with the same complexity.

4.5.2 Application benchmarks

We consider four complete benchmark applications representing more complex and compute intensive applications: a solver for the classic N-Queens problem, a face detection application, a Virus scanning application, and an application that combines two pictures into an unique large one.

Benchmark	BIV		Complexity	Data (bytes)	
	WiFi	3G		Tx	Rx
Fibonacci	18	19	$O(2^n)$	392	307
Hash	550	600	$O(n^2 log(n))$	383	293
Methcall	2500	3100	$O(n)$	338	297
Nestedloop	7	8	$O(n^6)$	349	305

Table 4.3: Boundary input values of benchmark applications, with WiFi and 3G connectivity, and the complexity of algorithms.

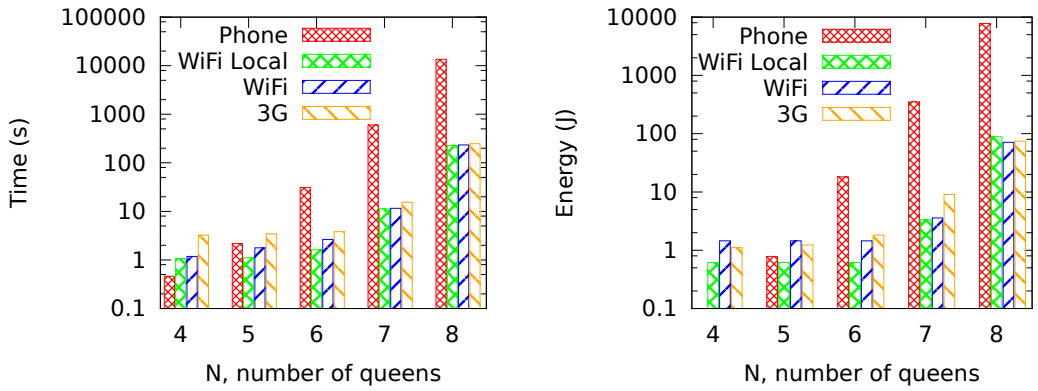


Figure 4.5: Execution time and energy consumption of the N-queens puzzle, $N = \{4,5,6,7,8\}$.

N-Queens Puzzle

In this application, we implement the algorithm to find all solutions for the N-Queens Puzzle and return the number of solutions found. We consider $4 \leq N \leq 8$ since at $N = 8$ the problem becomes very computationally expensive as there are 4,426,165,368 (i.e., 64 choose 8) possible arrangements of eight queens on a 8×8 board, but only 92 solutions. We apply a simple heuristic approach to constrain each queen to a single column or row. Although this is still considered as a brute force approach, it reduces the number of possibilities to just $8^8 = 16,777,216$. Figure 4.5 shows that for $N = 8$, the execution on the phone is unrealistic as it takes hours to finish. However, we consider the problem a suitable benchmark because many real problems get solved in a brute-force fashion.

Figure 4.5 shows the time taken and the energy consumed. We notice that the BIV

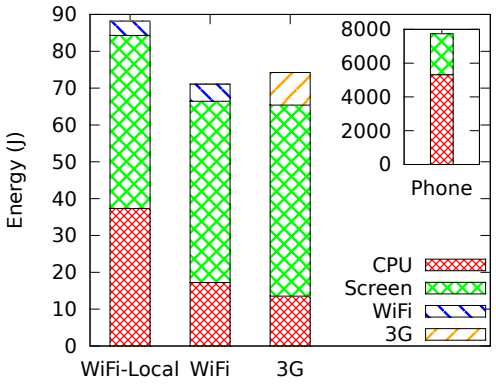


Figure 4.6: Energy consumed by each component when solving 8-queens puzzle in different scenarios.

is between 5: for higher N, both the time taken and energy consumed in the cloud are less than that on the phone. In general, WiFi-Local is the most efficient offload method as N increases, probably because the higher bandwidths lead to lower total network costs. Ultimately though, computation costs come to dominate in all cases.

Figure 4.6 breaks down the energy consumption between components for $N = 8$. As expected, when executing locally on the phone, the energy is consumed by the CPU and the screen: the screen is set to 100% brightness and the CPU runs at the highest possible frequency. When offloading, some energy is consumed by the use of the radio, with a slightly higher amount for 3G than WiFi. The difference in CPU energy consumed between WiFi and WiFi-Local is due to the difference of the CPU speed of the local device and cloud servers.

Face Detection

We port a third party program [7] towards a simple face detection program that counts the number of faces in a picture and computes simple metrics for each detected face (e.g., distance between eyes). This demonstrates that it is straightforward to apply the ThinkAir framework to existing code. The actual detection of faces uses the Android API `FaceDetector`, so this is an Android optimized program and should be fast even on the

[7] http://www.anddev.org/quick_and_easy_facedetector_demo-t3856.html

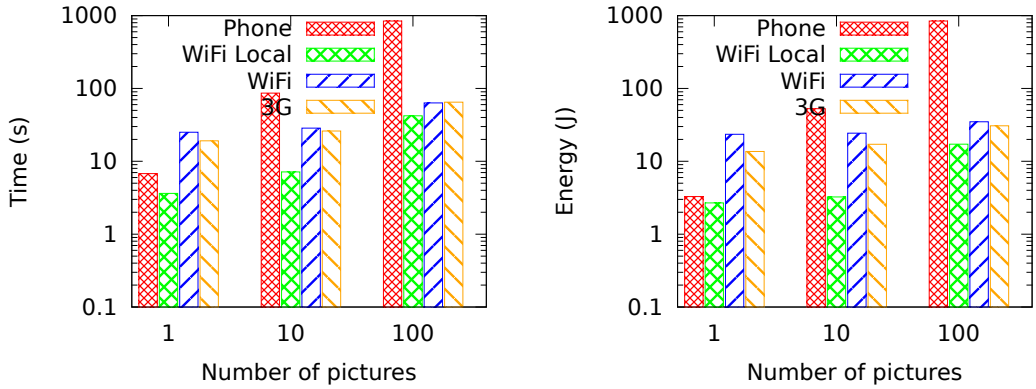


Figure 4.7: Execution time and energy consumed for the face detection experiments.

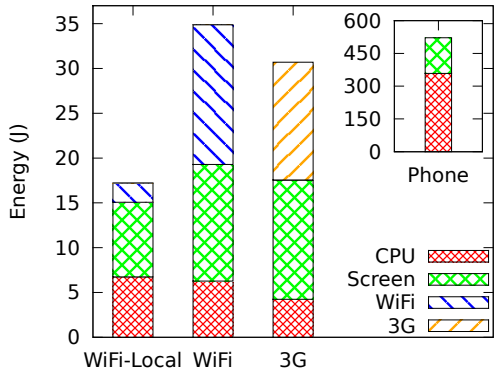


Figure 4.8: Energy consumed by each component for face detection with 100 pictures in different scenarios.

phone. We consider one run with just a single photo and runs with comparing the photo against multiple (10 and 100) others, which have previously been loaded into the cloud e.g., comparing against photos from a user's Flickr account. When running over multiple photos, we use the return values of the detected faces to determine if the initial single photo is duplicated within the set. In all cases, the execution time and energy consumed are much lower than that in the cloud.

Figure 4.7 shows the results for the face detection experiments. The single photo case actually runs faster on the phone than offloading if the connectivity is not the best: it is a native API call on the phone and quite efficient. However, as the number of photos being processed increases, and in any case when the connectivity has sufficiently high bandwidth

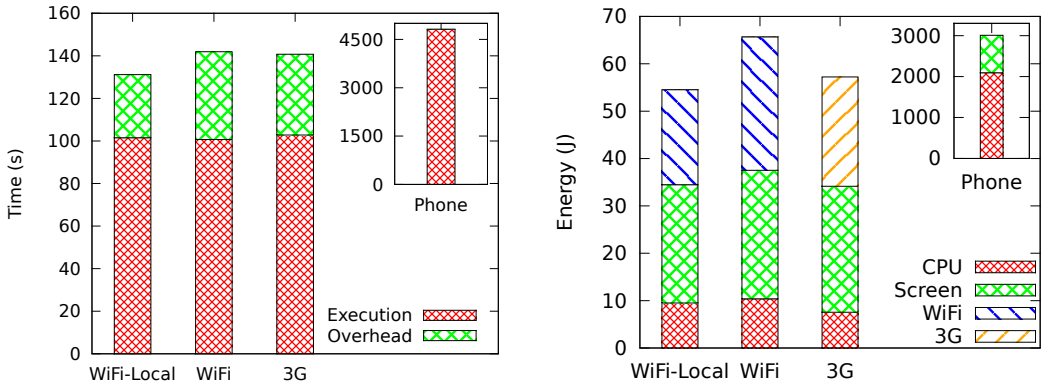


Figure 4.9: Execution time and energy consumption of the virus scanning in different scenarios.

and low latency, the cloud proves more efficiency. Figure 4.8 shows the breakdown of the energy consumed among components. Similar to the 8-Queens experiment results shown in Figure 4.6, the increased power of the cloud server makes the offloaded cases dramatically more efficient than that when everything is run locally on the phone.

Virus Scanning

We implement a virus detection algorithm for Android, which takes a database of 1000 virus signatures and the path to scan, and returns the number of viruses found. In our experiments, the total size of files in the directory is 10MB, and the number of files is around 3,500. Figure 4.9 shows that the execution on the phone takes more than one hour to finish, while less than three minutes if offloaded. As the data sent for offloading is larger compared to previous ones, the comparison of the energy consumed by the WiFi and 3G is more fair. As a result we observe that WiFi is less energy efficient per bit transmitted than 3G, which is also supported by the face detection experiment (Figure 4.8). Another interesting observation is related to the energy consumed by the CPU. In fact, from the results of all the experiments we observe that the energy consumed by the CPU is lower when offloading using 3G instead of WiFi.

Images Combiner

The intention of this application is to address the problem that some applications cannot be run on the phone due to lack of resources other than CPU, such as, the Java VM heap size is a big constraint for Android phones. If one application exceeds 16MB [8] of the allocated heap, it throws an `OutOfMemoryError` exception [9]. Working with bitmaps in Android can be a problem if programmers do not pay attention to memory usage. In fact, our application is a naïve implementation of combining two images into a bigger one. The application takes two images of size (w_1, h_1), (w_2, h_2) as input, allocates memory for another image of size $(\max\{w_1, w_2\}, \max\{h_1, h_2\})$, and copies the content of each original image into the final one. The problem here arises when the application tries to allocate memory for the final image, resulting in `OutOfMemoryError` and making the execution aborted. We are able to circumvent this problem by offloading the images to the cloud clone and explicitly asking for high VM heap size. First, the clone tries to execute the algorithm. When it does not have enough free VM heap size the execution fails with `OutOfMemoryError`. It then resumes a more powerful clone and delegates the job to it. In the meantime, the application running on the phone frees the memory occupied by the original images, and waits for the final results from the cloud.

4.5.3 Parallelization with Multiple VM Clones

In the previous subsection we showed that the ThinkAir framework can scale the processing power up by resuming more powerful clones and delegating the task to them. Another way of achieving the scaling of the processing power of a client is to exploit parallel execution. We have previously described how we expect to split parallelizable applications to tasks by using intervals of input parameters. In this section, we discuss the performance of three representative applications, 8-Queens, Face Detection with 100 pictures, and Virus Scanner, using multiple cloud VM clones.

Our experiment is setup as follows. A single primary server communicates with the

[8] http://developer.android.com/reference/android/app/ActivityManager.html#getMemoryClass

[9] The maximum heap size can be configured from the phone producers, so it can be different from 16MB, which is the default on the Android API

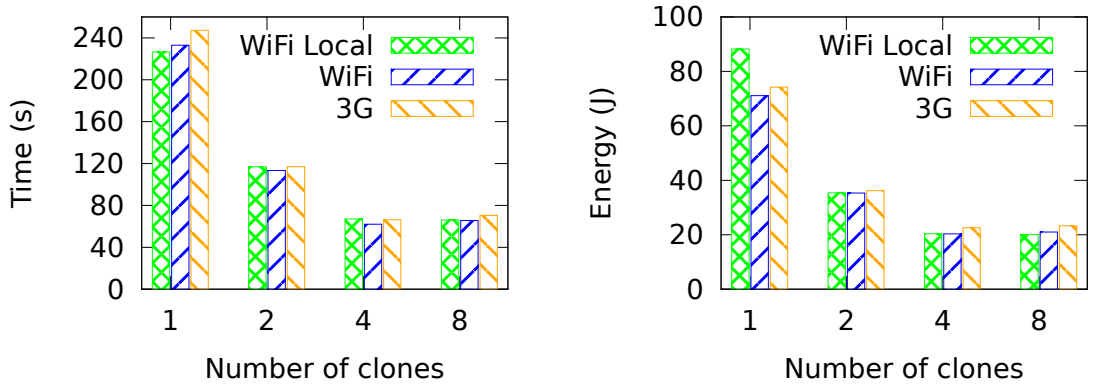


Figure 4.10: Time taken and energy consumed on the phone executing 8-queens puzzle using $N = \{1,2,4,8\}$ servers.

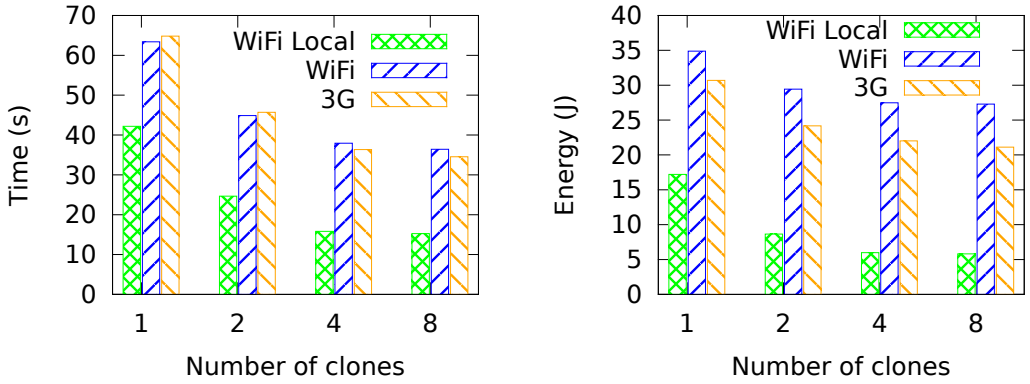


Figure 4.11: Time taken and energy consumed for face detection on 100 pictures using $N = \{1,2,4,8\}$ servers.

client and k secondary clones, where $k \in \{1,3,7\}$. When the client connects to the cloud, it communicates with the primary server which in turn manages the secondaries, informing them that a new client has connected. All interactions between the client and the primary are as usual, but now the primary behaves as a (transparent) proxy for the secondaries, incurring extra synchronization overheads. Usually the secondary clones are kept in pause state to minimize the resources allocated. Every time when the client asks for service requiring more than one clone, the primary server resumes the needed number of secondary clones. After the secondaries finish their jobs, they are paused again by the primary server.

The modular architecture of the ThinkAir framework allows programmers to implement

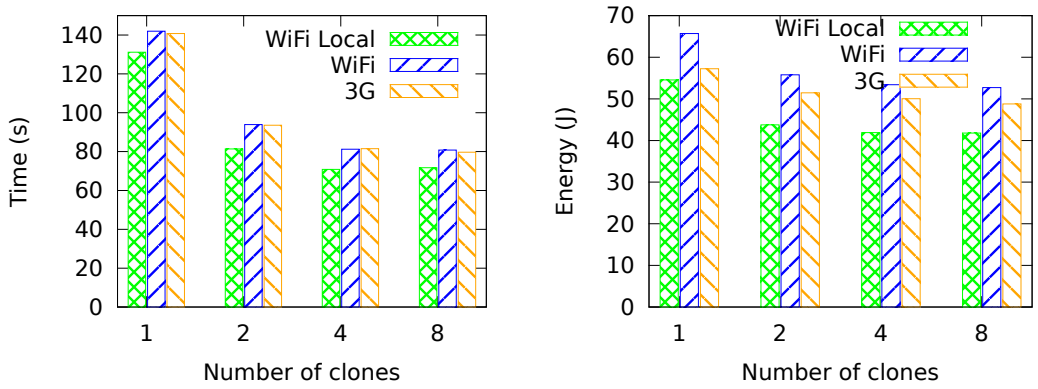


Figure 4.12: Time taken and energy consumed for virus scanning using $N = \{1,2,4,8\}$ servers.

many parallel algorithms with no modification to the ThinkAir code. In our experiments, as the tasks are highly parallelizable, we evenly divide them among the secondaries.

In the 8-Queens puzzle case, the problem is split by allocating different regions of the board to different clones and combining the results. For the face detection problem, the 100 photos are simply distributed among the secondaries for duplicates detection. In the same way, the files to be scanned for virus signatures are distributed among the clones and each clone runs the virus scanning algorithm on the files allocated. In all experiments, the secondary clones are resumed from the paused state, and the resume time is included in the overhead time, which in turn is included in the execution time.

Figure 4.10, 4.11, and 4.12 show the performance of the applications as the number of clones increases. In all 3 applications, the 4-clone case obtains the most performance benefits, since synchronization overheads start to outweigh the running costs as the regions which the board has been divided to become very small. We can also see that the increased input size makes the WiFi less efficient in terms of energy compared to 3G, which again supports our previous observations.

4.6 Discussion

ThinkAir currently employs a conservative approach for data transmissions, which is obviously suboptimal as not all instance object fields are accessed in every method and so do

not generally need to be sent. We are currently working on improving the efficiency of data transfer for remote code execution, combining static code analysis with data caching. The former eliminates the need to send and receive data that is not accessed by the cloud. The latter ensures that unchanged values need not be sent, in either direction, repeatedly. This could be further combined with speculative execution to explore alternative execution paths for improved caching. Note that these optimization would need to be carefully applied however, as storing the data between calls and checking for changes has large overheads on its own.

ThinkAir assumes a trustworthy cloud server execution environment: when a method is offloaded to the cloud, the code and state data are not maliciously modified or stolen. We also currently assume that the remote server faithfully loads and executes any code received from clients although we are currently working on integrating a lightweight authentication mechanism into the application registration process. For example, a device agent can provide UI for the mobile user to register the ThinkAir service before she can use the service, generating a shared secret based on user account or device identity.

Privacy-sensitive applications may need more security requirements than authentication. For example, if a method executed in cloud needs private data from the device, e.g., location information or user profile data, its confidentiality needs to be protected during transmission. We plan to extend our compiler to support `SecureRemoteable` class to support these security properties automatically and release the burden from application developers.

4.7 Conclusions

We present ThinkAir, a framework for offloading mobile computation to the cloud. Using ThinkAir requires only simple modifications to an application's source code coupled with use of our ThinkAir tool-chain. Experiments and evaluations with micro benchmarks and computation intensive applications demonstrate the benefits of ThinkAir for profiling and code offloading, as well as accommodating changing computational requirements with the ability of on-demand VM resource scaling and exploiting parallelism. We are continuing the development of several key components of ThinkAir: we have ported Android to Xen

allowing it to be run on commercial cloud infrastructure (Chapter 5), and we continue to work on improving programmer support for parallelizable applications. Furthermore, we see improving application parallelization support as a key direction to use the capabilities of distributed computing of the cloud.

Chapter 5

Clone2Clone (C2C): Enable Peer-to-Peer Networking of Smartphones on the Cloud

In this chapter we push the smartphone-cloud paradigm to a further level: We develop Clone2Clone (C2C), a distributed platform for cloud clones of smartphones. C2C associates a software clone on the cloud to every smartphone and interconnects the clones in a p2p fashion exploiting the networking service within the cloud. It shows the dramatic performance improvement that is made possible by offloading *communication* between smartphones on the cloud. Along the way toward C2C, we study the performance of device-clones hosted in various virtualization environments in both private (local servers) and public (Amazon EC2) clouds. We build the first Amazon Customized Image (AMI) for Android-OS—a key tool to get reliable performance measures of mobile cloud systems—and show how it boosts up performance of Android images on the Amazon cloud service. We then design, build, and implement Clone2Clone. Upon C2C we build CloneDoc, a secure real-time collaboration system for smartphone users. We measure the performance of CloneDoc on a testbed of 16 Android smartphones and clones hosted on both private and public cloud services and show that C2C makes it possible to implement distributed execution of advanced p2p services in a network of mobile smartphones.

The results presented in this chapter appear in [7].

5.1 The need for p2p smartphone networking

A wireless p2p network between smartphones is hard to realize: Smartphones connect to Internet through carrier private IP addresses. The cellular network base stations are responsible for distributing private IP addresses and translating them (through NAT) to public IPs. As devices move from one network cell to the other both private and public IPs change. So, the only way to open a p2p connection with a smartphone is to make the smartphone itself start a connection with a known IP address, and to keep it alive by always sending its current IP. This difficulty, combined with severe battery limitations and frequent loss of cellular coverage or Internet connections (subways, rural areas etc.) make p2p networking between smartphones almost impossible to realize. The C2C platform solves these problems—the clones are always on in the cloud and p2p connectivity is guaranteed by the high-bandwidth network of the cloud. C2C can help offload heavy mobile computational tasks by adopting techniques similar to ThinkAir [6], MAUI [20], CloneCloud [19], and SociableSense [67]. Most importantly, it helps realize communication offload among smartphones. In this way, C2C enables innovative services such as content sharing, search, and distributed execution among the users, and it makes possible to build p2p-based protocols for smartphones without the need of relying on a continuous connection between real devices.

On top of C2C we implement CloneDoc, a real-time collaboration system for smartphone users that work simultaneously on the same document. CloneDoc follows the motivation of DEPOT [81], Venus [82] and SPORC [83]: Creative applications such as concurrent group collaboration among users reading and editing some shared state (e.g. a document) should not need sacrifice privacy and security to exist. DEPOT [81] targets large files and is not suited for real-time group collaboration. Venus [82] does not support applications other than key-value storage. SPORC [83] targets real-time collaboration systems and relies on a single server to give a global order to concurrent user requests. SPORC uses Fork* consistency [84, 85, 86, 87, 88, 89, 90] to prevent misbehaving server from equivocating about the order of operations. Then, as in Google Wave [91], Operational Transformation [92, 93, 94, 95] is used to both recover from malicious forks and to bring users to the same shared state, even though the order of the operations applied locally by

each user is not the same.

These protocols necessitate p2p networking among the users to exchange a large number of messages, or public key cryptography to sign and verify messages required by the protocol, or both. Therefore, they are not suited for battery-limited smartphones. Conversely, we build our CloneDoc system upon the C2C platform. CloneDoc offloads cryptographic operations and p2p communication on the clones running on the cloud, thus alleviating the real device of many tasks. As a result, the C2C platform makes it possible to run non-trivial group collaboration systems and other communication and computation intensive distributed applications on networks of smartphones.

Our experiments show that CloneDoc can execute on smartphones by using a very limited amount of energy, thus demonstrating that C2C enables a whole new class of distributed applications on mobile devices.

Our contribution in this work is as follows:

- We study the performance of clones hosted on different virtualization environments (QEMU emulator [79], VirtualBox, Xen[1]) on both private (local servers) and public cloud (the Amazon EC2 platform).
- We are the first to build the *Ax86AMI bundle*: A *customized Amazon AMI (Amazon Machine Image)* for the *Android-x86 OS*. The Ax86AMI bundle boosts up performance of Android clones on Amazon's public cloud platform and is a fundamental tool to get reliable performance measures for mobile cloud systems.
- We design, build and implement the Clone2Clone (C2C) platform which exploits the most performing cloning methods as demonstrated by our study.
- On top of C2C we design, build, and implement CloneDoc, a system that makes *secure* and *real-time* group collaborative editing of files in a network of smartphones possible.
- We measure the performance of CloneDoc through a testbed of 16 Android phones connected wirelessly to their clones in the cloud—both local servers and Amazon's EC2.

[1] http://www.virtualbox.org, http://xen.org

This chapter is organized as follows: We explain the motivation and the intuition behind the C2C platform in Section 5.2. In Section 5.2.2 we show how to clone mobiles on both private and public cloud platforms and the challenges we faced while building the customized Amazon AMI for the Android-x86 OS. Then, in Section 5.3, we present the C2C design and architecture. Section 5.4 shows how to design and implement the CloneDoc, a secure and real-time group collaboration system, on top of C2C and the experimental results with the real testbed of 16 Android smartphones. These experiments confirm the energy saving of the C2C based protocols against protocols that do not make use of C2C. We then conclude with Section 5.5.

5.2 The C2C platform

5.2.1 Motivation and goals

Since their first introduction in the market, smartphones have changed the way we think of our mobiles. They are small, handy devices that outperform the most powerful desktops of just a few years ago in both computing power and memory. Nonetheless, even now, p2p networking protocols running on smartphones are still considered an utopia. Not because of lack of data—smartphones generate and handle an enormous amont of data (media, docs, etc.). Not because of the lack of networking capabilities—smartphones are equipped with WiFi and 3G technology. But because of the battery limit, that is the real obstacle in realizing systems that have a significant overhead in terms of computation and, especially, communication.

This is where our platform comes into the picture: C2C associates each smartphone with a software clone on the cloud (either private or public) and interconnects the clones in a p2p fashion using the networking service of the cloud. The peers (clones) of the C2C network clearly do not suffer of battery limitations unlike their mobile alter-egos—they are running in the cloud. Moreover, the high-bandwidth p2p network provided by the cloud has excellent availability and certainly does not have 3G or WiFi coverage problems. In the presence of C2C, whenever a user has to execute a heavy job on her device, she might

delegate the job to the clone by using methods like ThinkAir [6] or CloneCloud [19] (independently from other users), or can distribute the job to more clones of the C2C network for faster execution. Most importantly, C2C allows to offload communication between smartphones: A large file that is to be sent from smartphone *A* to many other smartphones can be uploaded to clone *A* on the cloud. Then, any other smartphone will seamlessly download the file through clone *A*, without involving the real device *A*. This process clearly alleviates battery consumption on the smartphone. C2C opens up the way for applications that enhance user experience such as distributed search over the C2C network, content sharing among users, and other applications that would be otherwise impossible on battery limited smartphones.

5.2.2 How to clone on the cloud

To realize a fully working version of C2C we make use of Android devices, being the Android OS open source. The first step towards the C2C platform is to generate a full operational clone for each smartphone on the cloud. To this end we consider two strategies: (1) Run the clone on a private cloud, or, (2) run the clone on a public (commercial) cloud computing service. The private cloud is made of our local servers running Unix/Linux (see Table 5.1 for more details). As for the public cloud service, we opt for the Amazon Elastic Compute Cloud (Amazon EC2) platform. As we will see, private and public cloud clones differ in performance. We use standard benchmarks to give precise evidence on the best performing clones in terms of CPU, I/O, and networking. This allows us to use the best cloning methodology on both local and public cloud for our C2C platform.

Note that, though is not necessary for the clones to be images of the Android OS running on the cloud—they could be Linux x86 OS running JVM applications—from a software engineering point of view running Android OS images makes it straightforward to install/uninstall user apps on the clone and use them for offloading.

Platform	Description	CPU	Memory
Amazon	High-CPU Med.Inst.	2 virtual cores 2.5 EC2	1.7 GB
Local	Intel(R)T5600	Core(TM)2 @ 1.83 GHz	1 GB

Table 5.1: Specification of Testing Servers.

5.2.3 Android clones: The private cloud case

The main hardware platform for Android OS devices is the ARM architecture. However, the Android-x86 project[2]–an Android port to the x86 architecture–makes it possible to run Android OS on other platforms like netbooks and ultra-mobile PCs. Also is it possible to run Android-x86 directly on workstation/servers. Even so, due to security concerns, it is a customary and reasonable choice to run Android-x86 as a virtualized OS (the clones are deployed in a controlled, virtualized environment independent from other clones). We consider three different virtualizing methods: Xen, VirtualBox, and by using QEMU [79]. Xen and VirtualBox are virtualizing environments. QEMU emulates a full computer system including peripherals and the Android Emulator runs on top of it.

We use CPU, I/O, and networking benchmarks to compare the performance of clones. More specifically, we are interested in measuring the performance at the application level. In Android, applications run on top of Dalvik, the Android Java Virtual Machine. So, for CPU performance we use the Java version of Linpack[3]. Linpack measures the number of floating point operations per second (Mflops). Rather, for I/O performance, we measure the number of bytes read/written on file per second. We use bufferized access to file through the standard Java API. During the test the clone makes use of one processor core. Each experiment is repeated 100 times—the average performance of each method is shown in Figure 5.1. VirtualBox clones outperform QEMU clones by a factor of 16 in terms of CPU Performance (MFlops) and by a factor of 18 (22) in terms of I/O Read (Write). This is not surprising—QEMU is an emulator. Xen clones are 1.47 times more CPU efficient and 1.3 (1.1) times more I/O Read (Write) efficient than VirtualBox clones. As a conclusion, Xen clones are the most performing among the methods we investigate for the private cloud

[2]http://www.android-x86.org/
[3]http://www.netlib.org/benchmark/linpackjava/

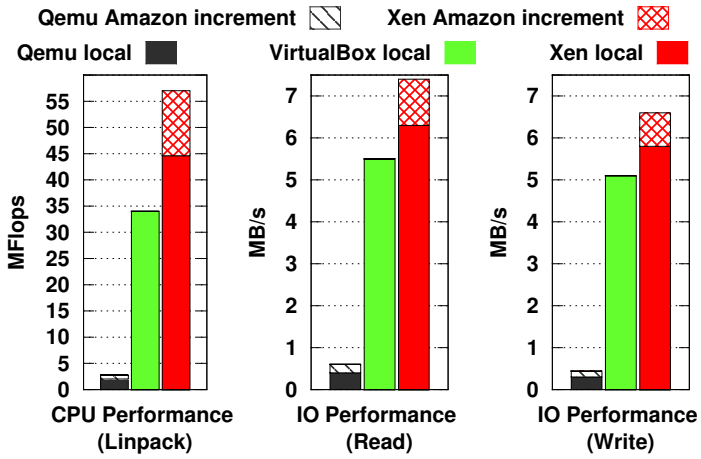


Figure 5.1: Benchmark performance on private and public cloud clones.

case.

5.2.4 Android clones: The public cloud case and the Android-x86 Amazon Machine Image (Ax86AMI)

The virtualization environment of the Amazon EC2 platform is Xen. On top of it run the so called Amazon Machine Images (AMIs). These images are special type of pre-configured operating systems and virtual application software used to create virtual machines within the Amazon Elastic Compute Cloud (EC2). One can either use a pre-configured AMI or create a customized one containing specific applications, libraries etc.

Unfortunately, an AMI for Android-x86 that runs on top of Xen does not exist. In addition, aside from the processor emulator QEMU [79], nor VirtualBox neither other virtualization environments are compatible with Amazon EC2. So, public cloud cloning so far is only possible by using the Android emulator which runs on top of QEMU. Note that QEMU runs on top of a virtualized Linux AMI, which in turn runs on top of Amazon's Xen. This is far from being efficient—there are two virtualization layers between Android and Xen—and experiments done in this way are not accurate. This is why we designed and built a *bundle*—a custom Amazon Machine Image for Android-x86 (*Ax86AMI*)[4]. Building

[4]Our Amazon Machine Image for Android-x86 is open source. If you need it to run experiments, just drop an email to one of the authors of this paper.

the Ax86AMI bundle is not an easy process for several reasons. Generally speaking, the process consists in making the Android system believe that it is actually running on a real mobile device and not in the cloud. Here we describe, step by step, the challenges we faced in building the bundle and how we overcame them.

Virtualization support

The Amazon EC2 platform is based on Xen. Xen uses a virtualization model known as para-virtualization—guests run a modified operating system using a special hypercall instead of certain architectural features. Therefore, any OS that needs to be run on it has to be properly modified at the kernel level so that it can support para-virtualization. To achieve this for the Android OS we built its kernel using a custom configuration file that enables Xen specific parameters.

No display, no bundle

The clone is not a real device. It is instead an emulator of the real smartphone on a virtualized environment. So, it lacks display and the related drivers that allow Android system services to interact with it. As a consequence, our modified Android-x86 kernel that supports Xen virtualization still cannot run properly. To understand why this is the case we shortly describe what happens during the boot of Android-x86 (see Figure 5.2): The bootloader loads the linux kernel and it starts the *Init* process. Init starts system demons and, most importantly, starts the *Zygote* process and the *runtime* process. Zygote initializes a Dalvik VM instance, and, among other tasks, it forks on request (of the runtime process) to create VM instances for managed processes. The runtime process initializes *Service Manager* that handles service registration and lookup. Then, the runtime process asks Zygote to start *System Server*, which starts the native system services one by one beginning with *Surface Flinger*—the abstraction for the 2D graphics engine. System server also starts *Activity Manager*, that finally starts all the other upper level services of the Android system.

Activity Manager depends on the native system services. If any of these services fails to start, Activity Manager dies ungracefully. This blocks the start of the rest of the upper level services, causing Zygote to die ungracefully as well. This is exactly what happens during

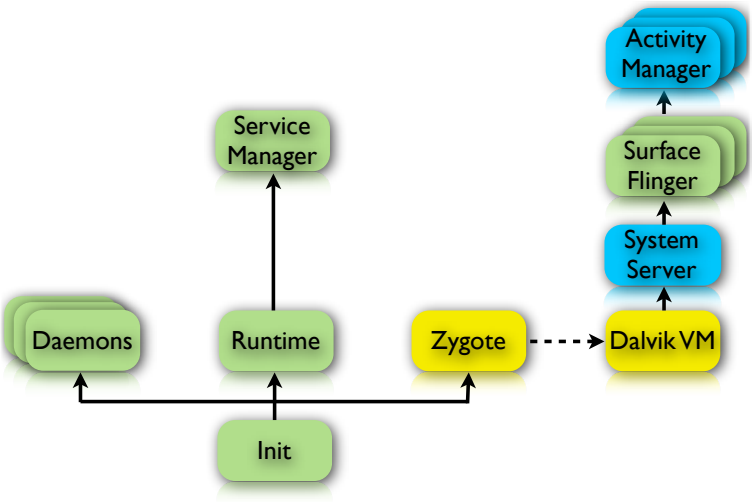


Figure 5.2: Flow of services started after boot in the Android System.

the boot on Xen: When the first of the native system services (Surface Flinger) is started, it asks the hypervisor (Xen) for the display driver. The clone is not running on a real device, so the display driver is missing. This causes Surface Flinger to die ungracefully. As a consequence, also Activity Manager dies ungracefully, before starting any other activity of the Android system. This brings down also System Server and the Zygote process. Then, Init is notified and it restarts Zygote, which restarts System Server. System Server tries again to boot the native system services, beginning from Surface Flinger, that again fails because the display drivers are missing. The above scenario is repeated again and again and Android does not start successfully. To overcome this problem we create Android Virtual Device Frame Buffer (AVDFB)—a virtual display driver through which all system services and activities access the graphic interface in our Ax86AMI bundle. To realize AVDFB we amend VFB[5], a virtual framebuffer device driver for Linux, as follows:

- We add APIs that allow for display access by Android services and activities.
- We modify options such as display dimension, resolution, etc. of VFB in such a way that they are standard for the Android system.
- We add buffering functionality: It buffers every incoming request, stores it in the

[5]http://www.kernel.org/doc/Documentation/fb/framebuffer.txt

dedicated memory, and gives a positive response to the requesting service/activity.

- Finally, we disable the *uvesafb* and *vesafb* drivers at kernel level and modify the Init load_modules script so that AVDFB is always loaded during the system boot.

Our customized frame buffer device driver AVDFB enables System Server to successfully start Surface Flinger and all the other services and activities that depend on it.

Sensors and other user interfaces

The cloned Android OS lacks of sensors present in the real device (e.g. microphone, camera, accelerator) and peripherals (e.g. blue-tooth, wireless, radio). Differently from the display, these interfaces are not crucial to the Android OS. Indeed, disabling the relative services does not block the successful boot of the system. What's more, this speeds up the boot of the system and makes it more lightweight. For these reasons, we choose to disable these services in our customized Ax86AMI bundle.

Preparing the final Amazon-runnable bundle

Before running our customized Android-x86 on the Amazon EC2 platform we need to package it in a bundle[6]—a custom AMI that respects Amazon's requirements. For this we compile the source code of our customized Android and create the .iso image. Then we extract its content in a directory, used as the root file system during the bundle creation process. Lastly, we modify the Init script so that the SDcard is mounted in read/write mode, instead of read_only—the way it is mounted by the original Android system source code. Now, the Ax86AMI bundle is ready to run on the Amazon EC2 platform.

Performance evaluation of public cloud clones

Here we present experimental results with Amazon clones. For the experiments we use a EC2 High-CPU Medium instance (see Table 5.1 for more details). We compare the performance of QEMU-clones with clones based on our Ax86AMI bundle. Again, during the test, the clone makes use of just one processor core. We use Linpack to measure CPU

[6]http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/UserProvidedkernels.html

performance and bufferized read/writes on files for I/O performance. Each experiment is repeated 100 times and the results are averaged and presented in Figure 5.1. Note that Amazon EC2 is incompatible with VirtualBox, so VirtualBox cannot be used.

By removing two virtualization layers and using our Ax86AMI bundle, we get an enormous boost of performance on the public cloud. Indeed, QEMU clones perform 23 times less in terms of CPU than Xen clones (the Ax86AMI bundle). Also the I/O performance is improved with the Ax86MI bundle. Our clones are more than 13 (13.5) times faster in performing I/O Read (Write) operations (see Fig. 5.1).

The results in Figure 5.1 also show that private cloud clones are outperformed by those on the public cloud. This is expected: Amazon's High-CPU Medium instances are more powerful than our local servers. However, keeping a clone always running on a commercial cloud platform has its cost. But so does the infrastructure and the maintenance of a private cloud. Latency is another important concern. Nonetheless, as we will see from the experiments, in networking applications like our CloneDoc this latency may be dominated by the latency due to the application itself (in the case of CloneDoc to coordinate with the remote peers) and thus is not a real problem. In any case, when deploying the clones, a tradeoff between performance, costs, and, communication latency has to be considered.

5.3 C2C: Architecture Design

To enable p2p networking among smartphone clones, the C2C platform needs a mechanism that "notifies" clones of the presence of others and gives information on how to connect to them. Here we stick to a simple and standard baseline architecture (see Figure 5.3). It includes a *directory service* (CloneDS in the figure) which takes care of mapping users to clones and clones to IPs. The CloneDS is always up and its IP is known (made public by e.g. the C2C platform builder). All the entities in the system—users, cloud providers, and the CloneDS—have a private/public key pair and they can securely verify the authenticity of public keys.

A user willing to join the C2C platform requires a virtual machine to her cloud provider of choice. The cloud provider generates the virtual machine with a standard Android-x86 image customized for C2C (the *clone*) and associates the machine with an IP address and a

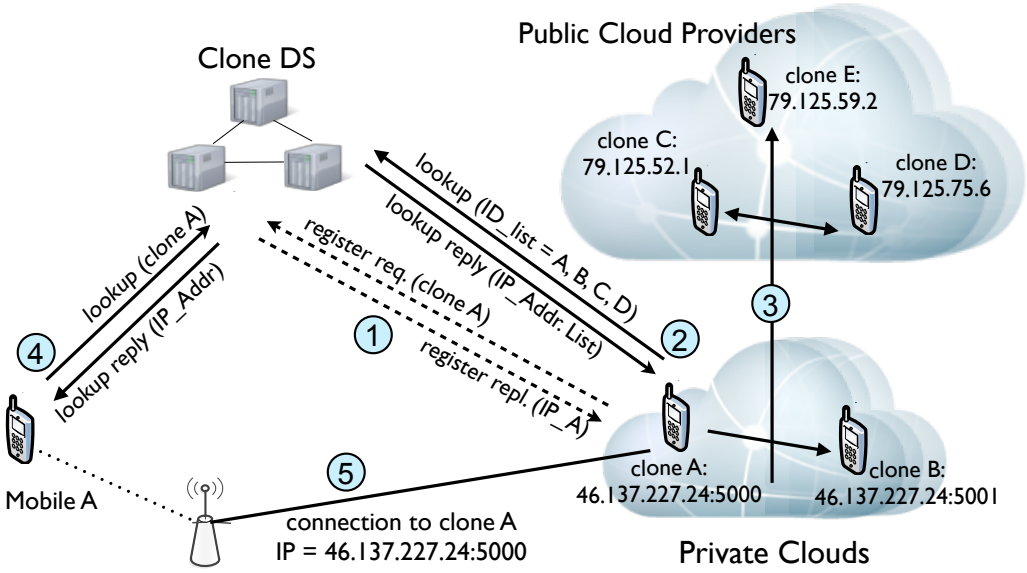


Figure 5.3: The C2C architecture and networking.The steps represent what happens in the system after the generation of a new clone.

private/public key pair. The public key is signed by the user, so that everybody can verify the owner of the clone. Then, the clone running in the cloud performs the following steps:

- *DS register:* The clone registers to the CloneDS (mobile A, Step 1 in Figure 5.3). The CloneDS maps the real user to the clone ID (clone A), its IP address, and its public key. This information is provided and signed by the clone itself.

- *DS lookup and C2C connect:* After clone A registers with the CloneDS, it receives a list signed by the CloneDS of the entries of the other clones in the platform through a lookup request to the CloneDS (step 2 in Figure 5.3). In this way, clone A learns the IP and the public key of the clone of its peers. Then, clone A starts p2p connections with the other clones (Step 3 in Figure 5.3).

- *User lookup:* If the mobile of user A fails, it can reconnect to her clone by getting its IP through a lookup at the CloneDS (step 4 in Figure 5.3).

- *User-clone connection:* The newly created clone is a default Android-x86 image running on the cloud, and does not contain any user data. The user is now ready to

connect to its clone through its public IP (Step 5 in Figure 5.3), and installs whatever she likes in her cloned devices, including apps that she already has in her real device. Moreover, the user negotiates a symmetric key with her clone (this is done in a standard way thanks to the authenticated private/public keys). This key will be used to encrypt and sign communication between the user and the clone.

The tasks handled by the CloneDS can be fulfilled in alternative ways. A possibility is to have an Internet service that provides the IP and the other info of the clones on request. Another way is to make the users send their info to other "friend" users by channels external to the C2C platform (e.g. email). The links on the C2C platform can also be bootstrapped from social network connections of the users on services like Facebook, or others. However, the presence of the CloneDS in the C2C platform has several benefits. The first is that it simplifies its architecture. In addition, it alleviates the user, the device, and the clone from providing the IP of the clone to the other clones.

5.3.1 Handling networking in C2C

The cloud is responsible for IP assignment to newly created clones, when asked by the user. In the current version of our architecture, when the clones reside on the Amazon EC2 platform, they are equipped with an EC2 Elastic IP address[7]. This address is generated simultaneously with the clone creation. Elastic IP addresses are mapped with the virtual machine rather than with the physical server. So, if the physical server fails and the clone is reallocated on another machine, the clone is again remapped to the same IP address. This process is internal of the Amazon EC2 service. So, is totally transparent to the user and to the C2C platform. Amazon does not charge for the Elastic IP addresses associated to a running instance (clone).

Clones can also reside in private clouds (see Figure 5.3) made of local servers. If this is the case, clones residing on the same machine are mapped to that machine's public IP, though to different ports (as in Figure 5.3).

[7] http://aws.amazon.com/articles/1346

5.3.2 C2C and security

In C2C, we assume that the users trust their own cloud provider. However, they do not trust the other users and their cloud providers. The interaction between the mobile device of a user and its clone is secured by using a shared symmetric key. In this way encryption of packets and signatures performed by the real device can be implemented efficiently.

In the architecture we described, the CloneDS is an external entity with respect to the cloud providers and is trusted by all the users in the system. Therefore, correct users can trust that the information provided by the CloneDS is consistent. Of course, malicious providers can forge information about their own users. But correct users and cloud providers can connect correctly. An alternative architecture for the CloneDS is to implement it as a distributed service among the cloud providers. In this distributed version, the CloneDS is replicated on the cloud providers, users connect to the replica on the cloud they trust, and the replicas are kept consistent by broadcasting signed updates among the cloud providers that are part of the system. This distributed alternative might be more available, since the cloud typically guarantees high availability.

5.4 CloneDoc: A mechanism for secure real-time collaboration

Secure group collaboration and file access among multiple clients can be efficiently deployed by making use of external servers running on the cloud [83, 81, 82, 90, 89]. These applications need p2p networking among the clients (that do not run on the cloud) and heavy cryptography to guarantee crucial security and system properties. This makes even SPORC [83], the state of the art of real-time collaboration systems, unfit for present-day smartphones. However, with the C2C platform the tables are turned. C2C delivers efficient p2p networking for smartphones by moving computation and, most importantly, communication on the cloud. We exploit these features of C2C and modify SPORC's architecture to build CloneDoc—The first energy-efficient and real-time collaboration system for battery constrained smartphones. We first give an overview of SPORC. Then we show the design of the CloneDoc architecture focussing on the differences with SPORC, and lastly

we compare the two systems experimentally. CloneDoc demonstrates that our idea of moving communication and enabling p2p networking on the cloud makes it possible to design energy-efficient non-trivial p2p applications for smart-phones.

5.4.1 Overview of SPORC

SPORC [83] is a system for group collaboration with an untrusted server. It allows users to generate a shared state called *document* and to edit it concurrently. SPORC guarantees that users see a coherent state of the document by using an untrusted server whose role is to force a global order on the concurrent users' operations. The server is potentially malicious—its goal may be to partition the clients in disjoint groups with different views of the document.

The main idea behind SPORC is as follows: The system is made of four states: (1) the *local state*—the client's current view of the document; (2) the *encrypted history*—the set of operations stored and ordered by the server; (3) the *committed history*—the set of plaintext operations shared among clients as ordered by the server; (4) the *pending queue*—the ordered list of each client's local operations that have already been applied to the local state, but that are still to be committed (they are still to get a global label by the server). When a client generates a new operation, the client first applies it to its local state and adds it to the end of its pending queue. Then the operation (encrypted and signed to prevent forgery) is sent to the untrusted server. The server gives a global sequence number to the operation, inserts it at the end of his encrypted history, and sends it to all the clients. Whenever the client that originally created the operation receives it back from the server, it extracts it from the pending queue and inserts it in its committed history. There is no need for this client to apply the operation to the local state since it has already been done before sending the operation to the server. However, the other clients cannot just apply the operation on the local state when they receive it from the server since this can lead to inconsistency. Let us explain why with the following example from [83, 84]: Suppose that two clients X and Y start from a common state "ABCDE" and locally apply $op_X = \mathrm{del}(4)$ and $op_Y = \mathrm{del}(2)$ respectively. X ends up with "ABCE" while Y with "ACDE". If they naively apply the other operation coming from the other user, X ends up with "ACE" and Y with "ACD".

Therefore, in SPORC clients use Operational Transformation (OT) [92, 93, 94, 95], which allows the execution of lock-free concurrent operations that preserve casual consistency and make the clients converge to a common shared state. When a client gets a labeled operation (not generated by himself), the client transforms the operation through OT mechanisms before applying it to the document. The transformation takes in consideration the optimistic updates due to local operations (generated by the client himself) yet to be labeled by the server. In the previous example, due to the OT mechanisms, Y transforms op_X and gets $op'_X = \text{del}(3)$, while X transforms op_Y and gets $op'_Y = \text{del}(2)$. This leaves both X and Y with the consistent state "ACE".

5.4.2 CloneDoc: System Architecture

CloneDoc's purpose is the same as SPORC's. Even though similar to SPORC, CloneDoc makes use of the C2C platform which introduces more complexity to the system overall, yet reducing battery consumption by liberating the mobile devices from many tasks. CloneDoc is a typical p2p application and not-so-light computation due to cryptography. Thus, we use it as a stress-test for the C2C platform.

The main idea in CloneDoc is to make the clone on the cloud receive operations from the mobile device, handle as many tasks as possible on the device's behalf and keep the device up-to-date. The clone (similarly to the client in SPORC) maintains two states: the *pending queue* and the *committed queue*. The clone behaves exactly as clients in SPORC: (1) submits to the server operations that he gets from the user's real-device and (2) appropriately transforms (using OT) the operations of the other users received from the server. It is thus responsible for correctly handling the queues so that its view of the document is coherent to that of other clones. Last, but not least, the clone sends back to the real device the operations such that the user's view is coherent to that of other users in the system. However, the real and the cloned device are not physically the same. This translates into an unavoidable delay in their communications that, if not appropriately managed, may introduce inconsistency.

Trust model

As in SPORC, in CloneDoc users trust their own cloud provider but they do not trust the other users and their cloud provider, except when they perform operations on a document on which they have the right access privileges. This trust model is coherent with the trust model of C2C, on top of which CloneDoc is implemented.

Clone–user consistency

We want a system that looks real-time to the user. So, the user's operations are applied optimistically on the device, before they are sent to the clone. The real device maintains the global label *seqNo* given by the server of the last operation it has seen, and a state, the *local pending queue*, containing all user operations optimistically applied by the device and not yet labeled by the server. Typically this queue is larger than the pending queue of the clone.

Similarly, the global label of the last operation in the clone's committed queue is typically larger than the one of the real device. The discrepancy can lead to inconsistency if not correctly handled. CloneDoc resolves the issue as follows: When the real device adds a new opportunistically applied operation in the pending queue, it sends it to the clone along with a local sequence number *usrSeqNo*. In addition, the message contains the seqNo of the last committed operation the device knows of. As soon as the clone gets the new operation, it first checks whether the seqNo sent by the user equals the seqNo of the last operation in the clone's committed queue. If not, it means that there exist committed operations submitted by other users that the real device has no knowledge of yet. If this is the case, the clone transforms the new operation past these and inserts it in the clone's pending queue. When the clone receives another user's operation labeled from the server, as in SPORC it transforms the operation past its pending queue. Then, when sending it (already transformed) to the real device, it also includes the usrSeqNo of the last operation in its pending queue. So, when the user gets the message, it is able to discriminate between operations in its pending queue known and unknown to the clone. The operation is further transformed past the operations unknown to the clone and applied to the document.

Number, type & OS	CPU	RAM
6×Samsung Galaxy S Plus (Android 2.3)	1.4 GHz Scorpion	512 MB
2×Nexus S (Android 4.0.1)	1 GHz Cortex-A8	512 MB
2×HTC Desire (Android 2.3)	1 GHz Scorpion	576 MB
6×HTC Hero (Android 2.1)	528 MHz ARM 11	288 MB

Table 5.2: Phone specifications.

Detecting misbehaviors

Differently from SPORC, in CloneDoc only the clones take part in the detection of misbehavior. Indeed, each time a new labeled operation is received from the server, all the checks on the sequence numbers, hash chains, etc. are done by the clone. The clone has the task of informing its user and the other clones when things are not as they should. What's more, also the check for possible group partitions of the correct clones (and respective users) is done by the clones, instead of having the device send by email its view of the history. The correctness of CloneDoc can be derived from the correctness of SPORC, under our trust model.

Dealing with disconnecting users

Differently from SPORC, even when a CloneDoc user goes temporarily offline her clone continues being part of the system. All the user has to do when she gets back is to pull from the respective clone the set of operations that she missed. This has many advantages. First, the user's device need not transform these operations past possible operations contained in its local pending queue (that the clone has). Indeed, the clone has already taken care of such transformations. Second, the clone optimizes the list of operations by canceling possible *add character* followed by a *del character* in the same position. Third, the clone sends the whole sequence encrypted in one bundle. So, the device does only one decryption, instead of one per operation as it is done in SPORC.

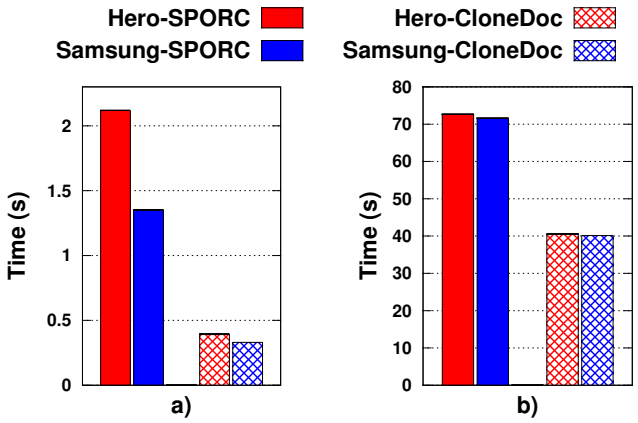


Figure 5.4: (a) Time to update the state of a temporarily disconnected user with SPORC and CloneDoc. (b) Time to apply all users' operations during the editing phase of the test.

5.4.3 Experimental Results

To test CloneDoc we compare it with SPORC. Unlike [83], that makes use of the source code of GoogleWave [91], we have implemented both systems from scratch—the source code of Google Wave is not compatible with Android.

Experimental setup

Our C2C testbed consists of 16 real devices (see Table 5.2 for more details) and an equal number of clones. We have chosen to deploy the system in a hybrid cloud platform: 14 of the clones run on Amazon as instances of our AAx86MI bundle, while two of them run on our private cloud on top of Xen. The untrusted server resides on our private cloud to make the scenario as heterogeneous as possible. All cellphones use WiFi to communicate with both the untrusted server and the clones. To avoid biased results due to other applications that are running concurrently, we disable all unnecessary phone services and applications. We made only four cellphones actually edit the document concurrently while the other 12 act as readers—it might not be realistic that 16 users edit a document at the same time. The goal is to test possible overhead due to the significant number of clones in the system. The test has four phases: (a) In the first phase the users edit the document by adding and deleting characters concurrently; (b) in the second phase a user disconnected for a certain amount of time reconnects to the system; (c) the third phase consists of an active user starting a

Fork* consistency check; (d) in the last phase the administrator removes the membership of one of the users. The test is repeated 10 times and the results on single and meaningful operations are averaged. The write operations executed during the first phase are generated automatically by a script so that measuring is not biased by possible difficulties of humans to type exactly at the same speed without mistakes. The four smartphones that actively edit the document are a Samsung Galaxy S Plus, a Nexus S, a HTC Desire, and a HTC Hero. We have collected a diverse set of cellphones on purpose—our goal is to measure performance on heterogeneous hardware. The clones for these specific smartphones run on Amazon EC2.

CloneDoc moves almost all the operations to the cloud. In particular, 99% of the signatures, 99% of the signature verifications, and 93.75% of the RSA encryption operations are offloaded to the clone. These operations are a fundamental and crucial part of the system since they recur in many tasks such as addition and removal of a user, client-server communication, user-user communication, update of a user's state after her reconnection, among others. Thus, offloading crypto operations to the clone directly translates into faster execution and energy saving. This is confirmed by the results in Figure 5.4, where we show the time it takes to a HTC Hero and Samsung Galaxy S plus to complete the first two phases of the test. Both phases impact the usability of real-time group collaboration: It is crucial that every user gets the operations generated by the other users as fast as possible and that a temporarily disconnected user gets its status updated quickly. Note that temporary disconnection of a smartphone can be very frequent for many reasons (e.g. low coverage, overload of the cellular network, etc.). With CloneDoc the execution of these processes is much faster than with SPORC—around 5 times faster for the status update and 1.8 times faster to apply all users' operations to the document. Most importantly, the experiment makes another important feature of C2C emerge: By using the computational and communication power of the cloud the performance gap between the outdated cellphones and the new more performing ones is much smaller.

Network Traffic

We have measured the number of bytes sent/received during the test when running SPORC and CloneDoc. The result is that the overall network bandwidth used by the smartphones

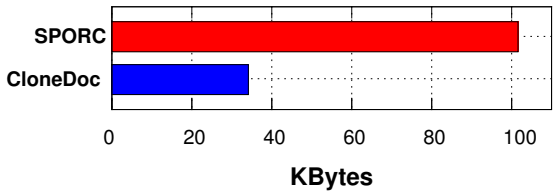


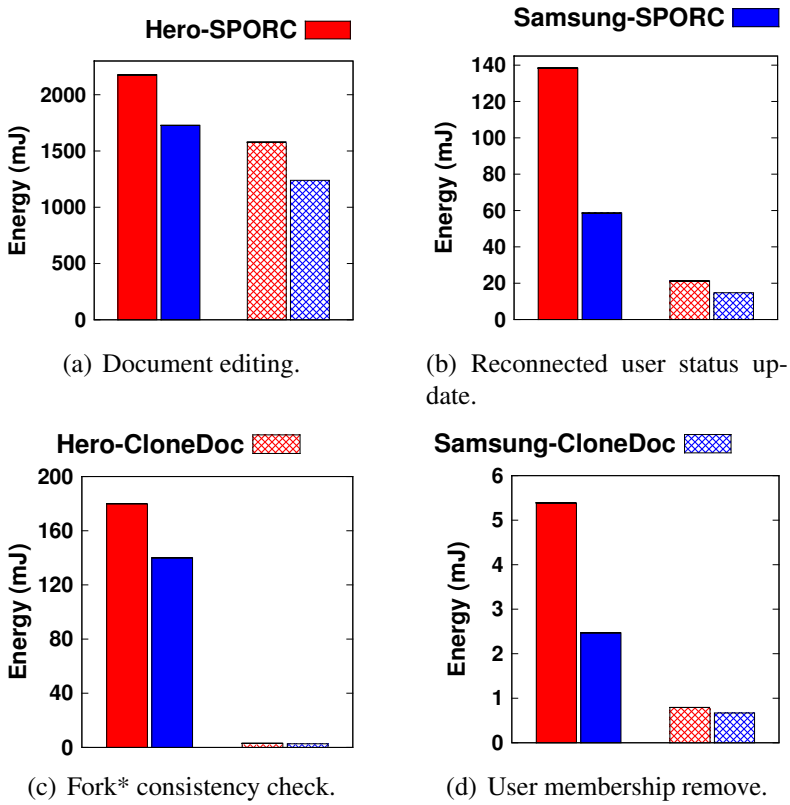
Figure 5.5: Network traffic from/to smartphone with SPORC and CloneDoc

with CloneDoc is 3 times less than with SPORC (see Figure 5.5). This is because the C2C platform enables offloading of both computation and communication. CloneDoc benefits of this feature by offloading most of the networking tasks on the p2p network of clones (e.g. communication with the server and with other users in the network). So, the smartphone uses its networking interface for a shorter period of time. As we will see in the next section, this also reduces the energy consumption of the device.

Energy

We measure energy consumed by the smartphones on both systems with the Mobile Device Power Monitor[8], used by many other works in the area [20, 80]. This device samples the smartphone's battery with high frequency (5000 Hz) so to yield accurate results on the battery's power, current, and voltage. Figure 5.6 shows the average energy used in every phase of our test for both SPORC and CloneDoc. The energy savings achieved with CloneDoc is high in all the phases—around 30% for editing, more than 80% for status update, more than 99% for Fork* consistency check, and more than 80% for the user removal. It is natural to ask what impact each of these operations has on the typical workload of the system. This is not a easy question: The load of editing operations depend on the use of the application by the typical users; the load due to status updates depend on how often our mobile disconnects, and therefore on the quality of our coverage; and the load due to Fork* consistency checks depend on the security level we need. Fork* consistency checkpoints may be used to rollback recovery in case of attack, and therefore the frequency depends on how much work we are willing to lose in this case. It might be reasonable to perform a Fork* consistency check every 1-5 minutes. Combining all these considerations, one can compute the savings in the typical workload of interest.

[8] http://www.msoon.com/LabEquipment/PowerMonitor/



(a) Document editing.

(b) Reconnected user status update.

(c) Fork* consistency check.

(d) User membership remove.

Figure 5.6: The energy consumption during the testing on SPORC and CloneDoc.

Finally, in all cases the gap between the energy consumed from HTC Hero with that consumed by the Samsung Galaxy S Plus is reduced on CloneDoc. This confirms that the C2C platform boosts up the performance of old-fashioned low-performing smartphones making them competitive with newer, more expensive, and more performing ones.

5.5 Summary, Lessons Learned, and Conclusions

In this work we gave several contributions. We built the first Amazon Customized Image (AMI) for Android OS. During the process we encountered many difficulties, mainly related to the fact that the cloud platform is not a real device, thus it lacks of natural components of smartphones such as keyboard, sensors, display etc. This makes us believe that

procedures similar to the one we followed to build the Amazon bundle can be used to make Android run on other commercial cloud platforms. By using our Ax86AMI, we were able for the first time to get reliable performance of public and private cloud clones of Android phones in terms of CPU and I/O with different virtualization environments—Xen, VirtualBox, and QEMU. Then we described Clone2Clone (C2C), a platform that realizes p2p networking of clones of smartphones on the cloud. To the best of our knowledge, this is the first time that distributed computing among smartphones is made possible by exploiting cloud services. We designed the system, implemented it, and tested its functionalities with CloneDoc—a non-trivial application for secure group collaboration. For the experiments we used a testbed of 16 android smartphones and the most performing clones on both public and private cloud, according to our performance study. An interesting lesson we learned is that our system is able to close the gap between old low-performing smartphones and newer, more expensive and performing ones.

C2C enables innovative applications such as content sharing and search in the huge amount of data stored by our devices and it makes it possible to implement distributed execution of advanced services in a network of mobile smartphones.

Chapter 6

CloudShield: Efficient Anti-Malware Smartphone Patching with a P2P Network on the Cloud

As we saw in the previous chapters, state of the art offloading architectures consider the possibility of using virtual copies of real smartphones, called *clones*. The clones run on the cloud, are synchronized with the corresponding devices, and help alleviate the computational burden on the real smartphones by running heavy software modules on behalf of the real devices. In Chapter 5 we presented C2C, a platform that organizes the clones in a peer-to-peer network in order to facilitate content sharing among the mobile devices and to enable the possibility of executing p2p application in a network of smartphones. We believe that P2P network of clones can be a useful tool to solve critical security problems. In particular, we consider the problem of computing an efficient patching strategy to stop worm spreading between smartphones. The worm infects the devices and spreads by using blue-tooth connections, emails, or any other form of communication used by the smartphones. The peer-to-peer network of clones is used to compute the best strategy to patch the smartphones in such a way that the number of devices to patch is low (to reduce the load on the cellular infrastructure) and that the worm is stopped quickly. We consider two well defined worms, one spreading between the devices and one attacking the cloud

before moving to the real smartphones; we describe CloudShield, a suite of protocols running on the peer-to-peer network of clones; and we show by experiments with two different datasets that CloudShield outperforms state-of-the-art worm-containment mechanisms for mobile wireless networks.

The results presented in this chapter are in collaboration with PhD colleague Marco Valerio Barbera and appear in [8].

6.1 Risk of malwares on alternative app–markets

The number of mobile apps available for smartphones has grown exponentially in the last few years. These apps are distributed by online stores such as the App Store for the iPhone and the Android Market for Android systems. The App Store makes a number of checks before making the applications available for download. Of course, the checks give some reasonable confidence that the applications run correctly but does not guarantee that they are immune to viruses and malware. The Android Market is using a different strategy that helps Android spread faster—the online store is open without particular limits or quality checks to application developers that want to distribute and advertise their applications. Clearly, this makes Android an even easier target to viruses and malware.

Recently, in order to correct this problem at least partially, Google has introduced the on-cloud check: Every app uploaded to the Android Market is made available only after running correctly on an Android image running on Google's Cloud. However, the Mobile Threat Report from F-Secure[1] shows that in the third quarter of 2012 have been detected 51,447 unique malware samples for Android on the Play Store. Furthermore, there are many third party markets out there that are becoming more and more popular and that do not implement any check at all. These markets are increasing the risk of virus/malware spreading among smartphones. Indeed, hackers select popular applications from official markets, inject them with malware, and upload them in third party markets [96]. This recently happened with the very popular Angry Birds game[2]: The fake app contained a

[1] http://tinyurl.com/buq3mtk

[2] http://gizmodo.com/5901691/psa-fake-angry-birds-space-android-app-is-full-of-malware

malware whose payload was hidden inside two .JPEG files. After the installation, the malware would start downloading additional malware to the phone and make it part of a botnet. Geinimi[3] was detected in the end of 2010, and was the first and the most sophisticated malware that displayed botnet-like capabilities. Geinimi is included into repackaged versions of legitimate applications, primarily games, and distributed mainly in Chinese Android app markets. The trojan collects private and sensitive informations that are sent to remote servers every 5 minutes. iCalendar is another real example of Android malware. The malware sends an SMS to the premium number 1066185829 and in the background blocks any incoming delivery reports. The SMS is sent only once (the 5th time the app is launched) in order to make the detection very difficult, and to not make the user suspicious of strange SMS charges[4].

In Chapter 5 we presented the C2C platform where every smartphone is associated with a software clone on the cloud, and where clones are interconnected in a peer-to-peer fashion exploiting the networking service within the cloud. The C2C platform is a peer-to-peer architecture that can be seen as a "facebook of the clones": The links in C2C are created automatically between clones whose users call/text/email each other frequently. The platform can be used in many ways, especially to share content among users without involving limited-battery smartphones.

C2C seems like a perfect candidate not only to offload heavy mobile apps but also to prevent malware spread. Indeed, the peer-to-peer network of clones on the cloud can be used both to check newly installed applications and to monitor virus spreading on the mobile devices. The clone can behave as a first check screen on newly installed apps: It can run the app for a while till it ascertains that everything is OK. After that, the app can be installed on the real smartphone. This way the users do not rely on the policies of the online markets and their smartphones are more protected.

In this chapter we advocate the idea that C2C, a peer-to-peer network of virtual smartphone clones running on the cloud, can help stop worm attacks spreading from smartphone to smartphone on the mobile network. The worm propagates by using bluetooth connection, mms messages, phone calls, or any other means of infection available among smartphones.

[3] https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/
[4] http://www.exploit-db.com/wp-content/themes/exploit/docs/17717.pdf

We work under the assumption that the links of the peer-to-peer network of the clones reflect the sociality among the real smartphone users (this is easily done since every clone on the cloud synchronizes with its corresponding real device).

The first problem that we consider in this work is to devise a mechanism to patch the smartphones in such a way that the number of devices patched is low, to make the scheme cheap, and that the probability of stopping the worm is high. As a solution, we propose CloudShield—a suite of schemes that cope with worm spread on cellular networks by using a peer-to-peer network of clones on the cloud to compute the patching strategy. The idea behind CloudShield is as follows: It selects few "important" clones in the cloud to patch; in turn, the patched clones transmit the patch to the respective smartphones by thus effectively containing the worm spread in the cellular network. We test our strategies on two different datasets—Facebook [97] and Live Journal [98, 99])—and compare them to the state-of-the-art worm-containment schemes on dynamic social networks [100, 101]. The experimental results show that our approaches outperform both [100, 101] in yielding lower infection rates after patching a smaller number of nodes.

Then, we consider the possibility that the weak link in the chain is the cloud itself. We assume that a worm attacks the peer-to-peer network of clones on the cloud with the goal of spreading on the smartphones. In this case, the attack is probably faster and we assume that the patch is not yet available. We attack this problem and come up with a two phase solution: At detection time, we make the clones weaken the incoming links from their peers in the cloud, so that the worm is contained as much as possible. Then, once the patch is released, we apply our CloudShield schemes to select effective patching sets, patch the respective clones, and finally, reset the capacity of the links. Our experimental results show that this combined strategy yields infection rates reduced with up to 20% of the nodes.

6.2 Worm-propagation in cellular networks

The research on worm-containment in wireless networks is divided into two main categories: Proximity-based and infrastructure-based schemes. The former assumes that the virus spreads through short-range communication technology such as Bluetooth [102, 103, 104, 105, 106]. These schemes assume that no central authority is involved, and thus nodes

cooperate in distributed way to detect and stop the infection [107, 108, 109, 110, 111, 112, 113]. They try to exploit social properties of the movement of network nodes in selecting efficient paths to spread virus signatures, so that infected devices are promptly healed and the infection is eventually stopped. In [111, 112] a simulation and analytic model for proximity-based worms is developed, and is shown that mobility has a significant impact on the propagation dynamics. The authors of [107] explore local detection of proximity worms and apply broadcast of proximity signature dissemination. However, the cost of the broadcast is high, and there is no guarantee that the solution is close to the global optimum. In [113] is described the containment and spreading of malware and patches over power law degree networks. The hosts considered are normal computers in this case, but the importance of automatic malware containment described is true also for the cellular networks. The solution proposed in [108] uses the community structure of the network. The social communities need to be computed in a centralized way, which poses severe limitations to the applicability of the scheme in real settings. Khouzani et al. [109] investigate the optimal dissemination of security patches in mobile wireless network to oppose proximity malware spreading. They make use of the SIR model to formulate the tradeoffs between the security risks and resource consumption as optimal control problems. Their work assumes homogeneous network setting, which is far from modeling well the human mobility. The worm spread considered in all the above schemes follows the Kermack-Mcendrick epidemic model [114], traditionally used in wireless networks. Works such as [102, 115, 116] show that, for large networks, the deterministic epidemic models can successfully represent the dynamics of malware spreading, which is demonstrated by simulations and matching with actual data.

Infrastructure based schemes deal with viruses that spread through the cellular network services (SMS and MMS) [117, 115, 118, 119, 120, 100, 101, 121], where a central authority, e.g. the network infrastructure, takes care of detecting and containing the worm. Fleizach et al. [122] evaluate the speed and severity of malware spreading by cellphone address books. In [118] the authors propose an automated procedure to identify the most vulnerable clients as well as a proactive containment method when an attack is in course. The work of Ruitenbeek et al. [119] considers the effects of MMS-based viruses that spread by sending infected messages to other phones. They model the virus spread through the

Möbius software tool, measure the propagation rate and the extent of virus penetration and compare the effectiveness of different mobile phone virus response mechanisms. In [120] the authors propose a behavioral-based methodology to detect mobile viruses, malware and trojans for the Symbian OS. By testing their machine-learning based detection methodology on a database composed by both simulated and real-world malware samples and show its high effectiveness (more than 96% of accuracy in detection).

To the best of our knowledge, the works in [100, 101] are to be considered the state of the art of infrastructure-based worm-containment schemes for cellular networks. Both works are based on the intuition that as we are more likely to open/download content from our friends, the stronger the social-relation between two users the more likely a virus passes from one to the other. In [100] the authors aim at patching the smallest number of devices to contain worm spreading. To do so they adopt a counter-mechanism which continually analyzes the network traffic to extract a social-relationship graph among mobile phones. This graph is then partitioned via two different methods—balanced and clustered partitioning—to select the optimal patching phone set as those with higher infection potential. Through trace-driven experiments using real IP packet traces from a cellular network in US the authors test the efficiency of both partitioning algorithms to contain mobile worms. The number of clusters k in which the network should be partitioned is a parameter of the scheme, and should be known in advance. The number N_k of nodes patched by the scheme intuitively decreases with the increasing of k. However, it is not possible to guess N_k given k, without actually computing the k partitions. What's most, the same k–value might yield different values of N_k in different networks of the same size. Actually, for particular networks, there might not even exist a k that yields exactly a given number of nodes to patch. So, a resource-limited network infrastructure that wants to patch not more than M nodes has to compute the possible patching sets for all possible k and see which is the one yielding the N_k closest to M. For how the scheme works, N_k should not be higher than M (the maximum patching number). Otherwise, it will leave "open bridges" to the worm to expand from one partition to the other. From the other side, N_k cannot be very much lower than M. Low values of N_k intuitively are given from low values of k (small k yields larger partition in the network). So, even by attacking just one partition the worm would spread on a large number of network nodes.

The approach in [101] targets worm containment in dynamic social networks, such as e.g. Facebook, Twitter or Google+, where users befriend or unfriend each other dynamically over time. Similarly to [100], also in [101] the social graph is partitioned, but with a different viewpoint—partitions represent the community sub-structures of the network, i.e., very well connected sub-graphs of friends. These sub-structures are adaptively kept updated so to reflect the evolution of the social network. Patches are then distributed to most influential users within single communities, i.e. nodes that have many links towards other communities are patched first. The authors test their approach on the Facebook network dataset [97] and show the superiority of their community-based solution to that of [100].

6.3 System model and motivation

In our system each cellphone is associated with a software clone in the cloud [19, 7]. The clone runs the same operating system and apps as the real device. The device sends updates to and gets updates from the clone whenever new data is generated and a new application is installed. Also does it use the clone to offload computation whenever possible so that to reduce battery consumption [19, 6, 7]. In addition, the clones are connected to clones of other "friend" devices on the cloud. Friendship is determined by the rate of communication between the real devices: Smartphones that call/text/email each other frequently have the respective clones connected in the P2P networks on the cloud [7]. Not only do clones help smartphones offload computation but also communication: Whenever a user A needs to receive a file from the smartphone of user B, the file is first transferred from clone B to clone A in the P2P network on the cloud and is then sent to the device A from clone A. Clearly, device B should have previously sent the file to clone B. However, B needs to do it only once. If successively device C needs the same file from B, it will get it through clone B, without involving device B anymore.

6.3.1 Why patching the clones

Previously presented infrastructure-based worm containment schemes consider worms that propagate smartphone-to-smartphone. They aim at individuating a set of crucial smartphones to patch so to contain the worm spread in the network. Once the set is selected the network infrastructure sends the patch to all the nodes. Small patching sets become very valuable in this context: Not only is it costly for the network infrastructure to send the patch to many nodes, but it also impacts negatively on the load of the already overloaded 3G/Cellular networks. From the other side, proximity-based schemes distribute the patch along efficient but slow blue-tooth paths. In this case as well, flooding the network with the patch may not be an option since patches are usually large files and smartphones are battery limited devices. As we already discussed in the related work section, both schemes try exploit the social relationships among users to better select fewer and more effective nodes to patch. Here we take a similar approach, but on the P2P network of clones on the cloud: We send the patch to few, effective clones, that in turn transmit it individually to the respective devices. Recall that the clones are up-to-date with statistics on the communication of the respective real devices. Indeed, a given clone contains information on calls/texts/emails received from and sent by the real device, as well as information on its geographical position in time and of blue-tooth communication opportunities with other mobile devices. This makes the P2P network of clones reflect very well the sociality of both the cellular-based and blue-tooth based communication among devices in the real world. Thus, by patching a few crucial clones on the cloud we are able to stop the worm spread on the real devices.

However, the P2P network of clones can potentially introduce a new attack to the real devices: A virus/worm that infects a clone, either during a synchronization with the (infected) device or during a communication with an (infected) friend clone, can propagate with enormous speed exploiting the P2P links in the cloud. The infection is then directly propagated to the real devices as soon as they communicate with their clones. Moreover, the worm propagation on the P2P network of clones can also take place without any action from the user. Indeed, after attacking a given clone, say that of real device A, the worm can pretend to be a file that is being sent from device A to device B, through the respective clones. Imagine for example a scenario in which clone A sends to friend clone

B a request containing: "Install this new cool app". The app is of course the worm itself. Remember that clones are supposed to act as screens to new apps, before the user installs them on the real device. If the newly installed application has not been discovered yet to contain a worm, clone B may get infected. One might think that this attack is mitigated if clone B does not accept digital content or does not install any new applications suggested by friend clones without consulting the user first. However note that users get bored of systems with features that require their intervention. Mr. Clippit for example, which was Microsoft's office assistant, was cut off the system because it made users very unhappy by frequently and regularly asking questions and giving suggestions[5]. Moreover, experience has taught us that people tend to blindly agree with settings that allow automatic download and installation of software suggested by already installed software or updates. Lastly, but probably most importantly, even if the user has to intervene and make a decision every time a friend clone suggests its clone to install an app, she will probably accept the suggestion. Past experience has shown that phenomenas such as viral marketing in social networks are successful because people tend to follow suggestions from their friends almost blindly. This was the case of the Koobface worm that spread on Facebook in 2008: it exploited the friends lists of victims sending them infected links[6].

Note that in this second attack scenario, the worm is not being transferred from smartphone to smartphone. Rather, it quickly propagates on the P2P network of clones on the cloud and then to the mobiles.

6.4 The Methodology

We consider the P2P network of clones to be a graph $G = (V,E)$, where V is the set of clones and E is the set of edges representing the communication links among them. The links, as we mentioned in the previous sections, reflect the frequency of communication between the devices in the real world. Note that many of the communication technologies possible through smartphones (texts/emails/calls) are not mutual. The amount of information between any two nodes is not the same in each direction. Not only that, in our address

[5] http://www.guardian.co.uk/media/2001/apr/11/advertising2
[6] http://gawker.com/5103848/why-the-koobface-virus-spread-so-fast

books there are numbers that we use more often and numbers that we tend to ignore. This is why the graph G representing the network is both directed and weighted. The weights are derived from the frequency of calls/texts/emails/sharing of files through blue-tooth between the two devices in the real world.

6.4.1 Characteristics of the Data-sets

Unfortunately we cannot replicate the experiments done in [100], since the dataset of cell-phone calls used in that work is not publicly available. Thus, we use two social-network datasets that are available and that can be used to replicate the experiments: (1) *FB* [97], a large Facebook subgraph of 63'392 nodes used also in [101], and (2) *LJ* [98, 99], a directed graph representation of the LiveJournal social network of 4'847'571 nodes. We generate the Facebook's dataset oriented version by transforming each friendship link among nodes u and v the two directed edges (u,v) and (v,u) of the same weight.

The FB dataset is enriched with information on over 1.5M wall posts between users for the period September 2006–November 2009. We use this information to derive the frequency of communication, thus, the weights of the directed edges, on FB's social graph. Conversely, the LJ dataset does not include information of user posts. Note though that the direction of the edge (u,v) on the Live Journal social network means that v has subscribed to $u's$ journal, and thus, gets all the posts published by u. So, in this case, we suppose that the weight of all the out-links of a node is 1.

6.4.2 Worm propagation model

We adopt the same worm propagation model used in [101, 100]: The worm is able to explore the social strength of the communication links among nodes. Once it infects a node, it tries to expand to its friends by exploiting communication opportunities from between the two to send infected files or suggest installation of malware apps. Thus, the probability of the actual infection happening depends on both the communication frequency (link weight) and on how likely is it that the possible victim accept suggestions sent by the infected neighbor. This factor is related to the level of trust that the possible victim has towards the infected node: We are more likely to follow links included in emails or messages received

by our close friends (which we trust more) than from strangers.

We measure the trust level of a node v towards its neighbor u as follows:

$$\tau_{v,u} = \frac{|OF_u \cap OF_v|}{|OF_v|}. \tag{6.4.1}$$

The intuition behind the above equation is that we deem as more valuable (and thus trust more) people that have many friends in common with us.

Then, the probability that a worm spreads from node u to node v through link (u,v) becomes:

$$p_{(u,v)} = w_{(u,v)} \cdot \tau_{v,u}, \tag{6.4.2}$$

where $w_{(u,v)}$ is the weight of the directed edge (u,v). Again as in [101, 100] we assume that the time that the worm takes to spread from an infected peer to another is inversely proportional to the communication frequency between the peer and this specific out-friend.

6.4.3 The CloudShield Scheme

We base our solution on the idea that the more a scheme chooses socially-important network nodes to patch the more effective it is in worm containment. Indeed, socially-important nodes are typically very influential in terms of information spreading in general, and in worm spreading in particular. This is due to a combination of two factors: Their position in the network and the large number of links towards other nodes. Moreover, we want the solution that implements this idea to be simple and computationally efficient in finding the subset of nodes to be patched. The simplest it is, the less it costs to deal with the dynamics of the network (links that appear and disappear).

With this in mind we build the following three versions of our CloudShield solution based on different methodologies on selection of nodes to patch:

1) *Page-rank CloudShield (PR-CS)*: It deems as important nodes with high page-rank in the network. The intuition behind this is as follows: A node is at risk of infection if it interacts frequently with other nodes with high risk of infection. Of course, this is very similar to the PageRank procedure [123], where a page has high ranking if it has many incoming links from other pages with high ranking.

2) *Degree CloudShield (DG-CS)*: It deems as important nodes with high out-degree weight in the network. Intuitively, once a node gets infected, the highest its out-degree, the highest its contribution in further spreading the worm to other nodes. So, patching first these nodes can be a reasonable choice to stop the infection.

3) *Greedy Degree CloudShield (G-DG-CS)*: In social networks, the nodes with highest degree tend to cluster. This is why the previous strategy is very likely to eventually misspend some patches to nodes surrounded by already patched nodes, while leaving other nodes of the network unprotected. Following this intuition, we present G-DG-CS, where, similarly to DG-CS, nodes with high out-degree are candidates to be patched first. However, the selection is different: After the highest out-degree weight node is chosen, all its in-links are dumped and the out-degree weight of its incoming friends is updated. The procedure is repeated till the required number of nodes to patch is reached.

As we will see from our experimental results, our simple schemes out-perform previous ones that require complicate computation of network clustering or community sub-structures (e.g. [100, 101]). This is due to the fact that the worm exploits the social links/-paths in infecting the network. The more a scheme manages to "destroy" such links by patching crucial nodes, the more the structure of the network changes, as far as the worm is concerned. Since the network has social properties, even applying a simple patching strategy based on page-rank or max-degree is enough to dramatically change its structure [124] (prolong the mutual distance among nodes, disconnect it etc.). Most importantly, our schemes require pretty simple computation and can be even computed in a distributed way—without the need of an authority—at the cost of inducing some traffic overhead on the C2C network. This makes it easier to handle network dynamics such as insertion or deletion of edges (new social relationships that start or old ones that end).

6.5 Experimental Results

To validate our CloudShield patching methods we compare them with the states of the art in terms of infrastructure- based patching schemes for cellular networks: Clustered

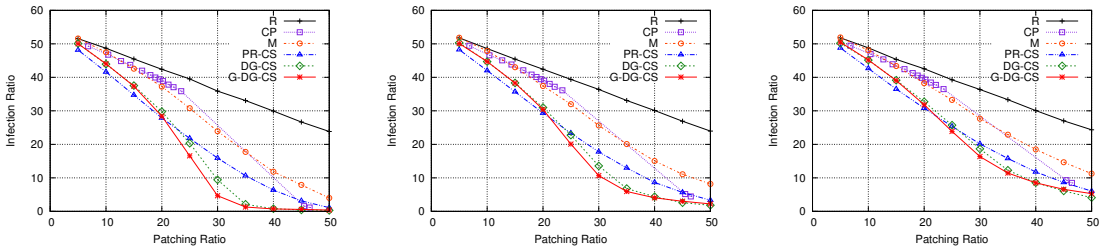
Partitioning (CP) [100] and Community-based partitioning (M) [101]. To the comparison we also add Random Partitioning, used as a benchmark in [100].

6.5.1 Worm attack model and patching threshold

To model the worm attack at its initial phase we follow the method in [100, 101]: We first induce the infection to a small number of users (0.02% of nodes), randomly and uniformly chosen on the network. This is to simulate the initial worm sources during the early stages of the infection. The worm starts spreading in the system till it reaches a certain number of infection rate (percentage of nodes infected), given by a patching threshold parameter α. This parameter represents the timespan between the very first infection phase and the moment in which the worm is detected and the patch is generated. Then, we apply the patches to nodes selected by any of the above schemes. The simulation finishes when the worm does not expand any further in the network. The performance of each scheme is measured by the infection rate reached by the worm as a function of the number of nodes patched with each of the patching schemes. Each experiment is run 1000 times and the results are averaged. As far as CP [100] is concerned, since it is impossible to derive the number of nodes to patch that each k value yields, we compute the patching sets for every possible value k.

6.5.2 Stopping the worm on the cellular network

Here we assume the first attack scenario: The worm is spreading in the cellular network, and does not attack the cloud. As soon as the infection rate reaches the patching threshold, the clones selected by the strategies are patched and pass the patch to their real alter-egos (smartphones). For each scheme we study how the infection rate changes as a function of the patching ratio (rate of the patched nodes). This is done for three different values of patching threshold α: 2%, 10%, and 20%. The experimental results for both FB and LJ are shown respectively in Figures 6.1 and 6.2. Let us start with considering the performance evaluation on the FB dataset (see Figure 6.1). First we note that our three schemes yield the lowest infection rates for any percentage of patched nodes. For patching rations up to

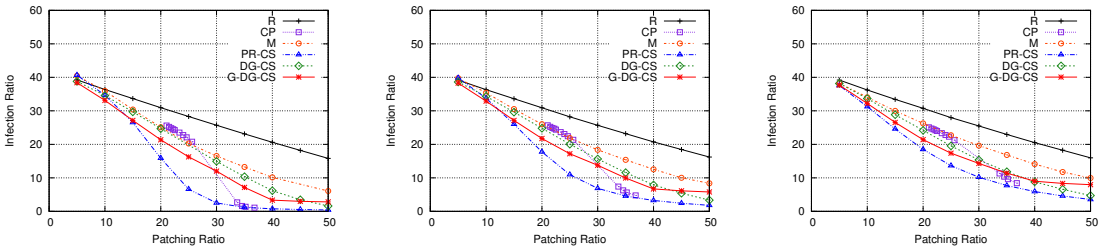


(a) $\alpha = 2\%$ (b) $\alpha = 10\%$ (c) $\alpha = 20\%$

Figure 6.1: Infection rate vs the percentage of patched nodes for the various schemes on the FB dataset. **"R"** denotes the random scheme; **"CP"** denotes the Clustering Partition; **"M"** denotes the Community-Based scheme; **"PR-CS"**, **"DG-CS"**, and **"G-DG-CS"** denote respectively the Page-Rank, Degree and Greedy Degree CloudShield schemes.

20%, both DG-CS and G-DG-CS are outperformed by PR-CS. This is because the Facebook graph is very dense. As a result, DG-CS tends, at least initially, to send the patch to nodes that are "close" in the graph. This effect is a bit mitigated by G-DG-CS; recall that, according to this scheme, after a node with high out-weight is patched its in-edges are dropped and weights are re-computed for its neighbors. Nonetheless, when the number of nodes to patch is low, this procedure does not get to distribute well the patches in the graph for very well connected graphs like Facebook. This effect is mitigated by the PR-CS scheme which succeeds in yielding more efficient patching sets for low values of patching ratios. When the values of patching ratio increase, we get a different scenario: For patching thresholds in the interval $[20\%, 35\%]$, G-DG-CS becomes the most performing. This is due to the fact that the higher is the number of the nodes patched, the larger is the number of edges dropped by the G-DG-CS scheme. This procedure makes so that very well connected clusters in the network get "destroyed" soon after a few cluster members are patched, and the scheme starts therefore selecting nodes in other parts of the network by better distributing the patches on the cloud. That said, this procedure eventually ends up "destroying" all the strongly connected clusters of the network: The graph results so sparse that the further selection choices do not impact much the infection rate. This is why for larger patching thresholds (from 40% and on) the simple degree scheme (DG-CS) yields better results.

Now, let us consider the comparative performance of all schemes on the LJ dataset (Figure 6.2). The respective graph is much larger and sparser than FB. In addition, the



(a) $\alpha = 2\%$ (b) $\alpha = 10\%$ (c) $\alpha = 20\%$

Figure 6.2: Infection rate vs the percentage of patched nodes for the various schemes on the LJ dataset. **"R"** denotes the random scheme; **"CP"** denotes the Clustering Partition; **"M"** denotes the Community-Based scheme; **"PR-CS"**, **"DG-CS"**, and **"G-DG-CS"** denote respectively the Page-Rank, Degree and Greedy Degree CloudShield schemes.

distribution of the out-degrees of the nodes in LJ decays faster than that of FB. So, high out-degree nodes are not only very well connected between them, but also are very central to the graph. Indeed, the average clustering coefficient for the LJ dataset is higher than that of the FB dataset (respectively 0.3123 for LJ and 0.22 for FB). So, the node sets chosen by our degree based schemes here tend to be even more clustered, thus yielding lower performance with respect to our PR-CS scheme. Note that the PR-CS scheme yields the best performance with respect to all considered schemes. Also note that the features of the LJ dataset make so that for low values of patching ratios the CP scheme performs very badly; then, its performance takes a quantum leap when passing from 25% ($k = 949461$) to around 33% ($k = 961431$) of patched nodes—recall that the size of the nodes patched depends on the number k of clusters in which CP partitions the network. This is due to the fact that till $k = 949461$ CP yields very large partitions. So, even though CP "cuts" the bridge edges between clusters, it does not manage to stop the worm infecting the many nodes of a given partition. When k passes from 949461 to 961431 these large partitions break into many smaller ones, thus making the scheme give less front to the worm and equaling the CP's performance to that of our PR-CS scheme.

Note finally that for higher values of patching thresholds all the schemes perform worse on both datasets. This is because, as the worm has already gotten to many nodes, it is more difficult to effectively contain it with the same number of patched nodes.

6.5.3 Stopping the worm on the cloud

Here we consider the second attack scenario in which the worm manages to break down the security of the cloud and overtake the clones. It firstly infects a subset of cloud clones, again considered to be as small as the 0.02% of the whole network. Then, it starts propagating towards other clones, exploiting the p2p cloud social links among them. As soon as it manages to infect the maximum number of clones possible, it makes the infected clones transmit the worm to the respective cellphones. In addition, this worm strategy might potentially infect all the p2p clones, and thus, all the real-world smartphones. Not only does the unpredictability and the strength of this attack make any infrastructure-based scheme ineffective, but it also induces a large amount of traffic to the cellular network. This is why it is particularly vicious and why it has to be stopped.

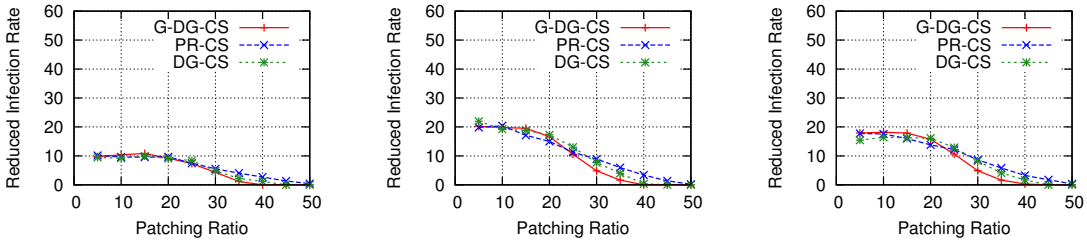
Applying one of our schemes to this scenario would contain the worm with the same efficiency as in the previous one. However, here we are dealing with a stronger worm, for which the patch might be more difficult and complicated to achieve. What's most, even if the patch is released early before the worm attacks the clones, no one guarantees that it is installed in time by users—recall the example of the SQL Slammer malware [125], for which the patch was released 6 months earlier than the attack time, and still managed to infect more than 75'000 users that had not installed the patch in time. So here we assume that the patch is applied way after the detection of the attack. With this in mind, we introduce another parameter in our system, the detection threshold δ, representing the fraction of nodes infected at the moment in which the attack is detected. We use this parameter to further improve the efficiency of our approach in worm containment in the following way: When the attack is detected, the clones enter in a "quarantine" state, during which they became more cautious and trust less their incoming friends. This assumption is indeed realistic: In real systems, when we know that a certain worm is spreading through e.g. email links, we pay more attention on what links we follow, or on what software we install. To simulate this behavior in the p2p network of clones we make each healthy clone diminish the trust τ towards its incoming friends. The goal of the quarantine phase is to contain the worm as much as possible from spreading, till the patch is released. However, as a side effect the bandwidth capacity of the incoming links of a clone decreases. This

Dataset	δ		
	2%	5%	10%
FB	$40\%\tau$	$30\%\tau$	$25\%\tau$
LJ	$50\%\tau$	$50\%\tau$	$45\%\tau$

Table 6.1: Values of quarantine trust τ_q that allow the virus to expand to infection rate $\alpha = 20\%$. τ denotes the initial trust value.

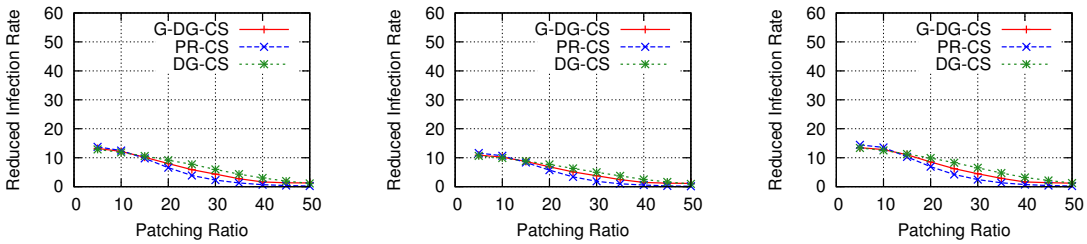
is why we make the quarantine phase last till the moment in which the patch is released. Then, the clones reset the trust on their incoming edges, and the capacity of the links goes up to its real value.

To study the impact of the quarantine phase on our schemes we fix the patching threshold to 20% and vary the detection threshold δ, which determines the moment in which the quarantine phase starts. We consider three different values for δ: 2%, 5%, and 10%. For each δ value we launch the following experiment: When an infection rate of δ is reached, clones diminish their incoming links trust from τ to τ_q until the worm infects a rate of $\alpha = 20\%$ of the network nodes. Clearly, if the value of τ is very low, e.g., 0, the worm will not propagate at all. However, this also means that the capacity of the p2p links between clones drops to 0, by thus making the p2p network of the clones useless. So, we set the τ_q value to be the smallest value that still allows the worm to expand enough to reach the infection rate $\alpha = 20\%$. The τ_q values we use on each dataset are shown in Table 6.1. Then, we patch the nodes according to each of our schemes (PR-CS, DG-CS, and G-DG-CS), restore the trust to its initial value, and wait till the worm propagation is stopped. Each experiment is run 1000 times and the results are averaged. Figures 6.3 and 6.4 present the reduction of the infection rate for each scheme when coupled with the quarantine phase, for three different values of detection rates δ. As can be noticed from the figures, the gain is very high for low patching rates (up to 20% for $\delta = 5\%$ on the FB dataset), and it lowers with the increase of the patching ratio. This is because patching a node is equivalent to lowering all its incoming links' trust to 0. So, when the number of nodes patched is very high, the number of paths "destroyed" is very high. This is why the quarantine phase does not make any difference in worm containment for large values of patching ratio.



(a) $\delta = 2\%$ (b) $\delta = 5\%$ (c) $\delta = 10\%$

Figure 6.3: Reduction of the infection rate on the FB dataset for the various schemes when introducing the quarantine phase. **"PR-CS"**, **"DG-CS"**, and **"G-DG-CS"** denote respectively the Page-Rank, Degree and Greedy Degree CloudShield schemes.



(a) $\delta = 2\%$ (b) $\delta = 5\%$ (c) $\delta = 10\%$

Figure 6.4: Reduction of the infection rate on the LJ dataset for the various schemes when introducing the quarantine phase. **"PR-CS"**, **"DG-CS"**, and **"G-DG-CS"** denote respectively the Page-Rank, Degree and Greedy Degree CloudShield schemes.

6.6 Conclusions

In this chapter we advocate the use of a peer-to-peer network of smartphones virtual clones on the cloud as a mechanism to worm containment. We consider the problem of stopping worms on the mobile network of smartphones, and then we consider the problem of stopping a worm spreading on the peer-to-peer network. We introduce simple mechanisms that can be computed quickly and easily by the P2P network of clones. Our mechanisms outperform the state of the art on worm containment schemes for mobile cellular networks and show some positive effect on the problem of stopping a more powerful and vicious worm that attacks the cloud itself before spreading to the smartphones.

Conclusions and Future works

We started the dissertation considering SWIM, a mobility model proposed by Mei et al. [15] that generates small worlds of mobile humans. The authors show that SWIM generates synthetic traces that have the same statistical properties of real traces. We extended the model introducing the concept of sociality and observed that SWIM is able to mimic the social community structure of real small-scale experimental traces. We executed complex community-based forwarding protocols on the real and SWIM-generated traces and showed that SWIM predicts very well the performance of these protocols [1]. Then we proposed a methodology to generate scaled scenarios starting from well-known real traces. Thanks to this feature we were able to analyse—for the first time, to the best of our knowledge—the scaling capabilities of different forwarding protocols, stating that the quest for a scalable forwarding protocol for pocket switched networks is still an open issue [2]. As a future work we plan to collect mobility data from large set of smartphone users and SWIM-simulate scenarios like districts, cities, and countries.

Later on, we treated individual selfishness, which is a psychological hurdle that users in an opportunistic network face. We presented a market based mathematical framework that enables heterogeneous mobile users in an opportunistic mobile network to compromise optimally and efficiently on their QoS demands [3, 4]. As a future work, we plan to implement the model and conduct a simulation study of our proposed theory.

Then, we focused our study in the area of mobile cloud computing. We presented ThinkAir [6, 5], a method-level mobile cloud computing framework that allows developers to easily offload computation to the VM clones on the cloud. ThinkAir extends previous works providing an efficient way to perform on-demand resource allocation and exploiting parallelism by dynamically creating, resuming, and destroying VMs when needed.

Experiments and evaluations with micro benchmarks and computation intensive applications demonstrate the benefits of ThinkAir for profiling and code offloading, as well as accommodating changing computational requirements with the ability of on-demand VM resource scaling and exploiting parallelism. As a future work we are working on integrating automating application parallelization with ThinkAir for better use of distributed computing on the cloud.

We pushed the cloud-smartphone paradigm a step further and designed, built, and implemented Clone2Clone (C2C) [7], a distributed platform for cloud clones of smartphones. We built the first Amazon Customized Image (AMI) for Android-OS, and studied the performance of device-clones hosted in private and public virtualization environment. Upon C2C we built CloneDoc, a secure real-time collaboration system for smartphone users, showing that thanks to C2C it is possible to implement distributed execution of advanced p2p services in a network of mobile smartphones.

As a straightforward application of C2C we implemented CloudShield [8], a worm-containment mechanism for mobile smartphone networks. CloudShield runs on the p2p network of the smartphone clones and is able to outperform state-of-the-art anti malware tools for mobile wireless networks. As a future work we are building other interesting applications—such as content sharing, distributed social networks, etc.—that make use of the Clone2Clone platform.

During the last work of my PhD we defined two types of clones—*off-clone* and *back-clone*—and gave a precise evaluation of the feasibility and costs of both types of clones. This work is described in [9] but was not included in this thesis. We provided results with a real testbed of 11 Android smartphones and associated clones running on the Amazon EC2 cloud platform. We studied the data communication overhead necessary to achieve different levels of synchronization (once every 5min, 30min, 1h, et.) between devices and clones for both types of clones [9]. As a future work we plan to extend our study to a larger set of users and gather more data to be used for a deeper and better analysis.

Bibliography

[1] S. Kosta, A. Mei, and J. Stefa. Small world in motion (SWIM): Modeling communities in ad-hoc mobile networking. In *IEEE SECON 2010*, 2010.

[2] S. Kosta, A. Mei, and J. Stefa. Large-scale synthetic social mobile networks with SWIM. *Transactions on Mobile Computing*, 2012. (accepted with minor revision).

[3] Ranjan Pal, Sokol Kosta, and Pan Hui. Settling for less: a qos compromise mechanism for opportunistic mobile networks. In *Proceedings of ACM SIGMETRICS MAMA*, 2011.

[4] Ranjan Pal, Sokol Kosta, and Pan Hui. Settling for less: a qos compromise mechanism for opportunistic mobile networks. *SIGMETRICS Performance Evaluation Review*, 39(3):49–51, 2011.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Unleashing the power of mobile cloud computing using thinkair. Technical report, arXiv:1105.3232v1 [cs.DC], 2011.

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

[7] Sokol Kosta, Claudiu V. Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clone2clone (c2c): Enable peer-to-peer networking of smartphones on the cloud. Technical Report TR-SK032012AM, T-Labs, Deutsche Telekom, 2012. url: http://www.deutsche-telekom-laboratories.de/∼panhui/publications/clonedoc.pdf.

[8] Marco Valerio Barbera, Sokol Kosta, Julinda Stefa, Pan Hui, and Alessandro Mei. Cloudshield: Efficient anti-malware smartphone patching with a p2p network on the cloud. In *Proceedings of 12th IEEE International Conference on Peer-to-Peer Computing (P2P '12)*, September 2012.

[9] Marco V. Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proceedings of IEEE Infocom '13*, 2013.

[10] P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket switched networks and human mobility in conference environments. In *Proc. of ACM SIGCOMM WDTN '05*, 2005.

[11] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of human mobility on the design of opportunistic forwarding algorithms. In *Proc. of IEEE INFOCOM '06*, 2006.

[12] P. Hui, J. Crowcroft, and E. Yoneki. Bubble rap: social-based forwarding in delay tolerant networks. In *Proc. of ACM MobiHoc '08*, 2008.

[13] V. Erramilli, M. Crovella, A. Chaintreau, and C. Diot. Delegation forwarding. In *Proc. of ACM MobiHoc '08*, 2008.

[14] Th. Spyropoulos, K. Psounis, and C. S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proc. of ACM WDTN '05*, 2005.

[15] A. Mei and J. Stefa. SWIM: A Simple Model to Generate Small Mobile Worlds. In *Proc. of IEEE INFOCOM '09*, 2009.

[16] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, 2000.

[17] A. Mei and J. Stefa. Give2get: Forwarding in social mobile wireless networks of selfish individuals. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 488 –497, june 2010.

[18] Lifei Wei, Zhenfu Cao, and Haojin Zhu. Mobigame: A user-centric reputation based incentive protocol for delay/disruption tolerant networks. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1 –5, dec. 2011.

[19] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys 2011)*, April 2011.

[20] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys*, 2010.

[21] I. Joe and Y. Lee. Design of remote control system for data protection and backup in mobile devices. In *Proc. of ICIS 2011*, 2011.

[22] V. Ottaviani, A. Lentini, A. Grillo, S. Di Cesare, and G.F. Italiano. Shared backup & restore: Save, recover and share personal information into closed groups of smartphones. In *Proc. of IFIP NTMS 2011*, 2011.

[23] C. Ai, J. Liu, C. Fan, X. Zhang, and J. Zou. Enhancing personal information security on android with a new synchronization scheme. In *Proc. of WiCOM 2011*, 2011.

[24] Jing Su, Alvin Chin, Anna Popivanova, Ashvin Goel, and Eyal de Lara. User mobility for opportunistic ad-hoc networking. In *Proc. of IEEE WMCSA '04*, 2004.

[25] T. Karagiannis, J.-Y. Le Boudec, and M. Vojnović. Power law and exponential decay of inter contact times between mobile devices. In *Proc. of ACM MobiCom '07*, 2007.

[26] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Pocket switched networks: Real-world mobility and its consequences for opportunistic forwarding. Technical report, Computer Laboratory, University of Cambridge, 2006.

[27] H. Cai and D. Y. Eun. Crossing over the bounded domain: from exponential to power-law inter-meeting time in manet. In *Proc. of ACM MobiCom '07*, 2007.

[28] P. Hui, E. Yoneki, S. Y. Chan, and J. Crowcroft. Distributed community detection in delay tolerant networks. In *Proc. of ACM/IEEE MobiArch '07*, 2007.

[29] A.J. Mashhadi, S. Ben Mokhtar, and L. Capra. Habit: Leveraging human mobility and social network for efficient content dissemination in delay tolerant networks. In *World of Wireless, Mobile and Multimedia Networks & Workshops, 2009. WoWMoM 2009. IEEE International Symposium on a*, pages 1–6. IEEE, 2009.

[30] J. Whitbeck, V. Conan, and M.D. de Amorim. Critical analysis of encounter traces. In *Proc. of ACM S3 2010*, 2010.

[31] M. Musolesi and C. Mascolo. Designing mobility models based on social network theory. *ACM SIGMOBILE Mobile Computing and Communication Review*, 2007.

[32] C. Boldrini, M. Conti, and A. Passarella. The sociable traveller: human travelling patterns in social-based mobility. In *Proc. of ACM MobiWAC '09*, 2009.

[33] I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the levy-walk nature of human mobility. In *Proc. of IEEE INFOCOM 2008*, 2008.

[34] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser. On Clustering Phenomenon in Mobile Partitioned Networks. In *The First ACM SIGMOBILE International Workshop on Mobility Models for Networking Research*, 2008.

[35] F. Ekman, A. Keränen, J. Karvo, and J. Ott. Working day movement model. In *The First ACM SIGMOBILE International Workshop on Mobility Models for Networking Research*, 2008.

[36] K. Lee, S. Hong, S. J. Kim, I. Rhee, and S. Chong. SLAW: A Mobility Model for Human Walks. In *Proc. of IEEE INFOCOM '09*, 2009.

[37] C. Boldrini, M. Conti, and A. Passarella. HCMM: modelling spatial and temporal properties of human mobility driven by users' social relationships. *Computer Communications*, January 2010.

[38] M. Musolesi and C. Mascolo. A community based mobility model for ad hoc network research. In *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality*, pages 31–38. ACM, 2006.

[39] N. N.S.-Djukic, M. Pidrkowski, and M. Grossglauser. Island hopping: Efficient mobility-assisted forwarding in partitioned networks. In *Proc. of IEEE SECON '06*, 2006.

[40] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 453:779–782, june 2008.

[41] Chen Zhao and M.L. Sichitiu. N-body: Social based mobility model for wireless ad hoc network research. In *Proc. of IEEE SECON 2010*, 2010.

[42] D. Fischer, K. Herrmann, and K. K. Rothermel. GeSoMo: A general social mobility model for delay tolerant networks. In *IEEE MASS 2010*, 2010.

[43] Aarti Munjal, Tracy Camp, and William C. Navidi. Smooth: a simple way to model human mobility. In *Proc of ACM MSWiM '11*, 2011.

[44] I. Rhee, M. Shin, S. Hong, K. Lee, S.J. Kim, and S. Chong. On the levy-walk nature of human mobility. *IEEE/ACM Trans. Netw.*, 19(3):630–643, jun 2011.

[45] Swim: The website. http://swim.di.uniroma1.it.

[46] J. Leguay, A. Lindgren, J. Scott, T. Friedman, and J. Crowcroft. Opportunistic content distribution in an urban setting. In *CHANTS '06: Proceedings of the 2006 SIGCOMM workshop on Challenged networks*, 2006.

[47] J. Leguay, A. Lindgren, J. Scott, T. Riedman, J. Crowcroft, and P. Hui. CRAWDAD trace upmc/content/imote/cambridge (v. 2006–11–17). Downloaded from http://crawdad.cs.dartmouth.edu/upmc/content/imote/cambridge, November 2006.

[48] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau. CRAWDAD trace cambridge/haggle/imote/cambridge (v. 2006–01–31). Downloaded from http://crawdad.cs.dartmouth.edu/cambridge/haggle/imote/cambridge, January 2006.

[49] D. Kotz, T. Henderson, and I. Abyzov. CRAWDAD data set dartmouth/campus (v. 2007-02-08). http://crawdad.cs.dartmouth.edu/dartmouth/campus.

[50] H. Cai and D. Y. Eun. Toward stochastic anatomy of inter-meeting time distribution under general mobility models. In *Proc. of ACM MobiHoc '08*. ACM, 2008.

[51] E. Yoneki, P. Hui, S. Y. Chan, and J. Crowcroft. A socio-aware overlay for publish/-subscribe communication in delay tolerant networks. In *Proc. of ACM MSWiM '07*, 2007.

[52] D.M. Endres and J.E. Schindelin. A new metric for probability distributions. *IEEE Transactions on Information Theory*, 49(7):1858–1860, july 2003.

[53] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, June 2005.

[54] Derek G. Murray, Eiko Yoneki, Jon Crowcroft, and Steven Hand. The case for crowd computing. In *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, MobiHeld '10. ACM, 2010.

[55] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of human mobility on opportunistic forwarding algorithms. *IEEE Transactions on Mobile Computing*, June 2007.

[56] Ramesh Johari and John N. Tsitsiklis. Parameterized supply function bidding: Equilibrium and efficiency. *Oper. Res.*, September 2011.

[57] Paul D Klemperer and Margaret A Meyer. Supply function equilibria in oligopoly under uncertainty. *Econometrica*, November 1989.

[58] Hal R. Varian. *Microeconomic Analysis*. Norton, 1992.

[59] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[60] J.P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proc. of Working IEEE/IFIP Conference on Software Architecture*, 2002.

[61] Rajesh Krishna Balan, Mahadev Satyanarayanan, SoYoung Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proc. of The 1st International Conference on Mobile Systems, Applications, and Services*, pages 273–286, 2003.

[62] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. The case for cyber foraging. In *Proc. of the 10th ACM SIGOPS European Workshop*, 2002.

[63] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP)*, 1997.

[64] O. Riva J. Porras and M. D. Kristensen. *Dynamic Resource Management and Cyber Foraging*, chapter Middleware for Network Eccentric and Mobile Applications. Springer Press, 2008.

[65] Andres Lagar-Cavilla Niraj, Niraj Tolia, Rajesh Balan, Eyal De Lara, and M. Satyanarayanan. Dimorphic computing. Technical report, 2006.

[66] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. of ACM MobiSys*, 2011.

[67] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proc. of Mobicom '11*, 2011.

[68] A.P. Miettinen and J.K. Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proc. of HotCloud 2010*, 2010.

[69] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th*

Annual Computer Security Applications Conference (ACSAC), Austin, Texas, December 2010.

[70] K. Kumar and Y. H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *IEEE Computer*, 43(4):51–56, April 2010.

[71] Y. Wen, W. Zhang, and H. Luo. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *Proc. of IEEE INFOCOM 2012*, 2012.

[72] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proc. of the 3rd IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[73] S. Gurun, C. Krintz, and R. Wolski. Nwslite: A light-weight prediction utility for mobile devices. In *Proc. of International Conference on Mobile Systems, Applications, and Services*, 2004.

[74] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14 –23, oct.-dec. 2009.

[75] Adam Wolbach, Jan Harkes, Srinivas Chellappa, and M. Satyanarayanan. Transient customization of mobile computing infrastructure. In *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*. ACM, 2008.

[76] E.Y. Chen and M Itoh. Virtual smartphone over ip. In *Proc. of IEEE WoWMoM '10*, 2010.

[77] Eric Y. Chen and Mistutaka Itoh. Virtual smartphone over IP. In *Proceedings of the IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6, Los Alamitos, CA, USA, jun 2010. IEEE Computer Society.

[78] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In

SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 164–177, New York, NY, USA, 2003. ACM.

[79] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[80] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, 2010.

[81] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. of OSDI '10*, 2010.

[82] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *Proc of ACM CCSW '10*, 2010.

[83] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. of OSDI '10*, 2010.

[84] D.A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. of ACM UIST '95*, 1995.

[85] M. Ressel, D. N.-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. of ACM CSCW '96*, 1996.

[86] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proc. of the international ACM GROUP '97*, 1997.

[87] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. of ACM PODC '02*, 2002.

[88] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of OSDI '04*, 2004.

[89] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc of ACM PODC '07*, 2007.

[90] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. of IEEE/IFIP DSN '09*, 2009.

[91] D. Wang and A. Mah. Google wave operational transformation. http://www.waveprotocol.org/whitepapers/operational–transform, 2010.

[92] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of CSCW '98*, 1998.

[93] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[94] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. In *Proc. of ACM CSCW '04*, 2004.

[95] A. Karsenty and M.B.-Lafon. An algorithm for distributed groupware applications. In *Proc. of ICDCS '93*, 1993.

[96] Zhi Wang Yajin Zho and, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of NDSS '12*, 2012.

[97] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *ACM SIGCOMM Workshop on Social Networks*, 2009.

[98] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, 2006.

[99] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[100] Zhichao Zhu, Guohong Cao, Sencun Zhu, S. Ranjan, and A. Nucci. A social network based patching scheme for worm containment in cellular networks. In *INFOCOM 2009, IEEE*, 2009.

[101] N.P. Nguyen, Ying Xuan, and M.T. Thai. A novel method for worm containment on dynamic social networks. In *Military Communications Conference (MILCOM 2010)*, 2010.

[102] James W. Mickens and Brian D. Noble. Modeling epidemic spreading in mobile environments. In *Proceedings of the 4th ACM workshop on Wireless security*, WiSe '05, 2005.

[103] Guanhua Yan and S. Eidenbenz. Modeling propagation dynamics of bluetooth worms. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, 2007.

[104] C. J. Rhodes and M. Nekovee. The opportunistic transmission of wireless worms between mobile devices. *PHYSICA A-STATISTICAL MECHANICS AND ITS APPLICATIONS*, 387(27):6837–6844, 2008.

[105] Hui Zheng, Dong Li, and Zhuo Gao. An epidemic model of mobile phone virus. In *Pervasive Computing and Applications, 2006 1st International Symposium on*, 2006.

[106] Wei Peng, Feng Li, Xukai Zou, and Jie Wu. Behavioral detection and containment of proximity malware in delay tolerant networks. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, 2011.

[107] Gjergji Zyba, Geoffrey Voelker, Michael Liljenstam, Andras Mehes, and Per Johansson. Defending Mobile Phones from Proximity Malware. In *Proc. of IEEE INFOCOM '09*, 2009.

[108] Feng Li, Yinying Yang, and Jie Wu. Cpmc: An efficient proximity malware coping scheme in smartphone-based mobile networks. In *INFOCOM, 2010 Proceedings IEEE*, 2010.

[109] MHR Khouzani, S. Sarkar, and E. Altman. Dispatch then stop: Optimal dissemination of security patches in mobile wireless networks. In *49th IEEE Conference on Decision and Control (CDC)*, pages 2354–2359. IEEE, 2010.

[110] Yong Li, Pan Hui, Depeng Jin, Li Su, and Lieguang Zeng. An optimal distributed malware defense system for mobile networks with heterogeneous devices. In *IEEE SECON 2011*, 2011.

[111] Guanhua Yan and Eidenbenz. Modeling Propagation Dynamics of Bluetooth Worms. *IEEE Transactions on Mobile Computing*, 8(3):1071, 2008.

[112] G. Yan, H.D. Flores, L. Cuellar, N. Hengartner, S. Eidenbenz, and V. Vu. Bluetooth worm propagation: mobility pattern matters! In *Proc. of ACM symposium on Information, computer and communications security*, page 44, 2007.

[113] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Transactions on Computer Systems (TOCS)*, 26(4):9, 2008.

[114] D.J. Daley and J.M. Gani. *Epidemic modelling: an introduction*. Cambridge Univ Press, 2001.

[115] M.H.R. Khouzani, S. Sarkar, and E. Altman. Maximum damage malware attack in mobile wireless networks. In *INFOCOM, 2010 Proceedings IEEE*, 2010.

[116] P. Wang, M.C. Gonzalez, C.A. Hidalgo, and A.L. Barabasi. Understanding the spreading patterns of mobile phone viruses. *Science*, 324(5930):1071, 2009.

[117] Liang Xie, Xinwen Zhang, A. Chaugule, T. Jaeger, and Sencun Zhu. Designing system-level defenses against cellphone malware. In *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*, 2009.

[118] Abhijit Bose and Kang G. Shin. Proactive security for mobile messaging networks. In *Proceedings of the 5th ACM workshop on Wireless security*, WiSe '06, 2006.

[119] E. Van Ruitenbeek, T. Courtney, W.H. Sanders, and F. Stevens. Quantifying the effectiveness of mobile phone virus response mechanisms. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, june 2007.

[120] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, 2008.

[121] J. Tang, H. Kim, C. Mascolo, and M. Musolesi. Stop: Socio-temporal opportunistic patching of short range mobile malware. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a*, pages 1–9. IEEE, 2012.

[122] C. Fleizach, M. Liljenstam, P. Johansson, G.M. Voelker, and A. Mehes. Can you infect me now? malware propagation in mobile phone networks. In *Proc. of ACM workshop on Recurring malcode*, page 68, 2007.

[123] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30:107–117, April 1998.

[124] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Robustness of social networks: comparative results based on distance distributions. In *Proceedings of the Third international conference on Social informatics*, SocInfo'11, pages 8–21, 2011.

[125] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, 1(4):33–39, August 2003.

[126] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, and Chulkoo Kang. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of USENIX ATC 12*, 2012.

[127] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, 1996.