Minimally Invasive Solutions to Challenges Posed by Mobility Changes

Joshua Reich

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2011

ABSTRACT

Minimally Invasive Solutions to Challenges Posed by Mobility Changes

Joshua Reich

The first computerized systems were completely immobile. During participation in computation, user, device, and software instance were tightly coupled: each had to remain in direct physical contact with the others.

Today, things have changed radically. As network technologies have proliferated and evolved, the components of, and participants in, computerized systems have become increasingly decoupled. Users travel and commute while connecting to their office computer or home media server. Hardware devices may be carried by users, move on their own, or reside in data centers, never to be seen or touched by end-users. Even *operating systems (OSes)* and applications may now migrate across the network while executing, thanks to advances in virtualization that are only just beginning to remake the computing landscape.

The decoupling of users, devices, and software has invalidated properties that enabled desired functionality: resulting in compromised function. Power interfaces utilize physical user interactions to determine when transitions between high and lower power states should occur; what happens when users are no longer physically present? Operating system execution often relies on components such as CPU and local disk responding with tightly bounded delays; what should be done when the OS itself is in the process of migrating between two separate physical machines?

The fundamental question explored by this dissertation is:
Can we find highly adoptable solutions to restore desired functionality that has been lost because of changed mobility characteristics?

Our emphasis on adoptability stems from pragmatic concerns: if a solution is difficult to adopt, it is highly unlikely to be used. Consequently, while many potential approaches may involve changes to the network itself, our work focuses on modifying end-point behavior.

We show that practical solutions *implemented solely in software* and *deployed only on network endpoints* can be developed for a wide problem range. We consider concrete challenges arising from user, device, and software mobility changes, affecting sub-disciplines spanning *cloud computing*, *green computing*, and *wireless networks*.

Cloud Computing: Users increasingly utilize *virtual machine (VM)* technology to migrate and replicate OS and software amongst networked hosts. Traditional execution required one VM image copy on each host's local storage. By transitioning to networked execution, dozens, if not hundreds, of VM replicas may now be distributed from a single networked storage location to a commensurately large set of physical machines. As these systems expand, they have come to be plagued by *boot storms* (and similar problems) caused when networked access to storage becomes a major bottleneck, drastically delaying VM distribution and execution. Can we develop techniques that resolve this network bottleneck without the need for expensive hardware over-provisioning?

Green Computing: Remote access technologies have enabled users to travel while still interacting with computational machinery left in the office or home. Yet, energy savings mechanisms have traditionally relied on the activity of attached peripherals to determine power usage. The shift to remote interaction, which bypasses physically attached peripherals, has effectively broken these energy savings mechanisms. Can we build an economic and practical system that accommodates energy efficiency without compromising the fluid remote interactions users have now come to expect?

Wireless Computing: Increasingly advanced mobile devices have provoked a shift towards heavy usage of 3G and 4G bandwidth use. Accordingly, the capacity of infrastructure wireless networks becomes increasingly strained. Can we find a way of supplementing this relatively low-latency infrastructure with high-latency, high-bandwidth opportunistic content exchange?

In each scenario, we design a solution that aims to strike the proper balance between adoptability and technical efficiency - producing what we believe are rigorous, practical and adoptable solutions.

Table of Contents

List of Figures

This page intentionally left blank

List of Tables

This page intentionally left blank

Acknowledgments

What a long and peculiar road this has turned out to be. When I started this PhD, I had no idea what I was getting myself into. Honestly, looking back, I was fairly clueless about quite a many things, including understanding what research actually was. If I had to go back and do it all over again, I probably would do almost everything differently. I guess that means I've managed to learn something after all!

Despite my many missteps, I couldn't be more pleased with the way things have worked out. These have been the best and most challenging years of my life. I am incredibly grateful to have had this time to grow, learn, explore, travel the world, and do research that I'm truly proud of. I never imagined things would work out as they have. It has been an amazing journey and one I'm looking forward to continuing. I so deeply appreciate the kindness, support, company, and guidance of all those individuals who helped me along the way.

The best place to start is with my advisers, Professors Vishal Misra and Dan Rubenstein. Although I started the MS-track PhD program in fall of 2002, I didn't start working with Dan and Vishal until spring of 2006. They took me on at a point in my academic career when it was unclear whether this document would ever be produced. I will always appreciate their betting on me. In the past half-decade that I've worked with them, they've taught me a tremendous amount about the discipline of networking (in which I had not even taken a class before I joined their lab), doing quality research, and being a true professional. Dan and Vishal have continually supported me, encouraged me to take risks, expose myself to different research styles through interning around the world, and provided a steady stream of constructive feedback. When I started working with Dan and Vishal, I barely had an idea of what real research looked like - today, I'm putting the finishing touches on a dissertation better than I could have imagined producing before I began working with them.

Having two advisers who work so well together has been a great gift. Dan has continually

encouraged me to think out of the box, urging me to pursue the ideas I care about most - instead of worrying whether a given project fits into the story I had planned to construct. Vishal has helped me hone my problem-selection instincts and has continually challenged me to think deeply about the fundamental issues my work explores. The unique combination of their advisement has its fingerprint on practically every page of this dissertation. They have given me the tools and guidance to explore the widely differing technical challenges to which I've been drawn, while simultaneously rooting these pieces of work in a common foundation that reveals their subtle inter-relations.

Of course, I would have never ended up working with Dan and Vishal if not for the mentorship and advice of several of my personal heros in the CS faculty. Professor Jonathan Gross has served for the better part of a decade as my secondary adviser. His classes, more than anything else, piqued my interest in CS as a discipline and provided the bridge to CS from my undergraduate studies in Mathematics. His support and mentorship have been invaluable to me. Without these, I doubt I would have started, or finished, this work.

Likewise, Professors Tal Malkin and Rocco Servedio have always been there to offer me advice and guidance. Both played an instrumental role in my transition to working with Dan and Vishal.

Since, I first took his OS class at the beginning of my studies, Professor Jason Nieh has always provided me good advice and been generous with his time. Jason is one of those rare researchers who makes every single question count. I've been continually impressed with his focus and incisiveness, qualities I am working to emulate.

Moreover, in working with Jason, I gained not one mentor, but two. Jason's student Dr. Oren Laadan has taught me more about systems work than anyone else with whom I've worked. His patience, good humor, and relentlessly constructive style of engagement have forced me to up my game time and time again. In watching Oren, I've learnt much about the kind of researcher and professional I aspire to be.

My peers in the department have both helped me learn the ropes and provided valuable guidance, feedback, collaboration, and, not least, commiseration. Vanessa Frias-Martinez shared some of the most eventful times with me, while Stelios Sidiroglou-Douskos and Abhinav Kamra were incredible helpful in easing my transition into the world of systems and networking. Alex Sherman and Salman Baset shared the long road with me, both as collaborators and good friends. Most especially, my long-time office mate, Eli Brosh, has always taken the time to listen, brainstorm,

and shoot the breeze. Eli has been a presence without whom my years at Columbia would have been far less rich.

The staff of our CS department is top notch, filled with helpful and wonderful folks. Mary, Rosemary, Pat, Elias, Remi, Jessica, Cindy, Sophie, Twinkle, and Alice have provided both assistance and contagious smiles. The staff of CRF have given me generously of their time and support, especially in the last two years over which I've had critical experiments running on their infrastructure. Daisy, Shlomo, Paul, Hi-tae, and Quy have helped me jump technical hurdles and become friends in the process.

Being at Columbia University has also exposed me to the fantastic visitors our department attracts. I've had the privilege of sitting in on seminars given by Dr. Pablo Rodriguez and Dr. Erich Nahum. I count myself lucky to have had access to each of their unique viewpoints. The counsel they've given me on research and career choices has proved invaluable.

Likewise, I've had the pleasure to work with Professor Gil Zussman from Columbia's Department of Electric Engineering. Gil has been a fantastic mentor, pushing me to refine my work and putting in long hours brainstorming and writing with me.

The time I've spent away from Columbia has proven to be as valuable as the time I spent at the University. I have been extremely fortunate in having had the opportunity to collaborate with so many top-notch researchers at fantastic institutions.

Dr. Venkat Padmanabhan offered me the chance to intern at Microsoft Research India, my first true research internship. I learned much from observing his gentle manner, good nature, and fantastic research insight.

Then Dr., now Professor, Augustin Chaintreau has been a long-term mentor and friend, ever since he supervised my internship at Technicolor's Paris research laboratory. The enthusiasm Augustin brings to all that he does inspires me, as does his analytic prowess.

Working with Dr. Jitu Padhye at Microsoft Research Redmond was both a privilege and a superb learning experience. We got a tremendous amount of work done in an amazingly short time. Jitu showed me what it looks like to always bring your A-game to the table.

At all the places I've interned, I've been fortunate to interact with terrific computer scientists. While I lack space to name everyone, several instances stand out: Brian Zill, Michel Goraczko, and Aman Kansal's patient help at Microsoft Research Redmond; Ram Ramjee's entertainingly

argumentative company at Microsoft Research India lunches; Christophe Diot's gregariousness and generosity, Laurent Massoulie's quiet consideration, and Fernando Silveira's quick mind and fast friendship at Technicolor.

Although, I've never met him face-to-face, I've come to think of Mike Hilber as a collaborator of sorts. Without Mike's help, I honestly do not know if the first section of this thesis would have been written. He and the other folks at Utah Emulab are truly fantastic.

To graduate, I've needed to pass my oral examinations, thesis proposal (which ended up having nothing to do with the document you currently read), and my thesis defense. I greatly appreciate participation and feedback provided by the Columbia faculty who served on these various committees and whom I haven't yet mentioned: Professors Gail Kaiser, Angelos Keromytis, Yechiam Yemini, and Henning Schulzrinne. I owe a special thanks to Professor Jennifer Rexford for serving as the external member on this dissertation's defense committee. Jennifer is an incredible researcher and has been amazingly generous with her time, providing some of the most detailed and thoughtful comments that I have ever received on my work.

Finally, I owe a deep debt of gratitude to my closest friends and family, without whom I would be lost. In particular, Perry Vais provided help and feedback during my most critical writing period, over the 4th of July weekend. Nancy Siegel has acted as a beacon, guiding me through some of the darker moments in my PhD, and lighting up the good ones more brightly. My brother Daniel and sister-in-law Mia have always been there for me. They are two of the best friends anyone could wish for. Despite our having lived thousands of miles apart for most of a decade, Daniel's presence in my life has been closer and more tangible than almost anyone else's. My dear mother-in-law Judit has been a constant friend these past several years; her confidence has continually bolstered me.

My parents have been an unwavering source of support. Their efforts made me the man I am today. My mother is possessed of an incredible generosity of spirit, it is amazing to have someone in my life who I know would do practically anything for me. My father has, over the past several years, become my biggest cheerleader and most steadfast supporter; his pride warms me. I love my parents dearly, as I do my grandparents of blessed memory. Grandpa Wilbur and Grandpa Harold did not live to see me start my studies, but Bubba did and I know the joy she took from knowing I was about to embark on a PhD. Grandma Dorothy bought me the computer I used for the first

several years of my studies and was always eager to hear the latest news of my unexciting life. I miss them all dearly, and wish they could have lived to see me graduate.

Last, I come to where I hope all my journeys end, the love of my life, Linda Lantos. As corny as it sounds, from the moment I met her, Linda seemed to be in my life to stay. She has pushed me to realize my potential, even when that has been painful and difficult. She is my closest friend and confidant. She has truly been an *ezer k'negdi*. Marrying Linda was the best decision I have ever made. My life with her is far richer than I could have imagined. This dissertation would not exist if she had not become part of my life. She deserves a degree for slogging through this experience with me, though all I can offer are my thanks and love.

Joshua Reich
New York, NY
24 August 2011

To my grandparents, who supported me.

To my parents, who raised me.

To my wife, who is with me every step of the way.

Chapter 1

Introduction

The first computerized systems were completely immobile. During participation in computation, user, device, and software instance were tightly coupled: each had to remain in direct physical contact with the others.

Today, things have changed radically. As network technologies have proliferated and evolved, the components of, and participants in, computerized systems have become increasingly decoupled. Users travel and commute while connecting to their office computer or home media server. Hardware devices may be carried by users, move on their own, or reside in data centers, never to be seen or touched by end-users. Even *operating systems (OSes)* and applications may now migrate across the network while executing, thanks to advances in virtualization that are only just beginning to remake the computing landscape.

The decoupling of users, devices, and software has invalidated properties that enabled desired functionality: resulting in compromised function. Power interfaces utilize physical user interactions to determine when transitions between high and lower power states should occur; what happens when users are no longer physically present? Operating system execution often relies on components such as CPU and local disk responding with tightly bounded delays; what should be done when the OS itself is in the process of migrating between two separate physical machines?

One approach to dealing with such problems advocates a complete redesign of the system from first principles [Stuckmann and Zimmermann, 2009]. Applied to our space, this type of re-design would not only incorporate the current mobility characteristics of the system, but also attempt to predict how these will evolve. In doing so, a sufficiently flexible set of assumptions may hopefully

be constructed so as to avoid need for another such re-design. Constructing such a flexible model appears to be an incredibly tough challenge in-and-of itself, especially given that no end to the trend towards increasingly complex mobility appears in sight. Adding to this difficulty, past performance demonstrates notoriously bad prediction as to how mobility will evolve. Finally, even if these hurdles can be overcome, there is still the question as to the practicality of such an overhaul. The IPv6 protocol [Deering and Hinden, 1998], successor to the first publicly used version of the Internet protocol IPv4 [Postel, 1981a], makes for an arguably incremental change. Yet, it is only now, almost a decade-and-a-half since its first commercial implementation [Jim, 1998], that IPv6 is finally being adopted in earnest [Society, 2011] - and only then in response to the complete exhaustion of the IPv4 address space [ICANN, 2011].

Consequently, this dissertation work takes another approach. We ask:
Can we find highly adoptable solutions to restore desired functionality that has been lost because of changed mobility characteristics?

Our emphasis on adoptability stems from pragmatic concerns: if a solution is difficult to adopt, it is highly unlikely to be used. Consequently, while many potential approaches may involve changes to the network itself, our work focuses on modifying endpoint behavior. Restricting modification to network endpoints encourages adoption in several ways. It is likely to be *economically advantageous*. If a solution requires greater expense to deploy and maintain than the savings that solution provides, the chances of successful adoption become vanishingly small. Avoiding changes to the core network also supports *incremental deployability*, as endpoints may be changed one at a time without dramatically affecting the network's functioning. Even if a solution proves economic and efficient in the long term, should the activation energy required to switch to that solution be too high, adoption will be delayed indefinitely.

These considerations inform our emphasis on *resolving*, not eliminating problems. A technical solution that provides the best performance with respect to raw metrics such as *energy efficiency*, *wait time*, or *input/output operations per second (IOPS)* will not necessarily be the most successful ones. In most cases, the consumer will decline to spend 2X more for a 10% performance gain. Rather, we focus on solutions that provide the right balance of technical performance and adoptability.

We show that practical solutions *implemented solely in software* and *deployed only on network*

endpoints can be developed for a wide problem range. We consider concrete challenges arising from user, device, and software mobility changes, affecting sub-disciplines spanning *cloud computing*, *green computing*, and *wireless networks*.

1.1 Cloud Computing

The first challenge we explore lies in the realm of cloud computing. Here, maturing *virtual machine (VM)* [Smith and Nair, 2005] technology combined with high performance networks has enabled migration and replication of an OS along with its software and settings, across many physical devices. A hypervisor decouples OS from hardware, while the network provides for high-speed movement of VM images. In this way, IT infrastructure, web application servers, and user desktops, may be moved onto shared hardware - allowing for reduction of administrative complexity and efficiencies of scale.

Whereas traditionally VMs have been relatively immobile, VM-based computation in the cloud may be moved based on hardware availability, energy costs, network constraints, etc. Likewise, by replicating VMs, services may be expanded quickly in response to demand.

However, challenges arise in applying VM technology to this new use paradigm. VMs have traditionally been executed from locally stored images. With the move to the cloud, VM images have come to be stored remotely on a *storage area network (SAN)* or *network-attached storage (NAS)*, making the network a serious potential performance bottleneck. Now in order to execute software, instructions must be read over the network, leading to pathologies such as *boot storms* in which VM-based *virtual desktop* execution slows to a crawl as network resources are overwhelmed by image access. Making this problem even more challenging is the unpredictability of demand for particular VM images in large public or private clouds. To achieve the full power of this model, VM instantiation, migration, and replication must all be accommodated with minimal delay, under potentially unpredictable changes in demand.

To address this problem, we design and implement of a system (vmTorrent) capable of quickly and scalably distributing and executing VM images. vmTorrent's custom front-end file server allows for VM quick-start in which VMs can execute while their images traverse the network. To this is attached a P2P back-end supporting efficient scaling. Finally, vmTorrent's intelligent

pre-fetching algorithms enable smooth streaming execution: balancing the needs of local execution (image pieces should be prefetched based on anticipated need) and swarm efficiency (prefetching may need to be randomized to ensure piece diversity sufficient to fully exploit swarm upload capacity).

1.2 Green Computing

Our second challenge is one of "green computation". Remote login/access capabilities have become widespread over the past decade. Thus a large number of enterprise users now expect they will be able to interact at will with their work machine, whether they are at home or on the road.

However, traditionally, desktop OS power management schemes assume users and machines will be physically collocated. If a user is present and active the machine stays awake, if not, the machine transitions to a low-power mode. If the user returns after the machine has fallen asleep, interaction with attached input devices will wake the machine. The introduction of remote access violates this model, as remote access bypasses physically attached input devices.

Sleeping machines now must be prepared to wake on appropriate network events in addition to those occurring on local peripherals, functionality unsupported by current power management designs. Consequently, most users adopt the wasteful, but convenient, strategy of idling their desktop computers 24/7 to support what is typically very occasional remote use [Nedevschi *et al.*, 2008; Agarwal *et al.*, 2009; Webber *et al.*, 2006; Allman *et al.*, 2007]. The environmentally-conscious remainder turn their machines off when leaving the office, but at the cost of potentially lost productivity.

Previously proposed solutions have not been adopted in the enterprise. Arguably, this is because these proposals have failed the adoptability test. The simplest solutions [amt,] merely extend Wake-on-LAN (WOL) [wol,] capabilities to Wide Area Network (WAN) accesses, requiring end-users to modify their behavior. In these solutions, before beginning to work, users needed to run a separate program that manually woke their machine. Despite requiring a relatively small change in behavior, even this bar proved too high for significant adoption. More recent proposals have been seamless, technically efficient [Agarwal *et al.*, 2010], and occasionally even elegant [Das *et al.*, 2010]. However, these proposals have required significant investment in new hardware as well as

relatively involved changes to client behavior. It is our belief that the costs for deployment and upkeep have outweighed the (currently) small monetary savings even the most efficient solution might provide. Consequently, these too have not been adopted.

In response, we architect and implement a non-intrusive, economical, network-based sleep-proxying system to address this problem. Each desktop machine runs a lightweight daemon that, immediately preceding transition to sleep states, informs a sleep proxy sitting on the local network. The network sleep proxy then redirects and monitors traffic incoming to the sleeping host, waking the host as appropriate.

While our lightweight design does not always provide as great power savings as other recent proposals, it does handle the frequently encountered cases well. Furthermore, by choosing a lightweight approach, our technique requires an order of magnitude less hardware and IT support - while still realizing up to 90% of potential energy savings.

1.3 Wireless Computing

Wireless computing is the final area we examine. While at first slow and expensive, cellular data plans have become increasingly more attractive in terms of price and performance. With the introduction of devices well-designed to take advantage of this connectivity - most notably Research in Motion's Blackberry in the enterprise, followed several years later by Apple's iPhone - users have jumped on board *en masse*. Earlier adopters of cellular data networks had mostly limited their use to low-bandwidth applications such as email, weather forecasts, news updates, and light web-browsing. Now, encouraged by telecoms, user demand for high-bandwidth content like apps and video while on the move is increasing rapidly - content which had previously been demanded solely over their cable or ADSL network feed. Further this demand comes from not only smartphones, but a plethora of devices including tablets, netbooks, chromebooks, and traditional laptops. This demand has overwhelmed the capacity of carriers' 3G [Cheng, 2008b] and fledgling 4G networks [Lawson, 2011; Wortham, 2011], leading to outages, widespread user dissatisfaction, lawsuits [Cheng, 2008a], and even organized grassroots protest [Heussner, 2009]. Carriers are struggling to improve their networks quickly enough, but the required overhaul of centralized and expensive nationwide infrastructure takes years, if not decades, while shifts in the growth of

demand are occurring at much shorter timescales [Arar, 2011].

Our work begins with the observation that not all demand is equal. Many of the most time-critical interactions, *e.g.,* email, chat, news, are relatively low-bandwidth. As demonstrated by the popularity of slow but inexpensive multimedia delivery mechanisms such as BitTorrent [Sandvine, 2010] and Netflix DVD [Seetharam *et al.*, 2010], users are willing to wait to obtain high bandwidth multimedia content - at least so long as price is right. If low-latency, medium-bandwidth centralized cellular infrastructure capacity can be supplemented by some other medium-to-high-latency, high bandwidth mechanism, then user demand might be satisfied both more successfully and more economically than the current cellular-infrastructure only solution.

Fortuitously, the very mobility characteristics of users, combined with the increasingly large storage and local radios of the devices they carry, may provide the material for building just such a mechanism. A large pool of bandwidth lies untapped in the chance contacts of mobile devices. At each such contact, meeting nodes might exchange and replicate locally stored content at high data rates for very low cost (essentially just battery drain). However, utilizing these opportunities poses a significant challenge; device memory is finite and this latent bandwidth is often unstructured and unpredictable.

Consequently, we model how such a system might work, focusing on user *impatience*: the function describing the decreasing utility users find as the wait for fulfillment of their demand increases. We then show how that under certain conditions the optimal memory usage policy can not only be described, but also approximated using a lightweight distributed mechanism. Moreover the information required to determine what content should be replicated by any pair of meeting nodes is entirely local, given knowledge of the impatience curve.

1.4 Contributions

The contributions of this dissertation break down naturally by the scenario explored. We begin with our contributions in the cloud.

1. We present the vmTorrent architecture, incorporating quick-start, scalability, and smoothness features into a coherent design. (Section 2.3)

2. We develop intelligent profiling and piece selection mechanisms tailored to the problem of VA

deployment and execution. (Section 2.4 and 2.5)

3. We implement a functional vmTorrent prototype. (Section 2.6.1)

4. We measure our prototype's performance on a variety of Virtual Desktop Infrastructure (VDI) tasks executed on both Windows and Linux-based VMs. For these we run our prototype on a hardware testbed, using up to 100 physical client peers. Our experimental results demonstrate our design choices produce a scalable system. vmTorrent delivers up to an **11X improvement** over a standard P2P approach that does not incorporate intelligent prediction and a **30X improvement** over demand-based streaming approach. In fact, the vmTorrent runtimes remain nearly as good as local disk execution for all workload sizes. (Section 2.6)

The next set of contributions are those made in reducing wasteful power use.

5. We design and prototype a lightweight and economical sleep-proxying system for use in the enterprise. (Sections 3.3 and 3.4)

6. We roll out the first substantial deployment of any sleep-proxying system in a corporate environment. We deploy our software on over 50 user machines in six subnets. Almost all of these machines are primary user workstations. (Section 3.6)

7. We measure the performance of our system, collecting over half-a-year's worth of data. We instrument our system extensively; capturing numerous details about sleep and wake periods, data which explains why machines wake up and *why they stay up*. Instead of using generic estimates of PC power consumption, we use a sophisticated *software*-based, model-driven system, *Joulemeter*, to estimate power draw. (Sections 3.5 and 3.7)

8. Additionally, we describe a number of practical issues we encountered when deploying a lightweight sleep proxy in a corporate network. Many of these have been overlooked by previous work. For example, our implementation must not only deal with vanilla IPv4 and IPv6 packets, but also tunneled packets. Our corporate network uses IPsec, and we find that *a seemingly minor implementation choice* in this setup, almost entirely removes the need to deal with this traffic. We describe race conditions that arise when the sleep proxy attempts

to redirect traffic from the sleeping client to itself, and provide a practical solution. We show how issues such as DHCP lease expiration and proxy failure can be handled without the need for the more complex mechanisms suggested by previous work. (Section 3.4)

9. Finally, we outline several unexpected insights provided by our work, the most significant of which indicates that the power of a sleep proxy's reaction policy - one of the major concerns of previous work - currently plays a secondary role in determining energy savings. The primary factor turns out to be the configuration of IT software and network services. (Section 3.7)

Lastly, we present our contributions in wireless.

10. We demonstrate that user impatience plays a critical role in determining the optimal allocation for disseminating content. We further find a surprisingly general behavior which holds over a wide variety of particular delay-utility functions: As the user population becomes increasingly impatient, the optimal allocation transitions steadily from uniformly dividing the global cache between all content items, towards a highly-skewed distribution in which popular items receive a disproportionate share of the global cache. We obtain these results by defining an optimal cache allocation in terms of delay-utility and global cache allocation. (Section 4.3)

11. Furthermore, we demonstrate that this optimal allocation is unique and can be computed efficiently in a centralized manner. Under the simplified assumption of homogeneous meeting rates, we show that the corresponding optimal cache allocation is known in closed form for a general class of delay-utility functions. (Section 4.4)

12. Inspired by these results, we develop a reactive distributed algorithm, *Query Counting Replication (QCR)* that for any delay-utility function drives the global cache towards the optimal allocation. Moreover QCR does so without use of any explicit estimators or control channel information. (Section 4.5.1)

13. We show the implementation of QCR in opportunistic environments is non-trivial and demonstrate a novel technique *Mandate Routing* to avoid potential pathologies that arise in insufficiently fluid settings. (Section 4.5.3)

14. Finally, we validate our techniques on real-world contact traces, demonstrating the robustness of our analytic results in the face of heterogeneous meeting rates and bursty contacts. We

find QCR compares favorably to a variety of heuristic competitors, despite those competitors having access to a *perfect control-channel* and QCR relying solely on locally available information. (Section 4.6)

1.5 Organization

As this dissertation's three areas of investigation are technically disparate, each topic is given one chapter which includes related work. Chapter 2 covers our work on efficient and scalable VM distribution and execution in the cloud. Chapter 3 presents our work on saving energy through the use of a lightweight and economical sleep-proxying architecture. Chapter 4 deals with our efforts to utilize abundant untapped bandwidth in mobile networks who infrastructure backbones are straining under the pressure of unprecedented demand patterns. Finally, Chapter 5 provides some brief conclusions and discusses directions for future research.

This page intentionally left blank

Chapter 2

Cloud Computing: Deploying Virtual Machines Scalably

2.1 Overview

Traditionally, *virtual machines (VMs)* have been run using VM images stored on the local disk and accessed by the *virtual machine monitor (VMM)*[1]. By combining host-level virtualization with modern networking and data center facilities, the *cloud computing* paradigm has enabled a wide range of new applications for VM technology. Virtualized hardware clouds such as Amazon's EC2 and Microsoft's Hyper-V Cloud have enabled businesses to move their operations from physical servers and machines dedicated to running required applications to shared infrastructure on which *virtual servers* and *virtual appliances* may execute. In doing so, money is saved both through economies of scale. Hardware can be utilized more efficiently by leveraging statistical multiplexing of physical resources. Likewise, users and administrators of VMs can focus on their core competency while leaving the headache of maintaining and updating physical machines to the cloud infrastructure provider. Formerly complex tasks such as adding or removing (physical) machines are now replaced by simple operations such as cloning a VM image and booting the cloned image. For these same advantages, enterprises have begun eagerly adopting *virtual desktop infrastructure (VDI)* technology in which end-user desktop machines are encapsulated in VMs stored in the cloud.

[1] Also known as a *hypervisor*.

However, with the move to the cloud, VM images have come to be stored remotely on a *storage area network (SAN)* or *network-attached storage (NAS)*: making the network a serious potential performance bottleneck. Now, in order for VMs to execute, their images must be read over the network by the VMM on which they execute. This leads to pathologies, such as *boot storms* (as they are known in hosted VDI), in which VM execution slows to a crawl as network resources are overwhelmed by image access requests sent by VMMs.

Current VM distribution techniques either utilize a remote file-system abstraction or require download of a VM's complete virtual disk image, only after which the VM can be run. Given that compressed VM sizes run anywhere from several hundred MB to a few GB, there can be significant delays from the time a user decides he/she wants to run a particular VM until the time that VM can be used. Likewise, most current setups are unable to service the peak *input/output operations per second (IOPS)* rates required for seamless remote file-system-based execution. These problems are only exacerbated when demand for particular VMs spikes and NAS/SAN/server bandwidth resources become the distribution bottleneck[2]. Of course, hardware over-provisioning can resolve these problems, but only at significant economic cost.

The goal of our work is to ***minimize VM execution time***, whether to improve VDI end-user experience or speed up deployment and execution of tasks run on virtual servers and virtual appliances in the datacenter. We attack the problem by producing a system design based on three critical observations:

- Typically only a small fraction of the VM image needs to be accessed in order for it to run to completion.

- At any given time, especially in a datacenter environment, a large number of nearly identical VMs are simultaneously executing. This presents an opportunity to tap into the unused resources of these VMs (e.g., upload bandwidth in a datacenter rack) to speed up the distribution and execution of other similar "peer" VMs.

- And finally, there is a strong correlation in the image accesses across similar workloads, enabling the creation of predictive models for VM image access. Image access is also typically

[2]Section 2.2 discusses why multicast is ill-suited to addressing this challenge.

bursty, providing an opportunity to prefetch blocks that will be needed soon, based on the predictive model.

Inspired by these observations, we have built a peer-to-peer solution, vmTorrent, to support quick and scalable launching of VMs on-demand. Our solution is applicable to enterprise, and data-center settings and agnostic of the particular VMM used (e.g., VMware Player [VMware Player,], VirtualBox [VirtualBox,], Kernel Virtual Machine (KVM) [Kivity *et al.*, 2007]). vmTorrent provides architectural support for efficient just-in-time deployment and execution of VMs. When a new guest VM is spawned, the system begins downloading the required disk image from a P2P swarm. vmTorrent aims to fetch sections of the disk image (encapsulated in BitTorrent *pieces*) just in time for their use by the VMM.

We have found the incorporation of *all three* of these observations into our our system design to be essential. While elements of each have been incorporated individually in different prior work (as detailed in Section 2.2), our results demonstrate that only by incorporating *all* the elements, can true scalability be produced. We start off by describing our system design in the next section.

In this chapter we make the following contributions:

1. We present the vmTorrent architecture, incorporating *quick-start*, *scalability*, and *smoothness* features into a coherent design. (Section 2.3)

2. We develop intelligent profiling and piece selection mechanisms tailored to the problem of VM deployment and execution. (Section 2.4 and 2.5)

3. We implement a functional vmTorrent prototype. (Section 2.6.1)

4. We measure our prototype's performance on a variety of VM/task combinations. For these we run our prototype on a hardware testbed, using up to 100 physical client peers. Our experimental results demonstrate our design choices produce a scalable system. vmTorrent delivers up to an **11X improvement** over a standard P2P approach that does not incorporate intelligent prediction and a **30X improvement** over demand-based streaming approach. In fact, the vmTorrent runtimes remain nearly as good as local disk execution for all workload sizes. (Section 2.6)

2.2 Related Work

There are only a few studies of efficient on-demand deployment of virtual appliances or machines in the *local area network (LAN)*. A straightforward approach to deploy VMs is to sequentially copy them to the target nodes. This approach is common in data centers, which employ provisioning servers to distribute and execute pre-customized images on-demand [Mietzner and Leymann, 2008; Shi *et al.*, 2008]. However, sequential distribution can lead to long distribution times and network hotspots when VM demand is high.

Alternative distribution methods were studied in the context of cloud computing and educational environments, including: multicast, tree, and P2P [Schmidt *et al.*, 2010; Wartel *et al.*, 2010; O'Donnell, 2008]. In multicast [Albanna *et al.*, 2001; Hinden and Deering, 2006], data is simultaneously delivered to multiple recipients, reducing the load on the network and the server. However, multicast is still not widely deployed in many IP networks [El-Sayed *et al.*, 2003], is not geared towards on-demand image distribution which requires asynchronous data delivery, and is not well suited to delivering data to geographically distributed networks. An alternative approach constructs a distribution tree (at the application-layer) between the downloading nodes [Hosseini *et al.*, 2007]. While this approach enables parallel image delivery, it is highly sensitive to node churn.

Another distribution method is P2P. Typically, P2P delivery is based on BitTorrent [Cohen, 2003], an efficient solution for scalable content delivery [Qiu and Srikant, 2004]. The work in [Chen *et al.*, 2009] leverages this strategy to speedup the provisioning of cloud infrastructure. However, their approach applies this idea naively, taking more than 20 minutes before a VM can even begin execution. Contrastingly, by incorporating profile-based prediction, vmTorrent can fully execute VMs in a fraction of this time at scale.

To obtain efficient piece exchange, vmTorrent seeks to balance the immediate download requirement of the VMM and maintaining a high level of piece diversity in the system. This approach to creating diversity is inspired by P2P video streaming systems which use a sliding window to download pieces needed for immediate playback, while still acquiring non urgent pieces for diversity [Vlavianos *et al.*, 2006; Zhou *et al.*, 2007].

Conceptually similar work to ours is that of [Zhang *et al.*, 2008]. This paper proposes a play-on-demand solution for desktop applications. The basic idea is to store user's data at a USB device, and at run time, download desktop applications using P2P (specifically, via BitTorrent). These

applications are then run in a lightweight virtualization environment. The main focus here is on how to run an application without installation. The downloaded images here are not standalone VMs, making this approach of more limited applicability. These images are also significantly smaller than those of VMs, consequently they do not explore execution during on-demand download, or VM profiling.

Similarly [O'Donnell, 2008] proposes the idea of using BitTorrent as a fast method for distributing VMs to student machines in a training environment. It provides a proof-of-concept, but also focuses on a niche application space and leverages neither on-demand download nor VM profiling. In the popular media [Roth, 2008] advocates the idea of using BitTorrent to perform large scale deployments of desktop images over the WAN. Also related is Twitter's Murder system [Twi, 2010] which dramatically cuts down the distribution time for software binaries by optimizing BitTorrent for the datacenter. Their techniques are complementary to our own and could be used to improve the performance of our BitTorrent backend.

Also within this space are the Collective [Chandra *et al.*, 2005], a server-based system delivery of managed desktops to personal computer (PC) users. The Collective stores a portion of the managed desktop in the local cache. While it does not profile the particular blocks needed to support different tasks, the Collective will pre-fetch and fill its cache with the most popular applications. Less popular applications are streamed on-demand. This work evolved into the commercial solution MokaFive [MokaFive,]. VM fork [Lagar-Cavilla *et al.*, 2009] provides a data-center oriented method for instantaneously cloning a VM into multiple replicas running on different hosts Internet Suspend/Resume [Kozuch and Satyanarayanan, 2002] was early work that allowed machines to be suspended on one hardware platform, transferred over the network and resumed on another using virtualization.

Finally, techniques such as IP multicast [Albanna *et al.*, 2001; Hinden and Deering, 2006] can allow the same data to be streamed simultaneously to multiple recipients with very low network overhead. However, we believe IP multicast is ill-suited to solving the problems faced in our domain for several reasons. (1) Virtual appliance execution paths will differ from instance to instance depending on both non-determinism of OS operation and more importantly differing usage patterns amongst users. As a result, different peers will need only partially overlapping sets of pieces, and on different schedules. (2) Multicast is poorly suited to delivering data to peer sets that

are geographically diverse (e.g., home users, nano data centers, multiple data centers or corporate locations). (3) Even if virtual appliances were in a single LAN and executed almost identically but still started at staggered times (e.g., in response to increasing demand), each packet would likely need to be multicast many times for each newly started machine that has not received that packet yet, significantly reducing efficiency and the effective throughput of the multicast server. (4) Even in a single network, multicast has significant setup overhead and is not used by many organizations as a result [El-Sayed *et al.*, 2003; Hosseini *et al.*, 2007]

2.3 System Design

Ideally, a VM executing on vmTorrent should have execution comparable to that of execution off the local disk. vmTorrent attempts to achieve this goal by adjusting peers' piece selection strategies, *predicting* piece accesses and pre-fetching to reduce execution delay, while still maintaining sufficient piece *diversity* to support efficient swarming. In this, vmTorrent resembles P2P on-demand video streaming [Huang *et al.*, 2008] which selects pieces from the playback window with higher priority, while still acquiring enough random pieces to ensure sufficient diversity.

However, unlike video streaming, playback of VMs is both less structured and less predictable. Non-sequential piece access is extremely frequent and user actions - each one potentially accessing different sections of the VM image - add to the inherent stochasticity of VMM/VM operation. Moreover, while most video streams will eventually use all pieces in the stream, we have found that across the workloads we have studied, only a fraction of the VM image is accessed (see Table 2.2); most programs, drivers, and files are never used in the average execution. Consequently, vmTorrent consults a *profile* of likely block access patterns associated with each guest VM to determine a preferable streaming order.

Given that individual peers can accurately predict which pieces will be needed, and in what order, to continue execution, these peers then must determine which pieces should be requested from the swarm. Naively requesting pieces earliest-first, even according to the highest probability profiled execution path, can backfire as swarms grow larger. vmTorrent aims to have peers arrange their requests to both ensure sufficient piece diversity for the swarm to function well and also avoid "over downloading" (bringing unneeded pieces, or needed pieces too quickly) which may

impair host performance by unnecessarily loading the host and diverting bandwidth from other peers.

Recall that our system design is motivated by observing that (1) only a small fraction of a VM image needs be accessed in order for it to run to completion, (2) peers have plenty of unused upload bandwidth, particularly in the datacenter, and (3) the portions of the image accesses in order to execute a given workload are strongly and positively correlated with the actions carried out in that workload.

Consequently, we design vmTorrent to take advantage of these properties. Firstly, due to observation (1), vmTorrent does not need to wait until the whole machine image is downloaded. When spawning a new VM, vmTorrent can make the VM image immediately available to the VMM, which can immediately begin booting the VM. However to do this without incurring unreasonable *transport delay* (i.e.,the time image accesses are stalled waiting for data to be downloaded from the network), vmTorrent must prefetch the portions of the image that will be needed. Observation (3) provides the basis for successfully predicting which portions of the VM image those will be. Finally, observation (2) provides us with the opportunity to do this at scale. By leveraging and modifying P2P mechanisms, we can turn each downloading peer into an uploading server that has many of the image sections other peers are likely to need.

In our approach, vmTorrent predicts which pieces will be needed by consulting a *profile* associated with the guest VM being executed. This profile summarizes the execution behavior(s) and corresponding disk image access patterns of that VM. These profile-based predictions of which file system *blocks* will be needed enable vmTorrent to *proactively* fetches the BitTorrent *pieces* in which these blocks are contained. vmTorrent also changes the download schedule *reactively* in response to real-time demand by the VMM for blocks that may not be in the profile or have been accessed earlier than expected.

vmTorrent instance may operate on hosts that are part of, or attached to a cloud. For instance, hosts can be users' computers connected to a cloud, or data center nodes inside a cloud. Each host comprises a VMM that can run guest VMs, as well as an instance of vmTorrent that provides file system services for the VMM. vmTorrent operates as a file server that stores and provides access to a locally modifiable copy of the VM disk image while concurrently downloading that image. Each vmTorrent instance keeps a pristine copy of the disk image blocks so that it

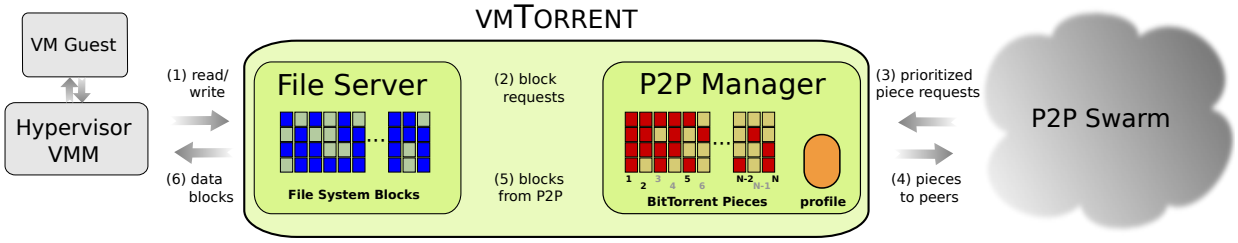


Figure 2.1: vmTorrent architecture.

can serve these copies to peer vmTorrent instances. The collection of vmTorrent instances operating on all hosts running a given VM forms a P2P swarm.

To support this mode of operation, vmTorrent is composed of two components: a *custom file server (FS)* and a *custom P2P manager (P2PM)*. Figure 2.1 illustrates the operation of the system. As the VM starts to execute, (1) the VMM tries to access the disk image in response to the guest's virtual disk accesses. If a block is not yet present, then (2) FS requests that block with high priority from P2PM. Meanwhile, (3) P2PM continuously downloads pieces of the disk image from the network and (4) uploads pieces already on disk to other swarm members in response to requests generated by their instances of vmTorrent. As pieces arrive, vmTorrent's P2PM stores each piece in a pristine copy of the VM image and (5) passes the file blocks comprising this piece to the file server. VMM access requests to these file blocks can then be served (6) by FS.

Sections 2.4 and 2.5 focus on profile creation and vmTorrent data exchange, respectively. The remainder of this section lays the groundwork for this, delving more deeply into vmTorrent's FS and P2PM components and their interaction with one another.

2.3.1 File Server

The vmTorrent FS is responsible for providing read and write access to the guest VM's disk image. FS operates similarly to traditional remote file systems: the root directory of file server is mounted in a designated location, making its files visible to the system. All disk accesses performed to the sub-tree under this mount point are intercepted by the operating system and delegated to the server. In this way FS enables a standard file system view that is indistinguishable from that of other common remote file systems such as NFS and CIFS - allowing vmTorrent to operate in a manner that is fully transparent to the VMM. Like other remote file systems, the accesses to FS

may sometimes incur significant delays. In Section 2.4 we discuss when such delays may impact the correctness of VM execution and how such special cases may be handled.

When a new guest VM is launched, FS creates a placeholder file for that VM's disk image, which from the VMM's view of the file system is indistinguishable from a complete local copy of the image. However, initially this file is empty; its content is gradually streamed in from the P2P network. As soon as the local placeholder is present FS can begin to serve requests to access the data - even if no data is yet present. For blocks that have already been downloaded locally, the server will respond to VMM requests immediately. However, if the VMM attempts to access blocks that are not yet present, FS will issue *demand requests* for these blocks to P2PM which will stall until the needed blocks have been received from the swarm. There may be multiple concurrent image accesses stalled at the same time, as VMM image accesses correspond to activity in the guest VM which itself may be running multiple processes.

File accesses arrive at the server as tuples of the form ≪`file,offset,length`≫, indicating that the VMM is attempting file access to a contiguous range of data of a particular size starting at a given offset. As accesses can be for ranges of arbitrary length, the server divides these accesses into a set of accesses to fixed-size chunks referred to as file-system blocks.

vmTorrent uses one bitmap for each file to track which blocks are exist locally. Initially this bitmap is empty corresponding to the empty placeholder. FS consults a file's bitmap to determine whether requested blocks are present, and updates the bitmap as additional pieces finish downloading and their blocks become locally available.

FS distinguishes between read and write accesses that fully align to a set of block (*aligned* accesses). Those accesses that only require part of the first or last disk block(s) we refer to as *unaligned* accesses. While read accesses always require a check to the bitmap to ensure the blocks being accessed are present (and download should some be missing), aligned write accesses can be served without checking the bitmap - since such write requests would have overwritten the original content of that block anyway. Unaligned write accesses, however, cannot safely overwrite the partially written blocks since it is possible that these blocks have not yet been downloaded. Thus the non-written portion of such blocks might need to be filled in later, which would require an order of magnitude more book-keeping complexity to support. Consequently, in such cases, vmTorrent first performs a standard bitmap check (and demand-request if needed) on any partially written

blocks to ensure they are in place, and if not, these unaligned write accesses stall until the blocks have been downloaded from the swarm.

2.3.2 P2P Manager

The vmTorrent P2PM is responsible for downloading the VM disk image asynchronously from the P2P swarm and uploading requested pieces to swarm peers. In order to upload the downloaded disk image to other peers, vmTorrent must maintain a pristine copy of the data downloaded from the swarm. Those blocks modified by FS writes cannot by uploaded to other swarm, since doing so would violate the consistency of the VM image being shared. To solve this problem vmTorrent provides a P2P-backed file system that stores two copies of the data: one copy for read-only access used by P2PM for uploading data to peers, and one copy used by FS to provides read-write access to the VMM. [3]

The primary goal of the P2P manager is to fetch the disk image data so that the data is available for use when needed by the VMM. vmTorrent employs two mechanisms that exercise intelligent piece selection strategies to minimize guest VM execution stalls due to missing disk blocks. Firstly, the P2P manager fetches blocks proactively based on when it estimates they will be needed. Secondly, if the VMM attempts access to a block that is not present, the file system will issues a *demand* request to the P2P manager, which in turn will immediately request the piece containing that block from the swarm.

vmTorrent implements proactive piece fetching through use of a *profile* associated with the VM that provides the P2P manager with the expected block access order and normalized inter-access intervals for common execution paths. By tracking the block access history and current progress, the P2P manager can use this profile to predict which blocks are most likely to be needed in the future, and estimate how much delay in receiving these blocks may be tolerated. When all information in the profile has been utilized, the download of the remaining pieces proceeds in the background. These non-profiled pieces are downloaded rarest-first, as in traditional P2P systems, albeit at a much lower rate and with lower priority so as not to divert resources from other swarm members downloading pieces critical to their execution path.

[3] A space optimizing version of vmTorrent could be constructed by doing *copy-on-write*, in which modified blocks would be stored by FS, while all other blocks would be read directly from the pristine image stored by P2PM.

Task	OS Type	Description
Boot-Shutdown	All	Boot, login, and shutdown.
Latex	Linux	Compile 30-page Latex document, view result in PDF viewer.
DocEdit	Linux	Create new OpenOffice document, save, reopen, edit, spell-check.
PowerPoint	Windows	View PowerPoint slide-show.
Multimedia	Windows	Play 30 second music file.

Table 2.1: Tasks.

When a demand request for data arrives from the file server, the P2P manager will attempt to fetch the corresponding piece(s) from the swarm. Moreover, the P2P manager will prioritize this request over other existing requests so as to satisfy this critical request as quickly as possible. As pieces arrive, the P2P manager pushes requested data back to the file server which notifies any threads that may be waiting for the data.

2.4 Profiling

To provide quick and scalable VM execution, vmTorrent's P2P manager intelligently prefetches pieces so as to minimize file system access delays experienced by the VMM. To do so, the P2P manager must have a way of predicting which pieces will be needed and when. vmTorrent uses *profiles* summarizing VM image access behavior to provide the P2P manager with this information.

To build these profiles, we analyze the VM disk image access patterns for a variety of VMs and workloads. For each VM/workload combination, we examine access order, access time, and frequency of use. We aim to identify those sections of the VM image that are common across the runs, and encode them in a *profile* that can be utilized by the P2P manager's prediction logic. In this work, we examine straightforward offline mechanisms for doing so (although we find they perform quite well in many scenarios). Future work will examine more sophisticated prediction utilizing recent execution history and explore online techniques for profile improvement. To better understand our profiling approach, we first discuss the set of tasks and VMs we examine.

VM	Size	% Accessed
Fedora	4.2 GB	7-9%
Ubuntu	3.9 GB	6-10%
Win7	4.3 GB	7-8%

Table 2.2: Virtual machines.

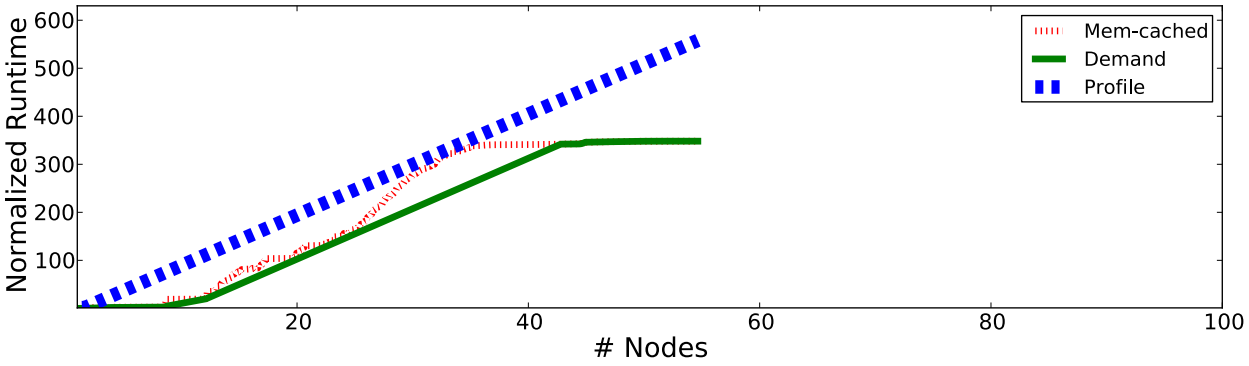
2.4.1 VMs and Workloads

We used a set of workload scenarios designed to simulate typical short VDI user tasks. Table 2.1 lists the usage scenarios for our experiments. These scenarios represent different user activities on desktop virtual appliances. Each benchmark consists of first booting the guest VM, then executing a script that performs a desired task by mimicking user actions and finally shutdown the guest VM. To automate the execution of these benchmarks, we configured guest VMs to auto-login once they boot, and then execute a script that selects the appropriate task to run.

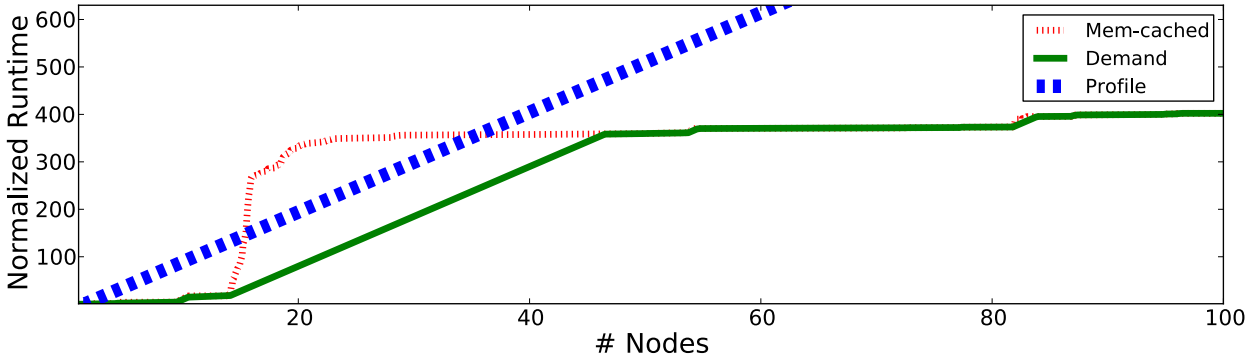
While the set of tasks we could explore in this work were necessarily limited, we believe the results produced are relevant to both longer VDI tasks, as well as those run on virtual servers and virtual appliances. For each of these, as we will shortly see, the most challenging portion of execution is the boot and login sequence - in both the latency and volume of image access. Consequently, vmTorrent's efficiency on long VDI tasks can be expected to mirror those of shorter VDI tasks. Moreover, many of the tasks we examine (*e.g.,* Latex compile, playing a music file) are essentially straightforward non-interactive workloads with similar characteristics to those virtual servers or virtual appliances would be expected to execute.

Figure 2.2 shows disc image access rates for VM execution from RAM disk on a modern host machine (Section 2.6.2) when running the *Boot-Shutdown* task on each of the guest VMs listed in Table 2.2. This execution performance represents a *best-case* scenario in which there is no network or disk I/O overhead. Each figure plots cumulative disk accesses in MB (y-axis) against time (x-axis). We see that the disk access patterns vary significantly by virtual appliance type and exhibit bursty behavior. Since the peak (local) access rate can easily exceed the throughput of a standard network, execution delays due to section of the VM image being unavailable are highly likely. These *network* delays arise inherently from the mismatch between network and local memory throughput.

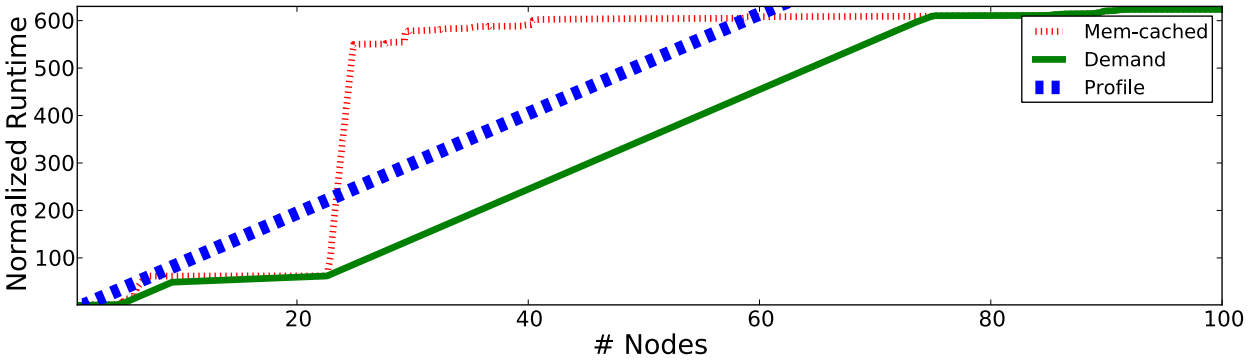
We illustrate these delays by computing the idealized cumulative delay for profile and demand



(a) Fedora.



(b) Ubuntu.



(c) Win7.

Figure 2.2: Best-case access rates and unavoidable network delay.

polices based on the collected image access traces and assuming a 100 $Mbps$ network operating at standard efficiencies (15% protocol/IP/loss overhead). We see that for both the Ubuntu and Windows 7 VMs, the access rate experiences a sharp spike at 13 s and 22 s, respectively - resulting in like delays for both idealized demand and profile network I/O. These spikes indicate a high level of parallelism in the boot sequence. However, the Fedora VM has a much more gradual boot behavior - likely indicative of a significantly more sequential boot sequence.

Finally, it should be noted that the delays shown only take into account the effect of throughput restrictions. Delays due to network latency, serving peer response, and similiar factors, will only increase those predicted in Figure 2.2. The extent to which these factors increase delay will depend on both the degree of parallelism in VM execution (greater parallelism will tend to mask latency-related delays in execution as resources shift to some non-I/O blocking thread), as well as the efficacy of prefetching (demand fetch requests will clearly be effected by latency-related sources of delay, while the latency of prefetch requests is more likely to be masked).

These computations provide insight into the baseline completion time we might expect from vmTorrent. For all of the virtual appliances we explore, with the exception of Fedora, we see that equaling performance of RAM disk with our approach is essentially unachievable in our experimental setup. There simply isn't time to get all the image content downloaded in time to continue uninterrupted execution. Here, the largest distance in time a given curve lags behind the best-case curve, serves as a lower bound on the delay an execution of that type is likely to experience, (precise numbers for the scenarios we consider are shown in in Section 2.6.5, Table 2.4). However, with the Fedora virtual appliance vmTorrent can (in theory) to always prefetch content before it is needed. However, as discussed above, latency-related delays will affect the Fedora VM's performance more greatly than that of the others. Consequently, while the Fedora VM's performance using our straightforward profiling and prediction will not match that of a memory cached system, it may still potentially outperform execution from local disk. In fact, we will see that this is precisely what happened in our experiments (Section 2.6.3).

2.4.2 Profile Creation

We use two profiles for our experiments per VM/task combination. The first is built using just one sample run. It simply lists the order in which blocks are accessed for the first time in the

sample run. The second is built using 1000 runs. Each block is ranked based on its access times and notated with the fraction of runs in which it was requested.

Rank values provide an easy to utilize encoding of "average" download behavior. The frequency values allow our prediction algorithm to easily adjust the aggressiveness of its prefetching strategy by filtering out those entries below a threshold frequency. We shall see how vmTorrent's piece selection logic incorporates these in Section 2.5.

To rank blocks, we determine their average access time across multiple runs. Then the blocks are ordered earliest-first with ties being broken randomly. We tried other statistics such as median, and min, but these other measures offered no noticeable improvement. When using runs from different machine types to create the profile, these runs would first need to be normalized. However, as we deploy on a single machine type in our experiments, we leave this challenge for future work.

An important caveat is that, like other remote file-systems, vmTorrent may introduce significantly larger than normal file system access delays as perceived by the VMM. Consequently problems may occur in which VM images fail to boot because a variety of bootup check timeouts may occur if VM execution is delayed during that check. We refer to a set of blocks accessed during such a check as *critical*. To deal with such checks, an additional level of analysis must be added to the profiler that identifies such sequences and marks them accordingly. Then vmTorrent's file server needs only to block on the access attempt immediately before the critical section, waiting until all pieces in that section are downloaded. Practically, we were able to avoid this exigency by setting the boot wait parameters of our VMs much larger than they might ever be stalled.

The only critical section for which we would have needed to implement this facility appears to be an undocumented timeout in Windows 7 auto-login. A slowdown of much above 4x memcached execution time will trigger this timeout, causing auto-login to fail. As this issue only affected techniques against which we compared vmTorrent's performance (see Table 2.6.2 in Section 2.6.2), we leave the development of automated techniques for identifying critical sections to future work.

We note that there is significantly more structure available to exploit. This can be seen even when using a relatively crude measure of clustering that defines a cluster as the set of all blocks that are completely positively correlated within one another (i.e., one is accessed if and only if all others are accessed). As Figure 2.3 - plotting the probability mass function (y-axis) of cluster

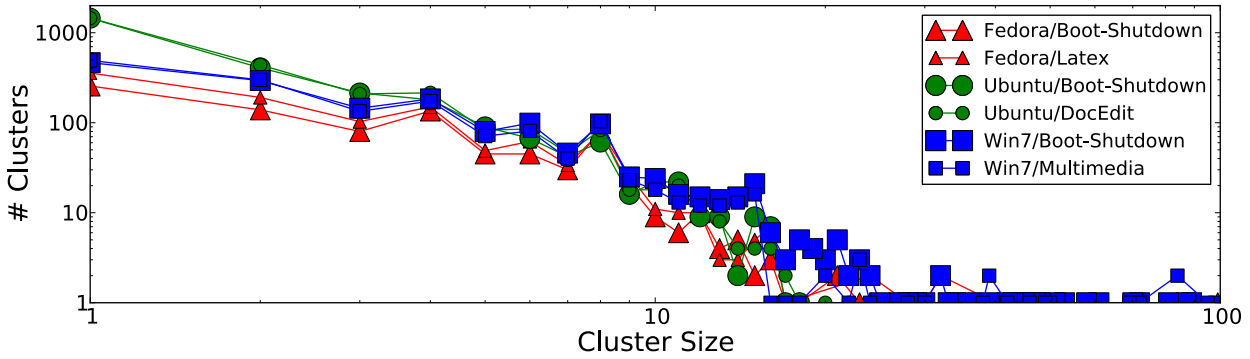


Figure 2.3: Profile access structure.

size (x-axis) for a variety of VM/workload combinations - shows there are a significant number of non-trivial clusters (y-axis). As brief inspection indicates, these clustering patterns are quite consistent across different VMs and workloads. Clearly a naive improvement to our policy would be to conditionally fetch all pieces in such a cluster as soon as the first had been accessed. We expect more sophisticated statistical clustering method may provide significant additional opportunity for prefetching improvement, which we leave to future work.

2.4.3 Mapping Blocks to Pieces

Thus far, this section's discussion has focused on FS blocks. However ultimately the profile will be used to predict which *pieces* will be requested by the P2PM. These data units are not necessarily of the same size.

For the P2PM, we leverage previous work [Marciniak *et al.*, 2008] which showed that piece size is a critical parameter for the overall performance, as it determines the degree of both the parallelism and overhead peers will encounter. In particular, small pieces are suitable for small-sized content, but may severely affect system performance for larger content. Smaller pieces produce proportionately larger meta-info (torrent) files and incur considerably higher communication overhead. In fact most BitTorrent codebases restrict the maximum number of pieces allowed. To meet these restrictions we need pieces of at least $32\,KB$ for typically-sized VMs and have chosen to use $256\,KB$ pieces.

For the FS, using a smaller granularity block for file accesses will allow vmTorrent to avoid

many unaligned writes. Since aligned writes do not require vmTorrent to fetch anything from the network, this clearly provides a performance gain. Consequently vmTorrent, block sizes should be chose so several fit in one piece. Our prototype uses blocks of $16\,KB$. To ensure correct operation, vmTorrent uses a mapping function to select which blocks should be packed into the same piece. We use a simple packing that maps contiguous ranges of blocks into pieces using a straightforward modulo operator on the block index. This mapping is useful for access patterns with high spatial locality, or when the access pattern is largely sequential, since it packs together adjacent blocks. As part of future work, we intend to examine more sophisticated mapping schemes. For example, a scheme that makes better use of the network bandwidth by avoiding downloading blocks earlier than needed when the access pattern exhibits weak locality at the piece-level. We are also exploring the use of data compression at the block-level.

2.5 Piece Selection

The goal of the vmTorrent piece selection policy is to balance the need to (1) ensure pieces needed to support execution *arrive with the least possible delay*, (2) *maximize swarm efficiency*, and (3) *avoid overuse of resources* and bandwidth that may hamper the performance of local processes (particularly the VMM) or other peers.

To achieve this goal vmTorrent's piece selection policy contains components for

1. *piece prediction*
2. *piece diversity*
3. *resource management*

vmTorrent implements its piece selection policy using three piece request types: (a) *demand* requests, (b) *profiled* requests, and (c) *default* requests. vmTorrent issues demand requests in response to missing blocks that are accessed through the file system. These requests are the highest priority and must be delivered immediately. Profiled requests are requests for blocks included in the profile and selected for prefetch by the prediction logic. These requests are high priority and may have varying deadlines. Finally, all other requests are default requests that have very low priority

and no latency requirements. P2PM will only issue default requests when the set of profiled blocks has been completely exhausted and no demand requests are enqueued.

2.5.1 Piece Prediction

Piece prediction errors come in three types:

1. *false positive* (pieces fetched but never used), resulting in wasted bandwidth and increased local overhead.

2. *false negatives* (pieces needed but never predicted) will result in pieces not being present when needed, causing increased latency as demand requests are issued and putting increased pressure on peers' queues as these demand requests are given priority over other pending requests.

3. *mistimed* requests either won't be prefetched in time for use, acting much as a false-negative, or will be prefetched too early and cause the same increased overheads as false-positives, but without the bandwidth waste.

Our piece prediction mechanism is straightforward. We select pieces in order of their profile ranks, filtering out those whose frequency of appearance lies below some cutoff threshold. We have found 0.25 to provide a good trade-off between aggressive prefetching that will result in an overabundance of false positives, and overly conservative prefetching that will suffer from over-many false negatives.

2.5.2 Piece Diversity

When VM instances begin execution in a highly staggered fashion, selecting pieces based on profile-based prediction alone may provide sufficient piece diversity to be exploited by the swarm - particularly if the VMs are running somewhat different tasks. If however, VM instances start as a flash crowd - a scenario for which our technique is designed, such policies will likely provide too little piece diversity, leaving swarm members without almost any pieces to exchange. This holds especially true during the boot sequence. As we will see in Section 2.6.6, this is one of the main causes of the significant performance gap between naive P2P provisioning and vmTorrent demonstrated in Section 2.6.3.

To provide additional diversity, we utilize a *window-randomized* selection policy that picks one of the first k pieces at the head of the prediction queue to request[4]. Here, the window size k is a tunable parameter that attempts to balance the urgency of pieces against the need for sufficient peer diversity. If a large k is chosen, the gain from accurately predicting piece accesses degrades as pieces fail to be prefetched in the order of predicted use. On the other hand, the choice of an overly small k will prevent the swarm from providing scalability since too little piece diversity will be achieved. Consequently, as the number of peers decreases, the time staggering of peers increases, or the usage pattern diversity increases the optimal k will tend to be lower (since the pressure on centralized servers is lower and the relative piece diversity in the network greater) and vice-versa.

2.5.3 Resource Management

On highly utilized end-hosts, allowing download bandwidth saturation by the P2PM may in fact interfere with VM execution by hogging resources needed by the VMM. This effect is due to potential competition over scare resources such as memory, bus, etc. Particularly, given that the majority of a VM is not used in most runs, downloading pieces comprised of such blocks at the same rate as higher priority pieces containing profiled-or-demanded blocks "litters" buffer cache. vmTorrent addresses this challenge by rate limiting the issuance of new piece requests when possible. Default priority requests are always highly throttled, as it is rather unlikely they will be of any use. Additionally in network regimes that provide more capacity than vmTorrent needs to prefetch every piece in time, peers should additionally throttle their download rates while prefetching in order to allow other peers, who may have a more urgent need, to utilize the swarm's upload capacity. Moreover, doing so reduces the load on the local system, thereby potentially avoiding local resource contention between the P2P manager and the VMM. To do this vmTorrent leverages *deadline-aware throttling*. If a sufficient buffer has been prefetched, further prefetch operations will be throttled significantly until additional pieces will soon be needed. We do not go into further detail here as our testbed environment did not provide sufficiently high data rates for this technique to be applicable.

[4]In order to avoid radical modification to the libraries used to build our prototype, our prototype broke this single k up into two different windows - one in the custom P2P manager code, the other in the BitTorrent library.

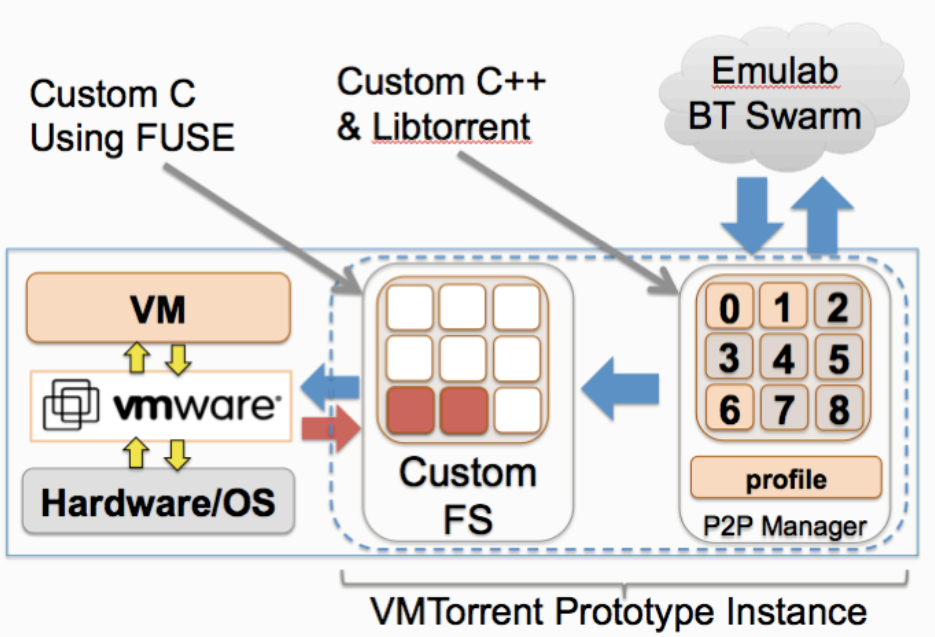


Figure 2.4: vmTorrent prototype.

2.6 Experimental Evaluation

To evaluate our design, we implement a prototype version of vmTorrent for Linux hosts and conduct extensive experimental evaluation. The foremost aim of our experiments is to determine vmTorrent's efficacy in providing quick and scalable virtual appliance distribution and execution. Our primary assessment metric is *completion time*: the time that it takes to execute a virtual appliance/workload combination. In addition, we characterize VM execution delays due to late arrival of image data. Our investigation focuses on two sources of delay: delay introduced by prefetching mistakes made by our profile based prediction; and delay introduced by P2P delivery inefficiency.

2.6.1 Experimental Setup

Our prototype, shown in Figure 2.4, comprises a user-space file server tightly integrated with a P2P client. This implementation is capable of transparent operation with any VMM/guest VM combination, and without requiring any changes to the VMM, guest VMs, or the underlying system.

Our file server builds on *bindfs* [bindfs,], a FUSE [Szeredi,]-based file system for mounting

a directory onto another location. The server provides a virtual file system that stores a locally modifiable copy of the guest VM image. While a user-level file server potentially introduces performance penalties over a kernel-based implementation, we show this overhead is sufficiently low for our purposes (Section 2.6.4).

Our prototype's P2P component is built on top of the *libtorrent* 0.15.5 [libtorrent,] library. Like most BitTorrent clients, libtorrent optimizes its piece selection policy to maximize file download throughput. Since our goal is to minimize the delay for downloading particular pieces needed for the VM execution, we modify libtorrent's default piece section policy (e.g., rarest-first selection) to support the low-latency prioritized piece download needed by the piece selection policies described in Section 2.5. We also implement a high-frequency rate-throttling mechanism needed to support both resource usage management and deadline-based policies.

We deploy vmTorrent on an Emulab network testbed [White *et al.*, 2002]. Each experiment consists of multiple hosts and an initial server which stays in the system for the duration of the experiment. The hosts are selected from a pool of 160 "d710" machines at the University of Utah's Emulab installation. Each host was equipped with a 64-bit $2.4\,GHz$ Quad-Core Intel Xeon X5530 CPU, $12\,GB$ RAM, and a Barracuda ES.2 SATA 3.0-$Gbps$/7200-RPM/250-GB local disk. The hosts run 64-bit Ubuntu 10.04.1 LTS with a modified 2.6.32-24 Linux kernel provided by Emulab. The hosts use VMware Workstation 7.1.0 build-261024 as the VMM.

Hosts and server are connected to a $100\,Mbps$ private experiment LAN. While we would have preferred to test vmTorrent on a 1 or $10\,Gbps$ network, $100\,Mbps$ was the upper limit offered by Utah's Emulab. Hence, peer upload and download capacity was limited $100\,Mbps$. All experiment control and logging are conducted on a separate control network interface. While, unless otherwise specified, peer metadata exchange is conducted on a full-mesh topology. Each host wis limited to actively upload to at most five other hosts.

2.6.2 Methodology

We evaluate the performance of vmTorrent across the set of VM/workload combinations introduced in Section 2.4. To study the system's scalability, we vary the number of concurrent physical machines hosting vmTorrent instances (i.e., the swarm size) from 2 to 100. For each set of parameters, we run 2-10 trials of each experiment (fewer trials were taken for larger swarm sizes as it

Method	Summary
demand	FS front-end + ram-disk $sshfs$ mount
p2p+demand	vmTorrent without prefetching
p2p+demand+profile	vmTorrent
local disk	FS front-end + local disk
memory-cached	FS front-end + ram-disk

Table 2.3: Deployment methods used for comparison.

was increasingly difficult to obtain a sufficient number of physical machines from the shared pool), presenting averaged results.

We compare the performance of four different scenarios summarized in Table :

1. *demand* refers to the scenario where pieces are fetched directly from a server via sshfs [Hoskins, 2006], run through the FS front-end mounted on a RAM disk (so all accesses after the first read are done locally).

2. *p2p+demand* refers to vmTorrent with profiling turned off, where pieces are fetched from the P2P swarm as they are demanded by the FS.

3. *p2p+demand+profile* refers to a full vmTorrent execution where piece selection is driven by the profiled prefetching sequence of pieces.

4. *local disk* where the VM image is present locally at the start of the execution. While comparing *p2p+demand+profile* against *local disk* is helpful in providing insight as to whether and how well a given cloud setup can support virtualized execution, our main comparison is against *demand* and *p2p+demand*, which as discussed on Section 2.2, are the current state-of-the-art solutions in this domain.

We normalize our results with respect to a *memory-cached* execution using a RAM disk. Thus *memory-cached* measurements provide a theoretical best-case scenario as neither network nor disk I/O is incurred.

2.6.2.1 Demand Performance Model

We also provide a back-of-the-envelope estimate of demand performance. Doing so allows us to both sanity-check the performance of our prototype and to provide a performance comparison when we are not able to empirically obtain demand numbers. Particularly, as discussed in Section 2.4.1, the Win7 VM fails to auto-login if slowdown becomes too great, breaking our automated testing infrastructure. Consequently, Win7 graphs will be missing points for both *demand* and *p2p+demand* scenarios on which normalized execution times are much greater than 4.

To produce our model, we break the network-execution time of a VM into two components. We refer to the first component as *base throughput delay*: the number of bytes needed to complete execution divided by the available network bandwidth. As was shown previously in Figure 2.2 (Section 2.4.1), peak throughput requirements may well outstrip available bandwidth - particularly in OSes such as Windows 7 whose memory access patterns peak very sharply, causing additional delay. Moreover, the RTT to request and obtain the first byte of a demand-fetched block will delay execution even further. For now, we combine these two sources into a second component we refer to as *fetch delay*. Given empirical measurements of both single-server, single-client execution time and number of blocks accessed for a given task, we can then describe the expected demand delay of a set of n peers as:

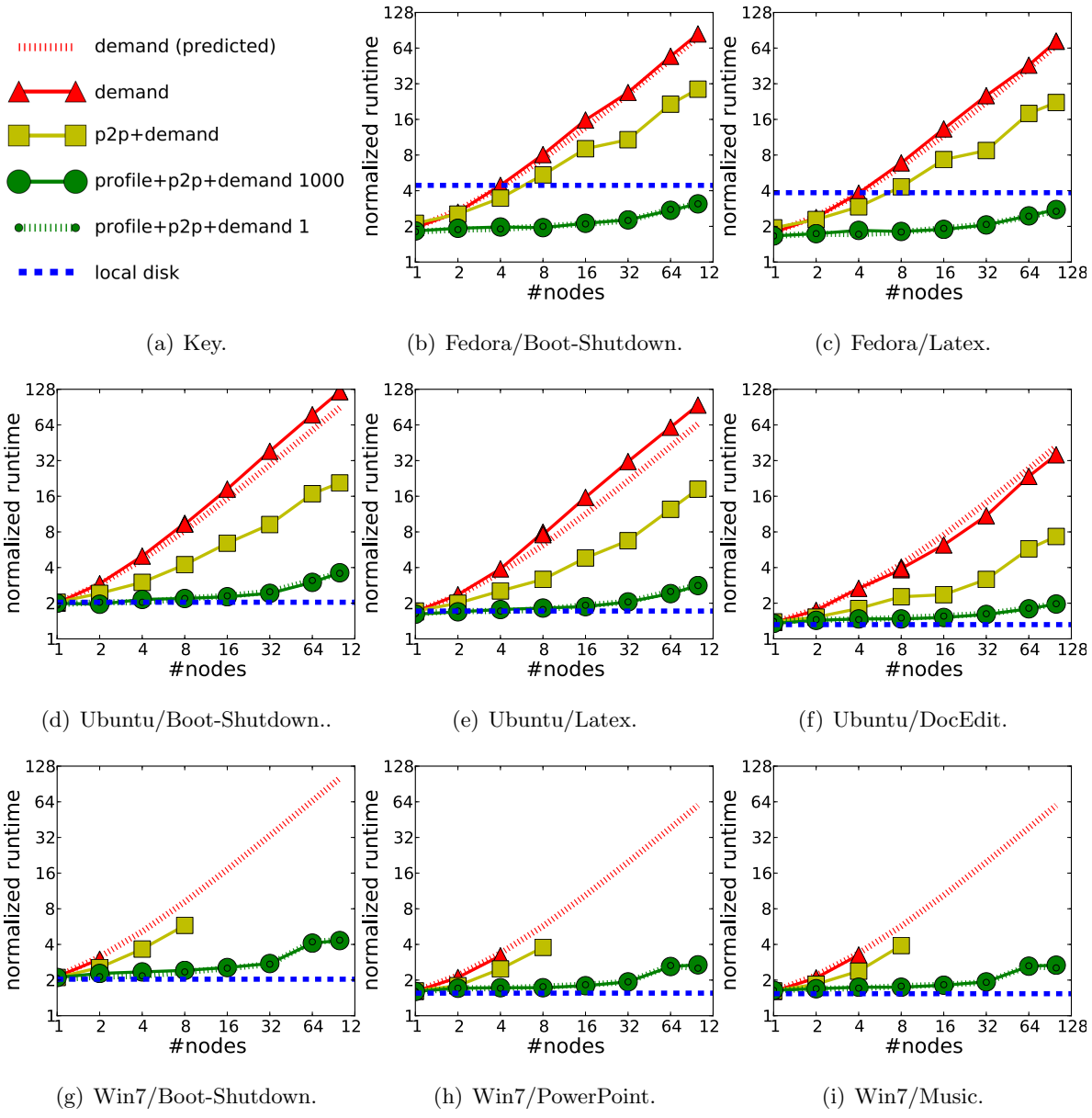
$$E[D_n] = D_{fetch} + nD_{throughput} \tag{2.1}$$

The intuition here is that as the number of nodes increases, the fetch delay will play an increasingly smaller role, becoming masked behind throughput delays.

2.6.3 Scalability

We show the results of running the four primary scenarios: *p2p+demand+profile*, *p2p+demand*, *demand* and *local disk* where a number of clients attempt to execute a VM workload. We show how closely our main VMTORRENT scenario (*p2p+demand+profile*) matches *local disk* performance, how well it scales with the number of participating clients, and how strikingly it outperforms other network-based scenarios.

We run a series of experiments consisting of a single server with a complete copy of disk image cached in memory and a set of peers running VMTORRENT instances. We consider two downloading

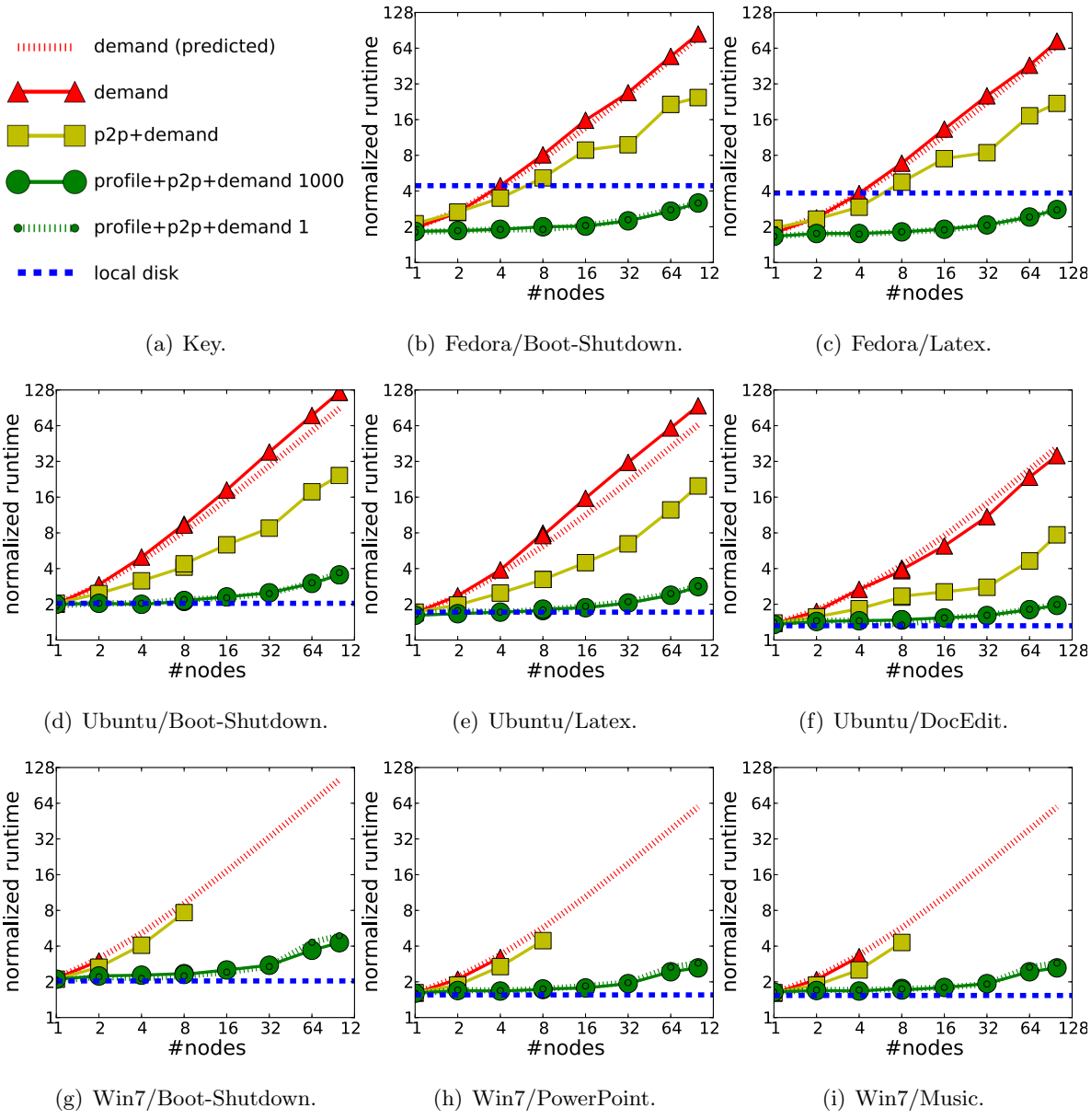


(a) Key. (b) Fedora/Boot-Shutdown. (c) Fedora/Latex.

(d) Ubuntu/Boot-Shutdown.. (e) Ubuntu/Latex. (f) Ubuntu/DocEdit.

(g) Win7/Boot-Shutdown. (h) Win7/PowerPoint. (i) Win7/Music.

Figure 2.5: Swarm-size vs. runtime: flash crowd/immediate departure.

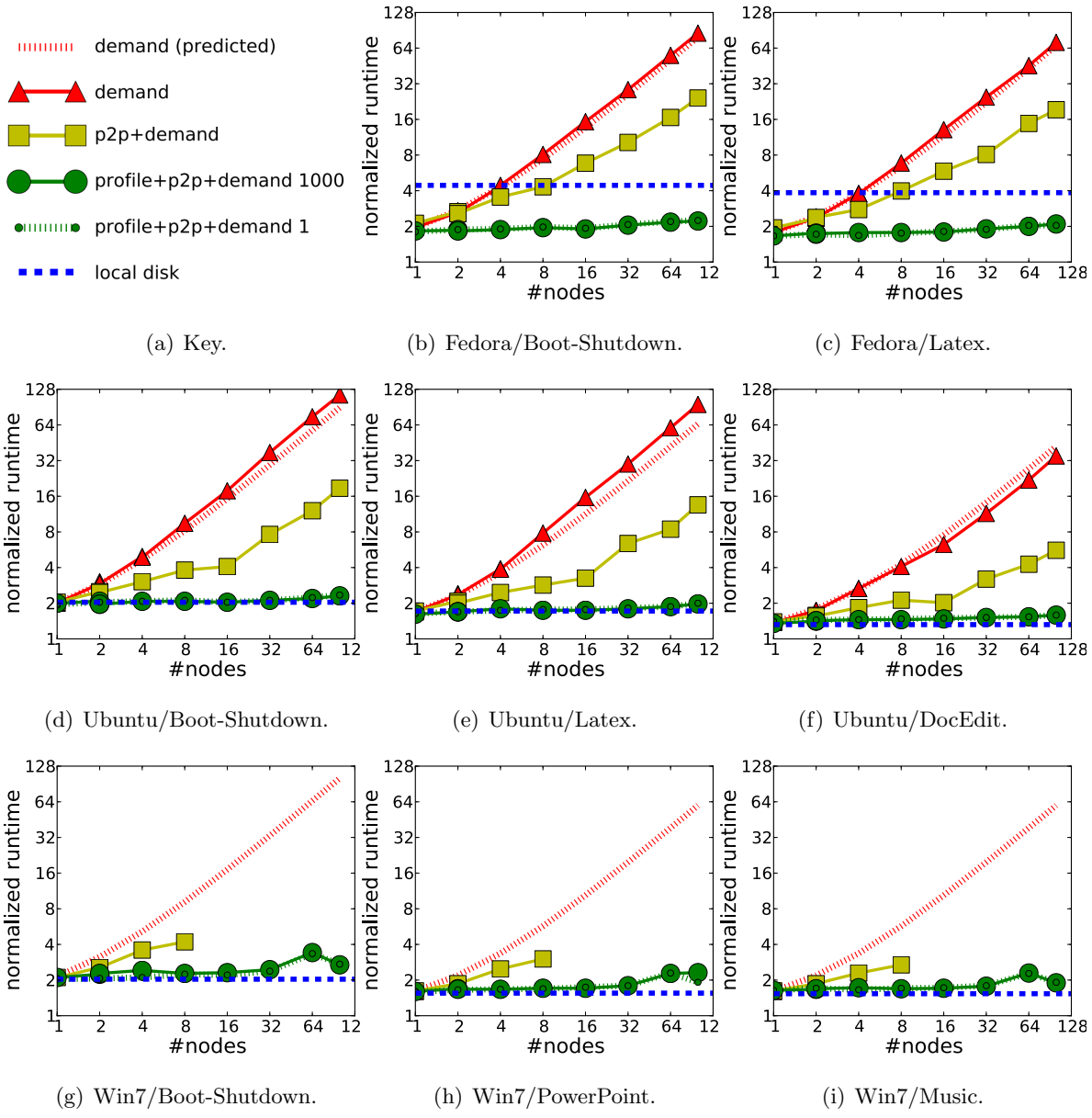


(a) Key. (b) Fedora/Boot-Shutdown. (c) Fedora/Latex.

(d) Ubuntu/Boot-Shutdown. (e) Ubuntu/Latex. (f) Ubuntu/DocEdit.

(g) Win7/Boot-Shutdown. (h) Win7/PowerPoint. (i) Win7/Music.

Figure 2.6: Swarm-size vs. runtime: flash crowd/delayed departure.

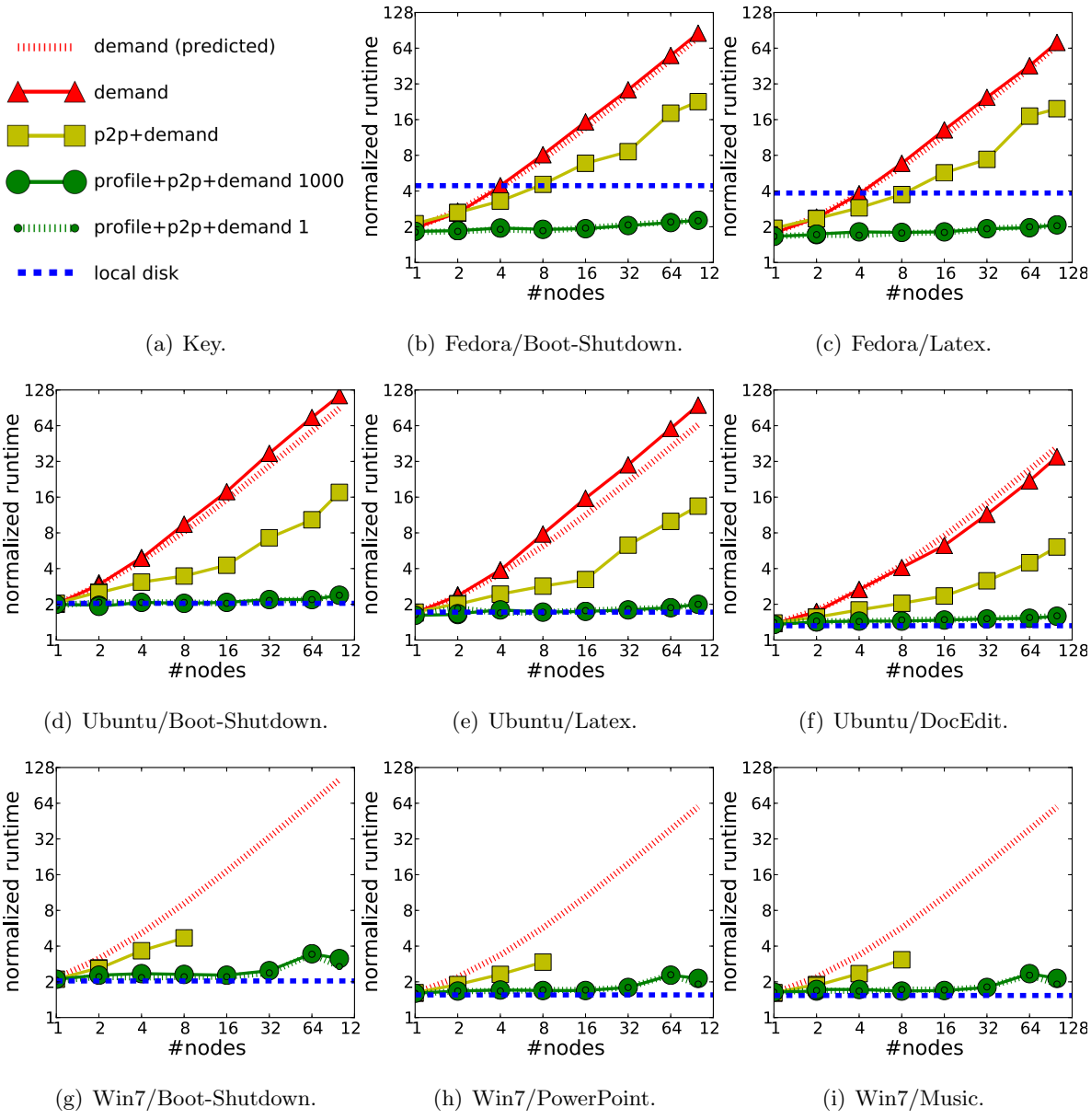


(a) Key. (b) Fedora/Boot-Shutdown. (c) Fedora/Latex.

(d) Ubuntu/Boot-Shutdown. (e) Ubuntu/Latex. (f) Ubuntu/DocEdit.

(g) Win7/Boot-Shutdown. (h) Win7/PowerPoint. (i) Win7/Music.

Figure 2.7: Swarm-size vs. runtime: staggered arrival/immediate departure.



(a) Key. (b) Fedora/Boot-Shutdown. (c) Fedora/Latex.

(d) Ubuntu/Boot-Shutdown. (e) Ubuntu/Latex. (f) Ubuntu/DocEdit.

(g) Win7/Boot-Shutdown. (h) Win7/PowerPoint. (i) Win7/Music.

Figure 2.8: Swarm-size vs. runtime: staggered arrival/delayed departure.

peer arrival patterns: *staggered*, in which a new peer joins the system one second after the previous peer had joined, and *flash crowd*, in which all peers arrive simultaneously. Likewise, we consider two departure patterns: *delayed* departure, in which nodes stick around until the experiment is over, and *immediate* departure, in which nodes leave the system as soon as they are finished.

The graphs in Figures 2.5-2.8 plot mean normalized runtime (y-axis) vs. swarm size (x-axis) for both vmTorrent (*p2p+demand+profile* - using both 1-run and 1000-run profiles) and comparison setups (*p2p+demand*, *demand*, and *local disk*). Each figure consists of eight plots, one for each VM/workload combination described in Section 2.4. Each figure details one combination of peer arrival and departure pattern.

From a scalability perspective, we would expect the most difficult of these for both p2p variants to be flash crowd arrival and immediate departure (Figure 2.5) and the easiest to be the combination of staggered arrival and delayed departure (Figure 2.8). The results shown bear this expectation out. However, as can be seen from visual inspection, vmTorrent's performance remains substantially the same across all arrival/departure patterns patterns for up to 32 nodes. For larger swarm sizes of 64 and 100 nodes, performance does differ between immediate and staggered arrival, while departure pattern appears to have minimal impact on both vmTorrent and *p2p+demand*.

While vmTorrent's scaling with staggered arrival appears essentially flat, realizing our design goals, for both 64 and 100 nodes, we observe a significant uptick in runtime (although vmTorrent still radically outperforms current state-of-the-art). In Section 2.6.6 we will show that swarming efficiency problems appear to cause this flash-crowd performance issue, and provide suggestions on how it may be ameliorated.

The lack of impact departure pattern has on runtime and scaling is less concerning, however we provide brief explanation here. What happens, at least in the scenarios we have explored thus far, is that peers arriving earlier take longer to run and all peers finish around the same time. Thus peers that immediately disappear have little negative impact as other peers in the swarm have already completed the vast majority of their download by that point. We observe a positive correlation between transport delay and arrival time which accounts for much of the disparity, but not all of it. Our hypothesis is that peers arriving earlier take a double performance hit: they benefit the least from swarm upload (as newer peers had less to share with them) while getting essentially the same share of the server (the portion of delay explained by transport delay), and they incur more

overhead uploading than other peers (the portion of delay unexplained by transport delay). In this way VMTORRENT appears to naturally provide a degree of fairness to execution of VMs in a given swarm.

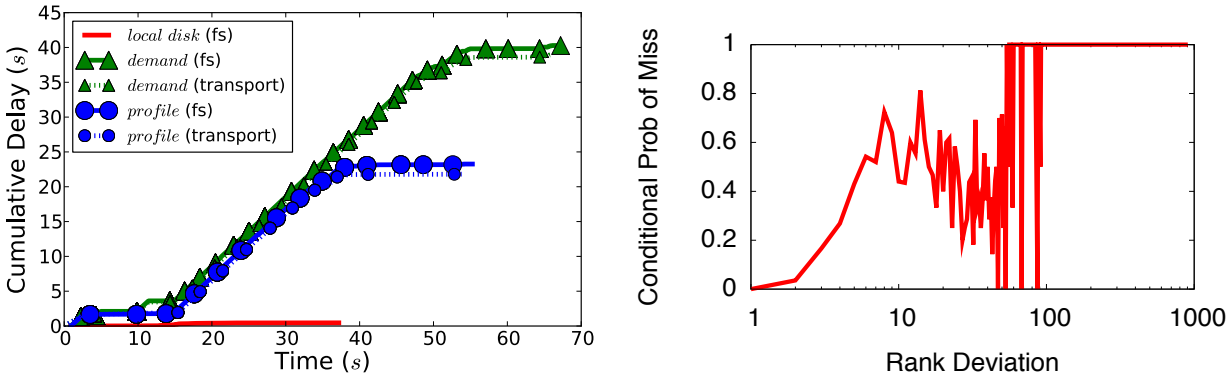
The most essential take-away from this section is that across all arrival/departure patterns and VM/workload combinations, the performance advantage of VMTORRENT over state-of-the-art competitors is best measured in orders of magnitude: **4-11X** better than *p2p+demand* and **16-30X** better than *demand*. Further, VMTORRENT's performance remains comparable to that of local disk for all swarm sizes examined and would seem likely do so for staggered arrival of significantly larger numbers of peers without the need to introduce any additonal features to our prototype[5]. All of the above observations appear to hold equally for the single-run and 1000-run profiles, indicating that, at least for the relatively straightforward profiling approach we currently take, a small amount of profiling goes a long way.

As predicted by Equation 2.1, the runtime of *demand* instances increases quickly with the number of clients as the server becomes the bottleneck. *p2p+demand* outperforms *demand* as it is able to utilize some of the peer upload bandwidth. However, *p2p+demand* also scaled poorly due both to sub-optimal bandwidth utilization and the effects of latency delays discussed in Section 2.4.1. This holds especially true when all peers arrive within a short time of one another (flash crowd), they will all request highly overlapping sets of blocks. Consequently, sufficient piece diversity is never achieved to allow for efficient use of swarm member bandwidth. Since peers have relatively little to exchange, each uses server bandwidth aggressively again causing bottlenecks.

This reasoning also explains why *p2p+demand* improves slightly under staggered arrival while *demand* performance remains essentially the same. In staggered-arrival scenarios VMTORRENT avoids this problem completely by randomizing prefetched content as discussed in Section 2.5. While for flash-crowd arrival, we will see in Section 2.6.6 that the abundance of high-priority demand requests limits the efficacy of such an approach.

Finally, we observe the small gap between VMTORRENT execution and that of local disk on both the Ubuntu and Win7 VMs across all tasks. On the Fedora VM this is even more pronounced with VMTORRENT significantly outperforming execution from local disk, confirming our guess from Section 2.4.1 that Fedora workloads on VMTORRENT might outperform those run from local disk.

[5]Section 2.6.6 discusses potential changes that would enable like scalability for flash-crowd arrival.



(a) Cumulative delay vs. runtime.

(b) Probability of mistimed errors vs. rank deviation.

Figure 2.9: Baseline performance: Ubuntu/Latex.

Moreover, we can see that because of the relatively serial execution of the Fedora VM (as opposed to the highly parallelized execution of the Ubuntu and Win7 VMs), the advantage of vmTorrent's profile-based prefetching over non-prefetching *p2p+demand* is roughly twice as great as that of the workloads run on the Ubuntu and Win7 VMs.

For the same reason, we can see the OS with the greatest peak-to-average access ratio, Win7, performs worst with respect to local disk - at $100\,Mbps$, even assuming perfect prefetching and no latency delay, there is still a substantial gap between what the network can provide and what the VMM's ideal access pattern demands. However, we will see in Section 2.6.5 that these same factors cause Fedora execution delays to extend significantly above the lower bounds shown in Section 2.4.1, while the Win7 VM is able to essentially achieve that lower bound.

It is worth noting that our prototype achieves this performance despite having only approximately implemented the selection policies specified in Section 2.5. (1) Due to its queuing design, libtorrent often reorders the pieces selected by our policy within a short time of one another. (2) A peer fulfills multiple piece requests from a given neighbors *highest priority first*, however, it allocates bandwidth amongst neighbors according to libtorrent's default mechanisms which are difficult to modify.

2.6.4 Baseline Performance

We now dive deeper to examine why vmTorrent performs as it does. We begin by examining the baseline overheads of vmTorrent running on a single client and server. In the subsequent

discussion we attempt to isolate the effect of profiling. In these single peer investigations the P2P component of the architecture is inactive. Consequently we will omit the prefix *p2p* since it conveys no additional meaning in this context.

We measure both the *file server* and *transport* delays. The *file server* delay is the cumulative time VMM file access operations wait before completing. The *transport* delay measures the portion of *file server* delay that is incurred while waiting for data to be downloaded from the network.

For this discussion, we consider the baseline performance of the Ubuntu VM running the Latex workload. We found the performance of other VM/workload combinations to be substantially similiar. Figure 2.9(a) plots both the total file system delay and its network transport component (y-axis) as functions of VM runtime (x-axis) using normal (*demand+profile*) and prefetch disabled (*demand*) delivery policies, as well as the total file system delay for execution from *local disk*. This figure shows that the dominant part of the delay experienced by the VMM is due to network delay. The shapes of the *demand+profile* and *demand* curve pairs will likely remind the reader of those shown in Section 2.4.1, plotting cumulative disk access against runtime. This is no coincidence - the delay spikes in this graph correspond directly to the access spikes shown in Figure 2.2(b).

Figure 2.9(a) shows that in the first five seconds of runtime all network-based execution variants experience a small spike in file server delay and an almost equally-sized spike in transport delay. From this we can conclude the the transport delay component of file server delay completely dominates other components during this period. Confirming this conclusion, we see that file server delay seen by execution from memory does not budge during this period. In fact, over the course of the entire run it accounts for slightly over 1% of the total runtime.

This early spike in delay experienced by network-based variants corresponds to the first set of file accesses made by the VMM on boot of the VM. Between five and 15 seconds, *demand+profile* encounters no additional delay. This is because during this period, prefetch over the network at 100 $Mbps$ stays ahead of download. *demand*, however, being unable to prefetch data suffers another delay spike at 10 seconds as it waits to fetch additional data for the boot process. From 15 to 35 seconds both *demand+profile* and *demand* experience linear increases in delay as they their maximum download rates are exceeded by those of the memory-cached run. However, after 35 seconds, *demand+profile* based execution has managed to download all of the data it will need (save for occasional false negatives), after which file access and transport delays diverge slightly (separated by

Test	Demand (s)	Profile (s)
Fedora	08 / 46	00 / 19
Ubuntu	24 / 27	13 / 18
Win7	32 / 36	21 / 21

Table 2.4: Transport delay by VM (lower bound / empirically observed).

less than 4% of total runtime). Meanwhile, *demand* lags behind, encountering additional transport delay all the way until shutdown. These transport delays translate directly to increases in runtime, from 37 s for mem-cached to 56 s for *demand+profile* and 66 s for *demand*.

In summary, the baseline overhead for both VMTORRENT's file server and P2PM (single-client mode) is minimal with respect to other causes of delay such as prediction errors, swarm inefficiency, which we now examine.

2.6.5 Prediction/Profiling Performance

We continue our single server and client investigation, exploring *demand+profile*-based prediction performance. We begin by comparing the runtimes of *demand+profile* and *demand* with the transport delay lower bounds described in Section 2.4.

These numbers are shown in Table 2.4 which list lower bound (from Section 2.4.1) transport delay / empirically observed file server delay pairs for *demand* and *demand+profile* respectively. The first thing to note is that our *demand+profile*-based prediction technique does provide significant improvement over *demand* across all VM types.

Additionally, the lower bounds calculated earlier appear to bound empirically observed delays quite well for both the Ubuntu and Win7 VMs, particularly for the latter. However, perhaps unsurprisingly by this point, the observed delay for the Fedora VM greatly exceeds the caculated lower bound. As previously, the difference here is attributable to the Fedora VM's relatively serial startup execution pattern. While the latency delays (*e.g.,* network RTT and peer request servicing time) discussed previously may be masked by the highly parallel execution behavior of the Ubuntu and Win7 VMs, the Fedora VM lacks such parallel threads behind which to mask these delays. Consequently, even utilizing profile-based prefetching, Fedora execution on VMTORRENT encounters significant delay - as any misprediction will directly impact performance. Conversely,

for the most highly parallelized VM, Win7, vmTorrent-based execution actually achieves the calculated lower bound (at measured granularity of one second), as latency delays due to less-than-optimal prefetching are hidden by other non-I/O-blocking threads that can utilize system resources during that thread's I/O wait.

Given that many VMs will not possess sufficient parallelism to completely mask latency-based delay (and some, like our Fedora VM, will be greatly effected by such delays), we now consider the impact of false positives and false negatives. Interestingly, here, we find that there were many pairings of ordering metric and frequency cutoff (Section 2.5), that produced the shortest runtimes for a given workload on the Ubuntu and Fedora VMs. The reduction in false negatives for frequency cutoffs below a certain point was more than offset by the increase in false positives. Likewise above a certain frequency cutoff, the reverse occurred. In between these, all cutoffs had essentially equivalent performance. Similarly, switching the ordering metric between mean, median, and min altered the distribution of delays but had relatively little impact on their sum. In order to do better, it appears a more sophisticated policy that predicts pieces accesses based on the execution trace will be needed.

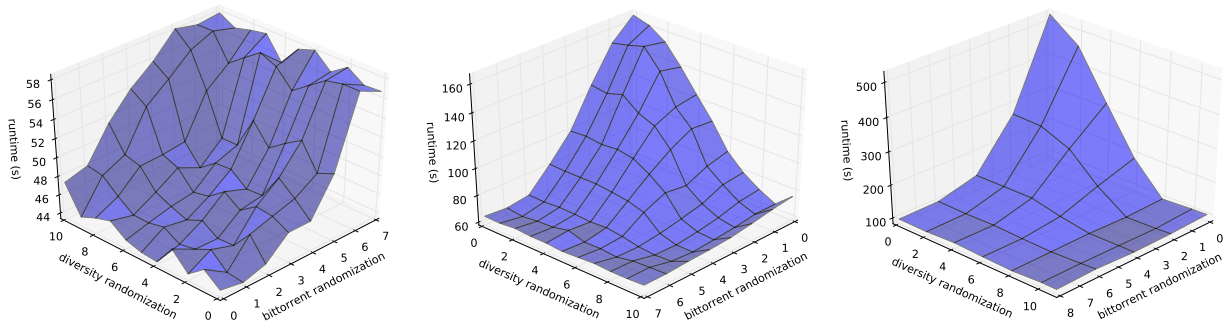
Mistimed errors present the final source of missed performance. From Section 2.4, we know that at 100 $Mbps$ networks, hosts that fail saturate their bandwidth with prefetching requests will experience linearly increasing delays. Consequently, we only need worry about mistimed errors on blocks that were need earlier than predicted. We use *rank deviation* as the measure of these mispredictions. *Rank deviation* for piece u is defined by

$$rankDev(u) = rank(u) - accessRank(u) \tag{2.2}$$

where $rank(u)$ is piece u's position in the profile and

$$accessRank(u) = |v| + |w| \tag{2.3}$$

for pieces v earlier in the profile that were actually accessed before u and false positive pieces w that came before u in the profile. Essentially, the larger the rank deviation, the earlier the piece was used with respect to those predicted to be needed before that piece. Figure 2.9(b) show the relationship between the likelihood of mistimed errors occuring (y-axis) and rank deviation (x-axis) for a representative vmTorrent-based VM execution. The essential, although perhaps unsurprising, insight here is that the less accurate our predictions are, the more likely we are to



(a) Single peer. (b) 16 peers. (c) 100 peers.

Figure 2.10: Diversity parameter settings.

incur overheads. This finding indicate that by improving the quality of our predictions we may avoid such errors. Here again, it appears that we require more sophisticated profiling and prediction techniques to considerably improve performance.

2.6.6 Swarming Efficiency

As discussed in Section 2.5.2, vmTorrent's scalability requires efficient swarming. If vmTorrent is unable to leverage the spare bandwidth of peers effectively, the potential gains from P2P will remain unrealized. Consequently, we conclude our investigation by examining the swarming efficiency of our vmTorrent prototype. In this study, we examine the swarming efficiency of the Ubuntu VM running the Boot-Shutdown workload. We expect that other VM workload combinations will demonstrate substantially similiar properties. The graphs shown here were produced for flash crowd arrival, as our current prototype implementation already possesses the scaling properties we desire for staggered peer arrival.

In Section 2.5.2, we discuss the theoretic trade-off between in-order and randomized prefetch, noting that increased randomization increases piece diversity at the expense of potential VMM delays due to pieces being fetched out of order. Roughly speaking, we would expect that for small swarms, in which swarming is less likely to be efficient and the server is likely to fill the bulk of requests, the best strategy for peers would be to keep their downloads fairly strictly time-ordered with respect to their predicted image access pattern. As swarm size increases, however swarm efficiency will become critical for peers to obtain good throughput and latency, implying that

diversity ought to be maximized, even at the cost of potential additional late positives.

The three plots in Figure 2.10 demonstrate this trade-off for one, 16, and 100 nodes respectively, plotting runtime (z-axis) against the pair of diversity windows used in our prototype's implementation[6] (in log-scale) on the x and y axes.

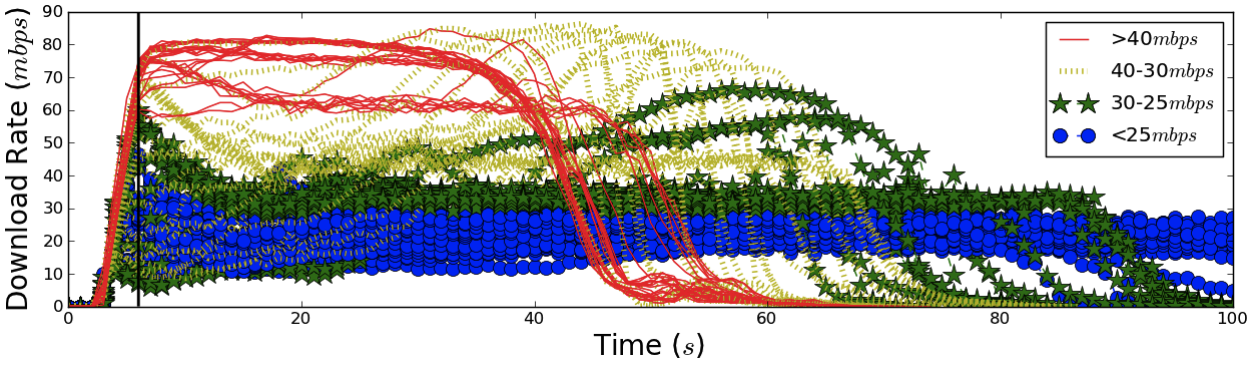
As expected, for a single node, increasing randomization simply results in additional delay as pieces arrive out of the required order (Figure 2.10(a)). However, in both Figures 2.10(b) and 2.10(c), we can see that in-order prefetch provides the very worst performance, since it fails to provide sufficient piece diversity to support efficient swarming. Moreover, the delay due to throughput losses from inefficient swarming exceeds that of the delay due to out-of-order piece arrival (as seen in the single node case) by orders of magnitude. While increasing diversity for both the 16 and 100 nodes cases past a certain point does result in slightly decreased performance, these decreases are tiny compared to the drastic gains achieved by increasing prefetch randomization (and thereby swarming efficiency). However, as can be seen from these figures, irrespective of how much prefetch randomization is introduced, the lowest achievable runtime using prefetch randomization increases greatly from 16 to 100 nodes.

To determine why this occurs, for each data point shown in Figures 2.10(b) and 2.10(c), we analyze the average peer download rate over time. These time-series are shown in Figures 2.11(a) and 2.11(b), respectively, plotting the average peer download rate (y-axis) against time (x-axis).

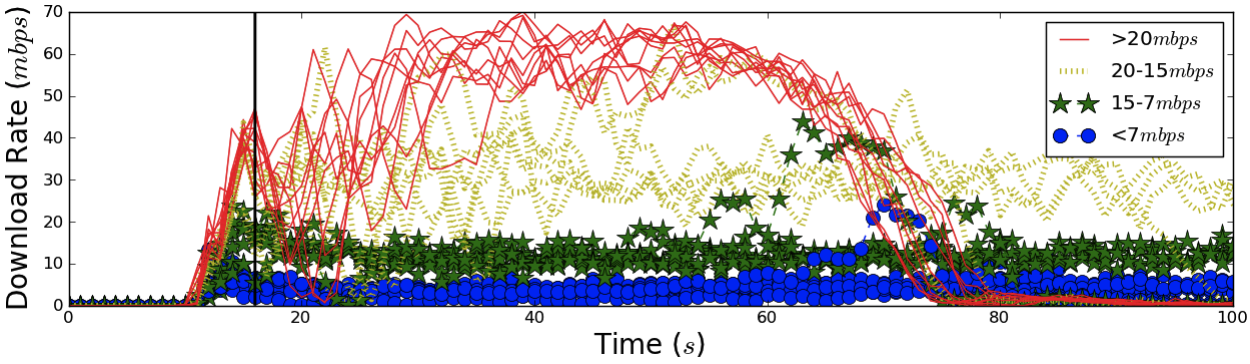
Examining Figure 2.11(a), we see that the best parameterizations shown in Figure 2.10(b) can achieve a sustainable peak rate of roughly $80\,Mbps$. Further this rate is reached after a relatively short six s startup period and remains relatively stable thereafter (until a drop-off when the profile is exhausted and vmTorrent transitions to non-prefetching mode).

Contrastingly, the behavior shown for 100 nodes in Figure 2.11(b) differs greatly. Firstly, the best parameterizations shown in Figure 2.10(c) only achieve roughly $60\,Mbps$ sustainable peak rates - a 25% drop from those achieved in 16 node swarms. Secondly, the startup period required until vmTorrent attains significant download speed is close to 3X longer than needed for the 16 node swarm. Finally, even after this point, significant fluctuation in the download rate occurs as it swings up and down until stabilizing (to the extent it does) at the peak rate roughly 35 seconds

[6]Recall from earlier discussion in Section 2.5.2 that randomization needed to be introduced in both our custom P2P manager code and the libtorrent library in order to minimize changes to that library.



(a) 16 peers.



(b) 100 peers.

Figure 2.11: Swarming efficiency.

after arrival of the flash crowd. While this clearly explains the flash-crowd arrival scaling problems seen for larger swarm sizes in Figures 2.5 and 2.6, can we use this information to devise a fix?

The behavior of this average rate indicates that for larger node populations and flash crowd arrival, randomizing prefetch requests will not provide sufficient piece diversity to support reasonably efficient swarming until a significant period of time has elapsed. The long period until the swarm begins providing significant benefit indicates that the node population lacks sufficient piece diversity for most of the first 20 seconds. Likewise, the increasingly small rate swings that occur thereafter indicate piece diversity increasing, being used up and then increasing again in fits and starts. We believe our design choice to deterministically prioritize demand requests over profile requests (see Section 2.3.2) bears responsibility for this pathology.

In smaller swarms, the server has enough capacity to fill a reasonable proportion of prefetch requests even during startup, while in larger swarms a significantly larger proportion of requests serviced are demand requests. But since all VMs in the swarm are issuing demand requests for essentially the same pieces (at the expense of low-priority randomized prefetch requests), it takes a relatively long time to achieve piece diversity in the swarm. Moreover, even once piece diversity has been realized, this diversity may quickly shrink when several of the quickest executing peers hit a section of the VM image for which their prefetch requests have not yet been filled - and thus are upgraded to demand requests, which again starve randomized prefetch requests for a short time.

We believe that this pathology can be avoided by introducing a more nuanced technique for servicing demand request and prefetch requests. Instead of giving demand requests deterministic priority over prefetch requests, the server may keep track of recently filled demand requests and demote subsequent ones for the same piece. Likewise, peers may keep local estimates of current swarming efficiency (based on upload and download rates to peers) and use these to reprioritize demand and prefetch requests. When swarming efficiency is low, a peer may increase the priority of prefetch requests and vice-versa. In future work, we plan to implementation and test such policies.

2.7 Summary

We have presented our design, implementation and evaluation of our VMTORRENT system. By incorporating *quick-start*, *scalability*, and *smoothness* features into our design, we have been able

to achieve close to perfect scalability in our quest to ***minimize VM execution time***. Our vmTorrent prototype outperforms current state-of-the-art by **up to 11X**. Further, we provide detailed examination of our prototype's functioning with concrete suggestions for future work likely to resolve current shortcomings.

Chapter 3

Green Computing: Enabling Desktop Energy-Efficiency

3.1 Overview

Remote login/access capabilities have become widespread over the past decade. Thus a large number of enterprise users now expect they will be able to interact at will with their work machine, whether they are at home or on the road.

However, traditionally, desktop OS power management schemes assume users and machines will be physically collocated. If a user is present and active the machine stays awake, if not, the machine transitions to a low-power mode. If the user returns after the machine has fallen asleep, interaction with attached input devices will wake the machine. The introduction of remote access violates this model, as remote access bypasses physically attached input devices.

Sleeping machines now must be prepared to wake on appropriate network events in addition to those occurring on local peripherals, functionality unsupported by current power management designs. Consequently, most users adopt the wasteful, but convenient, strategy of idling their desktop computers 24/7 to support what is typically very occasional remote use [Nedevschi *et al.*, 2008; Agarwal *et al.*, 2009; Webber *et al.*, 2006; Allman *et al.*, 2007]. The environmentally-conscious remainder turn their machines off when leaving the office, but at the cost of potentially lost productivity. Our own study at Microsoft Research finds hundreds of desktop machines awake, day or night – a significant waste of both energy and money. Indeed, potential savings can amount

to millions of dollars per year for large enterprises [Washburn,].

As businesses become more energy conscious, more desktops may be replaced by laptops. However, currently desktops comprise the majority of enterprise machines [eforcast,], with hundreds of millions additional desktops being sold every year [idc, ; Maisto, 2009; eforcast,]. Where users make heavy use of local resources (e.g., programming, engineering, finance), desktops continue to be the platform of choice. Hence, managing desktop power consumption is an area of both active research [Nedevschi *et al.*, 2008; Agarwal *et al.*, 2009; Webber *et al.*, 2006; Allman *et al.*, 2007] and commercial [Adaptiva Technologies, ; verdiem, ; rwt,] interest.

The most common reason that desktops are kept idling is that users and IT administrators want remote access to machines at will. Users typically want to log into their machines or access files remotely [Agarwal *et al.*, 2009], while IT administrators need remote access to backup, patch, and otherwise maintain machines. A number of solutions to this problem have been proposed [rwt, ; Agarwal *et al.*, 2009; Allman *et al.*, 2007; Nedevschi *et al.*, 2008]. The core idea behind these is to allow a machine to sleep, while a *sleep proxy* maintains that machine's network presence, waking the machine when necessary. Some of these proposals rely on specialized NIC hardware [rwt, ; Agarwal *et al.*, 2009]; others advocate use of network-based proxies [Allman *et al.*, 2007; Nedevschi *et al.*, 2008].

Unfortunately, most previous work has been evaluated either using small testbeds [Agarwal *et al.*, 2009; Nedevschi *et al.*, 2008; Allman *et al.*, 2007] or trace-based simulations [Nedevschi *et al.*, 2008]. We are not aware of any paper detailing the deployment of any of these proxying solutions in an operational enterprise network on actual user machines.[1] This is disconcerting: systems that work well on testbeds often encounter potentially serious challenges when deployed in operational networks.

This work aims to fill that gap. We describe the design and deployment of a network-based sleep proxy on our corporate network. Our design expands on the light-weight network proxy approach proposed in [Allman *et al.*, 2007; Nedevschi *et al.*, 2008], avoiding hardware modification [Agarwal *et al.*, 2009] and the overhead of virtualization [Das *et al.*, 2010]. Our architecture comprises two core components: a per-subnet sleep proxy, and a sleep notifier program that runs on each client. The sleep notifier alerts the sleep proxy just before the client goes to sleep. In response, the proxy

[1] Concurrent work [Agarwal *et al.*, 2010], provides the first study of sleep proxy deployment in an academic network.

sends out ARP probes [Cheshire, 2008] to ensure that all future traffic meant for the sleeping client is delivered to the proxy instead. The proxy then monitors this traffic applying a *reaction policy*: responding to some packets on the client's behalf, waking the sleeping client for certain specified traffic (using Wake-on-LAN (WOL) [wol,] packets), and ignoring the rest. Our reaction policy of waking for incoming TCP connection attempts on listening ports was chosen both in keeping with our goal for a light-weight, easily deployable system and based on the performance predictions of [Nedevschi *et al.*, 2008]. We provide in-depth discussion of the merits of this and alternative approaches in Section 3.3.

While our system has been in continuous operation since our first deployment in summer 2009, gaining both users and IT buy-in, this work presents the first six months of data on our initial deployment of slightly over 50 active users. Our software is deployed on user's primary workstations, not test machines. We have instrumented our system extensively; capturing numerous details about sleep and wake-up periods, why machines wake up and *why they stay up*. Instead of using generic estimates of PC power consumption, we use a sophisticated *software*-based, model-driven system, *Joulemeter*, to estimate power draw.

In this chapter:

2. We design and prototype a lightweight and economical sleep-proxying system for use in the enterprise. (Sections 3.3 and 3.4)

3. We roll out the first substantial deployment of any sleep-proxying system in a corporate environment. We deploy our software on over 50 user machines in six subnets. Almost all of these machines are primary user workstations. (Section 3.6)

4. We measure the performance of our system, collecting over half-a-year's worth of data. We instrument our system extensively; capturing numerous details about sleep and wake periods, data which explains why machines wake up and *why they stay up*. Instead of using generic estimates of PC power consumption, we use a sophisticated *software*-based, model-driven system, *Joulemeter*, to estimate power draw. (Sections 3.5 and 3.7)

5. Additionally, we describe a number of practical issues we encountered when deploying a light-weight sleep proxy in a corporate network. Many of these have been overlooked by previous work. For example, our implementation must not only deal with vanilla IPv4 and

IPv6 packets, but also tunneled packets. Our corporate network uses IPsec, and we find that *a seemingly minor implementation choice* in this setup, almost entirely *ameliorates the overhead* of dealing with this traffic. We describe race conditions that arise when the sleep proxy attempts to redirect traffic from sleeping client to itself, and provide a practical solution. We show how issues such as DHCP lease expiration and proxy failure can be handled without the need for the more complex mechanisms suggested by previous work. (Section 3.4)

6. Finally, we outline several unexpected insights provided by our work, the most significant of which indicates that one of the major concerns of previous work currently plays a secondary role in determining energy savings. The primary factor turns out to be the configuration of IT software and network services. (Section 3.7)

The highlights of our deployment experience and performance assessment are: **A light-weight system using a simple reaction policy can produce significant savings.** By analyzing trace data from our system, we find that our system allowed the clients to sleep quite well. Many machines slept over 50% of the time, despite use of a simple reaction policy. However, the average power savings was only 20%, casting a pall over the optimistic predictions made in [Nedevschi *et al.*, 2008; Agarwal *et al.*, 2009].

IT servers and applications proved a major impediment to sleep. The main cause of reduced power savings in our enterprise network was due to the IT setup. We find that while users do access their machines remotely, remote accesses by IT applications are the primary cause of both machines being woken (*fitful sleep*) and being kept awake (*insomnia*). IT server connection attempts repeatedly woke sleeping machines. In one extreme case, a single machine was contacted over 400 times within a two-week period. Additionally, some of the locally running IT applications (e.g., virus scanners) kept machines up by temporarily disabling sleep functionality. We also identify bugs in common software (e.g., Adobe Flash player) that interfere with proper sleeping. Fortunately, it appears there is significant room for improving the compatibility of IT setup and effective sleep. We discuss IT setup impediments and remedies further in Section 3.7.6.

The rise of cloud-based applications may demand more complex reaction policies. Three of our users required support for two popular cloud-based applications, LiveMesh and LiveSynch. Machines running these, or similar, applications must initiate TCP connections to the cloud server, which are used to inform them of any pending updates. These connections can either be periodic,

or long-lived, but it must be initiated by client. We refer to cloud applications of this type as *persistent*[2]. Consequently, to support such cloud-based applications the sleep proxy will need to keep some additional state which may be as simple as sending TCP keep-alives or as complex as running a virtualized client-side of the application. While this did not pose a major issue in our operational environment/population, as/where the predominance of persistent cloud applications increases, reaction policies supportive of this model will be needed.

Overall, we believe that the insights gleaned from our experience will be useful in guiding the design and deployment of future sleep solutions in enterprise networks.

3.2 Related Work

While the basic concept of sleep proxying has been known for some time [Christensen and Gulledge, 1998], it has received much renewed attention lately [Allman *et al.*, 2007; Nedevschi *et al.*, 2009; Agarwal *et al.*, 2009]. Among recent publications, the two most closely related to our work are [Agarwal *et al.*, 2009] and [Nedevschi *et al.*, 2009].

In [Agarwal *et al.*, 2009], the authors describe a hardware-based solution. They augment the NIC with a GumStix [gumstix,] device, which is essentially a small form factor, low-powered PC. Once the host machine goes to sleep, the GumStix device takes over. It handles select applications (e.g., file downloads) on behalf of the host PC, but wakes up the host PC for more complex operations. While this approach is more flexible than the sleep proxy we have built, it is far less practical for two reasons. Not only is additional hardware required on every PC, but both applications and host OS modification are required to enable state transfer between host PC and GumStix device. Both these requirements are a substantial barrier to widespread deployment of this technology. In contrast, our approach requires neither extra hardware, nor application modifications.

In [Nedevschi *et al.*, 2009], the authors carry out an extensive trace-based study of network traffic, arguing for a network-based sleep proxy. Their primary finding is that in an enterprise environment, broadcast and multicast traffic related to routing and service discovery cause sub-

[2]Many cloud-based applications including most "software as a service" applications (e.g., Google Docs) are not of this type

stantial network "chatter", most of which can be safely ignored by a sleep proxy. They also posit that most unicast traffic directed to a host after it has gone to sleep can also be ignored, so long as the host is woken when traffic meant for a set of pre-defined applications arrive (early work had focused on avoiding disrupting existing TCP connections [Christensen *et al.*, 2004; Jimeno *et al.*, 2008]). Based on these insights, they propose a number of sleep proxy designs.

While our proxy design builds upon the insights of [Nedevschi *et al.*, 2009], we make several additional contributions in this dissertation. First, unlike [Nedevschi *et al.*, 2009], our design includes a client-side agent, which considerably simplifies the overall architecture, making it robust, and virtually configuration-free. Secondly, we build and deploy our sleep proxies in a real operational network on users' primary workstations. In contrast, the prototype in [Nedevschi *et al.*, 2009] was tested only a small testbed without real users, and did not address challenges such as IPsec traffic and proxy failures. Third, our instrumentation measures sleep and wake-up behavior of operational machines. We document why machines do not sleep, when and why they wake, etc. Fourth, our deployment includes a model-based power measurement component. Since machine power usage can vary by 2.5x while awake, our power estimates provide significantly greater fidelity than the "one size fits all" model used by [Nedevschi *et al.*, 2009].

Two pieces of concurrently published work address alternative sleep proxying architectures that make use of a networked sleep proxy. [Agarwal *et al.*, 2010] implements a stub-based reaction policy along the lines of [Agarwal *et al.*, 2009] and evaluates it in an academic network, while [Das *et al.*, 2010] runs client machines within a hypervisor and migrates these to the sleep proxy machine. We provide further comparison in Section 3.3.3.

We now turn to commercial systems. Intel offers two hardware-based solutions, Remote Wakeup Technology (RWT) [rwt,] and Active Management Technology (AMT) [amt,], that can remotely wake up a sleeping machine. AMT is primarily meant for management tasks (e.g., out of band access for asset discovery, remote troubleshooting). RWT is more closely related to our work. RWT requires the NIC of the sleeping machine to maintain a persistent TCP connection to an authorized server. The NIC wakes up the host machine upon receiving a special message over this TCP connection. RWT requires modification of client applications and works only with Intel hardware. Even the wakeup service has to be digitally signed by Intel. In contrast, our solution is entirely software-based, hardware-agnostic, and requires no application modification.

Apple has recently released a sleep proxy geared toward home networks that works only with select Apple hardware [apple-wol,]. For enterprise networks, systems such as Adaptiva [Adaptiva Technologies,] and Verdiem [verdiem,] are available. The primary focus of these systems is to enable the system administrator to estimate power usage, and wake up sleeping machines to perform management tasks such as patching. A number of industry participants are trying to standardize basic sleep proxy functionality [Committee, 2009].

Several other approaches to saving power, such as power-proportional computing [Barroso and Hölzle, 2007], dynamic voltage and frequency scaling [Sinha and Chandrakasan, 2001], the TickLess kernel [tic, 2009] and OS-level power management [Snowdon *et al.*, 2009] have been investigated, and can be used in conjunction with our system. Researchers have also looked at networking hardware and software stacks as potential targets for power savings. Examples include [Gupta and Singh, 2003; Nedevschi *et al.*, 2008; Chabarek *et al.*, 2008; Blackburn and Christensen, 2009]. [Al-Fares *et al.*, 2008; Valancius *et al.*, 2009; Mahadevan *et al.*, 2009] examine data center power consumption and savings approaches.

Prior work has shown that CPU utilization and certain performance counters can be used to estimate computer energy use [Rivoire *et al.*, 2008; Bircher and John, 2007; Fan *et al.*, 2007]. Our power estimation technology provides enhanced accuracy by considering additional factors not considered in prior work, such as processor Dynamic Voltage and Frequency Scaling (DVFS) states and monitor power.

3.3 Design Goals & Alternatives

As discussed earlier, enterprise users often do not let their machines sleep as they may require remote access. Our goal in deploying a sleep proxy is to encourage users to allow their machines to sleep – by ensuring their machine will wake on remote access attempts. We now describe the basic functionality required from a sleep proxy, define our design goals, and describe design alternatives. Before we begin, we note that our use of the term "sleep" refers to ACPI S3 (suspend to RAM) [acpi,]. Our system supports ACPI S4 (hibernate) and S5 (power off) as well.

3.3.1 Basic sleep proxy functionality

A *sleep proxy* detects when a *sleep client* machine (M) has gone to sleep, typically because that machine's *idle timeout* had been reached.[3] The proxy then monitors network traffic destined for M. Based on a pre-defined *reaction* policy, the sleep proxy will, (a) respond to some of the traffic on behalf of M (e.g. ARP requests for M), (b) wake M for selected traffic (e.g. TCP SYNs for M) and (c) ignore the remainder.

3.3.2 Design goals

Our goal is to build a practical, deployable sleep proxy for typical corporate networks, composed of desktop machines with wired connectivity. In a typical usage scenario, the user's machine goes to sleep, and wakes automatically on remote connection attempts.

The design of our sleep proxy was directed by four goals. (a) The system had to save as much power as possible, (b) while minimizing disruptions to users. It is critically important to ensure a sleeping machine is always woken when the user desires access: otherwise no one would use the system. Furthermore, the system had to be (c) easy to deploy and maintain, since we operated without the benefit of a large IT staff. We explicitly decided not to add hardware to client machines, as it makes deployment significantly harder. Finally, we required the architecture be (d) scalable and extensible, since the system had to operate in a dynamic live network

It was not our goal to support laptops *per se* as they offered much less opportunity for power savings. They consume much less power when active, and are more often put to sleep by users. Thus, while some of the work we have done is applicable to laptops, we do not address laptop-specific challenges such as mobility in our work.

As all the machines in our network run Windows, some details of our implementation are Windows specific. However, our architecture is designed to be OS agnostic.

3.3.3 Design alternatives

We now consider three design alternatives, and evaluate them in light of our requirements.

[3]In Windows, the idle timeout is typically 30 minutes from power up / last user activity, and two minutes for any other wake cause (e.g., scheduled wakeup, WOL).

3.3.3.1 NIC Pattern Matching

The first potential approach is to simply use the combination of *Wake-On-Pattern+ARP Offload*. This capability is available on most modern wired NICs.

How it works: The NIC effectively acts as the sleep proxy for the machine. It responds to incoming ARPs on behalf of the sleeping machine (*ARP Offload*), thereby maintaining the machine's network presence. The NIC can be programmed to detect specific patterns in incoming traffic, and wake up the host machine if a packet with specified pattern arrives (*Wake-On-Pattern*). The interface for specifying patterns [ndis-pattern,] includes built-in support for IPv4 and IPv6 TCP-SYNs; one only need specify additional information (e.g., ports). Raw bit patterns can also be specified.

Pros: These NICs are available on most modern machines, so no additional hardware needs to be deployed.

Cons: We found that for our purposes the capabilities offered by these NICs were not adequate. Our corporate network is quite complex: it supports IPv4, IPv6, v6-over-v4 and requires IPsec. To ensure machines were woken whenever users required access, we had to handle packets requiring flexible inspection (e.g. a TCP SYN in an ESP packet carried in an IPv6 packet, tunneled in an IPv4 packet - Section 3.4.3.2). While such packets may be detected by explicit bit-pattern matching, the number of wakeup patterns needed is a multiple of the number of listening ports (to detect tunneled variations) plus several base patterns for standard WOL functionality. NICs on older machines can support as few as four wake patterns and are limited to detecting matches in the beginning of the packet which restricts the ability to detect tunneled packets. Moreover, future needs (e.g., support for persistent cloud applications) may dictate stateful reaction policies or deeper packet inspection, beyond current NIC capabilities. Thus, this approach fails criteria (b) and (d).

3.3.3.2 Virtualization

How it works: Users install a hypervisor on their desktop, and then install and use a VM on top of the hypervisor[Das *et al.*, 2010]. When the desktop machine needs to sleep, the VM is migrated to a hosting server. When necessary, (e.g. a CPU intensive application is run), the desktop machine is woken, and the VM migrated back.

Pros: This approach is attractive because if the migration can be made seamless, the desktop does

not have be woken up for transactions of even moderate complexity that can be carried out on the hosting server. As the machine can go to sleep without interrupting existing network connections, the machines can go to sleep much more often, and hence the power savings may be greater.

Cons: To deploy a system based on this approach, we would have had to install hypervisor on the end user systems and boot their existing OS as a VM. Most users would not have agreed to such a drastic change to their work environment. Apart from taking a performance hit, virtualization may encounter problems with a number of common end-user devices (e.g., cameras, external drives), whose drivers do not always work well when virtualized.

3.3.3.3 Network-Based Sleep Proxies

This approach was proposed in [Christensen and Gulledge, 1998], its feasibility recently given careful study by [Nedevschi *et al.*, 2009].

How it works: This approach relies on a separate machine acting as a sleep proxy for the sleeping machine. The sleep proxy detects when a client goes to sleep. It then modifies Ethernet routing (Section 3.4.3.1) to ensure that all packets destined for the sleeping machine are delivered to the sleep proxy instead. The proxy examines the packets, and wakes up the sleeping client when needed, by sending a Wake-On-LAN (WOL) [wol,] packet.

Pros: Very little hardware support is required from the client machine - the client NIC only needs to support WOL. As the sleep proxy runs on a separate, general purpose computer, it has great flexibility in handling incoming traffic for the sleeping machine. The sleep proxy can do complex, conditional packet parsing and can even wake the sleeping machine based on non-network events such as requests by system administrators, users entering the building (with support from building access systems), etc. This design also scales well (Section 3.7.5.2).

Cons: This design requires deployment of a sleep proxy on a separate machine (generally one per subnet supported). In most variations a client-side application must be installed as well.

We have chosen this approach as it is both very easy to deploy and requires minimal changes to user machines. It affords great scalability and flexibility as the sleep proxy can be changed without disturbing the client machines. We have chosen to use light-weight reaction policy which simplifies both client and proxy software complexity and allows a very large number of hosts to be handled by a single proxy. This reaction policy does cause existing network connections are broken. We

argue that this is not an issue for typical corporate workloads (Section 3.4.4), although this may change if persistent cloud computing applications play a greater role in corporate environments.

Contrastingly, [Agarwal *et al.*, 2010] uses a stub-based reaction policy, capable of maintaining existing network connections and waking the host somewhat more infrequently. This comes at the cost of implementation complexity and will allow fewer clients to be hosted on a sleep proxy. Their implementation uses an ESX server that would preclude either low-power sleep proxies (Section 3.6) or peered proxying (Section 3.8). Their reaction policy faces the same impediments from sleep-unfriendly IT setups as ours - by far the main source of lost sleep opportunities in our environment - as IT tasks generally require waking clients.

3.4 Architecture

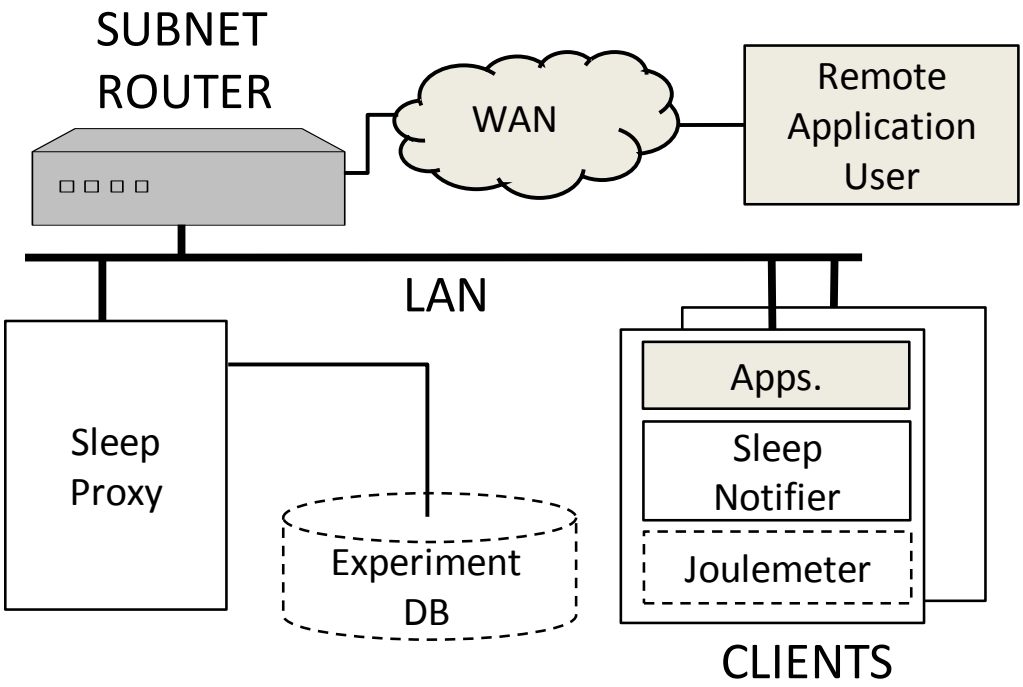


Figure 3.1: System block diagram. Blocks shaded gray represent existing components that are not modified in any way for the sleep proxy to work. Blocks with dashed outlines are part of our instrumentation setup.

The overall architecture of our system is shown in Figure 3.1. We require one sleep proxy per

subnet. We also required the clients to install a small background service[4], *sleep notifier*. In this section, we will focus on the design of the sleep proxy and the sleep notifier which form the core of our solution. We discuss Joulemeter in Section 3.5.1

As discussed earlier, a sleep proxy responds to some traffic, wakes the sleep client for other traffic, and ignores the rest. Our choice of reaction policy is similar to that of the proxy scheme (*proxy3*), which [Nedevschi *et al.*, 2009] found provided the highest simulated power savings. This reaction policy, whose rationale is discussed in Section 3.4.4, responds mechanically to IP resolution requests (e.g., ARP) and wakes the sleep client only on TCP connection attempts to listening ports[5], ignoring other traffic.

Before digging into design details (Section. 3.4.2 and 3.4.3), we provide a quick overview of how our system works.

3.4.1 System Overview

Imagine a *sleep client* M running sleep notifier. M's sleep notifier registers with the OS to receive notification when the machine is about to go to sleep. At such time, the OS alerts the sleep notifier.

M's sleep notifier then alerts the sleep proxy S that M is going to sleep, providing a list of M's TCP ports in the *listening* state (actively listening for incoming connections). Assume that the SSH port, 22, is one such port.

Upon receiving the notification, S adds M to its list of proxied clients and sends out an ARP probe (Section 3.4.3.1), re-mapping the switched Ethernet to direct future packets for M to the network port at which S resides. S now begins receiving traffic that was meant for M. S responds to ARP requests and IPv6 *Neighbor Discovery* packets as if it were M, thereby maintaining M's network presence and ensuring traffic for M continues to arrive at S.

Some time later a remote client C attempts to connect to the sleeping machine M, using SSH. As the first transport-layer action taken in establishing this new connection, C sends a TCP SYN on port 22 to M which the switched Ethernet routes to S.

Upon examining the packet, S determines that it is a TCP SYN meant for M and destined to a

[4]A daemon, in Unix terminology.

[5]There being no reason to wake the machine for connections to non-listening ports, which would just be ignored anyway.

port on which M was listening when it when to sleep. S therefore wakes M up by sending it a WOL packet (Section 3.3.3.3), removes M from the proxied client list, and drops the TCP SYN. As M wakes up, it sends its own ARP probes, which ensure that future traffic meant for M will arrive at M's network port. Meanwhile, C retransmits this SYN following the normal TCP timeout. The retransmitted SYN arrives at M, who responds as normal, thereby establishing the TCP connection without C being any the wiser, except for a small delay - quantified in Section 3.7.5.1.

3.4.2 The Sleep Notifier

Installing the sleep notifier on sleep clients greatly simplifies the overall design. As the service runs on user desktops, our aim is to make the sleep notifier code robust and stateless, requiring as simple configuration as possible.

The primary purpose of the sleep notifier is to notify the sleep proxy when the machine is going to sleep. Just before a machine is put to sleep, the Windows OS sends out a 'get ready for sleep" (a `Win32_PowerManagementEvent`) event to all the processes and drivers running on a machine, allowing them to prepare for sleep. The sleep notifier registers to receive this event. Upon receiving the event, the notifier immediately broadcasts a *sleep notification* packets (encapsulated in a UDP packet to port 9999), containing a "going-to-sleep" opcode and list of the sleep client's listening TCP ports, to the subnet broadcast address. For reliability it retransmits the packet three times.

In keeping with our light-weight approach, sleep notification packets are broadcast. The sleep client does not need to know the identity of the sleep proxy and requires no configuration nor stable storage, as there is no state to be kept. The sleep notification packet obviates the need for active probing sleep clients to determine sleep status (as done in [Nedevschi *et al.*, 2009]) or which ports should be proxied ([Nedevschi *et al.*, 2009] restricted proxied ports to a manually pre-configured set).

Since the sleep notifier may have less than two seconds in which to send the sleep notification packet before the machine falls asleep [6], it is possible, albeit unlikely, that the notification packets will not be sent in time. Consequently, the sleep notifier also sends out periodic heartbeats when the machine is awake. These heartbeats are identical to the sleep notification packet, save that they use a "heartbeat" opcode. In our current implementation, heartbeats are sent out every 5

[6]The sleep notifier cannot reliably force the system to remain awake once the notification is broadcast

minutes, with some jittering. When the sleep proxy misses two consecutive heartbeats from a client, it immediately sends a WOL packet to that client. If, after sending the WOL, neither a heartbeat nor a sleep notification is subsequently received from the client, the proxy assumes that the machine has left the network and removes it from the list of proxied sleep clients.

3.4.3 The Sleep Proxy

The sleep proxy needs to monitor incoming traffic to the sleep client and also wake that client by sending a WOL packet on the subnet broadcast address [7]. Redirecting traffic destined for a given machine to another machine outside of its local subnet requires substantial support from routers. Thus, the sleep proxy has to run either on the subnet router itself, or on some other subnet machine. Running a sleep proxy on the subnet router was not possible, so we use one dedicated machine per subnet to act as a sleep proxy for machines in those subnets.

3.4.3.1 Rerouting

Like most enterprise networks, our network is a switched Ethernet network. Thus, unicast traffic for a host is not generally visible to other hosts on the network. Thus, upon receiving the sleep notification from a client, the sleep proxy needs to ensure that the traffic destined for sleeping clients is re-routed to the sleep proxy's NIC.

While there are a few ways to affect such re-routing, we have found sending *ARP probes* [Cheshire, 2008], as shown in Figure 3.2, to be the most reliable method. A machine uses these ARP probes to advertise its MAC and IP address, and to perform duplicate address detection (DAD). Also, the subnet switches refresh/remap their internal routing tables upon receiving these probes.

Thus, when a sleep proxy receives a sleep notification from a client, it issues specially crafted ARP probes *pretending to be the sleep client* (refer again to Figure 3.2). This ensures that subsequent network traffic meant for the sleeping machine is delivered to the sleep proxy instead.[8]

[7] This packet must be broadcast since at the time it is sent, the subnet's routing is set to deliver all packets meant for the sleeping host to the sleep proxy.

[8] An alternate way of doing this would be to replace $M.MAC\_ADDR$ with the sleep proxy's MAC address, however this could cause the DAD mechanism to be triggered if the sleep client were to wake very quickly after sleep.

	Field	Value
Ethernet Header	Source Addr	$M.MAC\_Addr$
	Destination Addr	FF:FF:FF:FF:FF
ARP Request	Sender MAC Addr	$M.MAC\_Addr$
	Sender IP Addr	0.0.0.0
	Target MAC Addr	00:00:00:00:00:00
	Target IP Addr	$M.IP\_Addr$

Figure 3.2: ARP probe for sleep client M.

When a sleeping machine wakes (either because the sleep proxy woke it, or because it was woken for some other reason), it will naturally send out a fresh set of ARP probes generated by the OS to ensure that it can re-use the same IP address that it had before it went to sleep. This has two nice side effects. First, the subnet switches now begin forwarding traffic meant for the sleeping (and now awake) machine, back to that machine, instead of the sleep proxy. Secondly, as these probes are broadcast, the sleep proxy can see them, allowing it to immediately recognize when clients have woken and cease proxying.

3.4.3.2 Reaction Policy

As discussed earlier our sleep proxy reaction policy responds to IP address resolution traffic, examines incoming TCP connection attempts, and ignores all other traffic. This means that (a) current TCP connections are broken and (b) UDP applications are not supported.

Intuitively, the former would seem to be a safe strategy for many applications. The sleep proxy is not responsible for putting a machine to sleep. That decision is taken by the local OS. If the local OS deemed it safe to put a machine to sleep while it had TCP connections open, then clearly the applications to which those TCP connections correspond have not placed requests to prevent sleep (a standard feature of modern OSes). Moreover, most common corporate network applications are inherently disconnection tolerant (e.g., email, web browser).

As for the latter, in our network, practically all desktop applications use TCP. Users typically access their machines either via SMB (to retrieve files) or via Remote Desktop. Upon initiation, both these applications start new TCP connections, and hence send corresponding SYNs. Routine

maintenance is handled via RPC calls, and this traffic also goes over TCP. Additionally, it given the flexible parsing power of our sleep proxy, it should not be difficult to extend our technique to cover UDP traffic meant to initiate new connections for particular applications requiring such (e.g., NFS version 2).

The impact of ignoring non-TCP traffic and breaking currently existing TCP is difficult to estimate empirically. However, we believe the proof is in the pudding: after months of running our code, none of our users or IT staff have complained that their machines did not wake on remote access and the only applications which we received request support for were the two cloud-based applications run by a small minority of users. [Nedevschi *et al.*, 2009] provides a more detailed discussion of our reaction policy and comparison with other possibilities.

3.4.4 Implementation Challenges

IPsec:green: Responding to IP address resolution traffic is easy: the sleep proxy simply issue ARP responses and Neighbor Discovery advertisements as if it were the sleeping client. Handling TCP connection attempts is more complicated. To detect an incoming TCP connection attempt the sleep proxy must examine the packet's IP header confirming it was destined to a currently proxied machine, and contains a TCP SYN with a destination port on which that machine had been listening. While it is easy to parse a TCP SYN contained in a vanilla IPv4 or IPv6 packet, our network (like most corporate networks) is more complicated in both its use of IPv6 tunneling and IPsec ESP authentication[9]. Tunneling comes in three flavors, ISATAP, 6over4, and Teredo [teredo,]. Our current implementation handles ISATAP and 6over4. ISATAP packets are already unwrapped for the sleep proxy by the ISATAP router and arrive as IPv6 packets on the sleep client's ISATAP IPv6 address. Thus these packets require no additional processing. 6over4 packets arrive as IPv4 packets whose next protocol is 6over4. The inner packet is then removed and parsed as a standard IPv6 packet. Our current implementation does not handle Teredo wrapping, since it is being phased out in favor of the first two mechanisms.

The use of IPsec [win-ike,] presents a number of challenges. Imagine a remote machine C trying

[9]Note that tunneling and IPsec can be (and indeed are) used together. Our sleep proxy routinely sees and handles TCP SYNs that are encapsulated in an ESP payload, which is carried in an IPv6 packet, which is tunneled inside an IPv4 packet.

to connect to sleeping machine M using TCP. Let S be the sleep proxy. If IPsec is in use, there are two possibilities. Either C has not communicated with M in recent past, or it has.

If C has not recently communicated with M it would first try to establish a new security association by doing IPsec key exchange (IKE). The IKE packets are sent via UDP. The IKE sent by C end up at S. Recall, however, that our sleep proxy wakes up packets only on receiving TCP SYNs. Thus, the sleep proxy would never wake up M. However, Windows optimizes connection establishment by requiring C to send a TCP SYN "in the clear" as it begins the key exchange [win-ike,]. This is done to speed up the connection establishment: TCP handshake can happen in parallel with IPsec handshake. This works in our favor: the sleep proxy can detect the TCP SYN transmitted by C, and wake up M, which can then finish the key exchange. Otherwise, M would need to be woken for every IKE attempt. As we shall see later, in our network this would have lead to many spurious wake-ups.

Conversely, if C has recently communicated with M, it may have cached the security association information. Since our network uses Encapsulated Security Payload (ESP) [Kent and Seo, 2005] protocol, C would encrypt the TCP SYN it sends. While the TCP SYN would end up at S, there is no way for S to decode the packet. This would have been incompatible with our reaction policy, except that our network uses ESP only with integrity service: the payload itself is not encrypted. Thus, S can parse the packet, inspect it, and wake M if needed.

Thus by choosing an IPsec setup in which both ESP payload encryption is disabled and enabling TCP connection establishment optimization, the need for running a heavier-weight reaction policy is ameliorated.

ARP probe timing: The sleep proxy cannot simply send out ARP probes as soon as it receives the sleep notification from a client, as that client may send other packets before the network card sleeps. If ARP probes from the sleep proxy intermingle with traffic generated from the client that is about to fall asleep, the spanning tree protocol may end up in state where packets meant for the sleeping machine are not routed to the sleep proxy. In our early implementations, this problem created much heartache.

To avoid this problem, after receiving the sleep notification, the sleep proxy begins pinging that sleep client. The sleep proxy waits for five consecutive ping failures before sending out ARP probes and thereby taking over for the sleeping client.

Daily wake-up & DHCP lease expiration: Currently, the sleep proxy wakes all sleeping clients at 5AM. The primary reason is to allow these machines to initiate any backup or scanning activity. The wake-up also obviates the need for the sleep proxy to handle DHCP traffic on behalf of the clients. In our network, DHCP leases are valid for 30 days. When the client is awake, it renews the lease every day. Furthermore, it also renews the lease when it wakes up. As each client is guaranteed to wake up at least once a day, we did not need to implement DHCP renewal on our sleep proxy. The same mechanism also protects against address black-holing: whereby a sleep proxy keeps holding on to the address of a machine that has departed the network. If heartbeats are not seen for a sleep client after the daily wake-up, that machine is inferred to have left the network (as described earlier).

Failure of sleep proxy: In our current implementation, each subnet is served by a single sleep proxy. This creates a single point of failure. We have designed, but not yet implemented a primary-backup solution for ensuring additional reliability. Another possibility is to design a purely peer-to-peer solution (Section 3.8). Our design does offer protection against a sleep proxy crashing, and restarting. The sleep proxy stores the MAC addresses of all the machines that it is proxying for in a log maintained on non-volatile network storage. Upon restarting, the sleep proxy checks the log, and proactively wakes up all the machines by sending them WOL packets. This ensures that the sleep proxy starts operations in a consistent state.

Multi-homed machines: The sleep proxy architecture can easily handle multi-homed machines as long as (*i*) the sleep notification goes out on all interfaces and (*ii*) a sleep proxy is available on each network that receives incoming connection attempts.

Manual wake-up: Apart from the "automatic" wake-up described so far, we also provide for remote, manual wake-up of sleeping clients. This is achieved by maintaining a website outside our corporate firewall. Every sleep proxy maintains an open TCP connection to this web server. Users can type in the name of their machine on this website. The web service sends the name to every sleep proxy, and if a sleep proxy has the specified machine as a client, it wakes that machine up by sending it a magic packet. This service provides a "last resort" wake-up alternative and also allowed the small minority of cloud application users to manually reconnect cloud apps.

3.5 Instrumentation

Our sleep proxy keeps a detailed log of its interactions with clients, including when and why the clients go to sleep or wake up. On client side, we use *Joulemeter*, to estimate the power consumption of the clients, and gather information about why clients *stay* awake. Joulemeter is installed as a separate, optional service on clients.

3.5.1 Monitoring power consumption

To quantify the energy savings of our approach, we desired an accurate method of estimating our deployment's power consumption. Different machine makes and models consume power at differing rates. Further, a given machine consumes vastly different power depending on its CPU utilization level, P-state and whether its monitors are on or off. For instance, the power usage of an HP xw4300 workstation with two monitors varied from 141W to 240W with processor utilization, and changed by an additional 120W with monitor power state for a total variation of 2.5X.

However, desktop workstations do not typically have built-in instrumentation to measure power usage, and we wished to avoid attaching external power-meters to each machine for the same reasons we rejected hardware augmented sleep proxying approaches. Consequently, we used a software solution, *Joulemeter*, that produces power usage estimates based on hardware activity and pre-calibrated machine models.

The key principle behind Joulemeter's energy estimation is to use a machine specific power model. The model consists of a set of equations that relate the hardware configuration and resource utilization levels to power usage. Our current model takes into account processor P-states, processor utilization, disk I/O levels, and whether the monitor(s) are on or off. The power model for a specific hardware configuration is learned via *calibration* - controlled experiments in a laboratory settings. Once the power model is known, the machine's power consumption at run time can be estimated by monitoring CPU utilization (and P-state), disk utilization and monitor status. We omit the details of model construction due to lack of space. For a preliminary introduction see [Kansal and Zhao, 2008].

Figure 4.2 shows Joulemeter estimates versus measured power consumption (using a hardware power meter) for a HP d530 workstation with 2.66GHz Pentium CPU running a workload generator

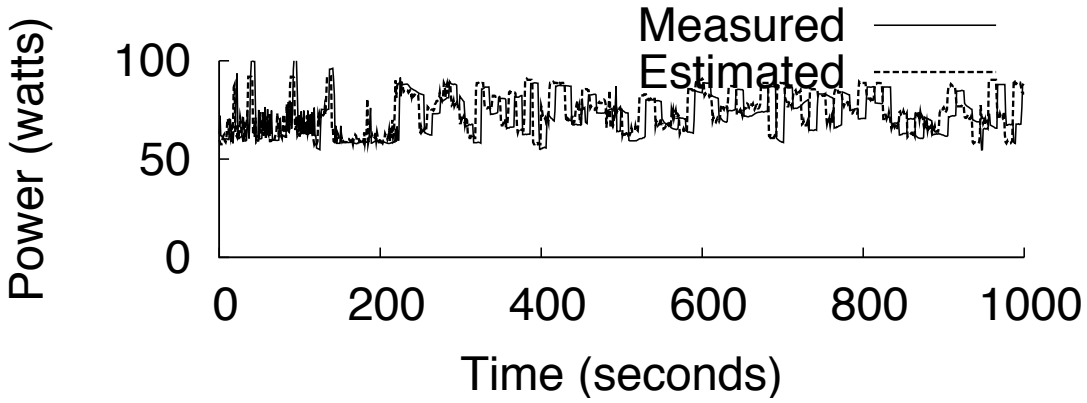


Figure 3.3: Measured and predicted power consumption.

that loaded the CPU and disk at random. The estimates were generated using the calibrated model produced from *a different* workstation with the same model and CPU. The results shown confirm Joulemeter's estimates track closely with the actual power consumption. In practice, no two systems are exactly alike. Still, in validation testing we found Joulemeter predictions to be accurate within 20%

In our deployment Joulemeter generated power readings were averaged over 30 second intervals and periodically uploaded to the database. We have built up a library of power models covering most of our client machines.

3.5.2 Monitoring machine insomnia

To determine why a machine is awake, Joulemeter relies on two sources. First, it periodically checks the *lastUserInput* timer provided by the OS. This timer provides the time of last user activity. We compare the value of this timer to the idle timeout (a typical Windows default value is 30 minutes). If user activity has occurred more recently than the idle timeout, we assume that the machine is being kept awake by user activity. We note that due to various technical issues this timer is not always available, so we cannot always determine whether the user is active.

We also find that machines often stay awake even when the idle period exceeds this duration. To determine the reasons behind this, we rely on *powercfg.exe* utility that ships as part of Windows 7. The utility can often (but not always) shed light on why a machine is staying up by detailing *requests* to the OS for the machine to remain awake. For example, a remote machine may be holding a file open or a defragmenting routine may be running. Joulemeter periodically collects

this information and reports it to the central database. Analysis of this information is presented in Section 3.7.

3.6 Implementation and Deployment

Our deployment consisted of 6 proxies (one for each of our network's 6 wired subnets), 51 clients, an SQL database, and the manual wakeup webservice mentioned earlier (standard IIS webserver with code written using ASP.NET). Most of the code is written in C# (5000 lines).

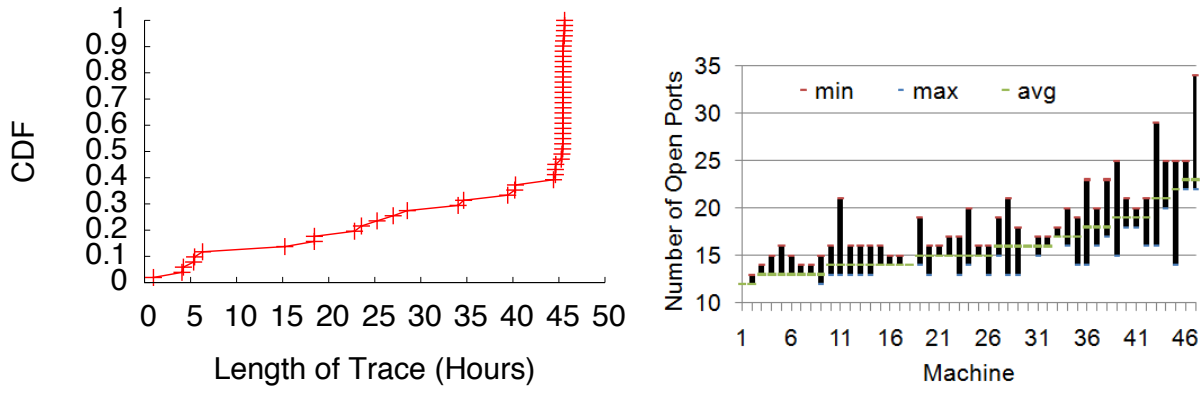
Only the sleep proxy contains any significant amount of unmanaged code. The sleep proxy relies on PCAP to capture and examine incoming packets. A small custom driver allows the sleep proxy to craft and inject ARP probes while bypassing the network stack. The primary data structure in the sleep proxy is a hashtable used to keep track of clients and their status. We first used ordinary desktop machines as proxies and have begun migrating to the low-powered, small-form-factor machines drawing less than 25 watts of power.

On client side, apart from the required sleep notifier service, the clients install three optional applications: Joulemeter, a GUI program displaying sleep statistics and estimated energy savings, and an auto-updater service that keeps client-side code up-to-date. During client installation, we ensured that Wake-On-LAN was enabled and ARP offload (which is enabled by default for certain cards in Windows 7) was disabled on the client's NIC. We also set the idle timeout to 30 minutes.

3.7 Results

This section is guided by several overarching questions. What is the sleep and wake behavior of machines in our system? How much power did our solution save? What might be done to obtain additional power savings? What impact did our setup have on user experience? Was the sleep proxy architecture scalable? For the impatient reader, we highlight our main insights at this section's end (Section 3.7.6).

We begin by describing the details of our dataset.



(a) CDF of machine trace length.

(b) Number of listening ports.

Figure 3.4: Trace length and listening port distribution.

3.7.1 Dataset Overview

While our deployment has been active for half a year in various stages, for the rest of this section we focus on the 45 day period from 19 November 2009 through 3 January 2010. During this time, we gathered data from 51 distinct machines belonging to 50 distinct users. As users installed our software at differing times, not all machines provided data for the entire period (although most did). Figure 3.4(a) shows the cumulative distribution of trace lengths of individual machines. Our users were a self selecting group, so their behavior may not be representative of all user populations.

3.7.1.1 Machines in our study

As we noted in Section 3.5.1, machine power consumption depends on the particulars of that machine's hardware configuration. The hardware configuration of machines in our deployment was varied, but not overly so. Of the 51 machines, 43 are HP and 6 were Dell. Only one of the machines has an AMD processor, the rest having Intel CPUs. Most of the machines are dual or quad cored. The CPU frequencies vary from 2-3.4GHz. Twenty seven machines had one monitor, 20 had two, and five had three. Five machines ran Windows Vista, all the rest ran Windows 7.

As we wake up machines for incoming TCP SYNs only on listening ports, it is worth examining the number of listening TCP ports on each machine. This number, of course, varies over time, as active processes and settings change. Figure 3.4(b) shows the min, max, and average number of listening ports by machine. One machine had 35 ports open simultaneously!

3.7.1.2 Traffic

Since all traffic destined for sleeping clients arrives at their sleep proxies, we can examine this traffic in centralized manner, without installing sniffers on individual machines. While we have deployed a sleep proxy on each of our six subnets, 59% of our machines are connected to the largest subnet. We have seen as many as 800 active machines on this subnet. We examined in detail a trace of all (23 million) packets arriving at the sleep proxy serving this subnet during a typical work week (5.5 days).

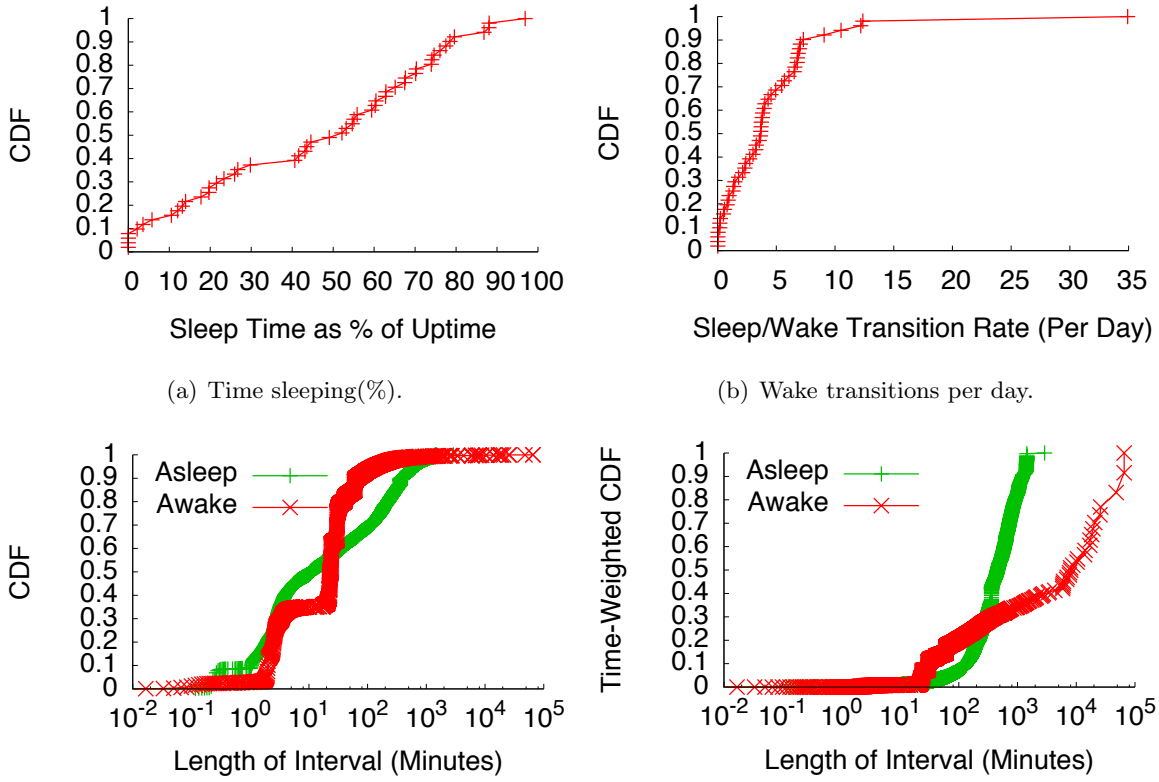
Of this traffic, 96% were multicast and broadcast packets. Of the multicast packets, 12.31% were ARP requests, which the sleep proxy examined and replied to as needed. The vast majority of the multicast traffic was safely ignorable [Nedevschi *et al.*, 2009]. The remaining 4% traffic was unicast: destined either to the proxy itself, or to the sleeping clients. 75% of these packets were wrapped by ESP and 8.4% were tunneled v6-over-v4 packets - underscoring the importance of parsing such packets. 7% of the total unicast packets were UDP (mostly IPsec related) and 3% were ICMP, which the sleep proxy ignores. Most of the remaining traffic was TCP, and the proxy was able to ignore the vast majority of it. During this time, we woke sleeping clients for just 747 TCP SYNs. Our analysis of the traffic data confirmed the importance of filtering TCP SYNs based on port. More than half of incoming TCP connection attempts were destined to ports on which the sleep client was not listening. If we had woken clients without filtering by port, we would have had more spurious wake-ups than valid ones.

3.7.2 Sleep/Wake Behavior

We note that five of our 51 clients did not sleep at all, as their their users manually disabled sleep functionality.

3.7.2.1 Aggregate sleep/wake behavior

Figure 3.5(a) shows the percentage of time each machine spent sleeping, as a CDF across all machines. The uniform slope of the CDF demonstrates that the average sleep time was quite variable, with 50% of the clients sleeping more than half the time. Figure 3.5(b) plots the CDF of the average number of sleep-to-wake transitions per day for the machines. Most machines average



(a) Time sleeping(%).

(b) Wake transitions per day.

(c) CDF of sleep/awake intervals.

(d) Time-weighted CDF.

Figure 3.5: Aggregate sleep/wake statistics.

fewer than seven daily wake-ups. Later, we will see that most of these wake-ups were caused by IT management traffic (e.g., updates) arriving for a sleeping machine.

We now examine the duration of sleep and awake intervals. Note that no sleep interval is longer than 1440 minutes because of the daily 5AM wake-up. The CDF of length of sleep and wake intervals is shown in Figure 3.5(c), while Figure 3.5(d) shows the time-weighted CDF (i.e., contribution of intervals at or below a given length to the total sleep or wake time). By comparing these two figures, we see that while most sleep and awake intervals are under one hour, the majority of both sleep and awake time comprises intervals over one hour. This implies that insomnia should be our first focus in attempting to reduce power usage (Section 3.7.3.2).

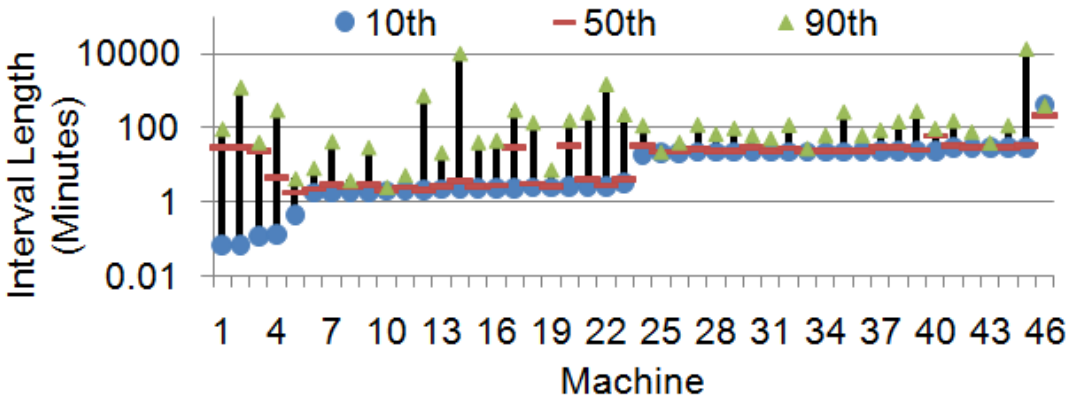
The awake interval CDF in Figure 3.5(c) demonstrates a bimodal distribution with abrupt changes in slope at around two minutes, and at 30 minutes. This indicates that awake periods of two and 30 minutes are prevalent in our trace.

3.7.2.2 Individual sleep/wake behavior

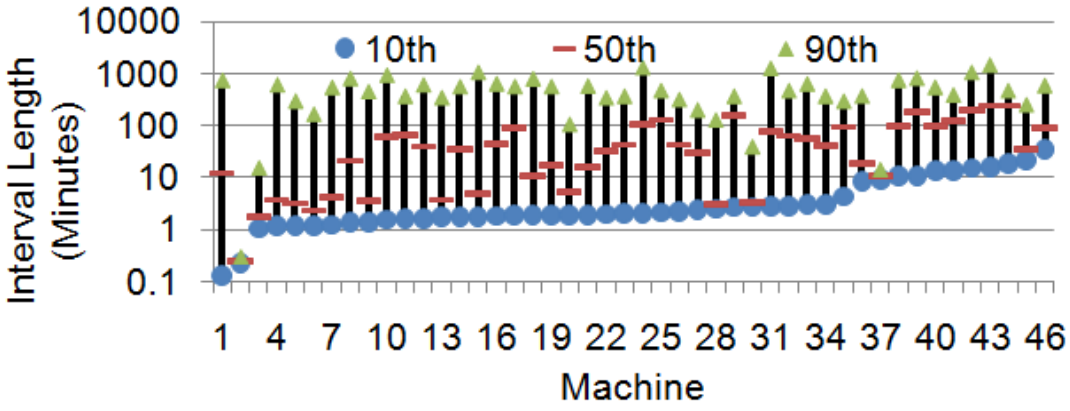
Figure. 3.6(b) and 3.6(a) show the 10th, 50th, and 90th percentile of wake and sleep intervals for each machine. The machines are sorted in order of 10th percentile. Notably, for around half of the machines the 10th percentile lies around two minutes, while for other half it lies around 30 minutes, corresponding to the jumps seen in Figure 3.5(c).

We closely inspected a number of these awake periods. The prevalence of both two and 30 minute awake periods is easily understood: these being the idle timeouts after WOL wake-up and user activity respectively. When looking at our special 5AM wake-up (which we know was not user-initiated - Section 3.4.4) we saw a much greater than normal proportion of two minute wakes which is precisely what we would expect.

Figure 3.6(b) shows that for about a quarter of the machines, the median sleep interval is under 10 minutes. For one machine all sleep intervals were under a minute. This machine appears to have some driver configuration issue that causes almost immediate wake upon sleep and was unique in our data set. Such intervals add very little to overall sleep duration and indicate potential sleep problems which will be examined further in Section 3.7.3.

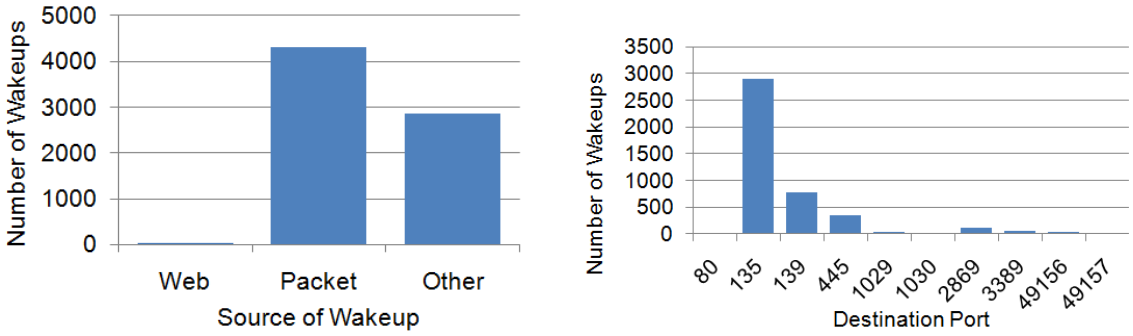


(a) Awake interval.



(b) Sleep interval.

Figure 3.6: Per-machine sleep/wake intervals.



(a) Cause of wake.

(b) Wake-up source port.

Figure 3.7: Wake cause.

3.7.2.3 Why do machines wake up?

Figure 3.7(a) shows the causes of wake-ups. We divide these into three categories: manual wake-up using our web site, wake-up by proxy due to incoming traffic, and other. The last bucket includes wake-ups caused by users walking up to the machine, any timer-based wake-ups caused by the BIOS, as well as occasional WOL packets sent by a commercial wake-up solution being tested by our IT department. We were able to confirm for 33% of these that the user did in fact initiate wake-up (by checking *lastUserInput* - Section 3.5.2) and for 50% of these the user definitively did not wake the machine. The remaining 27% could not be determined as *lastUserInput* was unavailable.

We see that while the web site was used in a few cases, it is not statistically significant. The majority of wake-ups caused by the sleep proxy are due to incoming TCP SYNs. The ports to which these SYNs were destined to are shown in Figure 3.7(b).

Remote Procedure Calls (port 135) were the overwhelmingly largest source of wake-up triggers, followed by NETBIOS (139) and SMB (445). SMB is the main mechanism used for remote file system access in our network. The two other notable ports are UPnP (2869) and Remote Desktop (3389). In our network, Remote Desktop is the primary mechanism for interactive remote machine access. We can see Remote Desktop is not a major wake-up source. In fact, only 39% of the machines were ever woken up due to Remote Desktop requests. Therefore, it would seem that while users leave their machines on for potential remote access, interactive remote access is used relatively rarely.

3.7.2.4 Who wakes up machines?

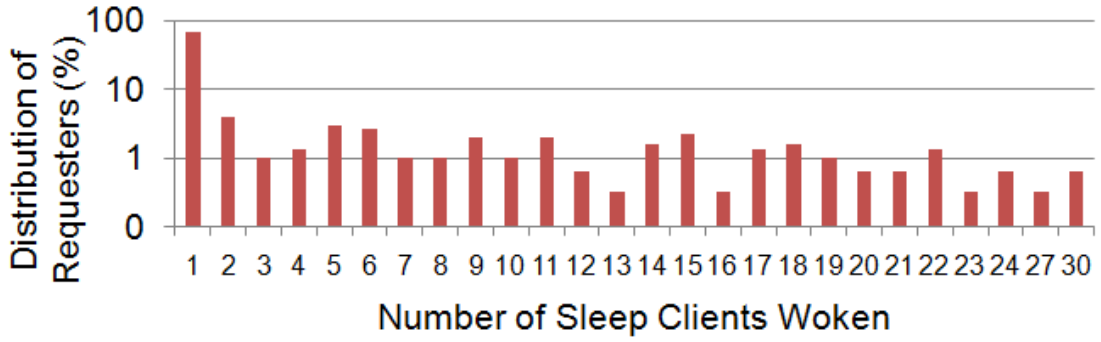
There were slightly over 300 IP addresses *requesters* whose incoming connection attempts caused wake-ups. Most of these only attempted to connect to a single sleep client. However, a sizable minority attempted to connect to multiple clients as seen in Figure 3.8(a). We were able to verify that all the requesters who woke 20 or more sleep clients were machines belonging to our IT department. These machines perform a variety of management actions such as verifying patch status and checking security policies. We will see later that our IT configuration is sleep-unfriendly in other ways as well (Section 3.7.3.2).

Figure 3.8(b) shows the number of wake-up events caused by requester. Just as most requesters only connect to a single machine, many only cause only one wake-up and most cause only a handful. However, again a large minority of requesters cause many wake-ups each. IT-owned machines again make a large portion of this group. Interestingly, several of the most active requesters actually connect to only one or a handful of machines. In fact, the most active requester with over 400 requests connected to only two machines, and that too in in a span of just two weeks! We are currently investigating the role of this requester further.

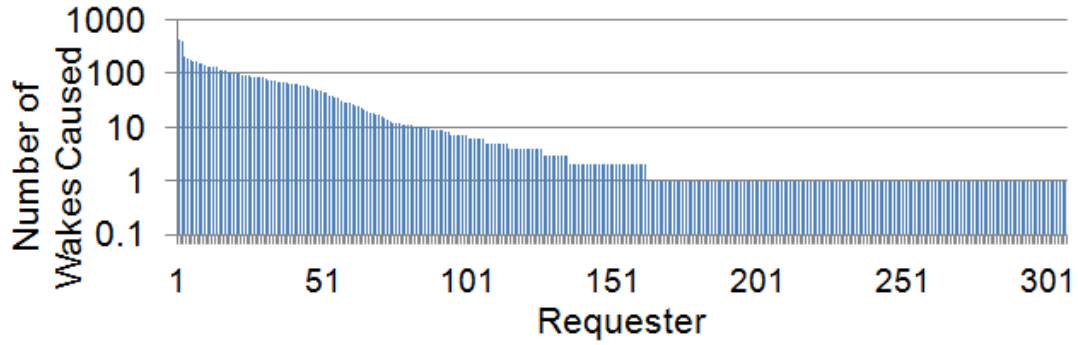
3.7.3 Why Machines Don't Sleep Better

While we have seen that our solution is fairly successful at enabling machines to sleep (Figure 3.5(a)), we wanted to investigate whether more idle time could be harvested. We begin by noting that most machines are not being woken overly often (Figure 3.5(b)). However, a small subset of machines suffer from "crying-baby-syndrome" being woken as soon as they fall asleep. Section 3.7.2.4. These machines are being bombarded by frequent connection attempts that interrupt their sleep often. If a machine with a standard 30 minute idle timeout wakes 12 times a day, one quarter of the day will have been spent awake due to wake-ups alone. It appears that configuration issues are responsible for much of this behavior.

However, most sleep clients are being kept awake for other reasons the majority of the time. In fact, when not being kept awake, these machines manage to sleep well, sustaining few wake-up events per day. We now consider whether these machines would have benefited from a more aggressive idle timeout, and then look at the problem of insomniac machines.



(a) Distribution of requesters by # clients woken.



(b) # Wake-ups caused By requester.

Figure 3.8: Who causes wake-ups?

3.7.3.1 Aggressive idle timeout

As mentioned in Section 3.7.2.2, it appears that setting the idle timeout more aggressively could result in some power savings. We now consider how much could be saved with a 5-minute idle timeout (this is 1/3rd the EnergyStar guidelines recommendation [Committee, 2009]).

To do so, we examined each wake interval to see why the machine was being kept up. Recall from Section 3.5.2 that a machine may be kept awake because the user is active, the machine has woken up recently, or a *stay-awake* request placed by a local application with the OS.

We divided the total awake time into three components, *recoverable*, *unknown*, and *unrecoverable*. *Recoverable* time was time in which the machine could have slept if the idle timeout had been set more aggressively. This time was the sum of periods in which the user had been active within the past five minutes or the machine had been woken within the past five minutes. The *unknown* time was the time for which insufficient data was available to diagnose cause of wakefulness. The

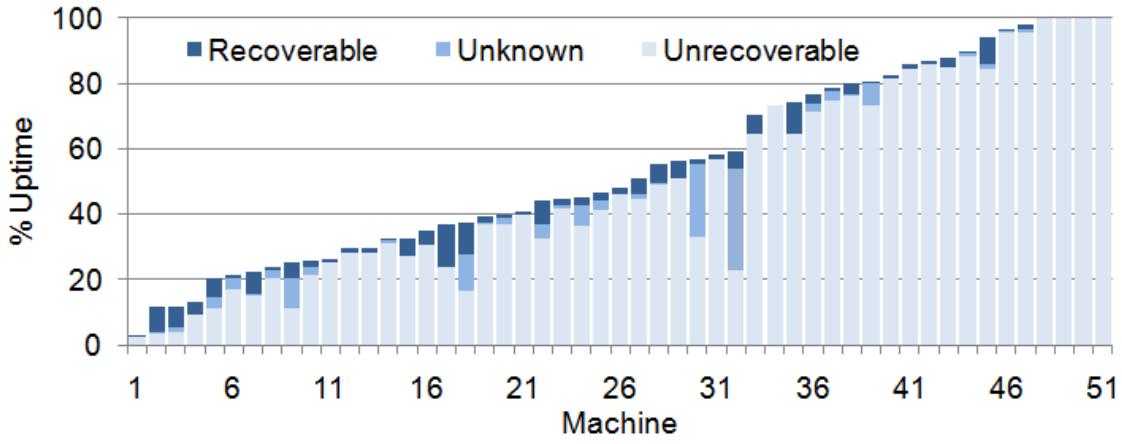


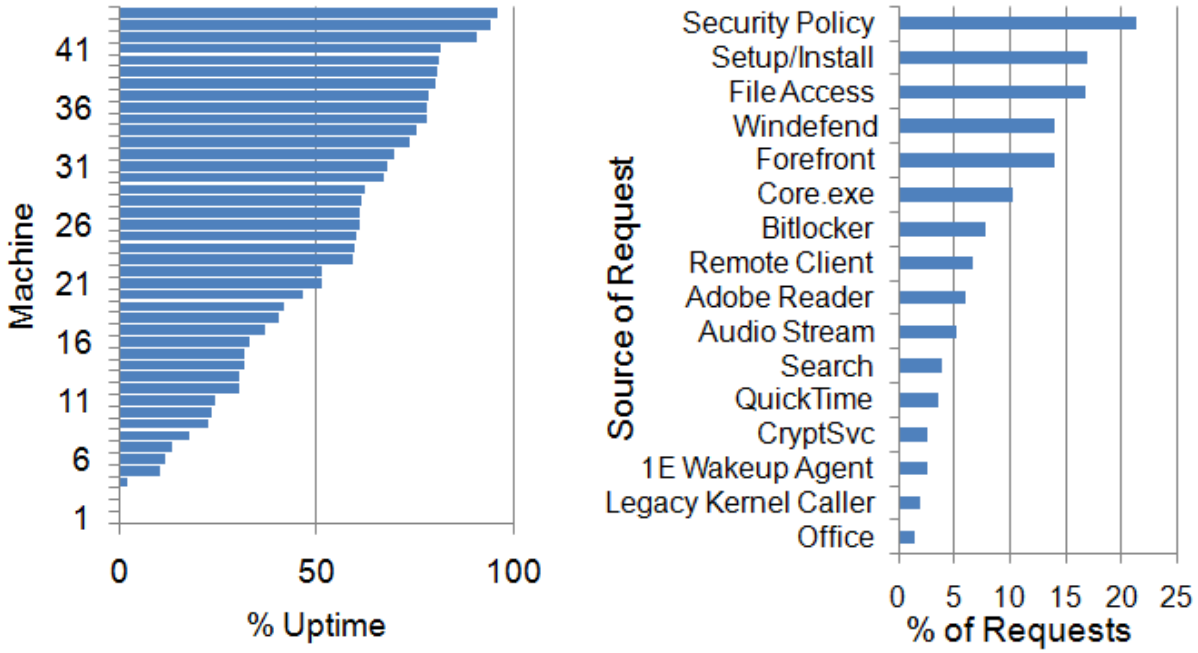
Figure 3.9: Awake time as % of uptime. Broken into components *unknown*, *recoverable*, and *unrecoverable* using aggressive idle timeout.

unrecoverable time consisted of all other time (i.e., an application had placed a *stay-awake* request with the OS).

Thus the recoverable time is a lower bound on the awake time that could have been saved by setting a more aggressive idle timeout. The sum of recoverable and unknown time provides the upper bound. Figure 3.9 breaks the total wake time as percentage of uptime into these three components on a per-machine basis. We see that on most machines the impact would be relatively small. These machines are being kept up by local application stay-awake requests, to which we now turn.

3.7.3.2 Insomnia

We now look more closely at which local applications keep machines awake. We label this phenomenon *insomnia*. Figure 3.10(a) plots the fraction of awake time a given machine was kept awake by local applications requesting OS to prevent sleep. We see that the majority of awake time is in fact due to such stay-awake requests. So which applications cause these stay-awake requests? Figure 3.10(b) shows the percentage of requests initiated by various applications. The news here is heartening. Four of the top sources (Security Policy Agent, Windefend, Forefront, and Bitlocker) are all applications mandated by our IT department. It may be possible to reconfigure or even re-write these applications to minimize and coordinate the duration of time they are active (and



(a) % awake time caused by requests.

(b) Source of requests.

Figure 3.10: Stay-awake request data.

thus preventing sleep). At least three more (Flash, Quicktime and Audio Stream) are the result of code or driver bugs. For example, certain older versions of Flash player may keep a machine awake by playing silence even after the audio clip has finished (Windows prevents sleep when audio streams are active). The third-highest request source is SMB. SMB's default behavior prevents a machine whose files are being accessed from sleeping. Careful changes to this behavior may allow for greater sleep opportunities.

3.7.4 Power savings

PC consumption varied, averaging from 89-143W for individual machines. The lowest draw we saw was 50W idling. The highest was 191W heavily loaded. While sleeping, all machines drew 1-2W. Monitors generally added from 30-60W when on.

Figure 3.11(a) illustrates the lower bound on power savings on a per machine basis. This lower bound is calculated with the assumption that had the machine stayed up instead of sleeping, it

Step	Time (s)	From→To	Packet Type
1	0	M1→M2	TCP SYN
2	0.04	S1→Broadcast	Magic Packet
3	2.48	M1→M2	TCP SYN
4	5.6	M2→Broadcast	ARP Probe
5	8.48	M1→M2	TCP SYN
6	8.49	M2→M1	TCP SYN-ACK

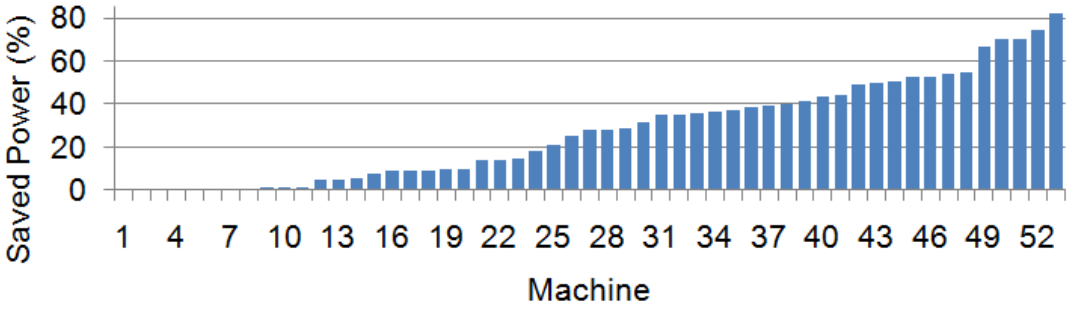
Table 3.1: Time line of a wake-up.

would have consumed power at the lowest rate seen in the entire non-sleeping portion of the trace. This represents part of the reason we saw less power savings than that predicted by previous work (which assumed machines consumed power at a constant rate irrespective of activity level). The average across all machines is about 20%, although variation is considerable.

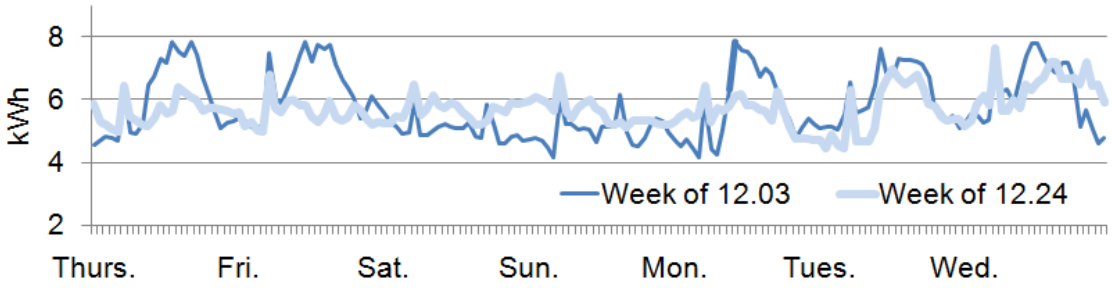
Figure 3.11(b), shows aggregate power consumption for a both a representative one-week period beginning 12/3/09 and the winter break (beginning 12/24/09). During the representative week, weekend power consumption is low, spiking only at the 5AM wake-up. During the work-week, power use peaks during the work day before declining into an overnight trough and bottoms out early on Friday. In contrast we can see a markedly different pattern for the Mid-Winter week with almost no increase in activity during the day from the day preceding Christmas (which fell on Friday) through the following Monday. By the Tuesday following the holiday, we begin to see a similar level of activity to that of the representative week, albeit at a lower amplitude, as employees begin returning from the holiday. Interestingly, the power consumption over the Christmas weekend (12/26-12/27) weekend was slightly higher than during a normal weekend (12/5-12/6).

3.7.5 Micro-Benchmarks

We now validate our architectural approach by examining wake-up delay time and sleep proxy scalability.



(a) Lower bound on per-Machine power savings



(b) Aggregate power draw for normal vs. mid-winter weeks.

Figure 3.11: Power draw and savings.

3.7.5.1 Wake-up delay

The energy saved by our system comes at a cost: the user experiences additional *startup* latency *the first time* a connection (e.g., ssh login or samba file access) to a sleep client is attempted since that client fell asleep. This happens because sleep client takes time to both wake and begin responding to an incoming TCP connection attempt. To make the system usable, we need to minimize the startup latency encountered by interactive transactions.

The user-perceived startup latency consists of several components: the delay involved in sending the WOL magic packet, the time required to wake up the machine, and the time required to perform any application-specific actions. To quantify these component latencies, we present a simple, but representative example.

Two machines, M1 and M2 were connected to the same subnet. M1 was ran a simple TCP sink, and was put to sleep. Thereafter, sleep proxy S1 started proxying for M1. From M2, we attempted to establish a TCP connection to the the sink on M1. The packet trace of the connection

establishment is summarized in Table 3.1.

The total latency is about 8.5 seconds, but the sleep proxy itself consumes only 40 milliseconds, even though it is on a busy subnet and proxying for several other machines. The largest component is the *wake-up* delay (i.e., time required for M2 to wake up and become active). This is roughly the delay between steps 2 and 4 (about 5.5 seconds). The remaining *TCP-retransmit* delay occurs between steps 4 and 5 (about 3 seconds). This delay is incurred while M1 waits to retransmit the TCP SYN the second time, following regular TCP timeout algorithm [Postel, 1981b].

Specific applications will usually encounter slightly higher latencies, as the machine needs to perform additional, application-specific actions. For example, when M1 tried to list a directory on M2 via SMB, the transaction took 13.37 seconds when M2 was asleep. The additional delay was incurred while M2 re-connected with the domain controller, and obtained security credentials to determine whether to allow M1 access.

We stress that this delay is incurred only for the transaction that wakes the machine. Subsequent transactions experience normal latencies. While our experience is that users do not mind this one-off penalty, both the wake-up and retransmit delays can be addressed. A number of research and engineering efforts are underway to address the former. The latter can be shortened either by having M1 retransmit TCP SYN more aggressively, or having S1 "replay" the TCP SYN.

3.7.5.2 Scalability

Our current deployment uses one sleep proxy per subnet. The load on these sleep proxies is a potential concern. We find that the CPU load on a sleep server rarely exceeds 5%. The total traffic (broadcast inclusive) seen by the sleep server is also quite low (90th percentile is 250 Kbps). We conclude sleep proxy operations do not require substantial resources, and a single sleep proxy could easily handle very large subnets if necessary. Conversely for reasonably sized subnets, the sleep proxy could be located on a client machine without noticeably degrading the user experience (Section 3.8).

3.7.6 Summary

Insomnia is the foremost cause of lost sleep. Thus improving the energy savings of systems like ours, the main focus should be on addressing sources of wakefulness.

IT applications are the main source of both insomnia and fitful sleeping. Several uncoordinated IT applications for patching, security, and network testing all woke machines and kept them awake. While we studied one particular IT setup, practically all IT setups will interfere with sleep to some extent - dependent on quantity, aggressiveness and degree of coordination of IT applications.

Misconfiguration can result in crying-baby syndrome Requiring administrators to diagnose and resolve the minority of machines suffering this issue.

Use of more aggressive idle timeouts is of secondary benefit. In enterprise systems behind firewalls, wake-ups will occur because of valid incoming TCP connection attempts and in well configured setups, the number of wake-ups caused by IT/misconfiguration will be minimal. Thus savings from more aggressive idle timeouts will be minor.

Incoming TCP connection attempts need to be filtered by listening port. More incoming TCP connection attempts arrived for non-listening ports, than listening ones.

3.8 Summary

We have designed and deployed a light-weight network-based sleep proxy in an operation enterprise network on over 50 user workstations - the first such deployment of which we are aware. During our work, we uncovered and addressed several practical issues that must be addressed by light-weight sleep proxying systems in enterprise networks. Our system has functioned both to user satisfaction and our own specification for the past several months, providing significant sleep opportunities and power savings using a simple reaction policy. However, we find that significantly more power savings could be achieved by altering the IT setup. Additionally, certain classes of cloud applications require specialized reaction policies. Should use of such persistent cloud applications become more widespread, our reaction policy would need adjustment.

This page intentionally left blank

Chapter 4

Wireless Computing: Supplementing Cellular Capacity

4.1 Overview

While at first slow and expensive, cellular data plans have become increasingly more attractive in terms of price and performance. With the introduction of devices well-designed to take advantage of this connectivity - most notably Research in Motion's Blackberry in the enterprise, followed several years later by Apple's iPhone - users have jumped on board *en masse*. Earlier adopters of cellular data networks had mostly limited their use to low-bandwidth applications such as email, weather forecasts, news updates, and light web-browsing. Now, encouraged by telecoms, user demand for high-bandwidth content like apps and video while on the move is increasing rapidly - content which had previously been demanded solely over their cable or ADSL network feed. Further this demand comes from not only smartphones, but a plethora of devices including tablets, netbooks, chromebooks, and traditional laptops. This demand has overwhelmed the capacity of carriers' 3G [Cheng, 2008b] and fledgling 4G networks [Lawson, 2011; Wortham, 2011], leading to outages, widespread user dissatisfaction, lawsuits [Cheng, 2008a], and even organized grassroots protest [Heussner, 2009]. Carriers are struggling to improve their networks quickly enough, but the required overhaul of centralized and expensive nationwide infrastructure takes years, if not decades, while shifts in the growth of demand are occurring at much shorter timescales [Arar, 2011].

Our work begins with the observation that not all demand is equal. Many of the most time-

critical interactions, *e.g.,* email, chat, news, are relatively low-bandwidth. As demonstrated by the popularity of slow but inexpensive multimedia delivery mechanisms such as BitTorrent [Sandvine, 2010] and Netflix DVD [Seetharam *et al.*, 2010], users are willing to wait to obtain high bandwidth multimedia content - at least so long as price is right. If low-latency, medium-bandwidth centralized cellular infrastructure capacity can be supplemented by some other medium-to-high-latency, high bandwidth mechanism, then user demand might be satisfied both more successfully and more economically than the current cellular-infrastructure only solution.

Fortuitously, the very mobility characteristics of users, combined with the increasingly large storage and local radios of the devices they carry, may provide the material for building just such a mechanism. A large pool of bandwidth lies untapped in the chance contacts of mobile devices. At each such contact, meeting nodes might exchange and replicate locally stored content at high data rates for very low cost (essentially just battery drain). However, utilizing these opportunities poses a significant challenge; device memory is finite and this latent bandwidth is often unstructured and unpredictable.

To address this challenge, we examine an alternative *opportunistic* content dissemination schemes for mobile devices. These schemes tap the potentially vast reservoir of capacity latent in currently unused communication opportunities between the short-range radios (*e.g.,* Bluetooth, 802.11) of smartphones. Leveraging short-range communication does not come without cost or complication. Particularly, users will need to tolerate both the *energy drain* from additional short-range radio use and the *fulfillment delay* encountered by nodes forced to wait until they meet peers who have the content needed to fulfill their requests. For now we assume that energy drain will be tolerable and focus on understanding how content can be disseminated so as to minimize the impact of fulfillment delay on overall user satisfaction. Once this is better understood, future work may return to rigorously address the issue of energy efficiency.

In order to understand the impact of delayed fulfillment on user satisfaction, we develop a model predicting fulfillment delay patterns and combine this with some monotonically decreasing *delay-utility* function mapping delay to utility. We then show how the aggregate expected utility can be calculated. This enables us to investigate how this quantity may be maximized (minimizing the impact of delay on our users) by manipulating local cache content. Additionally, one could utilize the aggregate expected utility to determine whether opportunistic content dissemination even makes

sense in a given scenario, by assessing whether it is above or below the system designer's chosen "break-even" point. Interestingly, our use of delay-utility functions enables us to answer these questions over the full spectrum of user *impatience* responses (*e.g.,* as delay increases, the user's likelihood of continuing to wait for content, decreases).

In the aforementioned model, the *allocation* of content in the *global distributed cache* comprising the union of all local caches, directly determines the pattern of demand fulfillment - and along with the delay utility function, the expected aggregate utility. By selectively replicating local content as node meetings provide the opportunity, the global cache can be driven towards a more efficient allocation.

As a motivating example, consider that an imaginary start-up *VideoForU* - having already noted that users are willing to use systems that require them to donate resources which provide them with delayed content at the right price (*e.g.,* Bittorrent) - decides to provide 15 minute video shows with embedded commercial content from a catalog of 500 available episodes (this catalog changes every so often - perhaps once a week). VideoForU manages to sign up 5000 users, who agree to dedicate a 3-episode cache on their local device's memory for use by VideoForU's protocol. VideoForU can now seed one or two copies of each episode into the global cache (by using cellular infrastructure, or base-stations run by VideoForU). They then let their protocol, running on the users devices, replicate content and fulfill user requests as chance meetings between users provide opportunity to do so. Assuming that the users's impatience is known (*i.e.,* the probability that a user, having waited time t, will not watch the content that she requested), via previous survey or feedback, VideoForU can design their replication protocol so as to maximize the total number of videos and embedded commercials, watched - the only question is how.

Making the answer to this question even more difficult is the fact that, for the same reasons as above (unpredictable mobility and resultant sporadic contacts), it may be difficult to gather global knowledge of the network's state. Consequently, we seek to develop distributed mechanisms capable of producing optimal or approximately optimal allocations, without needing to know the system's global state.

In this chapter we make the following contributions:

2. We demonstrate that user impatience plays a critical role in determining the optimal allocation for disseminating content. We further find a surprisingly general behavior which holds over a

wide variety of particular delay-utility functions: As the user population becomes increasingly impatient, the optimal allocation transitions steadily from uniformly dividing the global cache between all content items, towards a highly-skewed distribution in which popular items receive a disproportionate share of the global cache. We obtain these results by defining an optimal cache allocation in terms of delay-utility and global cache allocation. (Section 4.3)

3. Furthermore, we demonstrate that this optimal is unique and can be computed efficiently in a centralized manner. Under the simplified assumption of homogeneous meeting rates, we show that the corresponding optimal cache allocation is known in closed form for a general class of delay-utility functions. (Section 4.4)

4. Inspired by these results, we develop a reactive distributed algorithm, *Query Counting Replication (QCR)* that for any delay-utility function drives the global cache towards the optimal allocation. Moreover QCR does so without use of any explicit estimators or control channel information. (Section 4.5.1)

5. We show the implementation of QCR in opportunistic environments is non-trivial and demonstrate a novel technique *Mandate Routing* to avoid potential pathologies that arise in insufficiently fluid settings. (Section 4.5.3)

6. Finally, we validate our techniques on real-world contact traces, demonstrating the robustness of our analytic results in the face of heterogeneous meeting rates and bursty contacts. We find QCR compares favorably to a variety of heuristic competitors, despite those competitors having access to a *perfect control-channel* and QCR relying solely on locally available information. (Section 4.6)

4.2 Related Work

Networks that leverage local connection opportunities to communicate in a delay tolerant manner can be classified into two categories. The first category, featuring networks such as DieselNet [Balasubramanian *et al.*, 2007] or KioskNet [Seth *et al.*, 2006], involves nodes with scheduled or controlled routes, and routing protocols designed to communicate with predictable latency. [Chen *et al.*, 2006] extends the message ferrying paradigm to content dissemination in sparse MANETs. The second

category contains network featuring unpredictable mobility [Grossglauser and Tse, 2002; Chaintreau *et al.*, 2007] that may be used in an opportunistic manner. In this case, it is infeasible to provide strict guarantees on message delivery time. However, opportunistic contacts may greatly enhance the performance of many peer-to-peer (P2P) applications: as proposed for website prefetching in the 7DS architecture [Papadopouli and Schulzrinne, 2001], and podcast dissemination (series of content items on a channel), in the Podnet project [Lenders *et al.*, 2007]. It is into this second category that the content dissemination problem we investigate here falls. The performance of some of these systems have been analyzed from a hit-rate or delay standpoint [Lindemann and Waldhorst, 2005; Karlsson *et al.*, 2007] for the case of a persistent demand.

PodNet [Lenders *et al.*, 2007] is another opportunistic system that focuses on the dissemination of podcasts, or series of content items on a channel. Self Limiting Epidemic Forwarding (SLEF) [El Fawal *et al.*, 2007] aims at disseminating content in a limited space, and provides generalized TTL adaptation method.

Much previous work in the context of opportunistic networks has used utility functions as local states variables, both for unicast routing and publish-subscribe applications. The routing protocol PROPHET [Lindgren *et al.*, 2003] uses past information to predict delivery probability. The RAPID protocol generalizes this principle into an inference algorithm which accounts for several metrics related to delay [Balasubramanian *et al.*, 2007], while CAR [Musolesi and Mascolo, 2009] proposes the use of Kalman filtering to improve the prediction's accuracy. The impact of using different utility functions has been analyzed for single-copy routing schemes [Spyropoulos *et al.*, 2008], buffer management optimization [Krifa *et al.*, 2008], and the use of error-correcting code [Jain *et al.*, 2005]. In the context of pub-sub applications, utility functions were introduced to either predict user future demands [Sollazzo *et al.*, 2007], or leverage uneven distributions of demand and user proximity [Boldrini *et al.*, 2008; Costa *et al.*, 2008]. Other advanced cache management protocols includes utilizing filters [Greifenberg and Kutscher, 2008] and social relationships between mobile users in community [Yoneki *et al.*, 2007].

In general, local utility functions help a system to distinguish on-the-fly which intermediate node is the most likely to succeed (*i.e.,* for unicast routing, moving a packet closer to its destination, or, for pub-sub applications, facilitating dissemination to subscribing nodes). The complexity of these schemes makes performance analysis difficult. Moreover, the global effect of using local decisions

based on estimated utility often proves highly non-linear.

Our work significantly departs from previous approaches in two ways. The first is that instead of using (local) utility as an *intermediate* quantity used to estimate one or several parameters informing protocols, we take (global) utility as an *end-measure* for network efficiency (*i.e.,* the system's performance as it is perceived by users in aggregate). At no time during the course of the protocols is (local) utility estimated. Rather we study the effect of using light-weight replication protocols on the global utility of the network which the objective function we aim to maximize. The second difference is that we account for a general behavior of users with regard to delay, defining the global utility (or social welfare) as a function of any individually experienced delay-utilities (previous work either ignores user impatience or implicitly accounts for it using a fixed step function). A similar approach had been used for congestion control [Kunniyur and Srikant, 2003], and wireless scheduling [Liu *et al.*, 2003], but not so far for content dissemination in opportunistic networks.

Replication protocols were first introduced for unstructured P2P systems deployed on wired networks, as a way to increase data availability and hence to limit search traffic [Cohen and Shenker, 2002; Tewari and Kleinrock, 2006]. Assuming that nodes search for files in random peers, it was shown [Cohen and Shenker, 2002] that for each fulfilled request, creating replicas in the set of nodes used for the search (*i.e., path-replication*) achieves a square root allocation: a file i requested with probability p_i has a number of replicas proportional to $\sqrt{p_i}$ at equilibrium. This allocation was shown to lead to an optimal number of messages overall exchanged in the system. Assuming that nodes use an expanding ring search, an allocation where each file is replicated in proportion of its probability p_i was shown to be optimal [Tewari and Kleinrock, 2006].

[Hu *et al.*, 2009] presents an on-going effort to characterize a related channel selection problem. The algorithm proposed in this case uses an estimate of dissemination time and a Metropolis-Hasting adaptive scheme. One difference between the two approaches is that we show, because the optimal allocation satisfies a simple balance condition, that even simple algorithms which do no maintain any estimates of dissemination time or current cache allocation are optimal for a known delay-utility function. Another difference is that we also prove that the submodularity property for the cache allocation can be established even when contacts and delay-utility functions are not homogeneous.

4.3 Efficiency of P2P caching

Some nodes store content which they use to fulfill requests of the nodes they meet. In this section, we assume that the allocation of content to these nodes is fixed. We show that the global efficiency of such a system can be measured with an objective function parameterized by a delay-utility function representing the average user's impatience behavior.

4.3.1 Node Types, Content Cache

Each node in the P2P system may be a *client*, a *server*, or both. The set of client nodes is denoted by $\mathcal{C}$, we generally denote its size by N. Each client demands and consumes content as described in Section 4.3.3. The set of all server nodes is denoted by $\mathcal{S}$. Servers maintain a cache in order to make it available to interested clients (when such clients are met). This includes in particular the two following scenarios:

Dedicated nodes server and client populations are separate (*i.e.,* $\mathcal{C} \cap \mathcal{S} = \emptyset$).

Pure P2P all nodes act as both server and client (*i.e.,* $\mathcal{C} = \mathcal{S}$).

The dedicated node case resembles a managed P2P system, where delivery of content is assisted by special types of nodes (*e.g.,* buses or throwboxes [Balasubramanian *et al.*, 2007], kiosks [Seth *et al.*, 2006]). The pure P2P case denotes a cooperative setting where all nodes (*e.g.,* users' cellphones [Papadopouli and Schulzrinne, 2001; Lenders *et al.*, 2007]) request content as well as help deliver content to others. The motivating scenario, mentioned in the introduction, of VideoForU is likely to resemble the Pure P2P scenario, especially if as little content as possible is seeded with cellular infrastructure.

Caches in Server nodes The main variable of interest in the system is the cache content across all server nodes. In this section we assume it to be fixed; in practice the global cache dynamically evolves through a replication protocol (see section 4.5).

For any item i and m in $\mathcal{S}$, we define $x_{i,m}$ to be one if server node m possesses a copy of item i, and zero otherwise. The matrix $\mathbf{x} = (x_{i,m})_{i \in I, m \in \mathcal{S}}$ represents the state of the global distributed cache. We denote the total number of replicas of item i present in the system by $x_i = \sum_{m \in \mathcal{S}} x_{i,m}$.

In the remainder of this chapter, we assume that all servers have the same cache size so that they can contain up to ρ content items (all items are assumed to have the same size). This is not a critical assumption and most of the following results can be extended to caches or content items of differing sizes. It follows that a content allocation $\mathbf{x}$ in server nodes is feasible if and only if:

$$\forall m \in \mathcal{S}\,,\, \sum_{i \in I} x_{i,m} \leq \rho\,.$$

4.3.2 Representing Impatience as Delay-utility

In contrast with previous work in P2P networks, P2P content dissemination over an opportunistic mobile network induces a non-negligible *fulfillment delay* between the time a request is made by a client node and the time that it is fulfilled. This delay depends on the current cache allocation, as a request is fulfilled the next time the requesting node meets another node possessing a copy of the desired content. The term *impatience* refers to the phenomenon that users become decreasingly satisfied (or increasingly dissatisfied) with the delays they experience. A *delay-utility* function $h(t)$ can be used to characterize this phenomenon of user impatience in analytic terms, where the value of this function is monotonically decreasing with time (as increasing delay will not translate into increasing satisfaction).

Since different types of content may be subject to differing user expectations, we allow each content item i in the set of all system-wide content items available I, its own delay-utility function h_i. The value $h_i(t)$ denotes the gain for the network resulting from delayed fulfillment of a request for item i when this occurs t time units after the request was created. This value can be negative, which denotes that this delayed fulfillment generates a disutility, or a cost for the network. Note that t is related here to the user's waiting time, not to the time elapsed since the creation of the item. Currently, we decided to use the same set of delay-utility functions for all users. One can therefore interpret $h_i(t)$ as the average among users of the gain produced when a request is fulfilled after waiting for t time units. All the results we present generalize to users following different functions, but we choose to follow a simple average function to avoid notational issues, and to keep the system design simple.

We now present several examples of delay-utility functions corresponding to different perceptions of the performance of a P2P caching system by the users.

Advertising Revenue Assuming content items are videos starting with embedded advertisements, and that the network provider receives a constant unit revenue each time a commercial is watched by a user (a potential business plan for the scenario of VideoForU). In this case, the delay-utility function simply denotes the probability that a user watches a given video when she receives the content t time after it was requested. Two possible function families modeling this situation are:

Step function $h_\tau^{(\mathrm{s})} : t \mapsto \mathbb{I}_{\{t \leq \tau\}}$.

Exponential function $h_\nu^{(\mathrm{e})} : t \mapsto \exp(-\nu t)$.

The former models a case where all users stop being interested in seeing the item after waiting for the same amount of time. In the second case, the population of users is more mixed: at any time, a given fraction of users is susceptible to losing interest in the content.

Time-Critical Information Assuming the content exchanged by nodes deals with an emergency, or a classified advertisement for a highly demanded and rare product (*i.e.,* a well located apartment). In such cases, as opposed to the previous model the value of receiving this piece can start from a high value but very quickly diminish. It is possible to capture such a behavior by a delay-utility presenting a large reward for a prompt demand fulfillment.

Inverse power $h_\alpha^{(\mathrm{p})} : t \mapsto \dfrac{t^{1-\alpha}}{\alpha - 1}$. with $\alpha > 1$

Note that the value of delivering an item immediately in this case is arbitrarily large ($h(0^+) = \infty$). Such immediate delivery can occur when a node is both a server and a user, as the local cache may already contain the item requested. To exclude this case, we restrict the use of such delay-utility functions to the Dedicated node case.

Waiting Cost In some situations, such as a patch needed to use or update a particular application, users may request for an item and insist on receiving it no matter how long it takes, becoming with time increasing upset because of tardy fulfillment. As an example, the time a user spent with an outdated version of a software application may be related with the risk of being infected by a new virus, and hence incurring a high cost. One can consider to represent such cases a delay-utility

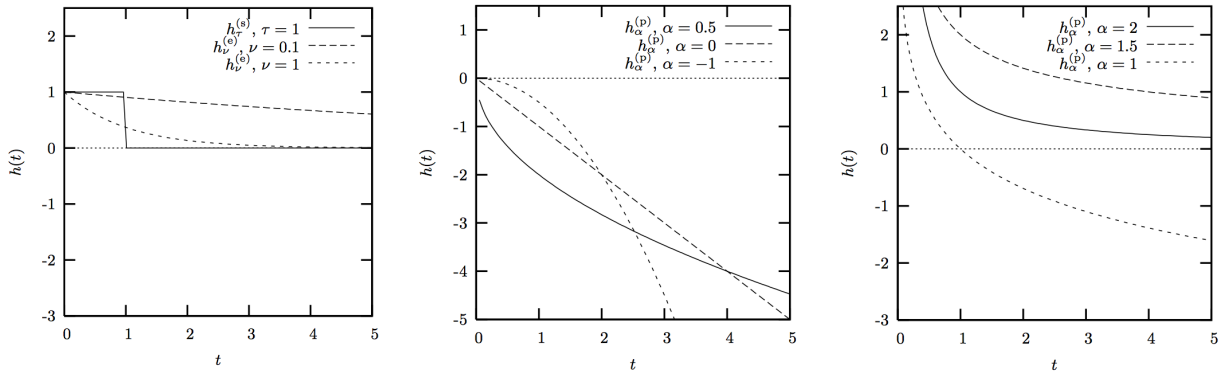
function that grows increasingly more negative with time, corresponding to a cost for the user and the network.

Negative power $h_\alpha^{(\mathrm{p})}$ as above with $\alpha < 1$

Negative Logarithm $h_1^{(\mathrm{p})} : t \mapsto -\ln(t)$.

The negative logarithm corresponds to the limit as α approaches 1. It features both a high value for fast fulfillment of request and a negative cost becoming unbounded as waiting time grows.

We plot on Figure 4.1 illustration of delay-utility functions for the three motivating examples presented above.



(a) Advertising Revenue (b) Time-Critical Information (c) Waiting Cost

Figure 4.1: Delay-utility functions used for advertising revenue (left), time-critical information (middle) and waiting cost (right).

To simplify the presentation below, we will assume in this chapter that h admits a finite limit at time $t = 0$, (*i.e.*, $h(0^+) < \infty$). This excludes the inverse power and the negative logarithm delay-utility functions introduced above. These functions can be considered in the dedicated node case where the exact same results hold, as shown in Appendix A.1.

4.3.3 Client Demand

Clients register their demand for content in the form of *requests*. As in previous work, we assume that the process of demand for different items follows different rates, reflecting differing content popularity. We denote by d_i the total rate of demand for item i. In the rest of this chapter, we assume any arbitrary values of d_i. As an example of demand distribution, one may use

Pareto with parameter $\omega > 0$: $d_i \propto i^{-\omega}$ for all $i \in I$.

In simulation we use a Pareto popularity distribution, generally considered as representative of content popularity.

We denote by $\pi_{i,n}$ the relative likeliness of a demand for item i arising at node n, where $\sum_{n \in \mathcal{C}} \pi_{i,n} = 1$. In other words, node n creates a new request for item i with a rate equal to $d_i \pi_{i,n}$. One can generally assume that different populations of nodes have different popularity profile, generally captured in the values of $\pi_{i,n}$. Otherwise, we can assume that items, especially the ones with the highest demand, are popular equally among all network nodes. This corresponds to the case where $\pi_{i,n} = 1/|\mathcal{C}|$.

4.3.4 Node Mobility

As all nodes (whether client or server) move in a given area, they occasionally meet other nodes - these meetings provide the opportunity for replication of cache content and fulfillment of outstanding requests. For simplicity and as a way to compare different P2P caching schemes, we focus on a case where contacts between clients and server nodes follow independent and memory less processes. In other words, we neglect the time dependence and correlation between meeting times of different pairs which may arise due to complex properties of mobility. In that case the process of contacts between two nodes m and n is entirely characterized by their contact intensity (the number of contacts between them per unit of time), which we denote by $\mu_{m,n}$.

Our model can be defined for any contact processes, this is what we simulate in Section 4.6 for a comparison using real traces. The memoryless assumption helps us to understand what are optimal strategies in a simple case before evaluating them using real traces for a complete validation of these trends. Two contact models can be considered:

Discrete time The system evolves in a synchronous manner, in a sequence of time slots with duration δ. For each time slot, we assume node contacts occur independently with probability $\mu_{m,n} \cdot \delta$ (for $m \in \mathcal{S}$, $n \in \mathcal{C}$).

Continuous time The system evolves in an asynchronous manner, so that events may occur in continuous time. We assume that node contacts occur according to a Poisson Process with rate $\mu_{m,n}$ (for $m \in \mathcal{S}$, $n \in \mathcal{C}$).

Note that when δ is small compared to any other time in the system, the discrete time model approaches the continuous time model. In this chapter, whenever space permits we write results for both contact model, focusing on the continuous case. Simulations results, which are based on discrete event processes, confirm the good match between our continuous time analysis and the discrete time dynamics of a real system.

The system is said to follow *homogeneous contacts* if we have $\mu_{m,n} = \mu$ for all nodes $m \in \mathcal{S}$ and $n \in \mathcal{C}$. This case corresponds to a population of nodes with similar characteristics where all meeting are equally likely, as for instance it may be between the participants of a special event.

4.3.5 Content allocation objective

Demand arises in our P2P system according to content popularity, and is served as a function of mobility and content availability, captured through variables $\mathbf{x} = (x_{i,m})_{i \in I, m \in \mathcal{S}}$.

We define $U_{i,n}(\mathbf{x})$ to be the expected gain generated by a request for item i created by client node n. Following our model of users' impatience, this expected gain is equal to $\mathbb{E}[h_i(Y)]$ where Y denotes the time needed to fulfill this request, which itself critically depends on the availability of item i in servers' caches.

The total utility perceived by all clients in the system, also called *social welfare*, may then be written as:

$$U(\mathbf{x}) = \sum_{i \in I} d_i \sum_{n \in \mathcal{C}} \pi_{i,n} U_{i,n}(\mathbf{x}) \,. \tag{4.1}$$

A good allocation $\mathbf{x}$ of content across the global cache is one that results in a high social welfare. Note that this objective combines the effects of delay on the gains perceived by users, the popularity of files, as well as the cache allocation.

In the remaining of this section, we derive an expression for $U_{i,n}(\mathbf{x})$, based on the *differential delay-utility function*, which will be instrumental in deriving some of its properties.

Differential delay-utility function We denote this function by c_i for the continuous time contact model (resp. Δc_i for the discrete time contact model). These functions are simply defined by

$$c_i(t) = -\frac{dh_i}{dt}(t) \,, \text{ and } \Delta c_i(k\delta) = h_i(k\delta) - h_i\left((k+1)\,\delta\right) \,.$$

The values of $c_i(t)$ and $\Delta c_i(k\delta)$ are always positive as h_i is a non-increasing function. The value of c_i (resp. Δc_i) represents the additional loss of utility, which is incurred per additional unit of time spent waiting (resp. the loss of utility incurred for waiting an additional time slot).

We present in the second line of Table 4.1 the expression for c_i for all the delay-utility functions introduced above. Note that when h_i is not differentiable (like for the step function), it may happen that c_i is not defined as a function but as the derivative measure in the sense of the distribution.

General expression for $U_{i,n}(\mathbf{x})$ Following a slight abuse of notation, we set by convention $x_{i,n} = 0$ when n is not a server node (*i.e.,* $n \notin \mathcal{S}$). With this notation, we find the following expressions for $U_{i,n}$.

Lemma 1 *In the discrete time contact model, $U_{i,n}(\mathbf{x})$ is*

$$h_i(\delta) - (1 - x_{i,n}) \sum_{k \geq 1} \prod_{m \in \mathcal{S}} \left(1 - x_{i,m}\mu_{m,n}\delta\right)^k c_i(k \cdot \delta)\,,$$

For the continuous time contact model, $U_{i,n}(\mathbf{x})$ is

$$h_i(0^+) - (1 - x_{i,n}) \int_0^\infty \exp\left(-t \sum_{m \in \mathcal{S}} x_{i,m}\mu_{m,n}\right) c_i(t)dt\,.$$

The proof follows from the memory less property of contacts and the expectation as obtained in integration by part:

$$\mathbb{E}\left[h(Y)\right] = h(0^+) + \int_0^\infty (1 - F_Y(t))h'(t)dh\,.$$

The term $(1 - x_{i,n})$ deals with possible immediate fulfillment (*i.e.,* request created by a node that already contains this item in its local cache). For more details, see Appendix A.1.

Homogeneous contact case If we assume homogeneous contacts (*i.e.,* $\mu_{m,n} = \mu$), the general expressions above simplifies. In particular, the utility depends on $(x_{i,n})_{i \in I, n \in \mathcal{S}}$ only via the number of copies present in the system for each item $(x_i)_{i \in I}$.

First, in the dedicated node case (*i.e.,* $\mathcal{S} \cap \mathcal{C} = \emptyset$), we have, respectively for the discrete time contact model and the continuous time contact model:

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(\delta) - \sum_{k \geq 1} (1 - \mu\delta)^{x_i k} c_i(k \cdot \delta)\right)\,. \tag{4.2}$$

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(0^+) - \int_0^\infty e^{-t\mu x_i} c_i(t) dt \right) . \tag{4.3}$$

Similarly, for the pure P2P case, if we further assume that all $N = |\mathcal{C}| = |\mathcal{S}|$ nodes follow the same item popularity profile (*i.e.,* $\pi_{i,n} = 1/N$), we have for the two different models of contact process:

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(\delta) - \left(1 - \frac{x_i}{N}\right) \sum_{k \geq 1} (1 - \mu\delta)^{x_i k} \, c_i(k \cdot \delta) \right) . \tag{4.4}$$

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(0^+) - \left(1 - \frac{x_i}{N}\right) \int_0^\infty e^{-t\mu x_i} c_i(t) dt \right) . \tag{4.5}$$

All these expressions follows from a simple application of Lemma 1 (see Appendix A.1 for complete details).

4.4 Optimal Cache Allocation

The *social welfare* defined above measures the efficiency of cache allocation which captures users' requests and impatience behavior. Finding the best cache allocation is then equivalent to solving the following optimization problem:

$$\max \left\{ \; U(\mathbf{x}) \; \middle| \; x_{i,n} \in \{0,1\} \, , \, \forall n \in \mathcal{S}, \sum_{i \in I} x_{i,n} \leq \rho \; \right\} . \tag{4.6}$$

4.4.1 Submodularity, Centralized computation

A function f that maps subset of $\mathcal{S}$ to a real number is said to be *sub-modular* if it satisfies the following property: $\forall A \subseteq B \subseteq \mathcal{S} \, , \, \forall m \in \mathcal{S} \, , \;\; f(A \cup \{m\}) - f(A) \geq f(B \cup \{m\}) - f(B) \, .$

This property generalizes to set functions the concavity property defined for continuous variables. Colloquially this is referred to as "diminishing returns" since the relative increase obtained when including new elements diminishes as the set grows.

The function $U_{i,n}(\mathbf{x})$ can be interpreted as a function that maps subset of $\mathcal{S}$ (*i.e.,* the subset of servers that possess a replica for item i) to a real number (the expected value of a request for item i created in client n). Similarly, U may be seen as a function that maps subset in $\mathcal{S} \times I$ (subsets denoting which servers possess which replica), to a real value (the social welfare). We then have the following result.

Theorem 1 *For any item i and node n, $U_{i,n}$ is submodular. As a consequence U is submodular.*

This result can be interpreted intuitively. On the one hand, in order to increase the value of $U_{i,n}$, creating a new copy of item i (*i.e.,* including a new element in the set of servers containing a copy of i) always reduce delays and hence increases utility. On the other hand the *relative* improvement obtained when creating this copy depends on the number of copies of i already present, and it diminishes as that item is more frequently found. What is perhaps less obvious is that this result holds for any mixed client/server population of nodes, heterogeneous contact processes, and any arbitrary popularity profile.

An interesting consequence is that one can deduce from submodularity, under some additional conditions, that a greedy procedure builds a $(1 - 1/e)$-approximation of the maximum social welfare for given capacity constraints (see [Nemhauser *et al.*, 1978]). A greedy algorithm is used in Section 4.6 to find a cache allocation for heterogeneous contact traces.

The proof of this result uses the general expression for $U_{i,n}$ found in Lemma 1 and a few observations: First, that the expression inside the integral multiplying the differential delay-utility function is a supermodular non-increasing and non-negative function of the set of servers containing i. Secondly, since the differential delay-utility function is positive, all these properties apply to the integral itself. Finally, that the product with $(1 - x_{i,n})$ preserves the supermodular non-increasing and non-negative properties. A complete formal proof can be found in Appendix A.1.

In the case of homogeneous contact rates, we can obtain an even stronger result, as the social welfare only depends on the number of replicas for each item, and not on the actual subset of nodes that possess that item.

Theorem 2 *In the homogeneous contact case, $U(\mathbf{x})$ is a concave function of $\{\ x_i \mid\ i \in I\ \}$.*

The optimal values of $\{\ x_i \in \{0, 1, \cdots, |\mathcal{S}|\} \mid\ i \in I\ \}$ are found by a greedy algorithm using at most $O(|I| + \rho|\mathcal{S}| \ln(|I|))$ computation s.

Moreover, the solution of the relaxed *social welfare maximization (*i.e., *maximum value of $U(\mathbf{x})$ when $(x_i)_{i\in I}$ are allowed to take real value) can be found by gradient descent algorithm.*

The concavity property is here not surprising, as it corresponds to submodularity when the function is defined using continuous variables rather than a set. Formally, the arguments used to prove this result are quite similar to the previous proof: one leverages previous expressions which

feature the product with the differential delay-utility function, and then use the fact that the family of convex non-negative non-increasing functions is closed under product.

The greedy algorithm follows a simple operation repeated once for each copy that can be cached ($\rho|\mathcal{S}|$ s in total): at each time step from the current cache allocation, it adds a copy for the item that brings the most significant relative increase in utility (assuming there does not exist already $|\mathcal{S}|$ copies of this item). By doing so, the algorithm is likely to select first popular items. As the popular items fill the cache with copies, the relative improvement obtained for each additional copy diminishes, and the greedy rule will choose to create copies for other less popular items. The diminishing return property ensures that this greedy algorithm selects the optimal cache allocation. For the same reason, starting from a cache allocation, a hill climbing algorithm with full knowledge can reach the optimal cache allocation only from local manipulation of cache between nodes that are currently meeting. A formal proof of these results can be found in Appendix A.1.

4.4.2 Characterizing the optimal allocation

In the homogeneous contact case, whenever x_i only takes integer values, it can be difficult to grasp a simple expression for the allocation maximizing social welfare, as it is subject to boundary and rounding effect. However, when the number of servers is large, x_i may take larger values, in particular for popular items. Hence, the difference between the optimal allocation and the solution of the *relaxed* optimization (where $(x_i)_{i\in I}$ may take real values, as defined in Theorem 2) tends to become small. The latter is then a good approximation of the former. In addition, when the number of clients N becomes large, the difference between the dedicated node case and the pure P2P case tends to become negligible, as the correcting terms $(1 - \frac{x_i}{N})$ in Eq.(4.4) and (4.5) approaches 1.

We show in this section that the solution of the relaxed optimization problem satisfies a simple equilibrium conditions. Although we only derive this condition in the continuous time contact model, a similar condition can be found in the discrete case model.

Property 1 *We consider the continuous time contact and dedicated node case. Let $\tilde{x}$ be the solution of the relaxed social welfare maximization (as defined in Theorem 2), then*

$$\forall i, j\,,\ \tilde{x}_i = |\mathcal{S}| \ \text{ or } \ \tilde{x}_j = |\mathcal{S}| \ \text{ or } \ d_i \cdot \varphi(\tilde{x}_i) = d_j \cdot \varphi(\tilde{x}_j)\,.$$

where we define φ as $\varphi : x \mapsto \displaystyle\int_0^\infty \mu t e^{-\mu t x} c(t) dt$.

This property states that, at the optimal solution of the relaxed cache allocation problem, the amount of copies created for all items depends on their popularity exactly in the same way: via a unique function φ defined *independently* of i. This equality holds only when the number of copies is not limited by the number of servers, otherwise it becomes an inequality.

This property follows from a simple derivation of the social welfare, as $\frac{\partial U}{\partial x_i}(\mathbf{x}) = d_i \varphi(x_i)$, which may be deduced from Eq.(4.3). At the optimal solution of the relaxed allocation problem, these derivatives should all be equal except for the value of x_i that are on the boundary of the domain (*i.e.,* when $x_i = |\mathcal{S}|$). If two points are in the interior and the derivative above differ, it is possible to modify $\tilde{x}$ slightly to remain under the capacity constraint and obtain an even larger social welfare, which would be absurd.

The function φ can always be defined a transform of the delay-utility function. For different choices of delay-utility, it leads to simple expressions which can be found in Table 4.1. As an example, when all items exhibit power delay-utility ($h_i = h_\alpha^{(\mathrm{p})}$), φ is a power function as well. The property implies then that, for all item i that are within the boundary conditions (*i.e.,* $x_i < |\mathcal{S}|$), the product $(\tilde{x}_i)^{2-\alpha} d_i$ is a constant that does not depend on i. We deduce that the optimal cache allocation for the relaxed problem resembles the distribution where $x_i \propto d_i^{1/(2-\alpha)}$, as shown in Figure 4.2.

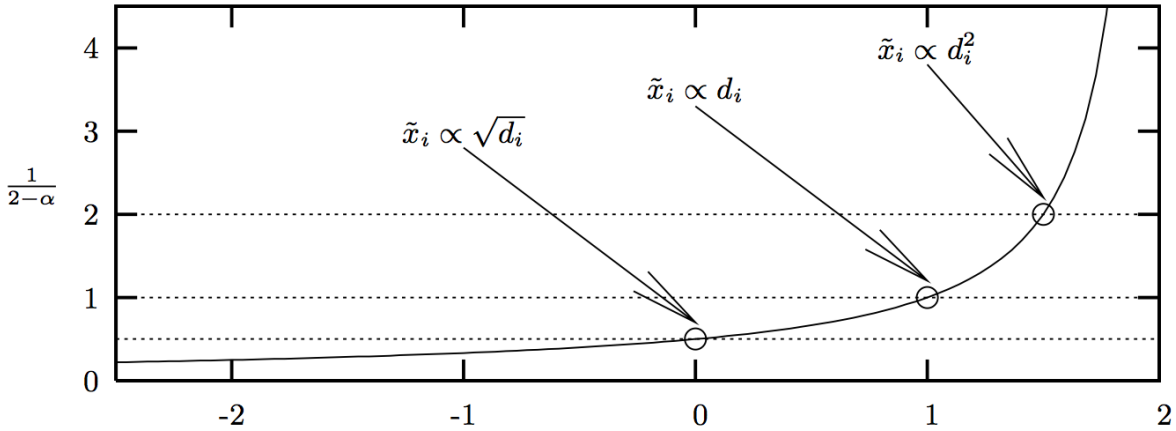


Figure 4.2: Coefficient of the optimal allocation for power delay-utility functions, as a function of α.

Note that, as α approaches 1 (*i.e.,* delay-utility is a negative logarithm), the social welfare is maximized when each item receives a fraction of cache proportional to its demand. For smaller

values of α (*i.e.,* waiting time cost) the optimal cache allocation becomes more egalitarian, it tends to uniform as α becomes arbitrary small. For larger values of α (*i.e.,* time-critical information), the optimal allocation becomes more and more skewed towards popular items (which are likely to give the best reward); as α approaches 2, the most demanded items completely dominate the cache. Similar qualitative observations hold for the step-function utility, the optimal allocation is more complex but again varies between these two extreme cases.

4.5 Distributed Optimal Schemes

The previous section establishes that the cache allocation problem admits an optimal operating point, which may in some cases be known in closed form, and can always be either computed directly or approximated in a centralized manner. When a highly available control channel is available, using such a centralized approach is feasible. However, making each local decisions based on global information maintained using this control channel seems to reach the optimal allocation only at a prohibitive cost.

In this section, we demonstrate that one does not need to maintain global information, or know the demand of items *a priori*, to approach the optimal cache allocation. This results in a drastic reduction of overhead and makes such caching service *possible* where no infrastructure is available.

We show that a simple reactive protocol, generalizing replication techniques introduced in the P2P literature, are able to approach the optimal allocation using only local information. In order to build a low-overhead reactive protocol for the opportunistic setting, two particular challenges need be overcome:

- We must understand how to construct a replication strategy that reacts naturally to the demand for and availability of content (Section 4.5.1), while also properly adapting our replication strategy to impatience of users (Section 4.5.2). A successful strategy will allow us to approach the optimal efficiency when the system reaches equilibrium.

- We must ensure that the replication technique is implemented in such a way that ensures the convergence towards the equilibrium. This challenge proves to be non-trivial in the opportunistic context for the strategies we examine (Section 4.5.3).

4.5.1 Query Counting Replication

We propose a general class of distributed schemes, that we call *Query Counting Replication (QCR)*. QCR *implicitly adapts* to the current allocation of data and collection of requests, without storing or sharing explicit estimators. QCR achieves this by keeping a *query count* for each new request made by the node. Whenever a request is fulfilled for a particular item, the final value of the query counter is used to regulate the number of new replicas made of that item. The function ψ that maps the value of the query counter to the amount of replicas produced is called the *reaction* function. We describe in Section 4.5.2 precisely how it should be set, given knowledge of user impatience.

As an example, consider a VideoForU client *Amy* who begins requesting a copy of video i. Each time Amy (or more precisely Amy's phone) subsequently meets another VideoForU node, Amy's phone queries the node met for a copy of item i and increments the query counter associated with i. If after nine meetings Amy's finally meets a node possessing a copy of item i and receives a copy of video i, according to the above rule, Amy's phone will create $\psi(9)$ replicas of this item and transmit them proactively to other nodes storing VideoForU content when the opportunity arise. This principle generalizes path replication [Cohen and Shenker, 2002] where $\psi(y)$ was a linear function of y.

Contacts between mobile nodes are unpredictable, hence, as Amy's phone distributes replicas, it may encounter nodes that already contain this item. We then distinguish two implementations: replication *without rewriting* where such contact is simply ignored, or replication *with rewriting* where such contacts decreases by one the number of replica to be distributed, even though no new copy can be made.

4.5.2 Tuning replication for optimal allocation

We now describe how to choose the reaction function ψ depending on users' impatience. We first observe that the expected value of the query counter for different item i is proportional to $|\mathcal{S}|/x_i$, since whenever a node is met there is roughly a probability $x_i/|\mathcal{S}|$ that it contains item i in its cache. Hence, we can assume as a first order approximation that approximately $\psi(|\mathcal{S}|/x_i)$ replicas are made for each request of that items. Inversely, as a consequence of random replacement in cache, each new replicas being produced for any items erases a replica for item i with probability $x_i/(\rho|\mathcal{S}|)$. When *rewriting* is allowed, one should account for all replicas created (including the one

created for the same item), we focus on this case for the analysis. Otherwise one should consider all replicas created for all other items. As a consequence, the number of copies for each item follows the system of differential equations:

$$\forall i \in I\,,\ \frac{dx_i}{dt} = d_i \cdot \psi(\frac{|\mathcal{S}|}{x_i}) - \frac{x_i}{\rho|\mathcal{S}|} \cdot \sum_{j \in I} d_j \psi(\frac{|\mathcal{S}|}{x_j})\,. \tag{4.7}$$

Assuming the system converges to a stable steady state, the creation of copies should compensate exactly for their deletion by replacement. In other words a stable solution of this equation satisfies

$$\forall i \in I\,,\ d_i \frac{1}{x_i} \cdot \psi(\frac{|\mathcal{S}|}{x_i}) = \frac{1}{\rho|\mathcal{S}|} \cdot \sum_{j \in I} d_j \psi(\frac{|\mathcal{S}|}{x_j})\,.$$

Note that the RHS is a constant that does not depend on i anymore, so that this implies

$$\forall i, j \in I\,,\ d_i \frac{1}{x_i} \cdot \psi(\frac{|\mathcal{S}|}{x_i}) = d_j \frac{1}{x_j} \cdot \psi(\frac{|\mathcal{S}|}{x_j})\,.$$

In other words, the steady state of this algorithm satisfies the equilibrium condition of Property 1 if and only if we have: $\forall x > 0\,,\ \frac{1}{x}\psi(\frac{|\mathcal{S}|}{x}) = \varphi(x)$ where φ is defined as in Property 1. Equivalently, $\forall y > 0\,,\ \psi(y) = \frac{|\mathcal{S}|}{y}\varphi(\frac{|\mathcal{S}|}{y})\,.$

Property 2 *The steady state of QCR satisfies the equilibrium condition of Property 1 if and only if*

$$\psi(y) \propto |\mathcal{S}|/y \int_0^\infty \mu t e^{-\mu \frac{t|\mathcal{S}|}{y}} c(t) dt\,.$$

The upshot of this result is that as long as the delay-utility function representing user impatience is known, we can always determine the number of copies QCR must make to drive the allocation towards its optimal. In particular, the optimal reaction function can be derived in a simple expression for all the delay utilities previously introduced, as seen in Table 4.1. This table was computed for the continuous time and dedicated node case. A similar table can be derived for the pure P2P case (see Appendix A.1). It is approximately equivalent to this one whenever the number of client nodes N is large.

Model	Step function	Exponential decay	Inv. Power ($\alpha < 1$)	Neg. Power ($1 < \alpha < 2$)	Neg. logarithm ($\alpha = 1$)
Impatience $h(t)$	$\mathbb{I}_{\{t \leq \tau\}}$	$\exp(-\nu t)$	$\frac{t^{1-\alpha}}{\alpha - 1}$		$-\ln(t)$.
Diff. Impat. c	Dirac at $t = \tau$	density $t \mapsto \nu \exp(-\nu t)$	density $t \mapsto t^{-\alpha}$		
Gain $U(\mathbf{x})$	$\sum_i d_i(1 - e^{-\mu\tau x_i})$	$\sum_i d_i(1 - \frac{1}{1+\frac{\mu}{\nu}x_i})$	$\sum_i d_i x_i^{\alpha-1} \frac{\mu^{\alpha-1}\Gamma(2-\alpha)}{\alpha-1}$		$\sum_i d_i \ln(x_i) - \texttt{cst}$
Cond. φ (Prop 1)	$d_i \cdot \mu\tau e^{-\mu\tau x_i}$	$d_i \cdot \frac{\mu}{\nu}\left(1 + \frac{\mu}{\nu}x_i\right)^{-2}$	$d_i \cdot x_i^{\alpha-2}\mu^{\alpha-1}\Gamma(2-\alpha)$		
Reaction ψ (Prop 2)	$(\mu\tau\lvert\mathcal{S}\rvert/y)\, e^{-\frac{\mu\tau\lvert\mathcal{S}\rvert}{y}}$	$\left(2 + \frac{\nu}{\mu\lvert\mathcal{S}\rvert}y + \frac{\mu\lvert\mathcal{S}\rvert}{\nu}\frac{1}{y}\right)^{-1}$	$y^{1-\alpha}\left(\mu^{\alpha-1}\lvert\mathcal{S}\rvert^{\alpha-1}\Gamma(2-\alpha)\right)$		

Table 4.1: Several delay-utility functions with associated equilibrium and reaction functions.

4.5.3 Mandate routing

Up to this point we have worked under the assumption that copies can be made more or less immediately, as in classical wired P2P networks. However, in an opportunistic context this is far from true. Particularly:

- Copies can only be made when another node is met, which happens only sporadically. Creating a replica may also takes additional time. For example, when rewriting is not allowed and the node met may already have a replica of that item.
- Since cache slots are overwritten randomly, it could be that, when a replica of the item needs to be produced, this item is no longer in the possession of the node desiring to replicate it.

Mandates & Pathologies Because replicas cannot be simply generated immediately, QCR mechanism deployed in an opportunistic context must inherently make (either implicitly or explicitly) a set of instructions for *future* replication of item i (*i.e.,* instructions to be used later, when the possibility for execution arises). We call such an instruction a *replication mandate* or *mandate* for short.

When a meeting occurs the mandate attempts to execute itself, but as we have already discussed, the circumstances may often not allow for its execution. This dependence of mandate execution on the state of the distributed cache may throw a monkey wrench in the dynamics outlined in Section 4.5.2 - for if the cache deviates to much from its expected state, the rates at which a given replica population evolves may be higher or lower than expected as well. As an example, if there are many fewer than expected copies of item i in the cache, and item i was erased by later random

replacement, item i may rarely be present again, so that mandates may not be executed soon in the future. An item i that, in contrast, is more frequently found, will execute its mandate more quickly and hence continue to dominate. Consequently, if mandates are simply left at their node of origin the allocation produced by any given run of QCR can diverge significantly from the target allocation, resulting in a loss of social welfare.

Our solution In order to address the above pathology, we need to ensure that the number of replication actions taken for each message is proportionally the same as the number of mandates produced for that message. This could be done in several ways, which all boil down to one of the following two approaches: (1) Move replicas to nodes with mandates for those replicas, or (2) Move mandates to nodes possessing the replicas which those mandates need in order to execute.

The former approach (*e.g.,* protecting items with current mandates from being erased by random replacement) violates the dynamics we are trying to protect and introduces significant implementation-level complexity - as we must now either replicate or protect against deletion particular messages based on locally existing mandates. While in practice these effects may be more or less severe, the second option of moving mandates to nodes with replicas provides us with a way of solving the problem *that involves no addition biasing of the overwrites, nor requires any adjustment to the mechanism of cache adjustment.* Additionally mandates are by nature quite small pieces of data, so moving them introduces little additional overhead in terms of communication and storage.

The mandate routing scheme used for the experiments shown in Section 4.6 is simple but can have significant impact as will be seen later. We assume that when two nodes meet, mandates are transferred with preference to the nodes possessing copies of the messages to be replicated. This ensures that most of the mandates that cannot be executed are soon transferred to appropriate nodes. Otherwise mandates are simply spread around - split evenly between the nodes. We demonstrate empirically that this simple modifications avoids divergence of QCR and is sufficient to converge towards an optimal point.

4.6 Validation

We now conduct an empirical study of different replications algorithms in a homogeneous contact setting, as well as several traces in various mobility scenarios. The goal of this study is threefold.

Firstly, to validate empirically that the rationale behind our distributed scheme does actually converge close to the optimal value we predict. Secondly, to observe quantitatively its improvement over simple heuristics. Thirdly, to test if the same scheme adapts well to contact heterogeneity present in real-world mobility traces, as well as complex time statistics and dependencies between contacts present in these.

4.6.1 Simulation settings

We have built a custom discrete-event, discrete-time simulator in C++ which given any input contact trace simulates demand arrival and the interactions of node meetings.

Data plots present below are the average of 15 or more trials with confidence interval corresponding to 5% and 95% percentiles. As said in Section 4.3.3 items are requested following Pareto distribution, here with parameter $\omega = 1$. By default we assume $\rho = 5$. Other values of ω and ρ can be found in Appendix A.2.

We do not consider the additional complexity of meeting durations. Instead we work on the premise that meetings are sufficiently long for nodes to complete the protocol exchange.

Implementation of QCR When two nodes meet they first exchange meta data. If either nodes have outstanding requests for messages to be found in the other's cache, then each of those requests is fulfilled. For each fulfillment a gain is recorded by the simulator, based on the age of this request and the delay-utility function. Nodes maintain query counters and makes a set of new mandates for each message fulfilled (as specified in Section 4.5.2). After fulfillment, the nodes then execute and route all of their eligible mandates (by sharing it equally if both nodes still possess a copy of the items, otherwise give it to the only node with a copy of this item). Rewriting of copy is not allowed, which means that contacts with a node already containing a copy of this item are simply ignored.

Each item i has one *sticky* replica which cannot be erased. This implementation detail has the effect of ensuring that we do not enter an absorbing state in which certain messages have been lost through discrete stochastic effects. We include them in mandate routing as preferred nodes (they will receive 2/3 of all mandates for this particular item whenever they meet a copy with this item, or all of them if the item has been erased on this node). We believe it is a reasonable

assumption for a fielded system, given that the initial seeder of a content item will likely keep that item permanently.

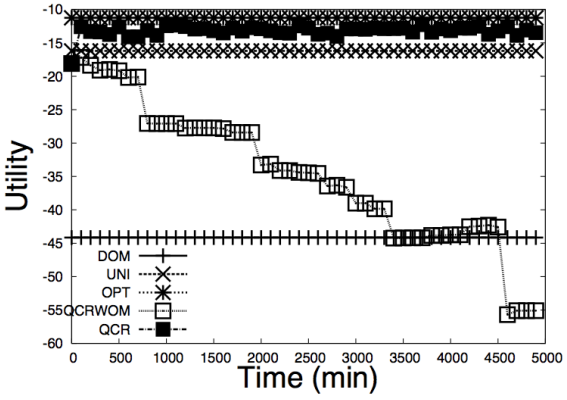
Competitor Algorithms We compare the performance of QCR against several heuristics using *perfect* control-channel information and the ability to set the cache *precisely and without restriction* to their desired allocation: **OPT** an approximation of the optimal obtained by a greedy algorithm optimizing Pb.(4.6). It is exactly optimal in the homogeneous case and approximately so in the heterogeneous ones; **UNI**: memory is evenly allocated amongst all items; **SQRT**: memory allocation proportionally to the square root of the demand; **PROP**: memory allocation proportional to demand; **DOM**: all nodes contain the ρ most popular items.

4.6.2 Homogeneous contacts

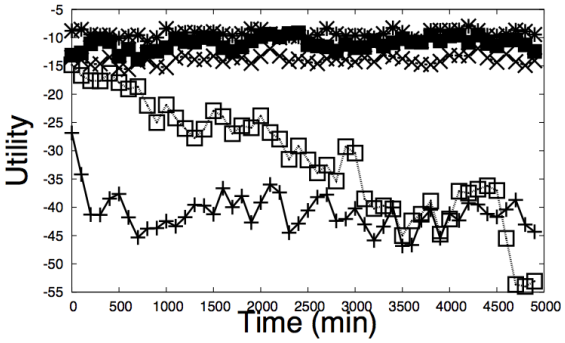
We simulate a network of 50 nodes with 50 content items ($I = 50$), meeting according to a rate $\mu = 0.05$ (the absolute value of μ plays no role in the comparison between different replication algorithm). As we wish to validate our analysis is not a mere artifact of the constraints used to generate it, we focus on the pure P2P case ($|C| = |S| = N = 50$), which is the furthest from the analysis we conducted. We tune the reaction function ψ according to Table 4.1.

QCR with and without mandate routing Figure 4.3 illustrates the need to implement mandate routing in query based replication. It was obtained for the power function with $\alpha = 0$. This result is representative of all comparison where mandate routing was turned on and off. As the time of the simulation evolves we see that the utility (as estimated in expectation on (a), and observed from real fulfillment in (b)) dramatically decreases with time when QCR does not implement mandate routing. Further investigations have shown that simultaneously the amount of mandate diverges for item less frequently requested. We see on (d), where the number of replicas is shown for the five most requested items, that QCR without mandate routing systematically overestimates their share and sometimes. In contrast, the number of replicas with mandate routing fluctuates around the targeted value, and QCR quickly converges and stay near optimal utility.

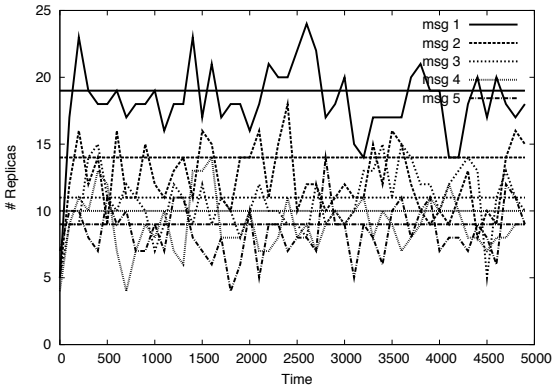
Comparison with fixed allocations Figure 4.4 presents the utility obtained with the both QCR and the competitor algorithms described previously. For each algorithm, we plot in the y-



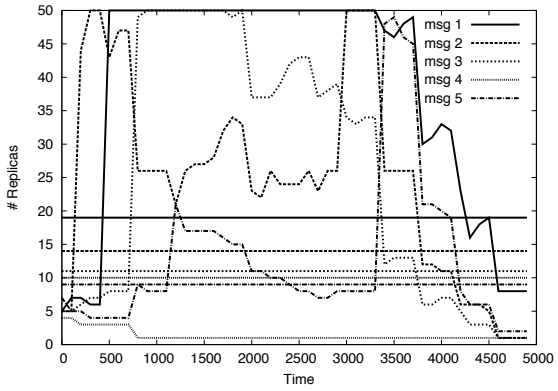
(a) Expected Utility



(b) Observed Util.



(c) Cache: Mandate Routing



(d) no Mandate Routing

Figure 4.3: Effect of mandate routing (homogenous contacts, power delay-utility function with $\alpha = 0$).

axis $(U - U_{\text{opt}})/|U_{\text{opt}}|$ where U is the utility obtained on average during the simulation by this algorithm and U_{opt} is the value obtained with the optimal allocation. Hence the plotted quantity is always negative (since as we expect no algorithm outperforms OPT). Value $y = -1$ corresponds to a utility 1% smaller than the optimal social welfare. Due to large variation of this quantity over the space and algorithms investigated, we used a logarithmic scale in the y-axis to present these results. For each algorithm, we consider two models of delay-utility (power and step function) with different parameters, varied along the x-axis.

We observe that for both delay-utility functions, the extreme strategies (*i.e.,* UNI and DOM) fail to approach the optimal in general. In particular it is the case for small value of α, when users are

sensitive to waiting delay and the decrease in social welfare can be high, and small value of τ where quick response is essential. While demand aware offline strategies (*i.e.,* PROP and SQRT) perform similarly to QCR, QCR does not require control-channel information to achieve this performance. We even observe that QCR outperforms PROP in many cases, sometimes very significantly. Across all heuristic competitors, QCR does not incur a loss of utility beyond 5% (for step function) and 60% in the worst case of power function. One unexpected result is that the square root allocation performs reasonably well in most cases studied, however this is an ideal performance observed when the allocation is fixed with *a priori* knowledge. In contrast, proportional allocation leads to much worse performance, in particular for power delay-utility function. Proportional allocation resembles a passive demand based replication where a fixed number of replicas (*e.g.,* one replica) are created whenever a request is fulfilled (as found in [Lenders *et al.*, 2007] and many other works). These results illustrate that such passive replication simply gives too much weight to popular items, and that compensating for this effect is both necessary and achievable.

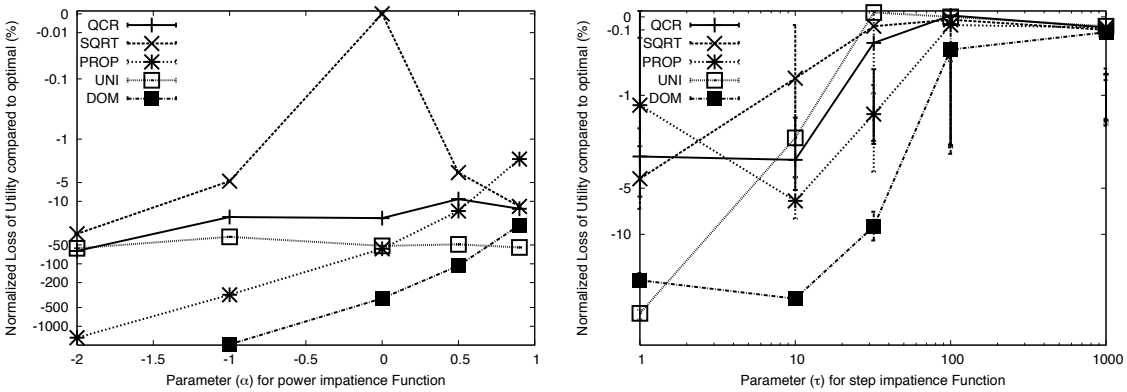


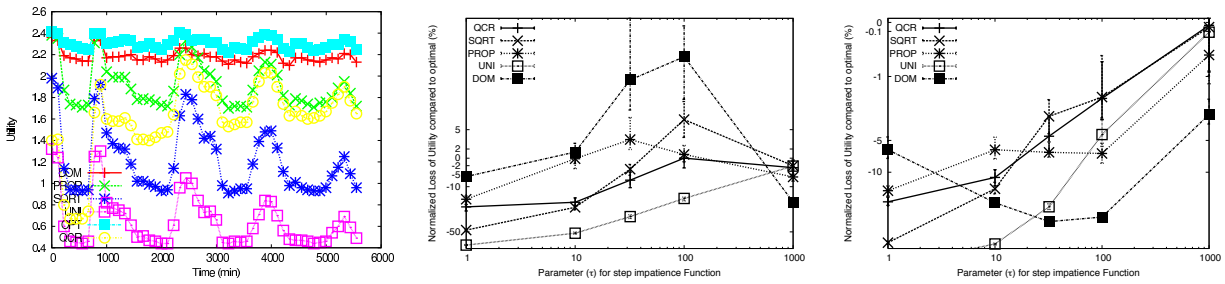
Figure 4.4: Comparison between QCR and several fixed allocations (homogeneous contacts): for power delay-utility function as a function of α (left), for step delay-utility function as a function of τ (right).

4.6.3 Real Contact Traces

We now abandon the homogeneous mixing assumption needed for our analysis and look at the performance of QCR on real-world contact traces to see if the spirit that our analysis still applies under more realistic mobility. As in the homogeneous experiments, we use $I = 50$ and $N = 50$ for evaluation of our techniques on both heterogeneous traces.

Conference scenario We use the Infocom '06 data set which measures Bluetooth sightings between 73 participants at the Infocom conference (see [Chaintreau *et al.*, 2007] for more details) over the course of thee days. To remove bias from poorly connected nodes, we selected the contacts for the 50 participants (numbered from 21 to 71 in the original data sets) with the longest measurement periods.

Figure 4.5 (a) presents the utility as seen over time (time averaged over an hour) for the competitor set and QCR (with mandate routing). We clearly observe the alternation of daytime and nighttime during the trace. Here, unlike in the homogeneous scenario, DOM and PROP perform the best. QCR performs very close to the latter, despite heterogeneity and complex time statistics. SQRT and UNI perform poorly until *tau* becomes quite large - as the delay requirement is too stringent to allow significant improvement on non-popular items that would offset the loss created by shifting the focus off popular content.



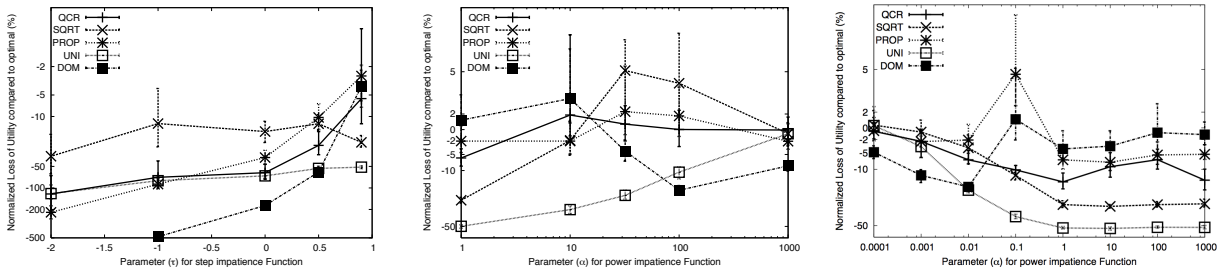
(a) evolution with time ($\tau = 1$) (b) vs. τ (actual trace) (c) vs. τ, (synthesized trace)

Figure 4.5: Utility for Infocom '06 dataset and step function model of impatience.

Figure 4.5 (b) and (c) presents the relative loss of utility for different algorithms (compared with OPT) as a function of τ. We separate the impact of heterogeneity *per se* by presenting the actual traces and a synthetic trace where contact rates of all pairs are identical but contacts are assumed to follow memoryless time statistics. Heterogeneity *per se* does not seem to greatly impact the performance of QCR. Indeed it appears QCR may even perform better under contact complexity, perhaps because its implicit reaction to content availability adapts well to heterogeneous cases. The most notable difference with the homogeneous case is that SQRT is not a clear winner anymore and that PROP and DOM seem relatively stronger. The results from actual traces show that time statistics greatly impact the performance of a fixed allocation. First, since OPT was

computed under the approximation of memory less contact, some competitors actually perform slightly outperform OPT on occasion. We also observe that the DOM greatly improves due to bursty statistics. However, the performance of QCR remains quite comparable, generally lying within 15% of OPT.

Vehicular networks We use contacts recorded between 50 taxicabs selected from the Cabspotting project contact traces. The data sets was extracted from a day of data and assumed that taxicabs are in contacts whenever they are less than 200m apart (see [Chaintreau *et al.*, 2009] for more details). Results, shown in terms of performance relative to OPT, may be found in Figure 4.6 (a) (b) (c). Again, we observe that OPT, which is based on a memoryless assumption, can be outperformed by some allocation (as in (b) for the step function case). Just as for the Infocom data set, we see that SQRT tends to produce degraded performance, while DOM improves as heterogeneity and complex time statistics are included in the contact trace. The performance of QCR, the only scheme based on local information, appears less affected by this change.



(a) vs. α (Power fn) (b) vs. τ (Step fn) (c) vs. ν (Exponential fn)

Figure 4.6: Comparison between QCR and several fixed allocations (Cabspotting dataset using actual traces): for power delay-utility function as a function of α (left), for step delay-utility function as a function of τ (middle), for exponential delay-utility function as a function of ν (right).

4.7 Summary

Our results focus on a specific feature which makes P2P caching in opportunistic network unique: users' impatience. From a theoretical standpoint, we have shown that optimality is affected by

impatience but can be computed and moreover satisfies an equilibrium condition. From a practical standpoint, we have seen that it directly affects which replication algorithm should be used by a P2P cache. Passive replication, ending in proportional allocation, can sometimes perform very badly, but one can tune an adaptive replication scheme to approach the performance of the optimal, based only on local information.

This page intentionally left blank

Chapter 5

Conclusions

In this dissertation, we have sought to find solutions for problems arising through changed mobility of user, devices, and software, that require **minimal modification** to existing designs We have considered concrete scenarios from sub-disciplines spanning *wireless networks*, *green computing*, and *cloud computing*. In each scenario, we design a solution that aims to strike an advantageous balance between adoptability and technical efficiency. We find that, not only this be done, but that adoptable and effective solutions *implemented solely in software* and *deployed only on network end-hosts* may be devised. The network itself and its protocols need not be changed at all.

We design and implement of a system (vmTorrent) capable of quickly and scalably distributing and executing VM images in the cloud. vmTorrent's custom front-end file server allows for VM quick-start in which VMs can execute while their images traverse the network. To this is attached a P2P back-end supporting efficient scaling. Finally, vmTorrent's intelligent pre-fetching algorithms enable smooth streaming execution: balancing the needs of local execution (image pieces should be prefetched based on anticipated need) and swarm efficiency (prefetching may need to be randomized to ensure piece diversity sufficient to fully exploit swarm upload capacity).

To evaluate this approach, we implement a functional vmTorrent prototype. We measure our prototype's performance on a variety VMs and short VDI tasks. For these we deploy our prototype on a hardware testbed, using up to 100 physical client peers. Our experimental results demonstrate our design choices produce a scalable system. vmTorrent delivers up to an **11X improvement** over a state-of-the-art P2P approaches and up to a **30X improvement** over demand-based streaming approach. In fact, the vmTorrent runtimes remain comparable to those

of local disk execution for all workload sizes. VMTORRENT provides such an effective solution to this problem that it is currently in the process of commercialization.

To save energy in enterprise networks, we architect and implement a non-intrusive, economical, network-based sleep-proxying system. Each desktop machine runs a lightweight daemon that, immediately preceding transition to sleep states, informs a sleep proxy sitting on the local network. The network sleep proxy then redirects and monitors traffic incoming to the sleeping host, waking the host as appropriate.

We roll out the first substantial deployment of any sleep-proxying system in a corporate environment. We deploy our software on over 50 user machines in six subnets. Almost all of these machines are primary user workstations. We measure the performance of our system, collecting over half-a-year's worth of data. We instrument our system extensively; capturing numerous details about sleep and wake-up periods, data which explains why machines wake up and *why they stay up*. Instead of using generic estimates of PC power consumption, we use a sophisticated *software*-based, model-driven system, *Joulemeter*, to estimate power draw. Additionally, we describe a number of practical issues we encountered when deploying a light-weight sleep proxy in a corporate network, providing new insight into how such systems may most effectively be deployed.

This work appears to have struck the right balance between technical efficiency and adoptability. Our sleep-proxying system has been in continual use since it was first rolled out two years ago, and continues to attract more users.

Finally in the wireless domain, we model how an opportunistic content distribution mechanism might function, focusing on user *impatience*: the function describing the decreasing utility users find as the wait for fulfillment of their demand increases. We then show how that under certain conditions the optimal memory usage policy can not only be described, but also approximated using a lightweight distributed mechanism. Moreover the information required to determine what content should be replicated by any pair of meeting nodes is entirely local, given knowledge of the impatience curve.

We validate these techniques on real-world contact traces, demonstrating the robustness of our analytic results in the face of heterogeneous meeting rates and bursty contacts. We find QCR compares favorably to a variety of heuristic competitors, despite those competitors having access to a *perfect control-channel* and QCR relying solely on locally available information.

Our work was the first in this area to take a first-principles approach that defines and maximizes a global objective function. Likewise, our work was the first to consider how users might respond to such a system. Since we published, less than two years ago, a stream of at least 15 publications from other researchers have cited our approach.

5.1 Future Directions

The trend towards increased mobility of, within, and between computerized systems appears to be accelerating. The rapid adoption of OS-level virtualization means that more and more computational machines can, and likely will, hop from device to device. We are currently seeing both the virtualization of the network itself and of an increasing range of devices that have traditionally been collocated. Finally the hardware in which computation resides becomes ever more ubiquitous, miniaturized and mobile - fully functional networked computers may masquerade as watches or sit in children's toys . Where this all ends is still a very open question, but we see abundant new challenges and opportunities arising in the years ahead, as we figure out how to repurpose existing technologies and build entirely new ones.

We now briefly outline some directions for future research related to the material covered in this dissertation.

Improved prefetching for vmTorrent: our current prefetching mechanisms are quite simple. In the future, we will look to explore improved techniques that determine prefetch order based on current execution pattern, more nuanced statistical profiling methods, and swarm awareness (*e.g.,* being able to adjust the degree of random re-ordering introduced into the prefetch requests based on current swarming efficiency).

On the fly compression: whereby image pieces are transferred in a compressed state, only being decompressed as needed. Not only will piece compression save network bandwidth, but if done correctly, it could both reduce metadata and save space in memory.

Cross-image swarming: by dealing with VM images at the file-system, instead of block, level, content overlap between similar VMs might be exploited.

VMs built on demand: Taking this same concept even further, VMs are already much more disposable than the bare-metal systems they replace. Might VMs be created on-the-fly from com-

ponents taken from pre-existing VMs already on hand, allowing for almost infinite flexibility? vmTorrent provides much of the technology infrastructure needed to do this, although significant challenges remain on both the networking (*e.g.,* solving the multi-swarm sharing problem) and systems ends (*e.g.,* determining which image components can be safely combined and how to do so).

Computer systems built on demand: As various components are virtualized might devices themselves become more fluidly defined, adding or deleting components as they move from one location to another? Already, Apple has built "Remote Disc" facilities directly into Mac OS X, allowing one machine to use another networked machine's physical drive, as if it were directly physically attached. Likewise, NAS designed for the home have gained widespread commercial traction. One might reasonably imagine a future in which devices incorporate additional CPU, RAM, monitors, etc., seamlessly on-the-fly, based on current execution requirements and geographic availability.

Multi-device VMs: Each individual could conceivably have a VM that hops from smartphone to tablet, to cloud, to car, and back again. Looked at another way, VM technology may lead to OSes that span multiple devices, migrating and activating functionality as needed.

Network-aware device OSes: In the nearer term, there appear to be opportunities to improve the underlying function of mobile devices to provide a more seamless network experience to the user. For example, most mobile browsers today will clear cached webpages and attempt to reload these pages, even though no connectivity is available. Several techniques, including smart virtualized network interfaces might be used to solve these types of problems.

Plug-and-play opportunistic mechanisms: Network-aware device functionality might be parlayed directly into truly plug-and-play opportunistic content exchange - providing a standardized API on which such applications might be built.

P2P sleep proxy: This type of functionality might also support more advanced messaging-based power control. Our current sleep-proxying architecture requires the use of a dedicated (albeit low-power) sleep proxy machine on each subnet. This could be replaced with a p2p architecture in which machines fall asleep one after the other, while the "last man standing" keeps watch for the entire subnet. Such a setup might even be extended to mobile devices, potentially extending the battery lives of smartphones, tablets, and similar devices.

IT application coordination and configuration: Currently, IT maintenance tasks are uncoordinated and consequently will keep machines awake during each of their separate execution time periods. Devising methodologies that schedule these tasks to overlap as much as possible can significantly increase sleep opportunities.

This page intentionally left blank

Bibliography

[acpi,] Advanced Configuration and Power Interfae, revison 4.0. http://www.acpi.info/.

[Adaptiva Technologies,] Adaptive Technologies. http://www.adaptiva.com/.

[Agarwal *et al.*, 2009] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *NSDI'09*, Berkeley, CA, USA, 2009.

[Agarwal *et al.*, 2010] Yuvraj Agarwal, Stefan Savage, and Rajesh Gupta. Sleepserver: A software-only approach for reducing the energy consumption of pcs within enterprise environments. In *USENIX ATC*, 2010.

[Al-Fares *et al.*, 2008] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[Albanna *et al.*, 2001] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper. RFC 3171: IANA Guidelines for IPv4 Multicast Address Assignments, August 2001.

[Allman *et al.*, 2007] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future internet through selectively connected end systems. In *Hotnets*. ACM SIGCOMM, Nov. 2007.

[amt,] Intel Active Management Technology (AMT). http://www.intel.com/technology/platform-technology/intel-amt/.

[apple-wol,] Apple Wake On Lan. http://www.macworld.com/article/142468/2009/08/wake_on_demand.html.

[Arar, 2011] Yardena Arar. Why wireless carriers both promote and dread 4g. `http://technologizer.com/2011/03/23/why-wireless-carriers-both-promote-and-dread-4g/`, March 2011.

[Balasubramanian *et al.*, 2007] Aruna Balasubramanian, Brian Levine, and Arun Venkataramani. DTN Routing as a Resource Allocation Problem. In *Proc. ACM SIGCOMM*, August 2007.

[Barroso and Hölzle, 2007] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007.

[bindfs,] bindfs. Bindfs: Mount a Directory to Another Location and Alter Permission Bits. `http://code.google.com/p/bindfs/`.

[Bircher and John, 2007] W. Lloyd Bircher and Lizy K. John. Complete system power estimation: A trickle-down approach based on performance events. In *ISPASS*, 2007.

[Blackburn and Christensen, 2009] J. Blackburn and K. Christensen. A simulation study of a new green bittorrent. In *Workshop on Green Communications*. IEEE Internation Conference on Communications, June 2009.

[Boldrini *et al.*, 2008] Chiara Boldrini, Marco Conti, and Andrea Passarella. Contentplace: social-aware data dissemination in opportunistic networks. In *Proc. ACM MSWiM*, 2008.

[Boyd and Vandenberghe, 2004] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[Chabarek *et al.*, 2008] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. Power awareness in network design and routing. In *INFOCOM 2008*, 2008.

[Chaintreau *et al.*, 2007] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, J. Scott, and R. Gass. Impact of human mobility on opportunistic forwarding algorithms. *IEEE Trans. Mob. Comp.*, 6(6):606–620, 2007.

[Chaintreau *et al.*, 2009] A. Chaintreau, J.-Y. Le Boudec, and N. Ristanovic. The age of gossip: spatial mean field regime. In *Proc. of ACM SIGMETRICS*, 2009.

[Chandra *et al.*, 2005] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A Cache-Based System Management Architecture. In *NSDI*, pages 259–272, Berkeley, CA, USA, 2005. USENIX Association.

[Chen *et al.*, 2006] Yang Chen, Jeonghwa Yang, Wenrui Zhao, M. Ammar, and E. Zegura. Multicasting in sparse manets using message ferrying. In *WCNC*, volume 2, pages 691–696. IEEE, April 2006.

[Chen *et al.*, 2009] Zhijia Chen, Yang Zhao, Xin Miao, Ying Chen, and Qingbo Wang. Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In *ICDCS Workshops*, pages 324–329, Washington, DC, USA, 2009. IEEE Computer Society.

[Cheng, 2008a] Jacqui Cheng. Apple hit with class-action lawsuit over iphone 3g flakiness. `http://arstechnica.com/apple/news/2008/08/apple-hit-with-class-action-lawsuit-over-3g-iphone-flakiness.ars`, August 2008.

[Cheng, 2008b] Jacqui Cheng. One month of the iphone 3g: what apple needs to fix. `http://arstechnica.com/apple/news/2008/08/one-month-of-the-iphone-3g-what-apple-needs-to-fix.ars`, August 2008.

[Cheshire, 2008] S. Cheshire. RFC 5227: IPv4 Address conflict detection., July 2008.

[Christensen and Gulledge, 1998] Kenneth J. Christensen and Franklin 'Bo' Gulledge. Enabling power management for network-attached computers. *Int. J. Netw. Manag.*, 8(2):120–130, 1998.

[Christensen *et al.*, 2004] K. Christensen, P. Gunaratne, B. Nordman, and A. George. The next frontier for communications networks: Power management. *Computer Communications*, 27(18):1758–1770, Dec. 2004.

[Cohen and Shenker, 2002] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):177–190, 2002.

[Cohen, 2003] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, May 2003.

[Committee, 2009] TC32-TG21 Committee. www.ecma-international.org/publications/files/drafts/tc32-tg21-2009-150.doc. Technical report, ECMA, Nov. 2009.

[Costa *et al.*, 2008] P. Costa, C. Mascolo, M. Musolesi, and G.P. Picco. Socially-aware routing for publish-subscribe in delay-tolerant mobile ad hoc networks. *IEEE Jsac*, 26(5):748–760, June 2008.

[Das *et al.*, 2010] Tathagata Das, Pradeep Padala, Venkat Padmanabhan, Ram Ramjee, and Kang G. Shin. Litegreen: Saving energy in networked desktops using virtualization. In *USENIX ATC*, 2010.

[Deering and Hinden, 1998] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, Dec 1998.

[eforcast,] Worldwide pc market. `http://www.tgdaily.com/slideshows/index.php?s=200903062p=1`.

[El Fawal *et al.*, 2007] Alaeddine El Fawal, Jean-Yves Le Boudec, and Kave Salamatian. Multi-hop Broadcast from Theory to Reality: Practical Design for Ad Hoc Networks. In *Proc. Autonomics*, 2007.

[El-Sayed *et al.*, 2003] A. El-Sayed, V. Roca, and L. Mathy. A survey of proposals for an alternative group communication service. *Network, IEEE*, 17(1):46 – 51, 2003.

[Fan *et al.*, 2007] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2007.

[Greifenberg and Kutscher, 2008] Janico Greifenberg and Dirk Kutscher. Efficient publish/subscribe-based multicast for opportunistic networking with self-organized resource utilization. In *Proc. AINAW*, 2008.

[Grossglauser and Tse, 2002] M. Grossglauser and D. Tse. Mobility increases the capacity of ad hoc wireless networks. *IEEE/ACM Trans. on Net.*, 10(4):477–486, 2002.

[gumstix,] Gumstix. `www.gumstix.com`.

[Gupta and Singh, 2003] Maruti Gupta and Suresh Singh. Greening of the internet. In *SIGCOMM*, New York, NY, USA, 2003.

[Heussner, 2009] Ki Mae Heussner. 'operation chokehold': Fake steve jobs rallies iphone users to cripple at&t network. `http://abcnews.go.com/Technology/GadgetGuide/fake-steve-jobs-rallies-iphone-users-cripple-att/story?id=9355447`, December 2009.

[Hinden and Deering, 2006] R. Hinden and S. Deering. RFC 4291: IP Version 6 Addressing Architecture, February 2006.

[Hoskins, 2006] Matthew E. Hoskins. Sshfs: super easy file access over ssh. *Linux Journal*, 2006, June 2006.

[Hosseini *et al.*, 2007] M. Hosseini, D.T. Ahmed, S. Shirmohammadi, and N.D. Georganas. A survey of application-layer multicast protocols. *Communications Surveys Tutorials, IEEE*, 9(3):58 –74, 2007.

[Hu *et al.*, 2009] Liang Hu, Jean-Yves Le Boudec, and Milan Vojnovic. Optimal channel choice for collaborative ad-hoc dissemination. Technical Report MSR-TR-2009-26, MSR, 2009.

[Huang *et al.*, 2008] Y. Huang, Tom Z. J. Fu, D. M. Chiu, J. C. S. Lui, and Cheng Huang. Challenges, Design and Analysis of a Large-scale P2P-VoD System. In *ACM SIGCOMM*, Seattle, WA, USA, August 2008.

[ICANN, 2011] ICANN. Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied. `www.icann.org/en/news/releases/release-03feb11-en.pdf`, February 2011.

[idc,] Idc netbook and pc sales projections. `http://www.tgdaily.com/slideshows/index.php?s=200903062p=1`.

[Jain *et al.*, 2005] S Jain, M Demmer, R Patra, and K Fall. Using redundancy to cope with failures in a delay tolerant network. *ACM SIGCOMM Computer Communication Review*, Jan 2005.

[Jim, 1998] Jim. Re: (IPng 5136) Congrats to IBM - IPv6 support shipping in AIX 3.3. `http://dict.regex.info/ipv6/6bone/6bone.mail-1998-01/0024.html`, Jan 1998.

[Jimeno *et al.*, 2008] M. Jimeno, K. Christensen, and B. Nordman. A network connection proxy to enable hosts to sleep and save energy. In *Performance Computing and Communications Conference*, pages 101–110. IEEE, Dec. 2008.

[Kansal and Zhao, 2008] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics08*, June 2008.

[Karlsson *et al.*, 2007] Gunnar Karlsson, Vincent Lenders, and Martin May. Delay-tolerant broadcasting. *IEEE Transactions on Broadcasting*, 53:369–381, 2007.

[Kent and Seo, 2005] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol., Dec. 2005.

[Kivity *et al.*, 2007] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux Virtual Machine Monitor. In *Ottawa Linux Symposium*, 2007.

[Kozuch and Satyanarayanan, 2002] Michael Kozuch and M. Satyanarayanan. Internet Suspend/Resume. pages 40–46. IEEE Computer Society, 2002.

[Krifa *et al.*, 2008] A. Krifa, C. Barakat, and T. Spyropoulos. Optimal buffer management policies for delay tolerant networks. *Proc. of IEEE SECON*, 2008.

[Kunniyur and Srikant, 2003] Srisankar Kunniyur and R. Srikant. End-to-end congestion control schemes: utility functions, random losses and ecn marks. *IEEE/ACM Trans. Netw.*, 11(5):689–702, 2003.

[Lagar-Cavilla *et al.*, 2009] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *EuroSys*, pages 1–12, New York, NY, USA, 2009. ACM.

[Lawson, 2011] Stephen Lawson. Lte outage irks users, dents verizon's reputation. `http://www.pcworld.com/businesscenter/article/226664/lte_outage_irks_users_dents_verizons_reputation.html`, April 2011.

[Lenders *et al.*, 2007] Vincent Lenders, Martin May, and Gunnar Karlsson. Wireless ad hoc podcasting. In *Proc. IEEE SECON*, 2007.

[libtorrent,] libtorrent. libtorrent: C++ Bittorrent Library. `http://www.rasterbar.com/products/libtorrent/index.html`.

[Lindemann and Waldhorst, 2005] Christoph Lindemann and Oliver P. Waldhorst. Modeling epidemic information dissemination on mobile devices with finite buffers. In *Proc. ACM Sigmetrics*, 2005.

[Lindgren *et al.*, 2003] A Lindgren, A Doria, and O Schelen. Probabilistic routing in intermittently connected networks. *SIGMOBILE Mobile Computing and Communication Review*, 7(3), 2003.

[Liu *et al.*, 2003] P. Liu, R. Berry, and ML Honig. Delay-sensitive packet scheduling in wireless networks. *Proc. of WCNC 2003*, 3, 2003.

[Mahadevan *et al.*, 2009] P Mahadevan, P Sharma, S Banerjee, and P Ranganathan. Energy aware network operations. In *Global Internet Symposium*. IEEE, April 2009.

[Maisto, 2009] Michelle Maisto. Global pc market suffering first decline since dot com crash. `http://www.eweek.com/c/a/Desktops-and-Notebooks/Global-PC-Market-Suffering-First-Decline-Since-DotCom-Crash-385323`, September 2009.

[Marciniak *et al.*, 2008] Pawel Marciniak, Nikitas Liogkas, Arnaud Legout, and Eddie Kohler. Small is not always beautiful. In *IPTPS'08: Proceedings of the 7th international conference on Peer-to-peer systems*, Tampa Bay, FL, February 2008.

[Mietzner and Leymann, 2008] Ralph Mietzner and Frank Leymann. Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. In *IEEE SERVICES*, pages 3–10, Washington, DC, USA, 2008. IEEE Computer Society.

[MokaFive,] MokaFive. `http://www.mokafive.com/products/components.php`.

[Musolesi and Mascolo, 2009] M Musolesi and C Mascolo. Car: Context-aware adaptive routing for delay-tolerant mobile networks. *IEEE Transactions on Mobile Computing*, 2009.

[ndis-pattern,] Adding and Deleting Wake on LAN Patterns. `http://msdn.microsoft.com/en-us/library/ff543710.aspx`.

[Nedevschi *et al.*, 2008] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI'08*, Berkeley, CA, USA, 2008.

[Nedevschi *et al.*, 2009] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: Reducing energy waste in networked systems. In *NSDI*, April 2009.

[Nemhauser *et al.*, 1978] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of the approximations for maximizing submodular set functions. *Mathematical Programming*, 14, 1978.

[O'Donnell, 2008] Chris M. O'Donnell. Using BitTorrent to Distribute Virtual Machine Images for Classes. In *SIGUCCS*, pages 287–290, 2008.

[Papadopouli and Schulzrinne, 2001] Maria Papadopouli and Henning Schulzrinne. Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices. In *Proc. ACM MobiHoc*, 2001.

[Postel, 1981a] J. Postel. RFC 791: Internet Protocol: DARPA Internet Program Protocol Specification, September 1981.

[Postel, 1981b] J. Postel. RFC 793: Tranmission control protocol., September 1981.

[Qiu and Srikant, 2004] Dongyu Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-like peer-to-peer Networks. In *SIGCOMM*, pages 367–378, 2004.

[Rivoire *et al.*, 2008] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. In *HotPower'08*, 2008.

[Roth, 2008] Michel Roth. Using BitTorrent to Solve Omnipresent Deployment Problems. *Thin-Computing.net*, 2008.

[rwt,] Intel Remote Wakeup Technology (RWT). `http://www.intel.com/technology/chipset/remotewake-qa.htm?iid=Tech_remotewake+qa`.

[Sandvine, 2010] Sandvine. `http://www.sandvine.com/downloads/documents/2010%20Global%20Internet%20Phenomena%20Report.pdf`, 2010.

[Schmidt *et al.*, 2010] Matthias Schmidt, Niels Fallenbeck, Matthew Smith, and Bernd Freisleben. Efficient distribution of virtual machines for cloud computing. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 0:567–574, 2010.

[Seetharam *et al.*, 2010] Anand Seetharam, Manikandan Somasundaram, Jim Kurose, Don Towsley, and Prashant Shenoy. Shipping to streaming: Is this shift green? In *SIGCOMM Workshop on Green Networking*, August 2010.

[Seth *et al.*, 2006] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav. Low-cost communication for rural internet kiosks using mechanical backhaul. In *Proc. ACM MobiCom*, 2006.

[Shi *et al.*, 2008] Lei Shi, Mohammad Banikazemi, and Qing Bo Wang. Iceberg: An image streamer for space and time efficient provisioning of virtual machines. In *ICPP - Workshops*, pages 31–38, Washington, DC, USA, 2008. IEEE Computer Society.

[Sinha and Chandrakasan, 2001] Amit Sinha and Anantha P. Chandrakasan. Energy efficient real-time scheduling. *Computer-Aided Design, International Conference on*, 0:458, 2001.

[Smith and Nair, 2005] J. E. Smith and R Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[Snowdon *et al.*, 2009] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for os-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, 2009.

[Society, 2011] Internet Society. World IPv6 Day. `http://www.worldipv6day.org/`, June 2011.

[Sollazzo *et al.*, 2007] Giuseppe Sollazzo, Mirco Musolesi, and Cecilia Mascolo. Taco-dtn: a time-aware content-based dissemination system for DTN. In *Proc. ACM MobiOpp*, 2007.

[Spyropoulos *et al.*, 2008] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi Raghavendra. Efficient routing in intermittently connected mobile networks: the single-copy case. *IEEE/ACM Trans. on Netw.*, 16(1), Feb 2008.

[Stuckmann and Zimmermann, 2009] P. Stuckmann and R. Zimmermann. European research on future internet design. *Wireless Communications, IEEE*, 16, October 2009.

[Szeredi,] Miklos Szeredi. FUSE: File System in Userspace `http://fuse.sourceforge.net/`.

[teredo,] Teredo tunneling. `http://en.wikipedia.org/wiki/Teredo_tunneling`.

[Tewari and Kleinrock, 2006] S. Tewari and L. Kleinrock. Proportional replication in peer-to-peer networks. In *Proc. INFOCOM*, 2006.

[tic, 2009] Tickless Kernel. `http://www.lesswatts.org/projects/tickless/`, 2009.

[Twi, 2010] Twitter's Murder Project at Github. `https://github.com/lg/murder`, Feb 2010.

[Valancius *et al.*, 2009] V. Valancius, N. Laoutaris, L. Massoulie, C. Diot, and P. Rodriguez. Greening the internet w/ nano data centers. In *CoNEXT*. ACM, 2009.

[verdiem,] Verdiem Technologies. `http://www.verdiem.com/`.

[VirtualBox,] VirtualBox. `http://www.virtualbox.org/`.

[Vlavianos *et al.*, 2006] Aggelos Vlavianos, Marios Iliofotou, and Michalis Faloutsos. Bitos: Enhancing bittorrent for supporting streaming applications. In *IEEE Global Internet Symposium*, Barcelona, Spain, April 2006.

[VMware Player,] VMware Player. `http://www.vmware.com/products/player/`.

[Wartel *et al.*, 2010] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. *IEEE CloudCom*, 0:112–117, 2010.

[Washburn,] Dog Washburn. How much money are your idle PCs wasting? Forrester Researech Reports, December 2008.

[Webber *et al.*, 2006] Carrie A. Webber, Judy A. Roberson, Marla C. McWhinney, Richard E. Brown, Margaret J. Pinckard, and John F. Busch. After-hours power status of office equipment in the usa. *Energy*, Nov. 2006.

[White *et al.*, 2002] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental

environment for distributed systems and networks. In *OSDI*, pages 255–270, Boston, MA, December 2002.

[win-ike,] How IPSec Works. `http://technet.microsoft.com/en-us/library/cc759130.aspx`.

[wol,] Wake-on-LAN. `http://en.wikipedia.org/wiki/Wake-on-LAN`.

[Wortham, 2011] Jenna Wortham. As networks speed up, data hits a wall, August 2011.

[Yoneki *et al.*, 2007] Eiko Yoneki, Pan Hui, ShuYan Chan, and Jon Crowcroft. A socio-aware overlay for pub/sub communication in DTN. In *Proc. ACM MSWiM*, 2007.

[Zhang *et al.*, 2008] Youhui Zhang, Xiaoling Wang, and Liang Hong. Portable Desktop Applications Based on P2P Transportation and Virtualization. In *LISA*, pages 133–144, 2008.

[Zhou *et al.*, 2007] Yipeng Zhou, Dah Ming Chiu, and John C.S. Lui. A simple model for analyzing p2p streaming protocols. *ICNP*, 0:226–235, 2007.

This page intentionally left blank

Appendix A

Wireless Computing

A.1 Proofs

A.1.1 General Expression for U

Proof of Lemma 1 We use the following two facts:

Let X be a geometric random variables in $\{ 1, 2, \cdots \}$ with probability of success r, then for any functions f

$$\mathbb{E}\left[f(X)\right] = f(1) - \sum_{k\geq 1}(1-r)^k(f(k) - f(k+1))\,. \tag{A.1}$$

It follows a simple Abel transformation of the series defining the expectation of this variable.

Similarly, let X be an exponential random variable with parameter λ, then we have, for any derivable function f defined on $[0, \infty[$ which admits a limit in 0^+:

$$\mathbb{E}\left[f(X)\right] = f(0^+) + \int_0^\infty \exp\left(-\lambda t\right) f'(t)dt\,. \tag{A.2}$$

It follows from a simple integration by part.

Whenever a demand for item i is created in client node n, two possible cases occur: Either the node is also a server and it contains a version of this item (*i.e.,* $n \in \mathcal{S}$ and $x_{i,n} = 1$), or the item has to be request from a server node. In the former case, the demand is fulfilled immediately in the next time slot, and the demand creates a gain $h(\delta)$ for this user. Otherwise, it may be fulfilled whenever a node meets a server node that stores this item. For any time slot, this client node's request may be fulfilled by server m with probability $(x_{i,m} \cdot \mu_{m,n} \cdot \delta)$. Hence the number of trials

before this request is fulfilled follows a geometric random variable in $\{ 1, 2, \cdots \}$ with a probability of success

$$r_{i,n} = 1 - \prod_{m \in \mathcal{S}} (1 - x_{i,m}\mu_{m,n}\delta) \ .$$

We may then write that the expected gain for an item i requested at node n is

$$\mathbb{E}\left[U_{i,n}\right] = x_{i,n}h(\delta) + (1 - x_{i,n})\mathbb{E}\left[h(\delta \cdot X)\right] \ ,$$

where X is a geometric random variable with success probability $r_{i,n}$, and $x_{i,n} = 0$ by convention whenever $n \notin \mathcal{S}$. Combining the above inequality with Eq.(A.1) we have:

$$\mathbb{E}\left[U_{i,n}\right] = h(\delta) - (1 - x_{i,n}) \sum_{k \geq 1} (1 - r)^k (h(\delta k) - h(\delta\,(k+1))) \, .$$

After replacing r and using the differential impatience function c, we obtain the expression of the lemma.

In a continuous time contact model, when the node i does not possess a copy of item i, the time elapsed before this request can be fulfilled is an exponential random variable with parameter:

$$\sum_{m \in \mathcal{S}} x_{i,m}\mu_{m,n} \, .$$

The result then follows from the exact same argument and Eq.(A.2).

Application to Homogeneous contact case In the homogeneous contact case, the expression for U simplifies: Let us start with the **dedicated node case** first. In the discrete time case, one can see

$$\prod_{m \in \mathcal{S}} (1 - x_{i,m}\mu_{m,n}\delta) = (1 - \mu\delta)^{x_i} \ .$$

Moreover, in the dedicated node case, for any $n \in \mathcal{C}$, $n \notin \mathcal{S}$ and hence $x_{i,n}$ is null by convention. The expression of $U_{i,n}$ from Lemma 1 hence can be rewritten:

$$U_{i,n}(\mathbf{x}) = h_i(\delta) - \sum_{k \geq 1} (1 - \mu\delta)^{k \cdot x_i} \, c_i(k \cdot \delta) \, .$$

This expression does not depend on n anymore, and we have

$$\begin{aligned} U(\mathbf{x}) &= \sum_i d_i \sum_{n \in \mathcal{C}} \pi_{i,n} U_{i,n} \\ &= \sum_i d_i \left(h_i(\delta) - \sum_{k \geq 1} (1 - \mu\delta)^{k \cdot x_i} \, c_i(k \cdot \delta) \right) \ , \end{aligned}$$

which implies Eq.(4.2).

In the continuous time case, we first observe that

$$\sum_{m\in\mathcal{S}} x_{i,m}\mu_{m,n} = x_i \cdot \mu\,.$$

As a consequence, following the same argument as before,

$$U_{i,n}(\mathbf{x}) = h_i(0^+) - \int_0^\infty \exp\left(-tx_i\cdot\mu\right) c_i(t)dt\,.$$

This expression does not depend on n any more, and implies that $U(\mathbf{x})$ can be written as Eq.(4.3).

We now consider the **pure P2P case**, where all nodes are server, and we assume that profile are uniform among users (*i.e.,* $\pi_{i,n} = 1/N$).

$$U_{i,n}(\mathbf{x}) = h_i(\delta) - (1-x_{i,n})\sum_{k\geq 1}\left(1-\mu\delta\right)^{k\cdot x_i} c_i(k\cdot\delta)\,.$$

For n such that $x_{i,n} = 1$ we have $U_{i,n} = h_i(\delta)$, this case occurs exactly x_i times. For all other value of n we have the same expression as in the dedicated node case above, this case occurs exactly $N - x_i$ times. Hence we deduce:

$$\begin{aligned}\sum_{n\in\mathcal{C}}\pi_{i,n}U_{i,n} &= \sum_{n\in\mathcal{C}}\frac{1}{N}U_{i,n}\\ &= h_i(\delta) - \frac{N-x_i}{N}\sum_{k\geq 1}\left(1-\mu\delta\right)^{k\cdot x_i} c_i(k\cdot\delta)\,.\end{aligned}$$

This proves Eq.(4.4). A similar argument for the continuous case yields Eq.(4.5).

A.1.2 Submodularity and Concavity property

Proof of Theorem 1 It is sufficient to prove that, for a given item i and a client node n, $U_{i,n}$ is a submodular function of the set $\{\ m\in\mathcal{S}\ \mid x_{i,m}=1\ \}$, since the sum of supermodular functions is supermodular.

Let us fix an item i and a client node n. For any subset A of $\mathcal{S}$, we introduce the following set function:

$$\sigma : A \mapsto \prod_{m\in A}\left(1-\mu_{m,n}\cdot\delta\right),$$

We may redefine $U_{i,n}$ as a set function such that $U_{i,n}(A)$ is given by:

$$h_i(\delta) - (1-\mathbb{I}_{\{n\in A\}})\sum_{k\geq 1}\sigma(A)^k c_i(k\cdot\delta)\,. \tag{A.3}$$

We first observe that, for any $k \geq 1$, the function $f : A \mapsto \sigma(A)^k$ is supermodular. Indeed, we have for any subset $A \subseteq \mathcal{S}$ and $m \in \mathcal{S}$:

$$f(A \cup \{m\}) - f(A) = \mathbb{I}_{\{m \notin A\}} \cdot \sigma(A)^k \left((1 - \mu_{m,n})^k - 1 \right) .$$

For $A \subseteq B$, we have

$$\mathbb{I}_{\{m \notin A\}} \left(\sigma(A) \right)^k \geq \mathbb{I}_{\{m \notin B\}} \left(\sigma(B) \right)^k \geq 0 ,$$

since both terms are positive non-increasing set function (w.r.t. set inclusion). As $\left((1 - \mu_{m,n})^k - 1 \right)$ is always non-positive, we deduce for $A \subseteq B$

$$\begin{aligned} f(A \cup \{m\}) - f(A) &= \left((1 - \mu_{m,n})^k - 1 \right) \mathbb{I}_{\{m \notin A\}} \cdot \sigma(A) \\ &\leq \left((1 - \mu_{m,n})^k - 1 \right) \mathbb{I}_{\{m \notin B\}} \cdot \sigma(B) \\ &\leq f(B \cup \{m\}) - f(B) . \end{aligned}$$

We deduce for any k, f is a supermodular function, which is also positive. A weighted sum, with positive weights, of supermodular function is a supermodular function. Hence, since $c_i(k \cdot \delta)$ is positive as the impatience function h_i is non-increasing, we deduce

$$A \mapsto \sum_{k \geq 1} \sigma(A)^k c_i(k \cdot \delta) ,$$

is a supermodular function. We also deduce that it is positive and non-increading.

Lemma 2 *Let f be a supermodular, non-increasing, non-negative set function defined on S. Then, for any $n \in S$, the set function $g : A \mapsto (1 - \mathbb{I}_{\{n \in A\}}) f(A)$ is supermodular, non-increasing and non-negative.*

Proof 1 *g is obviously non-negative and it is non-increasing since f is non-negative. It remains to be shown that it is supermodular. Let $A \subseteq B$ be two subsets of S, we wish to prove that, for any $m \in S$, we have*

$$g(A \cup \{m\}) - g(A) \leq g(B \cup \{m\}) - g(B) .$$

- *If $n \in B$, then the RHS above is always null. As g is non-increasing, the LHS is always non-positive, which proves the results.*
- *If $n \notin B$, then it is by definition not in A neither. Whenever $m \neq n$, we then have that the inequality above holds as g and f are equal on all these subsets. When $m = n$, the inequality simplifies to $-g(A) \leq -g(B)$, equivalently $g(A) \geq g(B)$ which holds as g is non-increasing.*

From Eq.(A.3) and Lemma 3, we deduce that $U_{i,n}$ may be written as a constant minus a supermodular function. It is hence by definition submodular.

The same argument holds for a continuous time model: We first introduce the following set function defined on all subset A of $\mathcal{S}$:

$$\tilde{\sigma} : A \mapsto \sum_{m \in A} \mu_{m,n} \,,$$

We may redefine $U_{i,n}$ for the continuous time model as a set function such that $U_{i,n}(A)$ is given by:

$$h_i(0^+) - (1 - \mathbb{I}_{\{n \in A\}}) \int_0^\infty \exp\left(-t \cdot \tilde{\sigma}(A)\right) c_i(t) dt \,. \tag{A.4}$$

We first observe that, for any $t \geq 0$, the function $f : A \mapsto \exp(-t \cdot \tilde{\sigma}(A))$ is supermodular. Indeed, we have for any subset $A \subseteq \mathcal{S}$ and $m \in \mathcal{S}$:

$$f(A \cup \{m\}) - f(A) = \mathbb{I}_{\{m \notin A\}} \cdot e^{-t\tilde{\sigma}(A)} \left(e^{-t\mu_{m,n}} - 1\right) \,.$$

For $A \subseteq B$, we have $\mathbb{I}_{\{m \notin A\}} \cdot e^{-t\tilde{\sigma}(A)} \geq \mathbb{I}_{\{m \notin B\}} \cdot e^{-t\tilde{\sigma}(B)}$, since both are positive non-increasing function of the set. As, for any m $\left(e^{-t\mu_{m,n}} - 1\right)$ is non-positive, we deduce for $A \subseteq B$

$$\begin{aligned} f(A \cup \{m\}) - f(A) &= \mathbb{I}_{\{m \notin A\}} \cdot e^{-t\tilde{\sigma}(A)} \left(e^{-t\mu_{m,n}} - 1\right) \\ &\leq \mathbb{I}_{\{m \notin B\}} \cdot e^{-t\tilde{\sigma}(B)} \left(e^{-t\mu_{m,n}} - 1\right) \\ &\leq f(B \cup \{m\}) - f(B) \,. \end{aligned}$$

We deduce that, whatever be the value of t, f is a supermodular function, which is also positive. A weighted sum, with positive weights, of supermodular function is a supermodular function, which also generalizes to integral of function multiplied by a positive term. Hence, since $c_i(t)$ is by definition positive for any value of t we deduce that

$$A \mapsto \int_0^\infty \exp\left(-t \cdot \tilde{\sigma}(A)\right) c_i(t) dt \,,$$

is a supermodular function. We also deduce that it is positive. From Eq.(A.4) and Lemma 3, we deduce that $U_{i,n}$ may be written as a constant minus a supermodular function. It is hence by definition submodular.

Proof of Theorem 2 With homogeneous contacts between nodes, the social welfare only depends on the total number of copies x_i of each item i. Let us start with the dedicated node case under the discrete time contact model. In this case, we have

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(\delta) - \sum_{k \geq 1} (1 - \mu\delta)^{x_i k} c_i(k \cdot \delta) \right) .$$

We first observe that, if we allow x_i to take real number value, U is a concave function of the $\{x_i | i \in I\}$. It comes from the fact that, for any $k \geq 1$, the function

$$x \mapsto (1 - \mu\delta)^{xk} ,$$

is convex. U is then a weighted sum of concave functions with positive coefficient (since $c(k \cdot \delta)$ is non-negative for all k). The same argument applies to show that the relaxed optimization for continuous time, as well as for the pure P2P case, satisfy the same property, from Eq.(4.3)-(4.5).

Lemma 3 *Let f, g be two convex, non-increasing, non-negative derivable functions, then the product function fg is convex, non-increasing and non-negative.*

Proof 2 *The function fg is obviously non-negative and non-increasing (as a product of two non-negative, non-increasing functions). In addition we have, after a derivation:*

$$\begin{aligned} \frac{dfg}{dx}(x) &= f'(x)g(x) + g'(x)f(x) \\ &= -\left(g(x)\left(-f'(x)\right) + f(x)\left(-g'(x)\right)\right) . \end{aligned}$$

The function $-f'$ is non-negative (as f is non-increasing) and non-increasing (as f is convex). We deduce that $g(-f')$ is non-increasing and non-negative. We deduce that the derivative of fg as shown above is non-decreasing, hence that fg is convex.

This proves the second half of theorem and directly implies that there exists a unique maximum of the relaxed optimization which can be found through gradient descent algorithm [Boyd and Vandenberghe, 2004].

The only remaining point to prove the theorem is to show that, for the original problem where x_i only takes integer values, the maximum can be obtained by a greedy procedure

For any $\mathbf{x} = \{x_i | i \in I\}$, we denote by $\frac{\Delta U}{\Delta x_i}(\mathbf{x})$ the *marginal improvement* obtained when another copy of item i is created. It is defined by

$$\frac{\Delta U}{\Delta x_i}(\mathbf{x}) = U(x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_M) - U(\mathbf{x}) \,.$$

One key observation is that, for a given $i \in i$, the marginal improvement $\frac{\Delta U}{\Delta x_i}(\mathbf{x})$ is independent of x_j for all $j \neq i$. Note that this is intuitive, as for a fixed number of copies of a given item, the delay or waiting time to find this item is not impacted by how much copies of *other* items are available. In other words, we may write $\frac{\Delta U}{\Delta x_i}(\mathbf{x}) = f_i(x_i)$, where

$$f_i(x) = d_i \sum_{k \geq 1} (1 - \mu\delta)^{xk} \left(1 - (1 - \mu\delta)^k\right) c_i(k \cdot \delta) \,.$$

In summary, the value of the social welfare can be decomposed, for each item i, as a sum of x_i terms that are values of a function f_i:

$$U(\mathbf{x}) = \sum_{i \in I} \left(f_i(1) + f_i(2) + \cdots + f_i(x_i)\right) \,. \tag{A.5}$$

Another important observation is that, owing to the concavity of the function U (established above), the function f_i above are all non-increasing.

We will now prove that a greedy procedure can find the allocation with maximum social welfare. Let us first define two optimization problems.

$$\textbf{OPT}(C): \quad \max \sum_{i \in I} \sum_{l \in \{1,2,\cdots,x_i\}} f_i(l) \;\text{ such that}$$
$$1 \leq x_i \leq |\mathcal{S}| \,, \sum_{i \in I} x_i \leq C$$

$$\textbf{SETOPT}(C): \quad \max \sum_{i \in I} \sum_{l \in A_i} f_i(l) \;\text{ such that}$$
$$A_i \subseteq \mathbb{N}^* \,, 1 \leq |A_i| \leq |\mathcal{S}| \,, \sum_{i \in I} |A_i| \leq C$$

The problem $\textbf{OPT}(C)$ with $(C = \rho \cdot |\mathcal{S}|)$ is exactly the problem we wish to solve. The problem $\textbf{SETOPT}(C)$ is only defined for the need of the proof: it is a slightly more general problem in the sense that it does not require to look at sum of values of f_i on contiguous integers, but can consider values of functions f_i on any collection of subsets.

We start by the two following simple results:

Lemma 4 *When the functions f_i are non-increasing for all i, the two optimization problems are equivalent:*

$$\text{We have } \forall i \in I\,,\; A_i = \{1, 2, \ldots, x_i\}\,.$$

where $\{\ x_i\ \mid i \in I\ \}$ (resp. $\{\ A_i \mid\ i \in I\ \}$) denotes the solution of $\boldsymbol{OPT}(C)$ (resp. $\boldsymbol{SETOPT}(C)$).

One can easily show that the A_i should be made of contiguous integers starting from 1 (If that is not the case, is is easy to construct an even better choice of A_i to maximize the sum). Owing to this fact, the two problems are equivalent and the result above holds.

Lemma 5 *Let $\{\ A_i \mid\ i \in I\ \}$ and $\{\ B_i \mid\ i \in I\ \}$ be the solutions of $\boldsymbol{SETOPT}(C)$ and $\boldsymbol{SETOPT}(C+1)$.*

- *for any $i \in I$, $A_i \subseteq B_i$.*
- *The only j such that $B_j \neq A_j$ satisfies*

$$j = \text{argmax}\left\{ \max_{l \notin A_i} f_i(l) \;\middle|\; i \in I, |A_i| < |\mathcal{S}| \right\}.$$

Since A cannot contain B due to size constraint, there exists j and a such that $a \in B_j$ and $a \notin A_j$. Let $B'_j = B_j \setminus \{a\}$ and $B'_i = B_i$ for all $i \neq j$. Then the collection of all subsets B'_j for all $j \in I$ satisfies the conditions of problem $\mathbf{SETOPT}(C)$. By optimality of B we should have $B'_i = A_i$ for all $i \in I$ (otherwise one could always construct an even better choice than B. This concludes the proof, as again, if the second property does not hold, one can construct an even better solution starting from the subsets A_i and adding one element.

As a consequence of the two lemmas, when all functions f_i for all i are non-increasing, we deduce that if $\{\ x_i \mid\ i \in I\ \}$ and $\{\ y_i \mid\ i \in I\ \}$ are the solutions of $\mathbf{OPT}(C)$ and $\mathbf{OPT}(C+1)$ then

$$\begin{cases} y_j = x_j + 1 & \text{if } \; j = \text{argmax}_i\{f_i(x_i+1) | x_i < |\mathcal{S}|\} \\ y_j = x_j & \text{otherwise} \quad . \end{cases}$$

This can be used as a recursive rules to deduce the optimal allocation for any cache size C. This is what Algorithm A.1, defined below, implements in at most $O(|I| + \rho|\mathcal{S}| \cdot \ln(|I|))$ steps (since the search for the maximum improvement can be run in at most $O(\ln(|I|)$ with a priority queue).

$x_i \leftarrow 1$;$\texttt{sum} \leftarrow M$;
$A = \{\ 1, 2, \cdots, M\ \}$;
for all $i \in A$ **do**
  $\texttt{imp}_i \leftarrow \frac{\Delta U}{\Delta x_i}(\mathbf{x})$;
end for
while $\texttt{sum} \leq \rho|\mathcal{S}|$ **do**
  pick $j = \arg\max_i \{\ \texttt{imp}_i\ \mid i \in A\ \}$;
  $x_j \leftarrow x_j + 1$; $\texttt{sum} \leftarrow \texttt{sum} + 1$;
  $\texttt{imp}_j \leftarrow \frac{\Delta U}{\Delta x_i}(\mathbf{x})$;
  if $x_j = |\mathcal{S}|$ **then**
    $A \leftarrow A \setminus \{j\}$;
  end if
end while

Figure A.1: Maximum welfare (Homogeneous contact).

The proof was conducted here in the discrete time for the dedicated node case, the same exact argument applies to pure P2P case, where f_i is defined as

$$f_i(x) = d_i(1 - \frac{x}{N}) \sum_{k \geq 1} (1 - \mu\delta)^{xk} \left(1 - (1 - \mu\delta)^k\right) c_i(k \cdot \delta)\,.$$

Similarly, we can prove the same result for a continuous time contact model, where f_i is defined in the dedicated node case as

$$f_i(x) = d_i \int_0^\infty e^{-\mu t \cdot x} \left(1 - e^{\mu t}\right) c_i(t) dt\,,$$

and in the pure p2p case as

$$f_i(x) = d_i(1 - \frac{x}{N}) \int_0^\infty e^{-\mu t \cdot x} \left(1 - e^{\mu t}\right) c_i(t) dt\,.$$

All these functions are positive and non-increasing, which allows to use the exact same proof.

A.1.3 Proof of Theorem 1

In the continuous time contact model and dedicated node we have:

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(0^+) - \int_0^\infty e^{-t\mu x_i} c_i(t) dt\right)\,.$$

Note that it implies $\frac{\partial U}{\partial x_i}(\mathbf{x}) = d_i\varphi(x_i)$. where φ is defined as in Theorem 1.

Let us assume that $\tilde{x}_i < |\mathcal{S}|$, $\tilde{x}_j < |\mathcal{S}|$. If we have $d_i \cdot \varphi(\tilde{x}_i) < d_j \cdot \varphi(\tilde{x}_j)$, then there exists $\varepsilon > 0$ such that $U(x') > U(\tilde{x})$ where we define x' as

$$\begin{cases} x'_j = \tilde{x}_j + \varepsilon \\ x'_i = \tilde{x}_i - \varepsilon \\ x'_k = \tilde{x}_k \qquad \text{for } \; k \neq i, k \neq j \,. \end{cases}$$

This contradicts the optimality of $\tilde{x}$, and proves that the equilibrium condition holds.

A.1.4 Singularity of h at $t = 0$

As shown in Lemma 1, $U_{i,n}(\mathbf{x})$ and $U(\mathbf{x})$ may be expressed in a simple form when $h(0^+) < \infty$. In the dedicated node case, where servers and clients are in disjoint sets, the same expression generalizes to other types of impatience function where $h(0^+) = \infty$. As an example, we treat here the case of the negative logarithm (*i.e.,* $h_\alpha^{(\mathrm{p})}$ with $\alpha = 1$) and the inverse powers (*i.e.,* $h_\alpha^{(\mathrm{p})}$ with $\alpha > 1$).

Since we are in the dedicated node case, we have

$$U_{i,n}(\mathbf{x}) = \mathbb{E}\left[h(X)\right] \,,$$

where X is an exponential random variable with parameter $\sum_{m\in\mathcal{S}} x_{i,m}\mu_{m,n}$, that we can denote by $\mu_{i,n}$ for short.

Expression of $U_{i,n}$ for the Negative logarithm When $\alpha = 1$, the impatience function is defined as $h_1^{(\mathrm{p})} : t \mapsto -\ln(t)$. Hence we have :

$$U_{i,n}(\mathbf{x}) = \int_0^\infty \mu_{i,n} \exp\left(-t\mu_{i,n}\right)\left(-\ln(t)\right)dt \,.$$

A simple change of variable leads to

$$\begin{aligned} U_{i,n}(\mathbf{x}) &= \int_0^\infty \left(\ln(\mu_{i,n}) - \ln(u)\right) e^{-u} du \\ &= \ln\left(\sum_{m\in\mathcal{S}} x_{i,m}\mu_{m,n}\right) - \int_0^\infty \ln(u) e^{-u} du \,. \end{aligned}$$

This allows to compute U thanks to

$$U(\mathbf{x}) = \sum_{i\in I} d_i \sum_{n\in\mathcal{C}} \pi_{i,n} U_{i,n}(\mathbf{x}) \,.$$

This expression simplifies for the case of homogeneous contact as follows

$$\begin{aligned} U(\mathbf{x}) &= \sum_{i\in I} d_i \ln(x_i) + \mathtt{cst}, \quad \text{where} \\ \mathtt{cst} &= \sum_{i\in I} d_i \left(\ln(\mu) - \int_0^\infty \ln(u) e^{-u} du \right) . \end{aligned}$$

Consequence on Theorem 1 and 2 We first note that it proves that $U_{i,n}$ is, as a set function, submodular (which extends the results of Theorem 1). Indeed, with similar notation than in Section A.1.2, we can write:

$$U_{i,n}(A) = \ln(\tilde{\sigma}(A)) - \int_0^\infty \ln(u) e^{-u} du \,.$$

which proves this result. We also deduce that the argument for the proof of Theorem 2 holds, since we can write $\frac{\Delta U}{\Delta x_i}(\mathbf{x}) = f_i(x_i)$ with

$$f_i(x) = d_i \ln\left(1 + \frac{1}{x_i}\right) .$$

which is a positive non-increasing function.

Expression of $U_{i,n}$ for the inverse powers When $1 < \alpha < 2$, we have that $h_\alpha^{(\mathrm{p})} : t \mapsto \frac{t^{1-\alpha}}{\alpha-1}$, and we may write in the dedicated node case:

$$U_{i,n}(\mathbf{x}) = \int_0^\infty \mu_{i,n} \exp(-t\mu_{i,n}) \frac{t^{1-\alpha}}{\alpha-1} dt \,.$$

Again, a simple change of variable yields

$$\begin{aligned} U_{i,n}(\mathbf{x}) &= (\mu_{i,n})^{\alpha-1} \int_0^\infty \exp(-u) \frac{u^{1-\alpha}}{\alpha-1} du \\ &= \frac{\Gamma(2-\alpha)}{\alpha-1} \left(\sum_{m\in\mathcal{S}} x_{i,m}\mu_{m,n} \right)^{\alpha-1} . \end{aligned}$$

This equation can be used to derive $U(\mathbf{x})$ in the general case. For homogeneous contact case, it simplifies to

$$U(\mathbf{x}) = \frac{\Gamma(2-\alpha)}{\alpha-1} \mu^{\alpha-1} \sum_{i\in I} d_i (x_i)^{\alpha-1} .$$

Model	Step function	Exponential decay	Neg. Power ($\alpha < 1$)
Impatience $h(t)$	$\mathbb{I}_{\{t \leq \tau\}}$	$\exp(-\nu t)$	$\frac{t^{1-\alpha}}{\alpha - 1}$
Diff. impat. c	Dirac at $t = \tau$	density $t \mapsto \nu \exp(-\nu t)$	density $t \mapsto t^{-\alpha}$
Gain $U_{i,n}(\mathbf{x})$	$1 - (1 - \frac{x_i}{N}) e^{-\mu\tau x_i}$	$1 - (1 - \frac{x_i}{N}) \left(1 + \frac{\mu}{\nu} x_i\right)^{-1}$	$\mu^{\alpha-1}\Gamma(1-\alpha)\left(-(x_i)^{\alpha-1} + \frac{1}{N}(x_i)^{\alpha}\right)$
Cond. φ'	$d_i \cdot \left(\mu\tau + \frac{1-\mu\tau x_i}{N}\right) e^{-\mu\tau x_i}$	$d_i \cdot \left(\frac{\mu}{\nu} - \frac{1}{N}\right) \left(1 + \frac{\mu}{\nu} x_i\right)^{-2}$	$d_i \cdot \mu^{\alpha-1}\Gamma(1-\alpha)\left((1-\alpha)x_i^{\alpha-2} + \frac{1}{N}\alpha x_i^{\alpha-1}\right)$
Reaction ψ	$\frac{\mu\tau + \frac{1-\mu\tau/y}{N}}{y} e^{-\frac{\mu\tau}{y}}$	$\left(1 - \frac{\nu}{\mu}\frac{1}{N}\right)\left(2 + \frac{\nu}{\mu} y + \frac{\mu}{\nu}\frac{1}{y}\right)^{-1}$	$\mu^{\alpha-1}\Gamma(1-\alpha)\left((1-\alpha)y^{1-\alpha} + \frac{1}{N}\alpha y^{-\alpha}\right)$

Table A.1: Optimal demand reaction and cache allocation for several impatience functions (dedicated cache case).

Consequence on Theorem 1 and 2 We have, with similar notation than in Section A.1.2,

$$U_{i,n}(A) = \frac{\Gamma(2-\alpha)}{\alpha - 1} \left(\tilde{\sigma}(A)\right)^{\alpha-1} .$$

For any $1 < \alpha < 2$, the function $x \mapsto x^{\alpha-1}$ is concave. The formula above proves that $U_{i,n}$ is again in this case, a submodular function, extending again the results of Theorem 1 to a new class of function.

Again the argument for the proof of Theorem 2 holds, as we have $\frac{\Delta U}{\Delta x_i}(\mathbf{x}) = f_i(x_i)$ with

$$f_i(x) = d_i \frac{\Gamma(2-\alpha)}{\alpha - 1} \left((x+1)^{\alpha-1} - x^{\alpha-1}\right) .$$

which is a positive non-increasing function.

As a conclusion, although the exact computation may differ, the exact same arguments why all the results of this chapter apply to a general impatience function with a limit in 0^+ holds for a general family of impatience function even when it diverges in 0. It should be possible to prove this result more generally, as well as to obtain similar expressions for the discrete time case, but this is beyond the scope of this current work.

A.1.5 Table condition in the pure P2P case

In the continuous time contact model an pure p2p case we have:

$$U(\mathbf{x}) = \sum_{i \in I} d_i \left(h(0^+) - (1 - \frac{x_i}{N}) \int_0^\infty e^{-t\mu x_i} c_i(t) dt \right) .$$

Note that it implies $\frac{\partial U}{\partial x_i}(\mathbf{x}) = d_i \varphi'(x_i)$ where

$$\varphi'(x) = (1 - \frac{x}{N})\varphi(x) + \frac{1}{N} \int_0^\infty e^{-t\mu x} c(t) dt \, .$$

In other words, just like Theorem 1 applies to the dedicated node case, a similar equilibrium condition exists for the optimality in the pure p2p case. We just need to define another function φ' which contains a correcting term, that is small when N is large.

The main consequence is that, just like Theorem 2, there exists a demand reaction function which is exactly optimal in the pure p2p case. This reaction function is shown in Table A.1. As it can be seen on the table, most of the time the correction term is small, which is why we did not use this version in the simulations.

A.2 Additional Results

In this section we present, for completeness, the result we obtain across all parameters we used for validation.

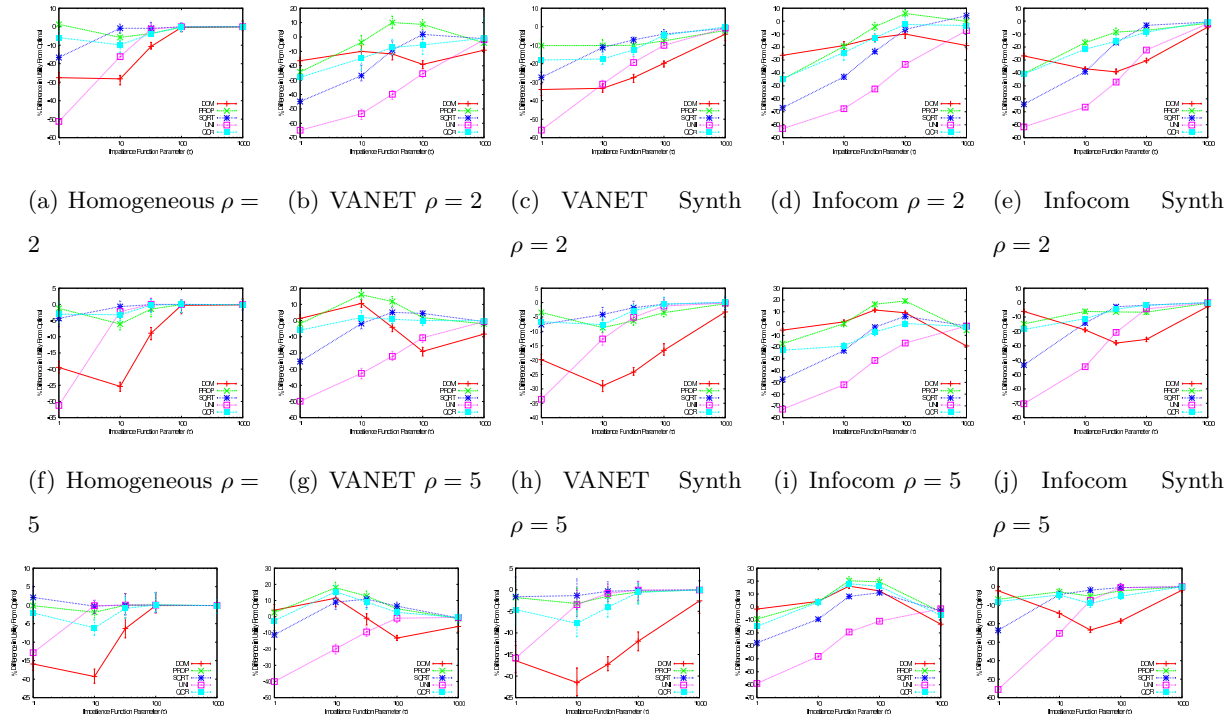
A.2.1 Impact of cache size and popularity distribution

Figure A.2 presents the difference in Utility as taken from optimal, for the step impatience function with different value of τ (on the x-axis), for the same strategies as in Section 4.6. The difference with previous results is that results are also shown for a smaller cache size $\rho = 2$ and a larger cache size $\rho = 10$. Note that we also reproduce the result for $\rho = 5$ (the default value) to allow for an easy comparison. As it can be seen, although there are small numerical differences, it plays almost no role in the qualitative behaviors, across all contact traces.

Figure A.3, presents the same result as above, except that we consider an highly skewed distribution (*i.e.,* $\omega = 2$) and a moderately skewed distribution (*i.e.,* $\omega = 1/2$), compared with the default case where ω is equal to 1. We recall that ω corresponds to the skewness of the popularity of file as: $d_i \propto i^{-\omega}$. As expected, a moderately skewed distribution tends to improve the performance of uniform and decreases that of the dominated allocation. A highly skewed distribution seems to even out most of the strategies across all data sets, except for uniform which performs badly.

A.2.2 Full results for the homogeneous case

We consider in this section a network with homogeneous contact among the nodes. Figure 4.4 already presented for different family of impatience functions the respective performance of the



(a) Homogeneous $\rho = 2$ (b) VANET $\rho = 2$ (c) VANET Synth $\rho = 2$ (d) Infocom $\rho = 2$ (e) Infocom Synth $\rho = 2$

(f) Homogeneous $\rho = 5$ (g) VANET $\rho = 5$ (h) VANET Synth $\rho = 5$ (i) Infocom $\rho = 5$ (j) Infocom Synth $\rho = 5$

(k) Homogeneous $\rho = 10$ (l) VANET $\rho = 10$ (m) VANET Synth $\rho = 10$ (n) Infocom $\rho = 10$ (o) Infocom Synth $\rho = 10$

Figure A.2: Impact of the cache size ρ (I=10, N=50, $\omega = 1$).

different algorithms. Figures A.4 and A.5 present in more details the content of the cache, and the utility obtained as a function of time for one run of each of this impatience functions.

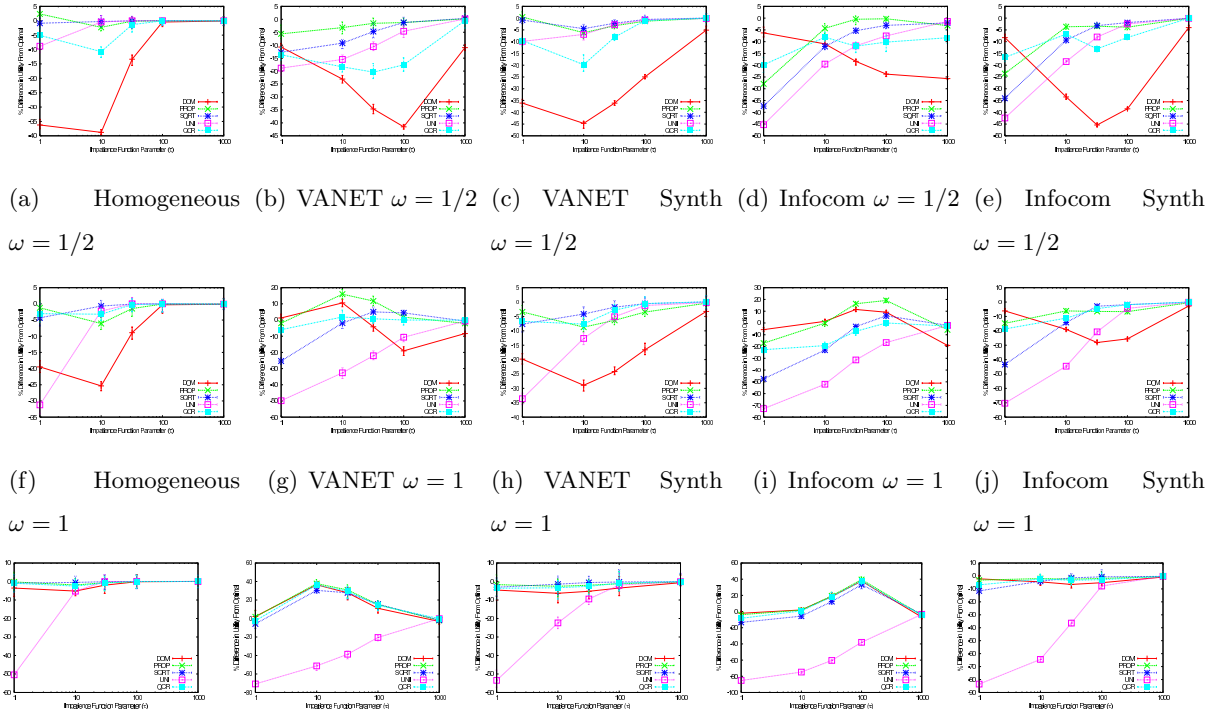
A.2.3 Full results for real contact traces

We now present the same full results (cache and utility perceived as a function of time for the contact traces (as well as the synthetic contact traces, which represents the heterogeneity of contact only).

Figures A.6 and A.7 present the result for the Infocom datasets and the step impatience function (with $\rho = 5$.

Figure A.8 and A.9 present the same results for the Vanet data sets with power impatience functions.

Finally A.10-A.15 present the results for the Vanet data sets, with step impatience functions and several values of ρ.

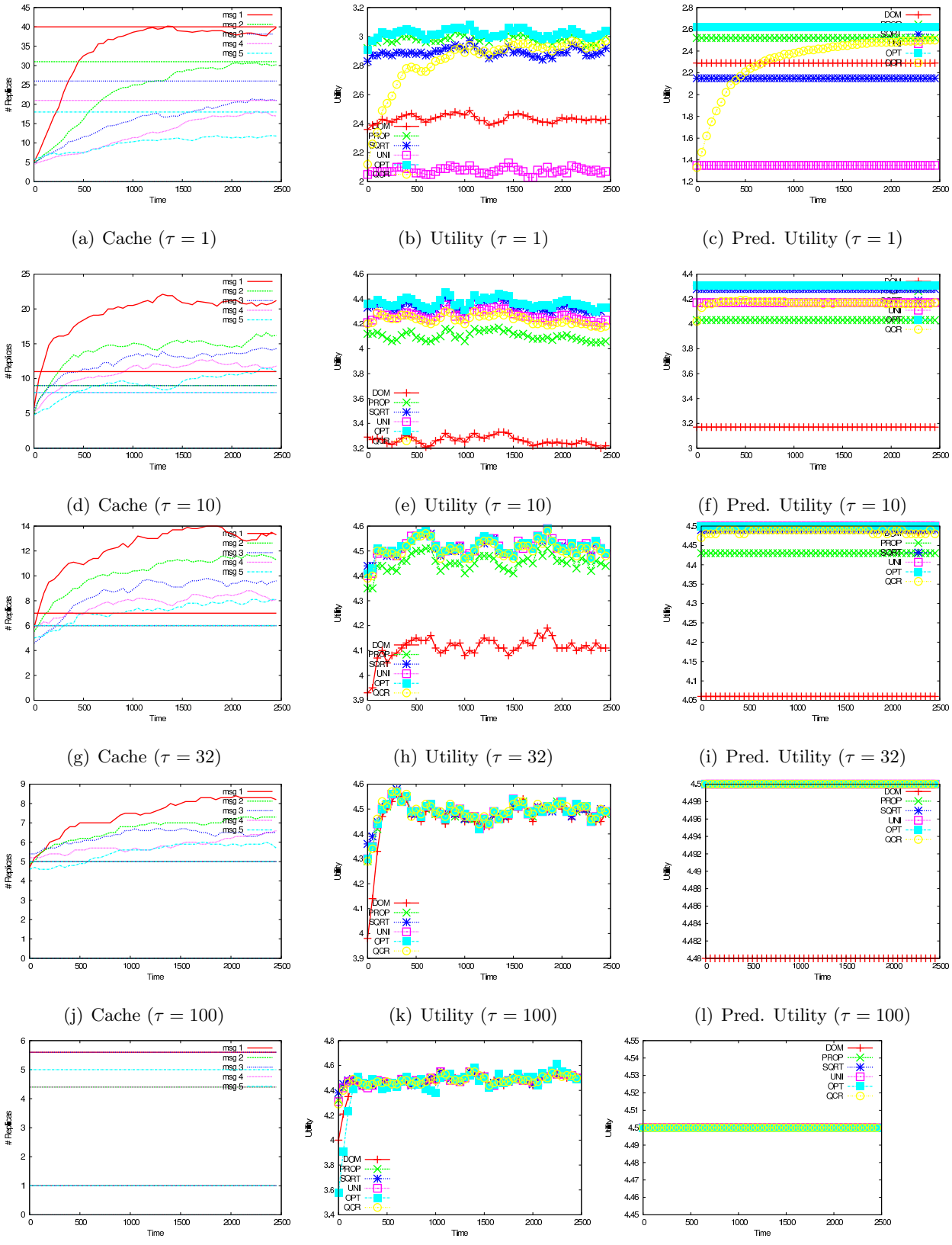


(a) Homogeneous $\omega = 1/2$ (b) VANET $\omega = 1/2$ (c) VANET Synth $\omega = 1/2$ (d) Infocom $\omega = 1/2$ (e) Infocom Synth $\omega = 1/2$

(f) Homogeneous $\omega = 1$ (g) VANET $\omega = 1$ (h) VANET Synth $\omega = 1$ (i) Infocom $\omega = 1$ (j) Infocom Synth $\omega = 1$

(k) Homogeneous $\omega = 2$ (l) VANET $\omega = 2$ (m) VANET Synth $\omega = 2$ (n) Infocom $\omega = 2$ (o) Infocom Synth $\omega = 2$

Figure A.3: Impact of the popularity distribution of items ω (I=10, N=50, $\rho = 5$).



(a) Cache ($\tau = 1$) (b) Utility ($\tau = 1$) (c) Pred. Utility ($\tau = 1$)

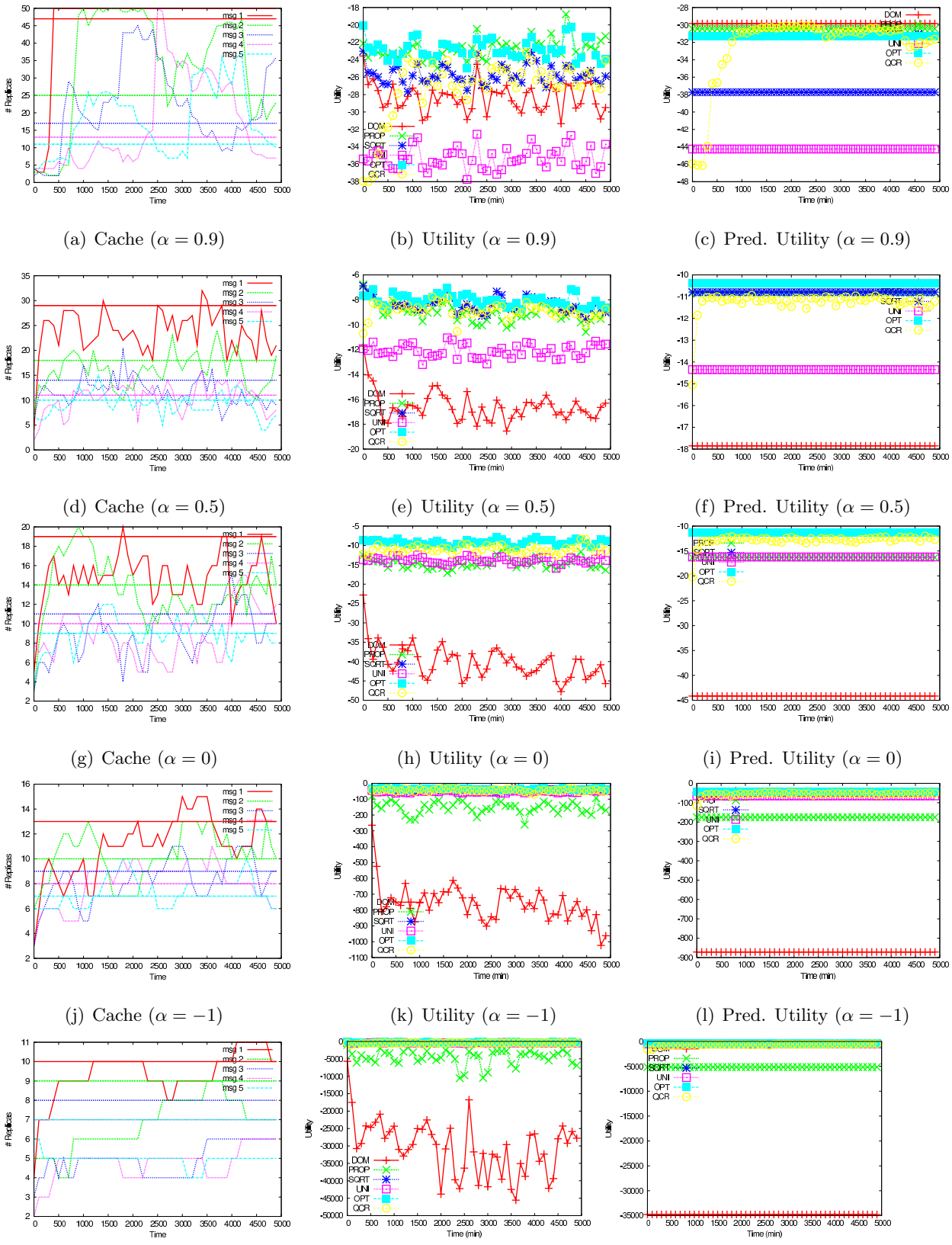
(d) Cache ($\tau = 10$) (e) Utility ($\tau = 10$) (f) Pred. Utility ($\tau = 10$)

(g) Cache ($\tau = 32$) (h) Utility ($\tau = 32$) (i) Pred. Utility ($\tau = 32$)

(j) Cache ($\tau = 100$) (k) Utility ($\tau = 100$) (l) Pred. Utility ($\tau = 100$)

(m) Cache ($\tau = 1000$) (n) Utility ($\tau = 1000$) (o) Pred. Utility ($\tau = 1000$)

Figure A.4: Homogenous mixing, step impatience: experienced utility, predicted utility, and cache evolution over time for $\mu = 0.05, \rho = 5, I = 50, N = 50, \omega = 1$.



(a) Cache ($\alpha = 0.9$) (b) Utility ($\alpha = 0.9$) (c) Pred. Utility ($\alpha = 0.9$)

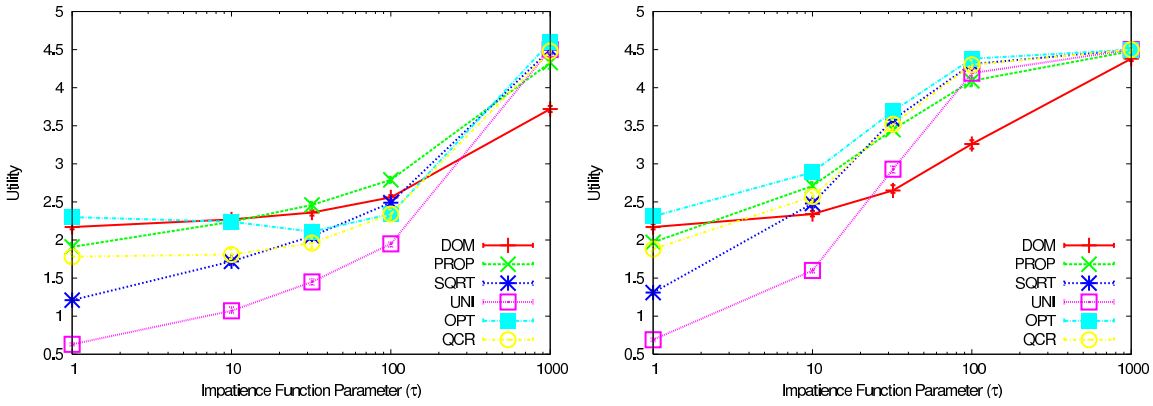
(d) Cache ($\alpha = 0.5$) (e) Utility ($\alpha = 0.5$) (f) Pred. Utility ($\alpha = 0.5$)

(g) Cache ($\alpha = 0$) (h) Utility ($\alpha = 0$) (i) Pred. Utility ($\alpha = 0$)

(j) Cache ($\alpha = -1$) (k) Utility ($\alpha = -1$) (l) Pred. Utility ($\alpha = -1$)

(m) Cache ($\alpha = -2$) (n) Utility ($\alpha = -2$) (o) Pred. Utility ($\alpha = -2$)

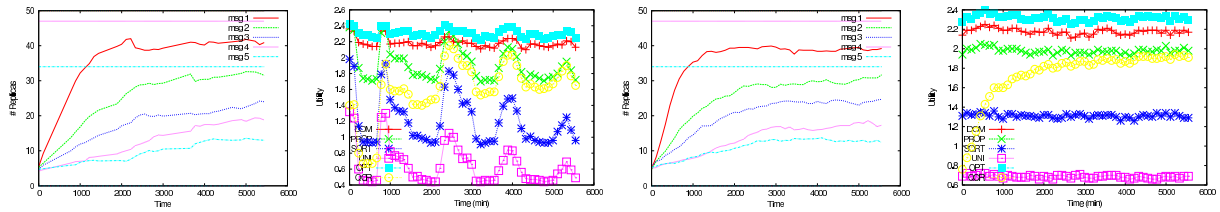
Figure A.5: Homogenous mixing, power impatience: utility, predicted utility, and cache evolution over time for $\mu = 0.05, \rho = 5, I = 50, N = 50, \omega = 1$.



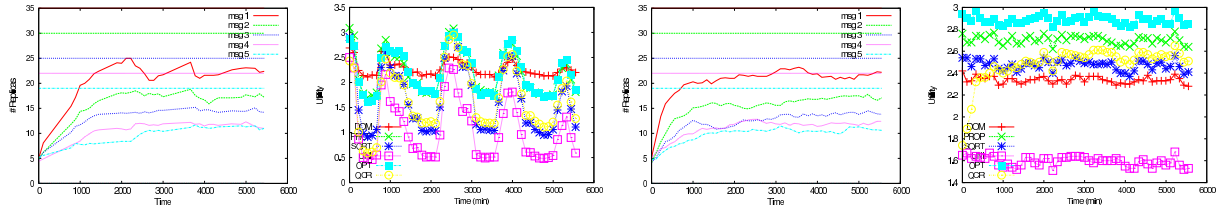
(a) Actual Trace

(b) Synthesized Trace

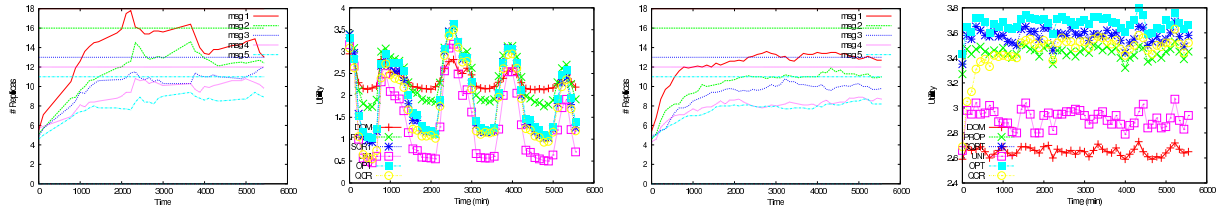
Figure A.6: Infocom '06 Trace, step impatience - understanding impact of time statistics on avg. behavior.



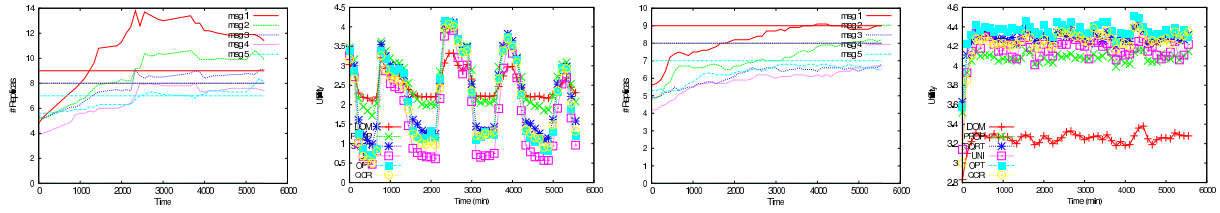
(a) Cache ($\tau = 1$) (b) Utility ($\tau = 1$) (c) (Synthetic) Cache ($\tau = 1$) (d) (Syntetic) Utility ($\tau = 1$)



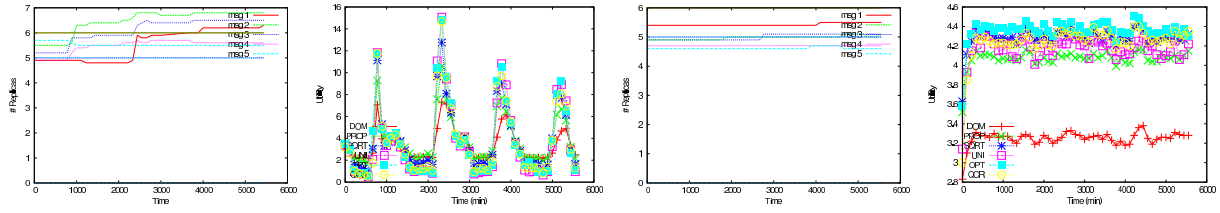
(e) Cache ($\tau = 10$) (f) Utility ($\tau = 10$) (g) (Synthetic) Cache ($\tau = 10$) (h) (Syntetic) Utility ($\tau = 10$)



(i) Cache ($\tau = 32$) (j) Utility ($\tau = 32$) (k) (Synthetic) Cache ($\tau = 32$) (l) (Syntetic) Utility ($\tau = 32$)



(m) Cache ($\tau = 100$) (n) Utility ($\tau = 100$) (o) (Synthetic) Cache ($\tau = 100$) (p) (Syntetic) Utility ($\tau = 100$)



(q) Cache ($\tau = 1000$) (r) Utility ($\tau = 1000$) (s) (Synthetic) Cache ($\tau = 1000$) (t) (Syntetic) Utility ($\tau = 1000$)

Figure A.7: Infocom '06, step impatience - timewise results for $\rho = 5, I = 50, N = 50, \omega = 1$.

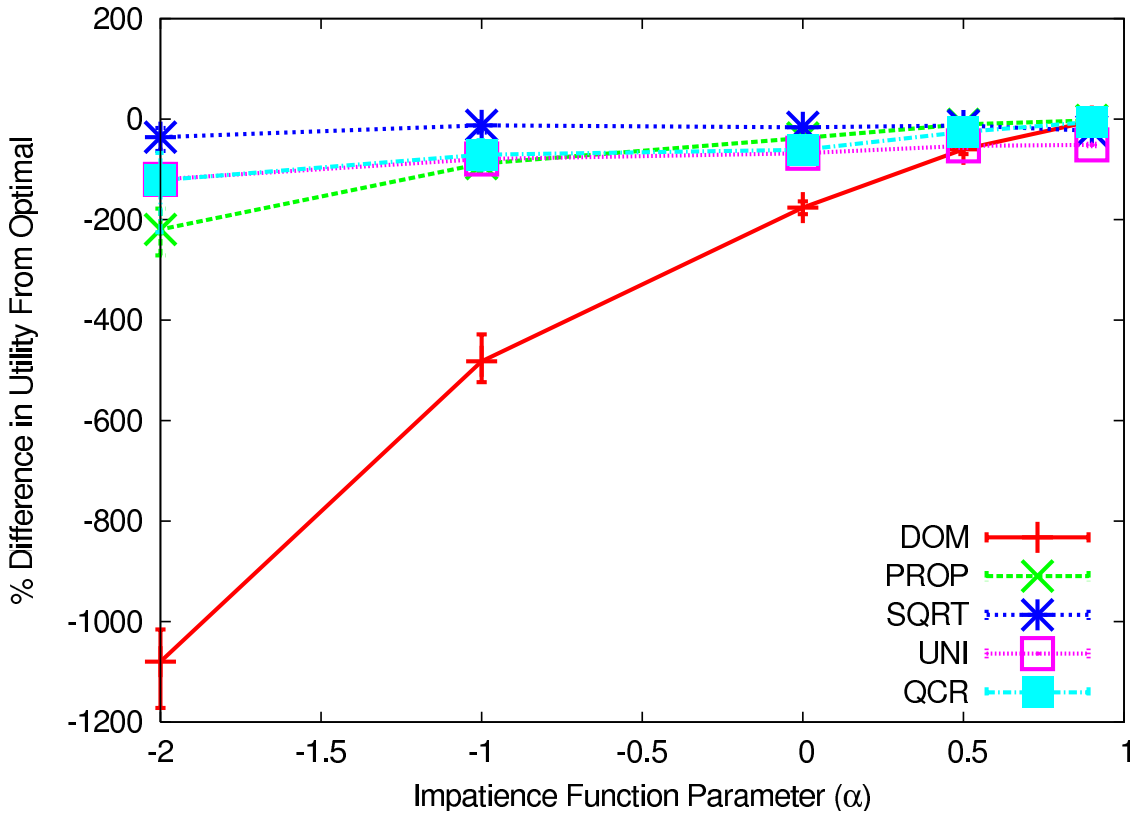
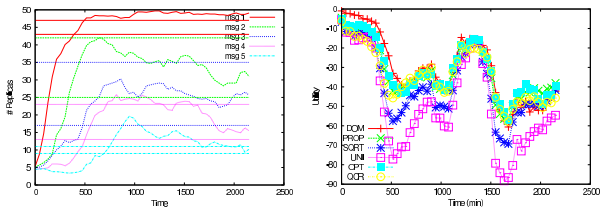
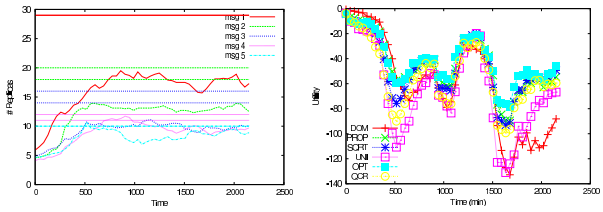


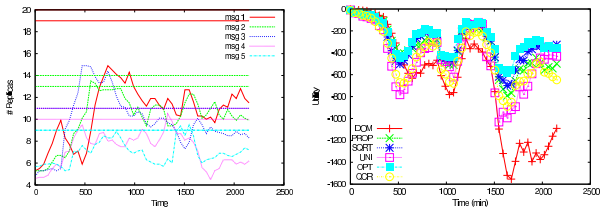
Figure A.8: VANET trace, power impatience, util vs. α.



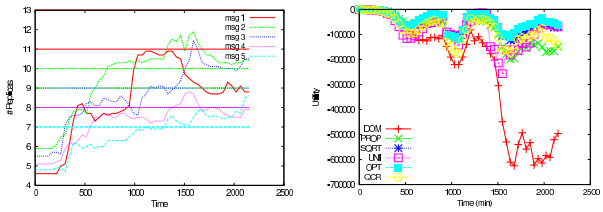
(a) MA VANET ($\alpha = 0.9$) (b) Cache VANET ($\alpha = 0.9$)



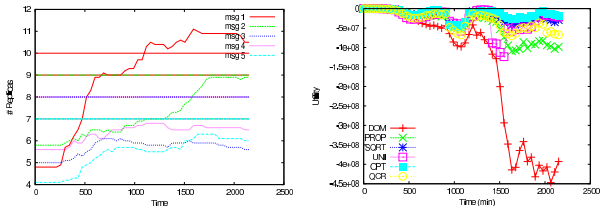
(c) MA VANET ($\alpha = 0.5$) (d) Cache VANET ($\alpha = 0.5$)



(e) MA VANET ($\alpha = 0$) (f) Cache VANET ($\alpha = 0$)

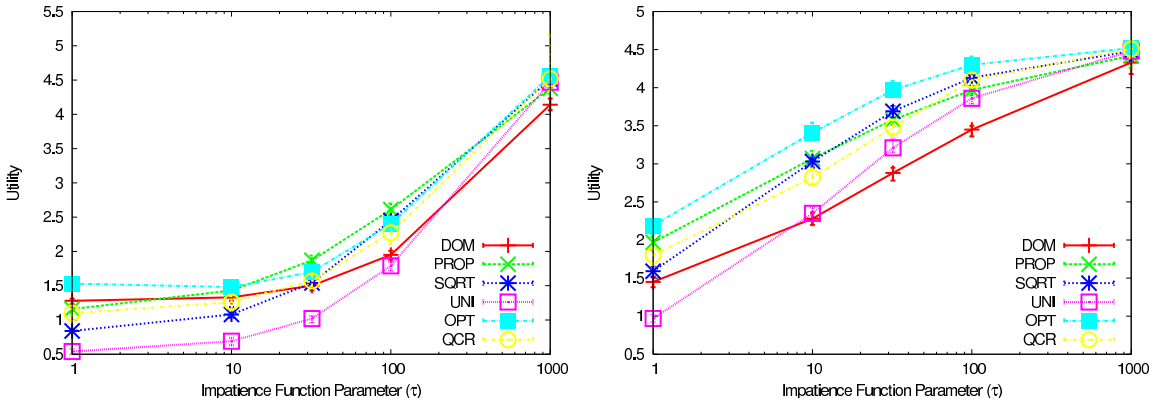


(g) MA VANET ($\alpha = -1$) (h) Cache VANET ($\alpha = -1$)



(i) MA VANET ($\alpha = -2$) (j) Cache VANET ($\alpha = -2$)

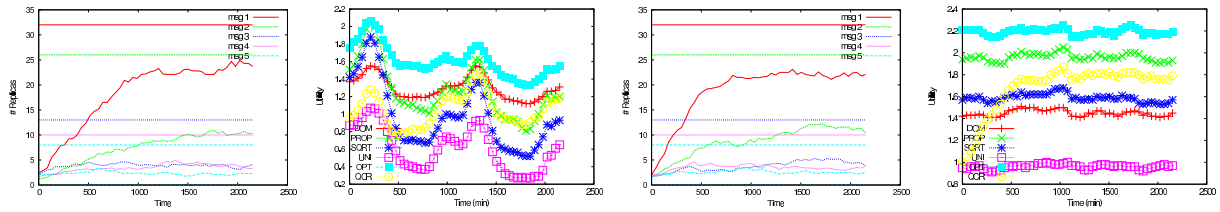
Figure A.9: VANET trace, power impatience for $\rho = 5, I = 50, N = 50, \omega = 1$.



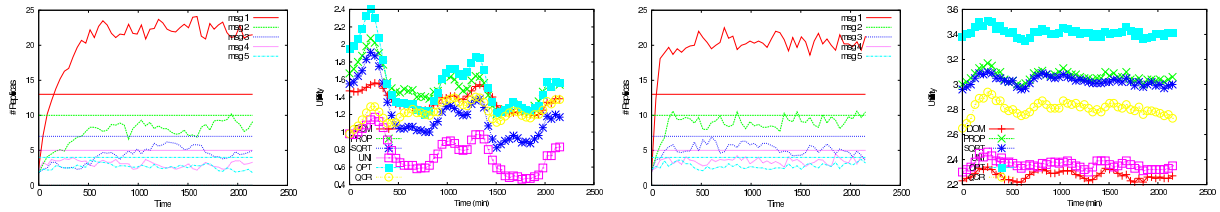
(a) Actual Trace

(b) Synthesized Trace

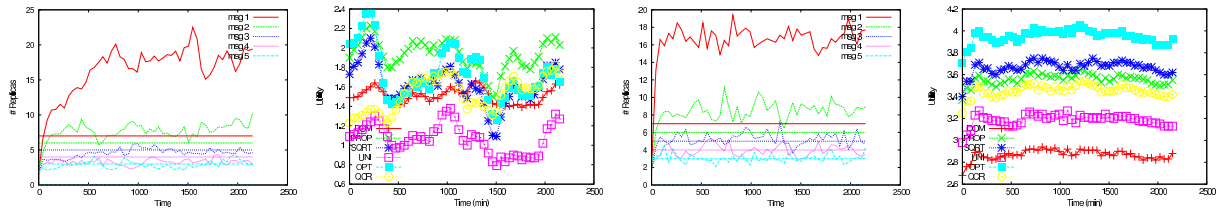
Figure A.10: VANET trace, step impatience ($\rho = 2$) - understanding impact of time statistcs on avg. behavior.



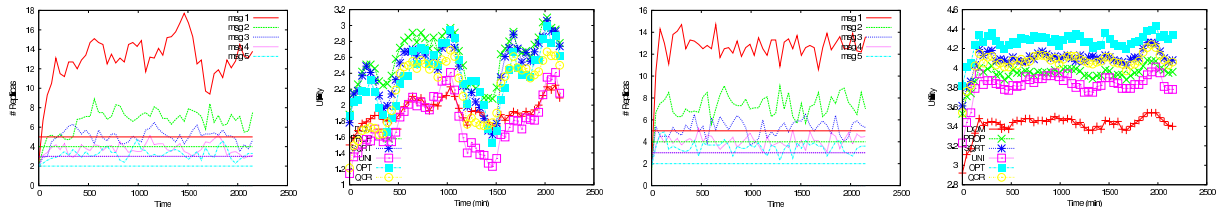
(a) Cache ($\tau = 1$) (b) Utility ($\tau = 1$) (c) (Synthetic) Cache ($\tau = 1$) (d) (Syntetic) Utility ($\tau = 1$)



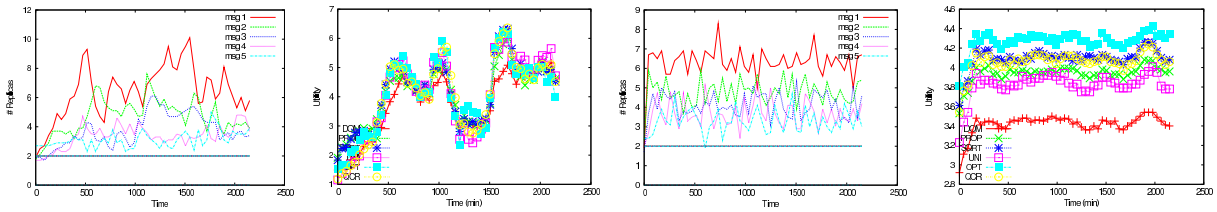
(e) Cache ($\tau = 10$) (f) Utility ($\tau = 10$) (g) (Synthetic) Cache ($\tau = 10$) (h) (Syntetic) Utility ($\tau = 10$)



(i) Cache ($\tau = 32$) (j) Utility ($\tau = 32$) (k) (Synthetic) Cache ($\tau = 32$) (l) (Syntetic) Utility ($\tau = 32$)

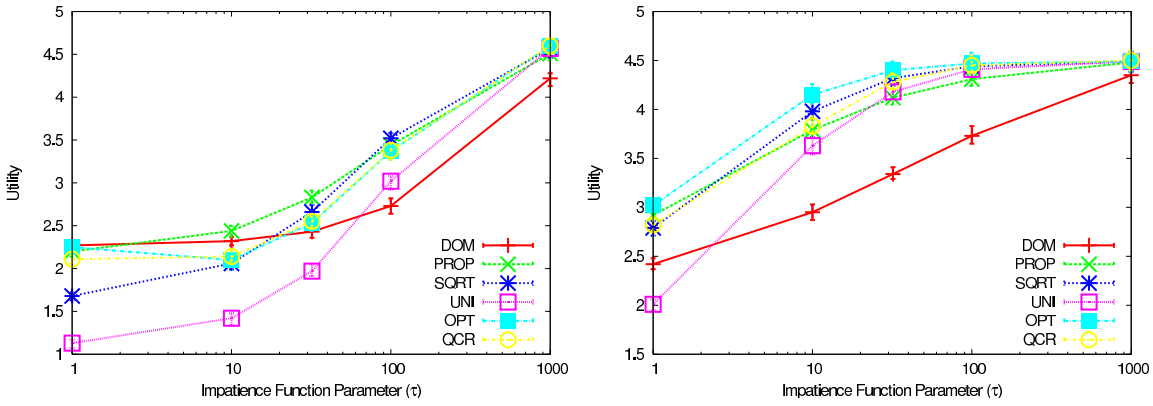


(m) Cache ($\tau = 100$) (n) Utility ($\tau = 100$) (o) (Synthetic) Cache ($\tau = 100$) (p) (Syntetic) Utility ($\tau = 100$)



(q) Cache ($\tau = 1000$) (r) Utility ($\tau = 1000$) (s) (Synthetic) Cache ($\tau = 1000$) (t) (Syntetic) Utility ($\tau = 1000$)

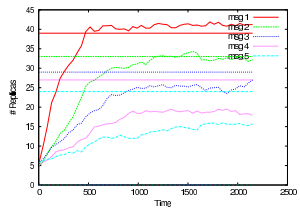
Figure A.11: VANET trace, step impatience - timewise results for $\rho = 2, I = 50, N = 50, \omega = 1$.



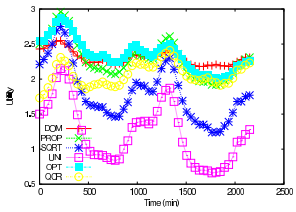
(a) Actual Trace

(b) Synthesized Trace

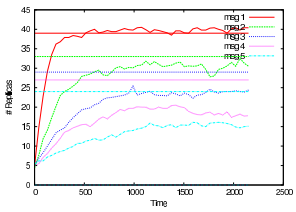
Figure A.12: VANET trace, step impatience - understanding impact of time statistcs on avg. behavior.



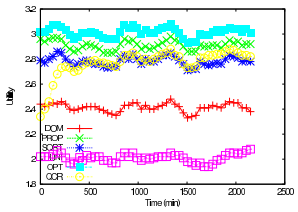
(a) Cache ($\tau = 1$)



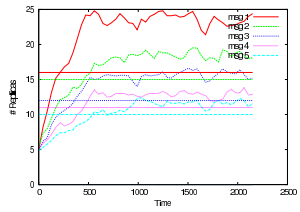
(b) Utility ($\tau = 1$)



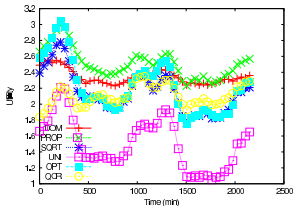
(c) (Synthetic) Cache ($\tau = 1$)



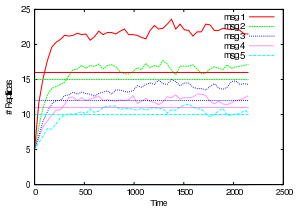
(d) (Syntetic) Utility ($\tau = 1$)



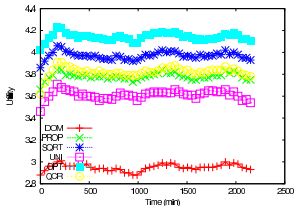
(e) Cache ($\tau = 10$)



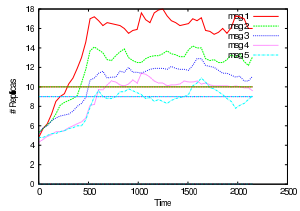
(f) Utility ($\tau = 10$)



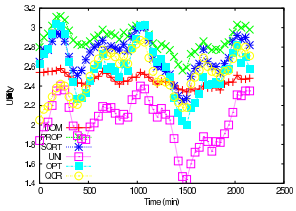
(g) (Synthetic) Cache ($\tau = 10$)



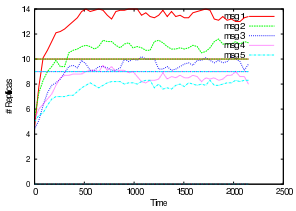
(h) (Syntetic) Utility ($\tau = 10$)



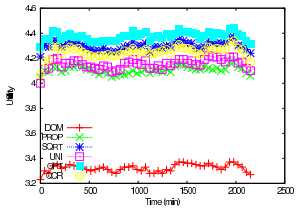
(i) Cache ($\tau = 32$)



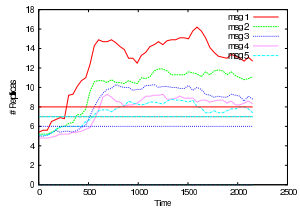
(j) Utility ($\tau = 32$)



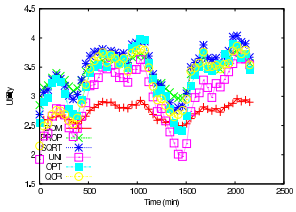
(k) (Synthetic) Cache ($\tau = 32$)



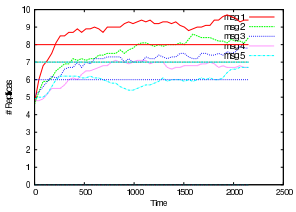
(l) (Syntetic) Utility ($\tau = 32$)



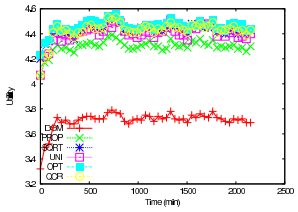
(m) Cache ($\tau = 100$)



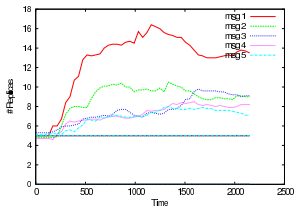
(n) Utility ($\tau = 100$)



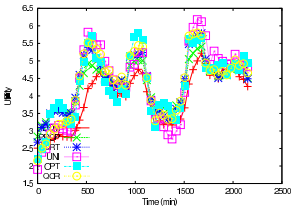
(o) (Synthetic) Cache ($\tau = 100$)



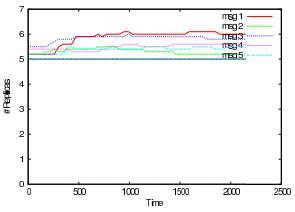
(p) (Syntetic) Utility ($\tau = 100$)



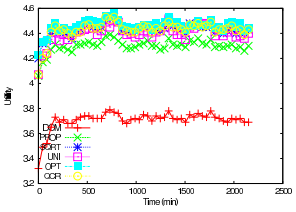
(q) Cache ($\tau = 1000$)



(r) Utility ($\tau = 1000$)

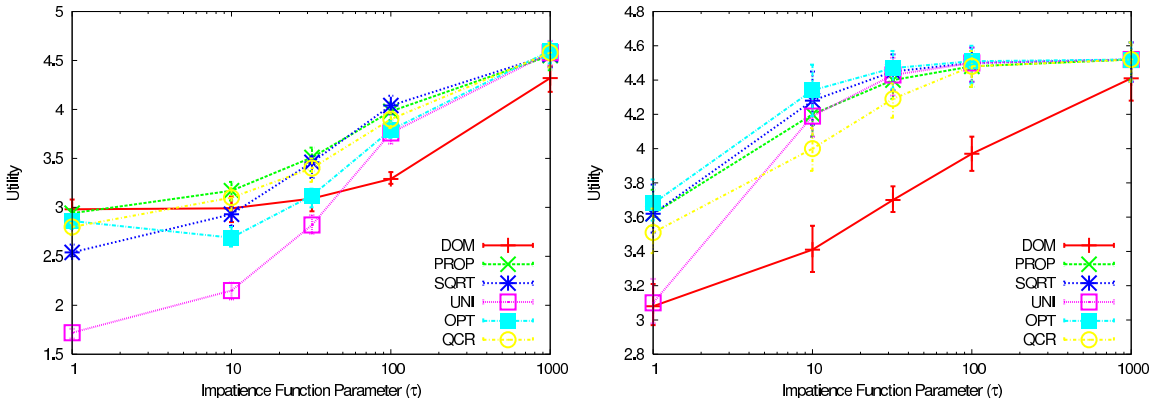


(s) (Synthetic) Cache ($\tau = 1000$)



(t) (Syntetic) Utility ($\tau = 1000$)

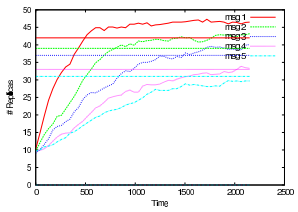
Figure A.13: VANET trace, step impatience - timewise results for $\rho = 5, I = 50, N = 50, \omega = 1$.



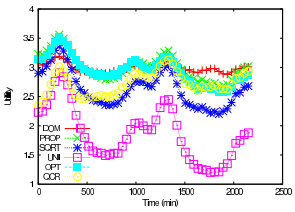
(a) Actual Trace

(b) Synthesized Trace

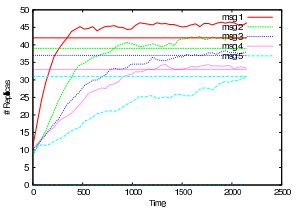
Figure A.14: VANET trace, step impatience ($\rho = 10$) - understanding impact of time statistcs on avg. behavior.



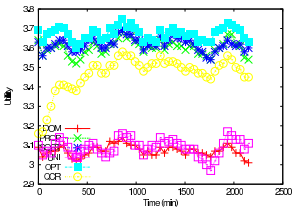
(a) Cache ($\tau = 1$)



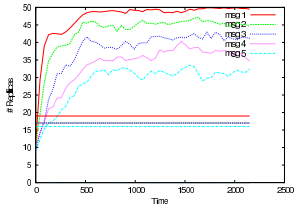
(b) Utility ($\tau = 1$)



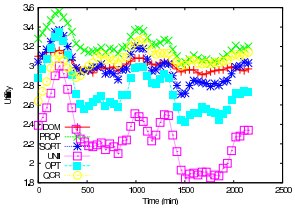
(c) (Synthetic) Cache ($\tau = 1$)



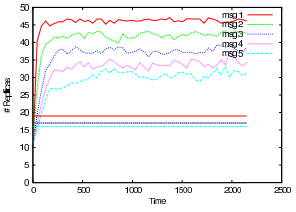
(d) (Syntetic) Utility ($\tau = 1$)



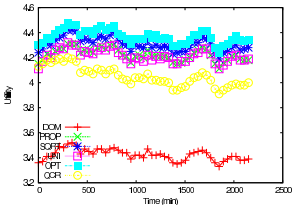
(e) Cache ($\tau = 10$)



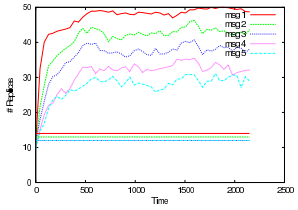
(f) Utility ($\tau = 10$)



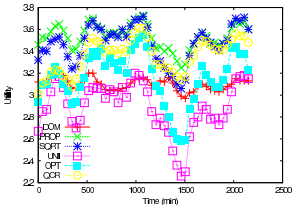
(g) (Synthetic) Cache ($\tau = 10$)



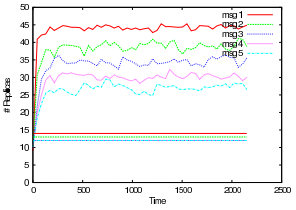
(h) (Syntetic) Utility ($\tau = 10$)



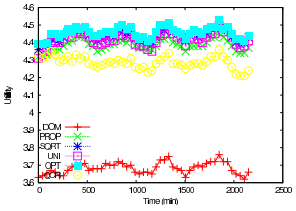
(i) Cache ($\tau = 32$)



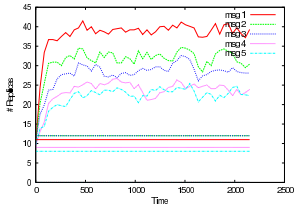
(j) Utility ($\tau = 32$)



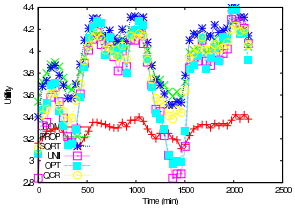
(k) (Synthetic) Cache ($\tau = 32$)



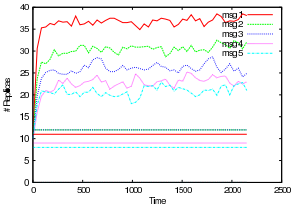
(l) (Syntetic) Utility ($\tau = 32$)



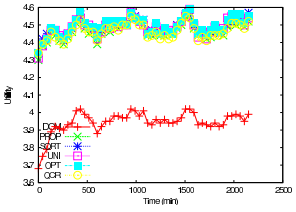
(m) Cache ($\tau = 100$)



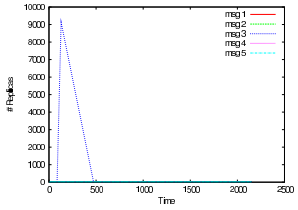
(n) Utility ($\tau = 100$)



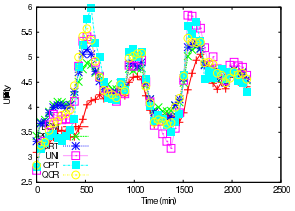
(o) (Synthetic) Cache ($\tau = 100$)



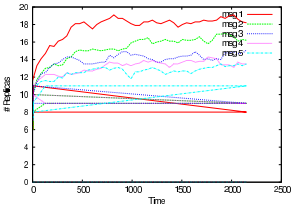
(p) (Syntetic) Utility ($\tau = 100$)



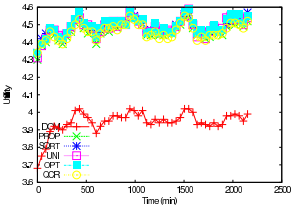
(q) Cache ($\tau = 1000$)



(r) Utility ($\tau = 1000$)



(s) (Synthetic) Cache ($\tau = 1000$)



(t) (Syntetic) Utility ($\tau = 1000$)

Figure A.15: VANET trace, step impatience - timewise results for $\rho = 10, I = 50, N = 50, \omega = 1$.