Empowering Mobile Devices Through Cyber Foraging
The Development of Scavenger, an Open, Mobile Cyber Foraging System

Mads Darø Kristensen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Empowering Mobile Devices Through Cyber Foraging

The Development of Scavenger, an Open, Mobile Cyber Foraging System

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the PhD Degree

by
Mads Darø Kristensen
June, 2010
Abstract

Cyber foraging is the opportunistic use of available computing resources by small, mobile devices. Put in another way, cyber foraging allows small, resource constrained devices, such as e.g., smart phones or small tablets, to off-load some of their resource intensive work to stronger surrogate computers that are reachable on the local network. While cyber foraging may be used to forage for a number of different resources, the focus in this thesis is purely on foraging for CPU power.

A number of cyber foraging systems have been proposed in the past decade, most of them offering a complete cyber foraging solution pursuing different goals—some offering high physical mobility within a constrained area by using pre-installed tasks and shared file systems, and others offering non-mobile usage in unprepared environments. In this thesis an approach is presented that offers both kinds of mobility; i.e., the user can be physically mobile while using cyber foraging, and when this mobility takes him into unknown territory, his applications will still be able to utilise cyber foraging.

This thesis presents the definition and design of a cyber foraging system supporting such highly mobile cyber foraging. Furthermore, the design has been realised in an implementation of a complete cyber foraging framework, and experimental data collected while working with this is presented.

An important aspect of cyber foraging is scheduling or task placement. Given a number of potential surrogate machines the mobile client must be able to select the best possible place to perform a given task. This scheduling is complicated by the heterogeneous nature of the surrogate machines, and a peer centric profile based approach collecting information about the performance of individual peers would therefore seem to be the obvious choice. While peer centric profiling works well when working within a small, constrained space, it is of no real use when the mobility of the user takes him away from that physical location. In this thesis a solution to this problem is presented, leveraging, amongst other things, task centric profiling that may be used to assess the running time of a task on any surrogate.

Cyber foraging involves mobile, distributed, and in some cases parallel computing—all fields within computer science known to be hard for programmers to grasp. This thesis presents a development model where applying cyber foraging to a resource intensive function can be done by adding a single line of code.
Acknowledgements

In this thesis I present work that I have done over a period of more than three years. During that long time numerous people have helped me, inspired me, or otherwise supported me in getting where I am today. Those people I would like to thank here.

First off, I would like to say a heartfelt thank you to my wife, who accepted that I switched to a job that meant a 33% pay cut, took care of our three little girls while I travelled to conferences and made extended visits to Finland, and who was always supportive when I needed to work long hours to meet some deadline.

Regarding the visits to Finland, I would like to thank professor Jari Porras of Lappeenranta University of Technology. I visited Jari three times during 2009 where we worked together on some of the scheduling aspects of my work—without a doubt the most productive time for me during my thesis work.

I would also like to thank my thesis supervisor Niels Olof Bouvin for dragging me back into academia, and for leaving me with a very high degree of freedom to direct my research as I saw fit. Niels has been a constant support when I needed to discuss some aspect of my research, or when I needed someone to proof read my papers. I would also like to thank my other supervisor Kaj Grønbæk.

Throughout the years a lot of source code has been written within the Locusts project, and I have not done all of it myself. In that respect I would like to thank Morten Holdflod Møller for his help implementing the first generation cyber foraging framework—the Locusts framework. Morten was not only a developer on the project, his insight and expertise have to a large degree helped shape and direct the research I have done. Thanks also goes to all my other colleagues, who have helped by discussing my work with me, proof reading papers, coming up with ideas etc.

Finally, I would like to thank Guido van Rossum, the Python community’s BDFL (Benevolent Dictator for Life), for providing me and thousands of other developers with the wonderful programming language Python. Without it I doubt I would have gotten so far in implementing my prototype system.
Preface

This report presents the work I have done within the Locusts project; a project funded by a research grant from the Danish Research Council for Technology and Production Sciences.

About the Author

During my scientific education I have conducted research on varying topics. Whilst on the bachelor and master’s level, most of which I took at the department of computer science at the University of Copenhagen (DIKU), I did work within algorithmics and database tuning which has been published as technical reports. After moving to Aarhus in 2004 I switched to Aarhus University where I finished my master’s degree within the topic of routing in mobile ad-hoc networks; a topic that is closely related to pervasive computing. The results from my master’s thesis were later on published as a peer reviewed workshop paper at a PerCom 2007 workshop.

After receiving my master’s degree in the summer of 2006 I went on a brief hiatus from academia and started working as a developer in the corporate world. This did not last long though, as I started my PhD fellowship in February of 2007.

During the latter part my PhD studies I have been working in close cooperation with professor Jari Porras from Lappeenranta University of Technology. This cooperation lead to the publishing of a book chapter in a scientific anthology and to an, as of yet unpublished, paper comparing three scheduling approaches that may be employed within cyber foraging. Our joint work took place both on-line via email and wiki communication, but also in the form of three two-week visits to Jari’s research laboratory in Lappeenranta in the first half of 2009. These were very interesting and fruitful visits, where I got a chance to see how research is carried out in other research laboratories, and to participate in that work.
Reading Guidelines

This thesis has been split into two parts. Part I is an overview of my work. This part summarises my work and relates it to relevant literature and research. Part I is structured as follows:

Chapter 1: Introduction presents cyber foraging as a computing technique, and gives background and motivation for why this technique is useful. A number of use-cases are presented along with an overlook of what is needed in order to realise such use-cases. Furthermore my research objectives within the field are presented.

Chapter 2: Research Method is concerned with describing my approach to research and discusses how this approach has been applied to the work I have done during my PhD. In this chapter my research theses are stated and I describe how these have been evaluated.

Chapter 3: Previous Cyber Foraging Research presents related work within the field of cyber foraging and discusses how it relates to my research focus and the cyber foraging systems I have developed. In this chapter the focus is on describing the work that has had inspirational value—a more stringent comparison of my work to related systems is deferred to later on in Chapter 8.

Chapter 4: Towards Highly Mobile Cyber Foraging discusses what is needed to design and implement a cyber foraging framework supporting highly mobile cyber foraging. Such a system must offer support for both the physical mobility of the user and the mobility of the computing tasks being performed.

Chapter 5: Dynamic Scheduling presents the two scheduling approaches we have worked with within the Locusts project. This chapter presents the task placement mechanisms of the Scavenger framework and the higher level task scheduling done in the Locusts framework.

Chapter 6: Development Support presents the development model of the Scavenger framework. It is shown by example how cyber foraging can be added to an existing application.

Chapter 7: Evaluation contains an evaluation of the scheduling and development support of the Scavenger framework. The schedulers are evaluated through extensive benchmarking, and the results of these benchmarks are discussed here. The development support is evaluated through the development of prototype applications using the Scavenger library.

Chapter 8: Comparison to Related Work discusses how the results described in this thesis relates to the existing work done within the area of research. It is shown how Scavenger and its approach towards cyber foraging adds a new level of mobility to the use of cyber foraging.
Chapter 9: Conclusion summarises the contributions of the work presented in preceding chapters.

Chapter 10: Future Work presents research topics and challenges for future work within highly mobile cyber foraging.

Appendices A to C instructs the reader in how to fetch, build, and work with the Scavenger framework.

Part II of this report contains four peer reviewed papers that I have published while working with my PhD, as well as a single unpublished draft that I am currently working on together with Jari Porras. Within Part I, references to these papers are on the form “Paper [I,II,III,IV,V]” and in digital editions of this report these references link directly to the paper.

**Paper I** Scavenger: Transparent Development of Efficient Cyber Foraging Applications Mads Darø Kristensen and Niels Olof Bouvin. Journal paper (conditionally accepted), Journal of Pervasive and Mobile Computing (PMC), 2010. Elsevier. This paper is an extended version of a conference paper [41] that I presented at PerCom 2010. This paper presents the Scavenger framework with a special focus on the development model used in that system. This paper also contains a detailed description of the scheduling done in Scavenger.

**Paper II** Dynamic Resource Management and Cyber Foraging Jari Porras, Oriana Riva, and Mads Darø Kristensen. Book chapter, Middleware for Network Eccentric and Mobile Applications, Chapter 16, Editors: Garbinato, Benoît and Miranda, Hugo and Rodrigues, Luís. 2009. Springer. In this paper Jari, Oriana, and I describe the process of cyber foraging in detail and present work done within the field along with a classification of this work.

**Paper III** Using Wi-Fi to Save Energy via P2P Remote Execution Mads Darø Kristensen and Niels Olof Bouvin. Workshop paper, The Sixth International Workshop on Mobile Peer-to-Peer Computing (MP2P), April 2010. IEEE. This paper presents some results we obtained by measuring the energy consumption of a modern mobile device while using Scavenger. The results show, that for this kind of device using Scavenger to offload CPU heavy tasks lead to considerable energy savings.

**Paper IV** Developing Cyber Foraging Applications for Portable Devices Mads Darø Kristensen and Niels Olof Bouwin. Conference paper, IEEE International Conference on Portable Information Devices (Portable), August 2008. IEEE. This paper presents the Locusts cyber foraging framework with a special focus on the development model used in that system. The first version of the AugIm demonstrator is presented in this paper.

**Paper V** Profile Based Scheduling of Compute Intensive Tasks in Mobile Networks Mads Darø Kristensen and Jari Porras. Unpublished draft. In this paper Jari and I describe the scheduling within Scavenger with a focus on
comparing the two different classes of profile based schedulers, task- and peer-centric, in order to show the benefits of using both kinds of profiling in a mobile setting.

Apart from the papers listed above I have published a number of other papers that, while still being relevant for the research I have done within my PhD, have not been included in this thesis. These papers are:

**Scavenger: Transparent Dev. of Efficient Cyber Foraging Applications**  
*Mads Darø Kristensen. Conference paper, Eighth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom). March–April 2010. IEEE [41].* This paper presents the Scavenger framework with a special focus on the development model used in that system. This paper also contains a somewhat detailed description of the scheduling done in Scavenger—though nowhere as detailed as the description given in Chapter 5 of this thesis. As of writing this, an extended version of this paper has been conditionally accepted for inclusion in Elsevier’s Journal of Pervasive and Mobile Computing (PMC).

**Enabling Cyber Foraging for Mobile Devices**  
*Mads Darø Kristensen. Workshop paper, 5th MiNEMA workshop: Middleware for Network Eccentric and Mobile Applications, September 2007 [38].* The first paper I published on the subject of cyber foraging. It presents the focus on highly mobile cyber foraging and discusses the use of task migration and mobile code to apply cyber foraging to such a setting.

**Scavenger – Mobile Remote Execution**  
*Mads Darø Kristensen. Technical report, DAIMI PB-587, University of Aarhus, October 2008 [39].* A technical report I wrote on the subject of peer discovery and securing a mobile code execution environment for mobile Python code. The discovery system described herein became the Presence framework that is used in both the Locusts and the Scavenger cyber foraging systems. The mobile code execution environment described in this report is also used in both systems.

**Execution plans for cyber foraging**  
*Mads Darø Kristensen. Workshop paper, Mobile Middleware: Embracing the Personal Communication Device (MobMid), December 2008. ACM [40].* This paper discusses the scheduling approach of the Locusts framework. In this paper the workflow structure of Locusts’ tasks is described in detail, and the ways that this structure may be used to facilitate scheduling and migration of tasks are presented.
# Contents

Abstract \(\text{v}\)

Acknowledgements \(\text{vii}\)

Preface \(\text{ix}\)

I Overview \(\text{1}\)

1 Introduction \(\text{3}\)
   1.1 Background and Motivation \(\text{3}\)
   1.2 Research Objectives \(\text{9}\)

2 Research Approach \(\text{11}\)
   2.1 Prototyping \(\text{12}\)
   2.2 Prototyping in the Locusts Project \(\text{14}\)
   2.3 Engelbart’s ABC Model \(\text{17}\)
   2.4 Evaluation Method \(\text{18}\)

3 Previous Cyber Foraging Research \(\text{21}\)
   3.1 Cyber Foraging Systems \(\text{22}\)
   3.2 Application Partitioning \(\text{39}\)
   3.3 Application Fidelity \(\text{41}\)

4 Towards Highly Mobile Cyber Foraging \(\text{45}\)
   4.1 Surrogate Discovery and Environment Monitoring \(\text{45}\)
   4.2 Scheduling for Mobility \(\text{47}\)
   4.3 Mobile Task Execution \(\text{47}\)
4.4 Contribution ............................................. 51

5 Dynamic Scheduling ............................... 53
  5.1 Task Placement ..................................... 53
  5.2 Workflow Scheduling ............................. 63
  5.3 Lessons Learned ................................. 66
  5.4 Contribution ..................................... 69

6 Development Support ......................... 71
  6.1 Creating a Mobile Code Task .................. 71
  6.2 Working with Larger Tasks .................... 76
  6.3 Contribution ..................................... 77

7 Evaluation ............................................. 79
  7.1 Scheduling ........................................ 79
  7.2 Developer Support ............................... 94

8 Comparison to Related Work .............. 105
  8.1 Mobility ......................................... 105
  8.2 Development support .......................... 109

9 Conclusion ............................................. 111
  9.1 Addressing the Research Theses ............. 111
  9.2 Realising the Use-Cases ....................... 113

10 Future Work ......................................... 117

A Installing and Using Presence ............ 121
  A.1 Getting the Source Code .................... 121
  A.2 Building the Binary ......................... 121
  A.3 Running Presence ............................. 122
  A.4 Installing the Presence Library .......... 122

B Installing and Configuring the Scavenger Daemon ........................ 125
  B.1 Getting the Source Code .................... 125
  B.2 Building and Running NBench ............. 125
## IV Developing Cyber Foraging Applications for Portable Devices

1. **Introduction** .................................................. 195
2. **The Locusts Framework** ............................... 197
3. **Developing Applications Using Locusts** .......... 199
4. **Demonstration** .............................................. 203
5. **Related Work** ............................................... 205
6. **Conclusion and Future Work** ....................... 205

## V Profile Based Scheduling of Compute Intensive Tasks in Mobile Networks

1. **Introduction** .................................................. 208
2. **Realising the Use Case** ............................... 209
3. **Scavenger** .................................................... 210
4. **Dynamic Scheduling** ....................................... 214
5. **Results** ....................................................... 222
6. **Related Work** ............................................... 231
7. **Conclusion** ................................................... 232
8. **Future Work** .................................................. 233

**Bibliography** .................................................. 235
Part I

Overview
Chapter 1

Introduction

This thesis addresses the requirements, design, implementation, and analysis of a cyber foraging system. Cyber foraging as a term was coined by Satyanarayanan [73] and further defined by Balan et al. [4] and it is construed as “living off the land”. More specifically, cyber foraging is the opportunistic use of available computing resources by small, mobile devices; i.e., mobile devices offloading some of their resource intensive work to stronger surrogate computers in their vicinity. A central concept here is opportunistic; the mobile device should be able to use surrogates when available, but when no such surrogates are available it should still be able to solve its tasks itself, so that the application does not cease to function when the mobile device is on its own. When on its own the mobile device may chose to perform a task at a lower fidelity, i.e., at a lower quality, a concept that has been worked on within the Odyssey system by Noble et al. [55].

In this introduction I will motivate the need for cyber foraging and for frameworks supporting this computing technique, and discuss the research objectives we have been working with within the Locusts project.

1.1 Background and Motivation

Mobile computing devices, such as the smart phone, are becoming ubiquitous and an increasing number of users are carrying such a device at all time. With the rapid improvement in display size, network connectivity, and input methods these devices have seen over the last couple of years, the smart phone is quickly advancing to become the full-fledged personal computing device of choice [7]. However, although relatively powerful, mobile devices will always be constrained in terms of physical size, thus leading to limitations in their computing and communication capabilities, battery lifetime, as well as screen and keyboard size. These constraints inhibit mobile devices from fully sup-
porting increasingly demanding mobile applications. Furthermore, although processing capabilities have followed Moore’s law for the last 30 years, the more critical resource on mobile devices is battery energy density, which has shown the slowest trend in mobile computing [59].

Modern smart phones, such as Nokia’s N900, HTC’s Android phones, or Apple’s iPhone all have large touch displays, high speed Internet connections, and some of them even have full QWERTY keyboards. Given such PC-like features it is only natural that users want to be able to do the same with their mobile phone, as they have been used to do on their PCs. Performing heavy tasks, such as e.g., simple image manipulation, is not feasible on even modern mobile devices though. While some devices may have the CPU power to perform these operations within a reasonable time frame, the delimiting factor remains the energy capacity. This is where pervasive computing techniques such as cyber foraging steps in; by leveraging otherwise unused resources at computing machinery in the vicinity, mobile devices may be able to perform resource intensive work much faster and without draining their battery.

1.1.1 Use-Cases

The uses of cyber foraging are many; as a computing technique it may be applied to many different use scenarios. In the following I will focus on its use in pervasive computing as that has been the focus in the Locusts project.

Image Manipulation

One area where the technological capabilities of modern smart phones do not match is this: all such devices have high quality built-in cameras capable of taking at least 4-5 megapixel photographs—some, such as Nokia’s new N8 smart phone, even taking 12 megapixel photographs. Through their large touch screens they also have the input capabilities needed to perform basic image manipulation on the photos taken, but they lack the raw processing power to actually perform such operations on these massive amounts of data. Even common operations such as sharpening or resizing an image is a very heavy task to perform on a mobile device. Using cyber foraging this use-case can easily be supported. Consider the following use-case:

A tourist is sitting in a café going through the pictures she has taken earlier in the day. The pictures were taken using the multiple megapixel camera in her smart phone, and she is browsing them to select the ones that she wants to upload to her online storage account, so that her friends and family may see them. Before uploading them she applies some filters to them—some need sharpening, others red-eye reduction, and yet others may need their brightness/colour/contrast adjusted. All of these operations are applied only on small previews of the photographs on her smart phone,
but when she presses the “apply” button cyber foraging is used to apply the image operations to the actual images. Her mobile device automatically scans its environment, finding a couple of surrogates provided by the café and some laptop computers owned by other customers, and quickly offloads both the image operations and the uploading of the resulting pictures to these devices, leaving her phone free for her to use—and, more importantly, leaving her phone’s battery at an acceptable level so that she may use it for the rest of the day. If no surrogates are available at the café, her mobile device will ask her to choose between performing the operations locally or queueing the operations until later on as surrogates become available.

This use-case was taken from a paper I presented at PerCom 2010 in Mannheim, Germany [41]. The image manipulation use-case is one we have worked on throughout the Locusts project, and we have two working demonstrators running on the two cyber foraging systems we have developed.

Continuous Speech Recognition

Speech recognition is in use on some mobile devices already—e.g., on the Google Nexus One that offers speech to text functionality for messaging as well as voice controlled navigation on the map application. By utilising cyber foraging such speech recognition could be brought into the domain of wearable computing [48]; i.e., to even smaller, wearable devices. Even for the devices that are capable of performing the recognition the use of remote resources would have the potential of yielding results of much higher quality while using less energy on the mobile device. Consider the following use-case:

A doctor doing house calls is wearing a small headset (similar in size and form to the well-known Bluetooth headsets for mobile phones). Using this headset he is able to access the patient’s journal, having parts of the journal read into his ear upon request, and he is also able to enter new information about his patient into the journal. This means that the headset is faced with the difficult task of continuous voice recognition. The headset is unable to perform this translation task itself, so instead of performing the actual voice recognition it merely records the utterances made by the doctor. Whenever the headset comes within range of usable computing resources (surrogates) it forwards some of the recordings to these machines who respond by returning the translated text. If the surrogate has an Internet connection it may even be given the task of updating the patient’s journal directly. After translation the headset may discard the recording and thus free storage for additional recordings.
Chapter 1 Introduction

This use-case was taken from a paper I presented at a MiNEMA workshop in Magdeburg, Germany in 2007 [38]. Speech recognition is a use-case that has been used by numerous cyber foraging systems. At an early stage of developing the Locusts cyber foraging framework we had a small cyber foraging enabled speech recognition demonstrator, but some problems with the speech recognition software made us abandon this demonstrator early on.

Augmented Reality

Another use-case is augmented reality systems. In augmented reality the vision is often the one of the user wearing a pair of glasses that are capable of performing complex feature recognition, while maintaining a head-up display that augments the users vision. Without the use of something like cyber foraging, such a scenario is not feasible, unless the user is asked to wear a blade server on his back. Consider the following use-case:

Ron has recently been diagnosed with Alzheimer’s disease. Due to the sharp decline in his mental acuity, he is often unable to remember the names of friends and relatives; he also frequently forgets to do simple daily tasks. He faces an uncertain future that’s clouded by a lack of close family nearby and limited financial resources for professional caregivers. Even modest improvements in his cognitive ability would greatly improve his quality of life, while also reducing the attention demanded from caregivers. This would allow him to live independently in dignity and comfort for many more years, before he has to move to a nursing home.

Fortunately, a new experimental technology might provide Ron with cognitive assistance. At the heart of this technology is a lightweight wearable computer with a head-up display in the form of eyeglasses. Built into the eyeglass frame are a camera for scene capture and earphones for audio feedback. These hardware components offer the essentials of an augmented-reality system to aid cognition when combined with software for scene interpretation, facial recognition, context awareness, and voice synthesis. When Ron looks at a person for a few seconds, that person’s name is whispered in his ear along with additional cues to guide Ron’s greeting and interactions; when he looks at his thirsty houseplant, “water me” is whispered; when he looks at his long-suffering dog, “take me out” is whispered.

This use-case was described by Satyanarayanan et al. [75] in 2009.
1.1 Background and Motivation

Other Uses

The three use-cases presented in detail here are but a small sampling of the ones found in the related literature. Uses of cyber foraging in literature include but are not limited to:

- Video streaming [55, 21].
- Web browsing [55, 54, 21].
- Speech recognition [55, 77, 25, 22, 21, 6, 38].
- 3D rendering [54, 6].
- Selective, application specific fetching of large data sets [15, 21].
- Data mining [25].
- Document preparation [22].
- Natural language translation [22, 6].
- Facial recognition [6, 75].
- Text to speech [6, 75].
- Optical character recognition [6].
- Data staging [4].

1.1.2 The Need for a Framework

In the preceding sections a number of use-cases have highlighted the need for cyber foraging as a computing technique. There is a long way from knowing that mobile applications may benefit from opportunistic use of remote execution, to actually employing such a technique in practice. Cyber foraging entails an excessive amount of scaffolding that must be in place before remote execution can take place. In [Paper II] Porras, Riva, and I describe the process of cyber foraging in detail as consisting of six steps: surrogate discovery, application partitioning, cost assessment, trust establishment, task execution, and environment monitoring; as depicted in Figure 1.1.

Surrogate discovery In order to use resources the mobile device has to be aware of their existence. Thus a service announcement framework suitable for mobile applications is needed.

Application partitioning When working with cyber foraging enabled applications, the developer has to identify the parts of the application that may be targets for remote execution. In doing so a framework is needed in which these remote executable tasks can be expressed.
Cost assessment Doing remote execution comes at a cost—before execution can commence the input data must be moved to the surrogate, and upon completion the output data must be fetched. Therefore a scheduler is needed that can assess the costs of doing remote execution for the specific task in the current environment. Sometimes local execution is the better choice.

Trust establishment A host of trust/security issues are present when working with cyber foraging. Mobile users must be able to trust that a surrogate does not misuse the data that it is entrusted with, and that it returns truthful results. The surrogate, on the other hand, must be able to trust that the client does not misuse its resources or snoop on private data stored on the surrogate. There is also the risk of eavesdropping etc.

Task execution Some mechanism that enable the remote execution of tasks is of course needed; e.g., by use of pre-installed RPCs or mobile code.

Environment monitoring A client must continually monitor the state of the surrogates surrounding it, so that it may make an informed decision when scheduling a task.

As should be apparent by now, cyber foraging is an involved technique with many facets that must be considered. Because of this, it is clear that a framework is needed if cyber foraging is ever going to be used outside of academia. Furthermore, a framework is needed in order to unify the cyber foraging enabled applications so that surrogates may be able to support multiple applications.
1.1.3 Why Not Cloud Computing?

In recent years the term cloud computing has become increasingly popular, so it is only natural to ask the question: “Why not use cloud computing instead of bothering with discovering and using local computing resources?” There are multiple answers to this question:

**Cost** In most cases, when using a mobile device accessing the Internet comes at a cost, which accessing a locally available Wi-Fi network does not.

**Latency** While network bandwidth when connecting to the Internet has been increasing rapidly over the past decade, the latency has not been brought down with the same alacrity (due in part to the speed of light). This makes cloud computing infeasible for many classes of problems where response time is crucial; e.g., as noted recently by Satyanarayanan et al. [75].

**Bandwidth** While mobile Internet speeds are increasing they are still orders of magnitude slower than the speed of local WLAN networks. This renders cloud computing useless in data intensive scenarios such as image manipulation, speech recognition, etc. This point is also being made by Satyanarayanan et al. in [75].

**Energy efficiency** Finally, by using the unused resources of machines that are already running, cyber foraging uses little or no extra power, whereas cloud computing applications are run in large data centers where hundreds (if not thousands) of dedicated servers are running 24/7.

Whereas cloud computing on its own is not directly usable in the scenarios catered for by cyber foraging, it could be used in some scenarios to complement cyber foraging. One could imagine scenarios where task schedulers would consider local execution, execution at available surrogates, and execution in the cloud when choosing where to perform a task. In that way cloud computing could be employed when no locally available surrogates exist. Satyanarayanan et al. [75] have recently proposed such an architecture, where cloud services are replicated onto local surrogates—which are then called cloudlets.

1.2 Research Objectives

As have been shown in the preceding section, cyber foraging is a complex computing technique containing many aspects that could be research projects in their own respect. Because of this we had to choose only a few aspects to focus on in the Locusts project—foci that have moved only slightly as we moved along. Initially we chose to focus on the mobility aspect of cyber foraging [38], as none of the existing systems seemed to support the kind of mobility we were after. After doing initial prototyping and research within the subject matter this focus shifted to being a focus on task scheduling [40] and execution, since these
aspects are the ones most determinant for the mobility of a cyber foraging system. Another focus, that has been with us all along, is that of developer support [Paper IV, Paper I]. It was immediately clear, that building mobile, distributed, parallel applications was a very hard thing to do, and we thus wanted to ease the process of developing cyber foraging enabled applications as much as possible. The research questions we wanted to ask were thus:

**Mobility** What is needed to support highly mobile cyber foraging—highly mobile meaning both that the user of the mobile device is physically mobile and that this mobility leads the user into unknown environments, i.e., environments where no previously known surrogates are available.

**Scheduling** How can a scheduler be designed that supports this high degree of mobility while still utilising multiple surrogates in an efficient manner?

**Development** How can we simplify the process of developing cyber foraging enable applications? Can it be done to a degree where novice programmers are capable of adding cyber foraging?

Choosing to focus on one or two aspects of a system of course means that other aspects are not considered. In Locusts we needed to develop a full cyber foraging system, so that we could test the theses we put forward about the mobility, scheduling, and development support. This meant, that we had to implement a discovery mechanism, a mobile code execution environment, and a plethora of other software components. In doing so we have done some research on discovery systems as well as the security aspects of executing mobile Python code. While not our main focus, this work was still of great interest to us and has therefore been published as a technical report [39].
Research Approach

Throughout my work I have taken an experimental approach to research. This means, that experiments and practical results supporting my theories have been favoured over purely theoretical discussions. To put it in my supervisor Niels Olof Bouvin’s words: “In experimental computer science we only do three things: 1) We think it, 2) we build it, and 3) we evaluate it.”. To this end I have implemented an entire cyber foraging system, starting from the bottom with a custom mobile code execution environment and service discovery, and advancing all the way to the top to a development library that application programmers can use to implement cyber foraging enabled applications, and prototype applications developed using this library.

In developing this cyber foraging system I have had two main foci; one user centred, where the focus is on supporting the development process of application programmers, and the other system centred, where the focus is on the inner workings of a cyber foraging system, especially the scheduling process within such a system. Within both of these, the prototype cyber foraging system has been used to conduct experiments.

Taking an experimental approach to research means that we, not unlike the practitioners within the classical sciences like e.g., chemistry and physics, spend a lot of time creating an experimental setup—the experimental setup in computer science being the software that is developed and the hardware it is running on. After building the setup we test it and observe how it acts under different circumstances; the aim of these tests being testing the theses we have about the setup.

The theses we have been working with within the Locusts project are:

**System support** It is possible to build a cyber foraging system supporting efficient use of remote resources in a mobile setting where 1) the user is physically mobile while execution of tasks takes place, and 2) this physical mobility takes the user into unknown territory necessitating a mobil-
ity of the tasks themselves.

**Performance gains** Even with having to cope with such a high level of mobility, such a system will be usable on small, mobile devices and will yield significant performance improvements enabling the mobile devices to take on tasks otherwise deemed unsuitable on such platforms.

**Development support** Such functionality can be presented to developers in a form such that even novices with little or no experience with distributed computing can build efficient, cyber foraging enabled applications.

**Energy efficiency** By using such a cyber foraging framework mobile devices will be able to save energy.

In order to test these theses a complete cyber foraging system has been prototyped. How I have approached this prototyping is the subject of Section 2.1.

The work done within the Locusts project is not focused on solving a single problem; rather it is focused on providing a software architecture, system, and development model that provides the tools needed to solve a multitude of problems involving resource intensive work. Engelbart [17] describes how building such tools supports the A-, B-, and C-level work of an organisation—in Section 2.3 I briefly analyse how the work presented in this thesis fits into Engelbart’s ABC Model of Organisational Improvement.

Finally, in Section 2.4 the methods used when evaluating the theses are presented.

### 2.1 Prototyping

As stated above, prototyping has been a very important part of the research work I have conducted during my PhD studies. By implementing prototype systems I have been able to test my theses with regards to both system and developer support.

In [24] Floyd discusses the use of prototyping within software engineering. According to Floyd, prototyping consists of four phases: *functional selection, construction, evaluation, and further use.*

**Functional selection** refers to the functionality chosen for the prototype. In general, the chosen functionality should be a subset of the functionality one would expect to exist in the final product. Within functional selection Floyd identifies two differing ways of prototyping: *vertical prototyping*, where the implemented functionality is presented in its intended final form, but only a small subset of the total functionality is included. Alternatively *horizontal prototyping* can be employed, where the entire functionality is represented, but the functions are not implemented in detail (often omitting aspects such as error handling, security, etc.).
Construction refers to the actual implementation of the prototype. The effort involved constructing a prototype should be much smaller than that involved in building the final product.

Evaluation is the phase where the implemented prototype is tested and evaluated in order to inform the development process of the final product. The evaluation could be said to be the raison d'être of prototyping.

Further use may vary depending on the kind of prototype being developed. In some projects the prototype is used exclusively as a learning vehicle, and is thus thrown away after prototyping. Other prototypes may be matured and then used fully or partially as a component in the final product.

Prototyping may be employed for a number of reasons; as a vehicle for learning about a new problem domain, as a way to develop a product in close cooperation with the users by providing them with early prototypes or mock-ups, or as a part of an iterative development process where early, executable prototypes are in use as a part of the development strategy, as is the case in e.g., agile software development [8]. Floyd works with three different classes of prototyping: exploratory prototyping, experimental prototyping, and evolutionary prototyping.

Exploratory prototyping has an emphasis on clarifying requirements and desirable features of the product under development. Working with exploratory development is e.g., useful when diving into a new problem domain, where the developer and architects need to learn about the capabilities of the chosen tools as well as what the requirements of a final product could be. Exploratory prototyping is just that: exploratory. Therefore the process of prototyping becomes circular, as depicted in Figure 2.1(a), switching rapidly between functional selection, construction, and evaluation.

Experimental prototyping can be used when the problem domain and requirements are known. In this case the prototype is used to determine the adequacy of the proposed solution before investing the time necessary to develop the finished product. Working with experimental prototyping is thus a more linear process, as depicted in Figure 2.1(b), where the functional selection is known beforehand, and prototyping is used to show the viability of the chosen solution.

Evolutionary prototyping is arguably not even real prototyping. Rather evolutionary prototyping in Floyd’s terminology refers to incremental development, a development model that is in extensive use today. In Floyd’s words, evolutionary prototyping is “… adapting the system gradually to changing requirements, which cannot reliably be determined in one early phase.”. Evolutionary prototyping is depicted in Figure 2.1(c).

A project need not choose just one of these classes of prototyping; in fact
most projects would benefit from using all three classes in different parts of the development process.

2.2 Prototyping in the Locusts Project

In the Locusts project we have made heavy use of both exploratory and experimental prototyping. Table 2.1 gives an overview of how prototyping has been employed in the Locusts project.

In the beginning of the project a lot of time was spent identifying what was needed to build a cyber foraging system, and at the same time reading up on related work within the field. Once we had a clear idea of what the other systems were capable of, we soon identified two main weaknesses of all other cyber foraging systems: development support was very weak or completely lacking, and none of the proposed systems supported the kind of mobility that we sought after.

After gaining this insight into the problem domain the initial exploratory prototyping began. A bit untraditionally perhaps, we started out by implementing a vertical prototype of the envisioned Locusts daemon and library. Parts of the system were replaced by placeholders; e.g., the execution environment which was simulated by simply spawning a new Python interpreter each time a task was performed, but the skeleton of the desired system was quickly built.
2.2 Prototyping in the Locusts Project

<table>
<thead>
<tr>
<th>Exploratory prototyping</th>
<th>Locusts daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Locusts library</td>
</tr>
<tr>
<td></td>
<td>Scavenger library</td>
</tr>
<tr>
<td></td>
<td>Mobile code execution environment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experimental prototyping</th>
<th>Scavenger daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scavenger library</td>
</tr>
<tr>
<td></td>
<td>Presence (service discovery)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evolutionary prototyping</th>
<th>Scavenger daemon.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scavenger library.</td>
</tr>
</tbody>
</table>

Table 2.1 Use of prototyping in the Locusts project.

After building this first prototype system the first demonstrator application, the augmented image manager (AugIm), was implemented using the Locusts library. This prototype was tested with regards to both performance and development support. While the demonstrator did show very large speed-ups when offloading work to surrogates, it quickly became apparent that the mobile code execution environment needed to be replaced. Developing this vertical prototype of an execution environment, which was dubbed Scavenger, was begun in parallel with the development of the Locusts daemon and library. The same was the case with the Presence mobile service discovery system; the functionality was initially hard coded into the Locusts daemon, but in order to make this component more flexible and easier to test a separate vertical prototype was started.

Having gone through the exploratory prototyping of Scavenger, the mobile code execution environment, it became clear that our focus in the scheduling parts of the Locusts daemon had become skewed. Some design decisions, such as representing tasks a directed, acyclic graphs and keeping the scheduling within the daemon, had complicated the overall system design to a unnecessary degree. At that point a hard decision was made; the entire Locusts daemon and library code base was left behind, and a new system based entirely around Scavenger was built. This does not mean that the work with the Locusts daemon and library was wasted; by no means. Because of the extensive lessons learnt working on these exploratory prototypes, the development of the Scavenger daemon and library was finished in a matter of months, yielding a system that performed much better with regards to efficiently using available computing resources, and also a system that was orders of magnitude easier to
develop for.

Because of the complexity of the Locusts cyber foraging framework, daemon and development library alike, these prototypes were abandoned and all development focused on the Scavenger system. It is therefore only the Scavenger daemon and library that have entered the evolutionary prototyping phase. These software components are still in active development today—being worked on by four different developers at the moment.

2.2.1 Locusts, Locusts and Scavenger

In order to clarify what is meant when referring to Locusts and Scavenger throughout this thesis consider these three distinctions:

**The Locusts Project** Locusts is the name of the research project that my supervisor Niels Olof Bouvin started in February 2007. My research has been funded by this project and I have thus worked on realising the ideas put forward within that project. Whenever I refer to Locusts as a project I have written “Locusts project” so as to avoid confusion with the Locusts system. Locusts, by the way, are swarming grasshoppers that travel great distances, rapidly stripping fields of their crops, i.e., living off the land.

**The Locusts Framework** We named the first cyber foraging system we developed after the project. As mentioned above, this framework was abandoned and the Scavenger framework took its place. Whenever I refer to the Locusts framework I write either “Locusts framework” or “Locusts cyber foraging framework” to avoid confusion with the Locusts project.

**The Scavenger Framework** Scavenger was initially the name of the mobile Python code execution environment created for use within the Locusts framework. While developing Scavenger a small RPC front-end was attached to it, in order to ease the debugging process, and it was soon realised that this light weight approach towards cyber foraging has some major benefits. The Scavenger execution environment therefore became the Scavenger cyber foraging framework. Whenever I use the name Scavenger throughout this text I am referring to the Scavenger framework.

The main difference between the Locusts framework and the Scavenger framework is their approach towards task scheduling. Locusts represents a task as a directed, acyclic graph consisting of interconnected subtasks (a workflow representation), and tries to do a dynamic min-min scheduling of such tasks. Scavenger, on the other hand, has no view of the larger task; it merely places the task(s) currently available for execution at the surrogate(s) most capable of performing it at that point in time. From this description it may seem that Locusts would be able to offer the best scheduling, given the extra information available to it, but as will be shown later on this is not necessarily the case. So, conceptually, the only difference between the Locusts framework and the
Scavenger framework that the reader is encouraged to remember, is the difference in the scheduling.

2.3 Engelbart’s ABC Model

To shed light on why research on software architectures and development models, such as the research done within the Locusts project, is important, I will in the following briefly analyse the development work done within the Locusts project using Engelbart’s ABC Model of Organisational Improvement [17].

Engelbart’s ABC model talks of work within organisations as being performed on three levels: A, B, and C.

**Level A** A-level activities are the core business activities of the organisation. In order to perform these activities a number of high level capabilities are needed, and these capabilities are supported by an Augmentation System (an idea that stems from Engelbart’s work on the Augment system [16]). The A-level work is the bread and butter of the organisation, and being able to perform this work efficiently is thus paramount.

**Level B** B-level activities focus on improving an organisation’s ability to perform its A-level activities. This is e.g., done by improving the Human-Tool Augmentation System, and the goal is to streamline the A-level work so that it may be performed more efficiently.

**Level C** C-level activities focus on improving an organisation’s ability to improve its B-level activities. The aim here is to bring down the improvement-cycle time, so that the organisation is continually improving its internal tools.

Applying the ABC model to computer science, Nürnberg [56] argues that work within this field is mostly concerned with B- and C-level activities. To most organisations the computer is first and foremost to be seen as a tool to accomplish some, other primary task. It is also at these levels the work performed with the Locusts project is situated—in fact most of the work is situated at the lowest level, C.

Development tools, libraries, software architectures and the like are all created on in order to support more efficient development of the Human-Tool Augmentation System; i.e., they are created to ease the development on the B-level that is needed to support the activities being done on the A-level. In the Locusts project the focus has been on these C-level activities, and we have created two cyber foraging systems, Locusts and Scavenger, and two accompanying libraries encompassing our proposed development model.

The only B-level activities we have done is the development of demonstrator applications. At this level we have developed two versions of the Augmented Image Manager: one using the Locusts daemon and library and run-
ning on an N800/N810 Nokia Internet Tablet, and a more recent version for the N900 Nokia smart phone. Apart from that a Mandelbrot fractal viewer for the N800/N810 and numerous scripts, testing the viability of the development model and the efficiency of the underlying cyber foraging system, have been developed.

It follows from Engelbart’s ABC model, that doing research on the C-level, e.g., by inventing new architectures and models, improves and/or eases the work done on the B-level, and thus also in the end the primary, A-level activities that are supported by the tools created on the B-level. Furthermore, the work done on the C-level is very likely to be usable to many organisations—not just one, which may very well be the case with B-level activities. We therefore find, that doing research on software architectures and development models is very important, which is why we focus on this in the work presented here.

2.4 Evaluation Method

As mentioned in the beginning of this chapter, the focus within the Locusts project has been on both developer support and on creating an efficient, lightweight cyber foraging system. Evaluating whether or not one has been successful with regards to these two objectives must be done in completely differing ways.

I have a background in experimental system development and algorithmics, having written my master’s thesis on results obtained simulating MANET\textsuperscript{1} routing protocols and having written technical reports on benchmarking various algorithms/systems throughout my studies. I am thus no stranger to the process of evaluating system performance. Evaluating the effects of a development library is another case altogether. In order to test the effects of using a development library to solve some task, a lot of resources are needed: first and foremost a number of developers are needed, and these should not be experts within the field, because that would skew the results etc. Within my PhD I have not had these resources available, and I have therefore not been able to perform as thorough an evaluation of developer support as I would have liked to. This does not mean that this aspect has not been evaluated at all though—more on that shortly.

2.4.1 Evaluating the Cyber Foraging System

In performing my experimental evaluation of both Locusts and Scavenger, I have been inspired by the work by Kurkowski \textit{et al}. [45]. The Kurkowski paper is on MANET simulations but many of the points it makes may be applied to most experimental system evaluation.

\textsuperscript{1}Mobile Ad-Hoc Network
One of the most important aspects when performing and reporting on results from system benchmarks is reproducibility; a person reading about your results should be able to reproduce your test results using his or her own machinery—and this should be possible with as little overhead as possible. Since we are working within computer science, our test setups are software and hardware. Reproducing hardware may of course be hard, but by using common hardware components there is a good chance that a peer may be able to reproduce the hardware environment to an acceptable degree. To this end, all papers published that report on experimental results using Locusts or Scavenger have detailed specifications of the hardware devices used whilst performing the tests.

The software setup is easier to reproduce—if the developer has designed towards this from the beginning. Far too many (almost all) research papers on experimental system development report on results obtained using systems that are: 1) not publicly available, 2) tailor made to suit only the exact setup of the authors’ test environment, or 3) near impossible to setup, configure, or compile. In the Locusts project, and especially with the Scavenger system, we have made a virtue of designing our system, both with regard to software and hardware, so that it may be possible for others to scrutinise our results. The Scavenger daemon and development library are available as open source software as is the Presence service discovery component. Both are hosted at Google Code: http://code.google.com/p/scavenger-cf/ and http://code.google.com/p/presence-discovery/. Links to these pages have been included in papers reporting on results obtained using the systems. The test scripts used when performing benchmarks for a paper are also available in the source repositories at Google Code, so that everything needed to reproduce the tests should be readily available.

Another important factor when performing benchmarks is of course how the results are measured. In our tests the measurement has been either energy usage or execution time. Measuring both of these is inexact; e.g., when measuring execution time some test runs may be off because of other processes being run on the system. Because of this multiple runs of each test are needed, and the results reported on must be averages of these runs—perhaps potted along with the standard deviation of the runs. In all papers we have written reporting on test results we have specified exactly how the data has been handled; how many test runs, whether outliers have been removed etc.

### 2.4.2 Evaluating the Development Model

As mentioned a thorough evaluation of the high level development library has not been performed. I have myself used the library extensively, and have written papers on the development model that argue the usefulness of the approach by example. That in particular Scavenger’s development model is simple, can be shown and argued for by presenting code examples. Using Scavenger a developer can add cyber foraging to an application by merely annotating a
function that is a viable target for remote execution; i.e., by writing two lines of code (one importing the scavenger module and one annotating the function) cyber foraging is applied. If coding efficiency equals lines of code this could be said to be efficient, but many other aspects play a part as well.

We have given Scavenger to a number of students; some have done small projects using it while others have based their entire master’s thesis on it. All of these students have been satisfied with the functionality and ease of use of the system, but they have not been using it from a pure “user” perspective; i.e., they have been using the low level Scavenger APIs instead of the automated, high level API. They have been experimenting with e.g., security models, and scheduling algorithms, and have not as such created cyber foraging enabled applications with the system. We have therefore chosen not to include interviews with these students as an evaluation of the development model. What these students have evaluated is the low level API of the Scavenger system, which will be reported on in Chapter 7.

In conclusion, doing a proper evaluation of the high level development model, e.g., where a group of developers are asked to solve a number of cyber foraging related problems with and without Scavenger, is something we have left for future work because of resource and time constrains.
Chapter 3

Previous Cyber Foraging Research

In this chapter I present scientific work that is related to cyber foraging and relevant for the research I have conducted within my PhD studies.

As mentioned in the introduction, the cyber foraging process consists of six separate steps: surrogate discovery, application partitioning, cost assessment, trust establishment, task execution, and environment monitoring; as visualised in Figure 1.1 which is replicated here in Figure 3.1.

Not all of these subjects will be covered in this chapter; e.g., surrogate discovery is not covered at all, but the interested reader is referred to [39] for...
an overview of peer discovery mechanisms. What will be presented in this chapter is four full-fledged cyber foraging systems that will be described in detail, outlining how they work and how that relates to the systems we have designed. In describing the existing systems the weight of the analysis will be on assessing the mobility of the system, its scheduling capabilities, and its development support. This means that application partitioning, cost assessment, task execution, and environment monitoring will be covered for each system. The four cyber foraging systems Spectra, Chroma, Slingshot, and the system by Goyal and Carter are presented and discussed in detail in Section 3.1.

After describing those systems some related work done within the area of application partitioning is presented. One of our research foci in the Locusts project was development support, which in cyber foraging e.g., means giving the developer easy access to application partitioning. Some work done within this area of research is presented in Section 3.2.

Related work done on the concept of application fidelity is presented in Section 3.3. These systems are also cyber foraging systems in some respects, but their focus is on providing tools to adapt an application to varying resource levels, in order to create applications that can operate in any environment regardless of the amount of resources available.

Cyber foraging may be foraging for a plethora of resources: processing power, energy, network connectivity, storage, displays, printers, locality information etc. In the Locusts project we have chosen to focus solely on cyber foraging for processing power. That offloading compute-intensive tasks to available surrogates also in some cases preserve precious energy resources at the mobile client is just a pleasant, although not unexpected, side effect; a side effect that we have explored briefly in [Paper III].

3.1 Cyber Foraging Systems

The focus in the Locusts project is on cyber foraging for processing power which is thus also the main focus when examining the systems in the following sections. Interesting factors to consider, when examining a cyber foraging system, are:

- The overhead induced by using the system with regards to:
  - Initialisation—how long does it take from the client’s initial wish to utilise a surrogate till the execution commences?
  - Remote execution overhead—how much overhead lies within each call to the surrogate once the initialisation has been done?

- The level of mobility supported by the system:
  - Initialisation, again—is the system capable of discovering and utilising surrogates while the client is in motion?
3.1 Cyber Foraging Systems

- Usage in unknown/unprepared environments—is the system able to perform in environments other than the user’s home/office?
- Task/process migration support—is the system able to move units of work around?

- The efficiency with which the system utilises available computing resources:
  - Does the system use any kind of intelligent scheduling?
  - Does the system employ fairness and load-balancing?
  - Is the system capable of using multiple surrogates in parallel?

- Development issues:
  - What does the system do to ease the development of cyber foraging enabled applications?

Of course there are numerous other factors that could be considered when studying a cyber foraging system, but this review will focus mainly on the items listed above. When describing a cyber foraging system a general description of the system is given in the “System Description” subsection, followed by a discussion of how the system matches up with regards to the metrics presented here in the “Metrics” subsection.

The systems presented here are, in chronological order:

**Spectra** Presented by Flinn *et al.* in [22] and by Balan *et al.* in [4]. This system is described in Section 3.1.1.

**Chroma** Presented by Balan *et al.* in [4, 6, 5]. This system is described in Section 3.1.2.

**Goyal and Carter** An unnamed cyber foraging system devised by Goyal and Carter in [25]. The review of this system may be found in Section 3.1.3.

**Slingshot** The Slingshot system was presented by Su and Flinn in [77]. This system is presented in Section 3.1.4.

### 3.1.1 Spectra

Spectra is the first true cyber foraging system described here. It uses the Odyssey system described in Section 3.3.1 to vary the application fidelity. Spectra was presented by Flinn *et al.* in [22] and also by Balan *et al.* in [4].

The concept of application adaptability, as defined by Noble *et al.* [55] within the Odyssey system, is of key importance in Spectra. When working with cyber foraging, the resource availability may fluctuate between resource-rich environments with an abundance of available computing resource, and resource-impoverished environments with no or poor connectivity and little
infrastructure support. Developing applications for operation in such changing environments can be quite challenging. In order to rise to that challenge an intelligent, adaptive scheduling of computing tasks is needed, and this is the main focus of Spectra.

System Description

The mobile clients in Spectra are running a specific Spectra client, which is tightly integrated with the Odyssey system. When an application starts it statically registers its operations with the Spectra client using the `register_fidelity` call in the Spectra API. For each operation it specifies a set of possible execution plans, corresponding to the different fidelities the operation may be performed at. An execution plan uses one or more services, which are the actual application code that is executed at Spectra servers. When local execution of services is needed, these must be run on a local Spectra server—a client device must therefore be running both the Spectra client and server along with the applications using the cyber foraging services. Apart from this, all machines participating in the Spectra network must also participate in a shared Coda file system. This file system is used when passing in- and output of services between client and surrogate. Coda was chosen for its admirable performance in the face of disconnections; when a Coda client is disconnected from its file server it may continue its operations working now only within the cache of the client, and when network connectivity reappears the changes applied to the cache are automatically applied to the file server.

Development

The process of cyber foraging is very much a manual task in Spectra: When an application reaches a point where an operation must be performed, it must signal this to the Spectra client by calling `begin_fidelity_op`. In this call the application specifies the values/sizes of any input parameters of the operation, so that these may be taken into consideration when Odyssey chooses a fidelity and Spectra chooses an execution plan and, possibly, a remote server. After this it seems that the Spectra client returns the execution plan to the application, because it is now the responsibility of the application itself to perform the local and/or remote services needed by using the `do_local_op` and `do_remote_op` calls. Furthermore, when the operation is done, the application must call `end_fidelity_op` to signal that the operation has ended.

Defining Services

Services are defined manually in Spectra, and developers are encouraged to create only services that perform relatively coarse-grained operations of at least a second in duration—services of a certain size is required to keep the relative overhead of using remote execution down. Spectra offers a simple API that these services must use. Services are called upon by use of remote procedure calls (RPC) that are called directly from the application. Each service runs as a separate process on the Spectra server, and, as far as I can tell,
these processes are spawned upon start-up, meaning that an installed service is always running and thus ready to serve a client. This of course gets very expensive with regards to resources when the number of installed services grows, since each service will at all time use an OS process. But, on the other hand, this degree of readiness of the services mean that the processing overhead of performing a service can be kept small.

**Scheduling** At the core of the Spectra system lies the scheduler. Both clients and servers are equipped with resource monitors in Spectra. These monitor the CPU usage, the energy consumption, the available network bandwidth, and the cache state of the underlying Coda file system. Using these resource measurements the Spectra client may make informed decisions on how and where to perform an operation. This is done by looking at the current state of resource availability on both the local device, and on remote Spectra servers by contacting them and asking them for their current resource measurements. But this snapshot-state of the current execution environment is not the only information applied when choosing an execution plan within Spectra; history based profiling is also used. The Spectra client is responsible for collecting information about resource usage whenever an operation is executed, which is done through the calls in the Spectra client API. When an operation is started by the application calling `begin_fidelity_op` the client starts its measurements of resources, and when `end_fidelity_op` is called the results of these measurements are collected. In the meantime, the application may have used remote resources by invoking services through `do_remote_op`, and resource measurements of these are also collected as part of the RPC response given by the services. Using all these measurements a history profile of each operation and its execution plans is built, and after some initial profiling it is expected that this profile of the operation should enable Spectra to choose the optimal execution plan under all circumstances. That this approach works is shown in the quite extensive tests presented in [22] where Spectra is tested in use of a speech recogniser, a document preparation system (\LaTeX{}), and a natural language translator.

**Metrics**

**Overheads** The overheads induced by using Spectra are relatively small when compared to the large size of the operations performed. The overhead of operation execution can in Spectra be divided into 1) the overhead of performing the initial scheduling, i.e., of choosing amongst available execution plans and surrogates, and 2) the overhead of using remote execution. The overhead of performing the scheduling can be quite sizeable in Spectra though if multiple surrogates are available, since a extensive search through the space of possible servers, execution plans, and fidelities is performed. In [22] they measure an overhead of 74 ms when choosing an execution plan for a null operation that returns immediately after being invoked when five surrogates are available.
When only a single surrogate is available this overhead is brought down to 21.4 ms. Compared to the larger size (at least 1 second in running time) of the operations, the initialisation time of Spectra is fairly small, but it may not scale well when more surrogates are introduced into the environment.

The next part of the overhead is that induced by the remote execution of services. Disregarding the transfer of in- and output data this overhead is very small in Spectra; services are already loaded at the servers and are thus ready to be called upon by clients. However, one can not disregard the in- and output transfers when doing remote execution. As mentioned Spectra uses the Coda file system for transfer of in- and output. This means, that when a service is called its input must be written into the client’s Coda file system mount. To gain a better understanding of exactly how services are performed in Spectra consider Figure 3.2. In this figure Spectra’s architecture is depicted in detail.

![Figure 3.2 Spectra’s architecture.](image-url)

On the mobile device the application, the Spectra client and server, some Spectra services, and the Coda file system daemon are running. The dedicated surrogate is running only the Spectra server, some Spectra services, and the Coda daemon. When a service is invoked the client writes its input into its local Coda file system mount, if it is not already there. After writing this file it will be automatically replicated over the network to the servers of the file system and thus onto the surrogate. After writing the input to the Coda file system the service may be called at the remote surrogate, adding a very small overhead, and during execution the surrogate writes the output file into its replica of the Coda file system, which is replicated to the servers and onto the client device. In the case of unprepared surrogates, i.e., a scenario where the client has just entered into range of the surrogate, this deteriorates into a simple push-pull ap-
3.1 Cyber Foraging Systems

...where input is pushed to the surrogate, the service is performed, and the output is pulled from the surrogate. If the next service, using the output just created in the previous service execution, is performed on another surrogate, the push-pull routine may be needed again since its Coda cache will not be updated yet. If, on the other hand, the same surrogate is used for performing multiple services the approach of using a shared file system will have one great advantage: it will not be necessary to push service input more than once. This means, that the overhead of performing operations consisting of multiple services can be kept very small, if the same surrogate is used for all services.

Mobility  Regarding mobility the Spectra system does not fare so well. On the positive side, the initialisation time in Spectra is low. In Spectra surrogates are defined statically in a configuration file on the client, and if the client has a network connection it can reach the surrogates, meaning that no discovery is necessary. This is a good thing when cyber foraging is only ever performed in prepared environments, but as soon as the client devices leaves the safe confines of the corporate network, its ability to use cyber foraging disappears completely. In [4] Balan et al. mention a service discovery framework called VERSUDS that could be employed within Spectra and Chroma, but none of the papers describing the systems seem to be using this.

It is not only the lack of a discovery mechanism that is responsible for the lack of mobility within Spectra. For one, all computers participating in the Spectra network must share a Coda file system, and such a file system may only reside on servers that are connected to each other, i.e., within a corporate or home network. Furthermore, in Spectra services are pre-installed onto surrogates, which means that all surrogates must be manually prepared in advance to be able to do work on behalf of their clients. This also would not work in an unprepared environment. Finally, migration of operations or services is not possible in Spectra, which also inhibits its use in highly mobile scenarios.

Efficiency  Efficiency-wise Spectra fares a bit better. Its scheduler enables it to make informed decisions about how to perform operations. Load-balancing is not a consideration in Spectra though, services are executed as separate OS processes, so it is up to the OS to multiplex between computing resources. Parallelisation is also not considered in Spectra; an operation is always performed in co-operation with a single surrogate.

Development  Development using Spectra is a fairly manual task. The developer must use the Spectra API to register operations, and when these operations are needed he must signal this to the Spectra client. The client then returns an execution plan, and it is then up to the application to follow this plan by calling local and/or remote services via RPC. Finally, when the operation is done the developer must also signal this to the Spectra client. The partitioning of applications into local and remote executable code is also quite
involved; all possible partitions (i.e., Spectra operations) must be identified at
development time; there is no support for dynamically generated operations.
When these operations have been identified the services they consist of must be
implemented using another Spectra API, and finally manually installed onto
all surrogates.

3.1.2 Chroma

The architecture of Chroma was described by Balan et al. in [5], and the process
of developing cyber foraging enabled applications using Chroma is described
in [6].

Chroma is a refinement of the Spectra system described in Section 3.1.1,
and as such has many similarities with that system. What is new in Chroma is
the use of tactics. In Chroma a tactic is a file that describes all the ways that an
operation may be performed, i.e., it describes the different fidelities, that the
operation may be performed at.

System Description

As mentioned, Chroma builds on the Spectra system, which means that its
primary focus is still on intelligent scheduling using history-based profiling
and local and remote resource measurements. This also means, that the sys-
tem is still using pre-installed services that run as OS processes on surrogates,
that no discovery mechanism is employed, that operation migration is not sup-
ported etc.

**Tactics** What Chroma brings to the table is its notion of tactics—a new way
of defining the execution plans that made up operations in Spectra. The tactics
files define “the full range of meaningful partitions”[5] of an operation, and
for each operation within an application such a tactic file must be created. In a
tactics file the developer specifies the RPC functions that may be called during
operation execution, and the different ways that these functions may be com-
bined to solve the operation at different fidelities. An example tactic file, taken
directly from the original paper, is shown in Figure 3.3.

In this example an operation performing natural language translation is de-
scribed. This operation uses four different services defined by the RPC keyword.
Service has the same meaning in Chroma as in Spectra, i.e., it is a piece of
code that is installed on a surrogate and may be called upon via RPC. After the
definition of services comes the tactic definitions; marked by the DEFINE_TACTIC
keyword. A tactic defines how services can be combined to perform the oper-
ation at different fidelity levels. The syntax for tactic definitions is simple: 1)
only defined services may be used, 2) services listed as a comma separated list
within brackets can be performed in parallel, and 3) when services are delim-
ited by an ampersand the latter of the services depend on the execution of the
### 3.1 Cyber Foraging Systems

```plaintext
RPC server_gloss (IN string line, OUT string gloss_out);
RPC server_dict (IN string line, OUT string dict_out);
RPC server_embt (IN string line, OUT string embt_out);
RPC server_lm (IN string line, IN string embt_out, IN string dict_out, IN string gloss_out, OUT string translation);
DEFINE_TACTIC gloss = server_gloss & server_lm;
DEFINE_TACTIC dict = server_dict & server_lm;
DEFINE_TACTIC embt = server_embt & server_lm;
DEFINE_TACTIC gloss_dict = (server_gloss, server_dict) & server_lm;
DEFINE_TACTIC gloss_embt = (server_gloss, server_embt) & server_lm;
DEFINE_TACTIC dict_embt = (server_dict, server_embt) & server_lm;
DEFINE_TACTIC gloss_dict_embt = (server_gloss, server_dict, server_embt) & server_lm;
```

Figure 3.3 An example tactic file describing a natural language translation operation. This example is taken from Balan et al. [5].

former (i.e., they may not be executed in parallel). Even though some services may be performed in parallel using this notation, it seems that this is not used in Chroma, which is a bit odd.

**Parallel Execution** Parallel execution is indeed used, as the use in over saturated computing environments is one of the main foci of Chroma, but in another form as could be expected after reading about the syntax of the tactic files. When multiple surrogates are available the Chroma scheduler may apply three different kinds of optimisation techniques using parallel execution:

**Fastest Result** Using this optimisation the scheduler chooses a fidelity (one of the defined tactics within the tactic file), and then when performing the chosen tactic it asks multiple surrogates to perform each service. The first result that is obtained using this technique is then used, and all other results are simply discarded; hence the name “fastest result”.

**Data Decomposition** In order to use this optimisation the developer must provide Chroma with a function that splits up the input data, and a function that can combine the resulting output data. When a service is performed using this technique Chroma finds a number of surrogates that can be used, chops up the data into the appropriate number of pieces, invokes the service at the surrogates, and combines the result before handing it over to the application.

**Best Fidelity** The final optimisation technique is employed on the tactic level. When an operation is started a number of tactics are chosen and shipped off to different surrogates. Chroma then waits for a specified number of seconds (or until the highest fidelity result has been returned), and returns the highest fidelity result that it has received within that period of time.
Scheduling  The architecture of Chroma can be seen in Figure 3.4 where the different components and their interrelations are depicted. Scheduling of operations in Chroma is done in much the same way as in the Spectra system. The execution history of the operation in question is fetched from the “resource demand predictor”; the current resource availability, both locally and at remote surrogates, is fetched from the “resource monitors”; using this information an exhaustive search is performed through the problem space to choose the optimal tactic given the current information. This search may, as we have seen, choose to use multiple surrogates in parallel given an over saturated execution environment. Once the tactic(s) have been chosen these are given to the “operation executor” which is responsible for performing the tactic. This means that the Chroma system is responsible for invoking the RPCs on remote surrogates, which is an improvement to the more manual approach taken in Spectra, where the application developer would himself have to invoke the remote functions. While the operation is being performed its resource consumption is measured and logged, and upon completion this log is given to the resource demand predictor, so that it may update its profiling data.

A final thing that should be noted about Chroma is the introduction of utility functions. When the tactics selection engine performs its search, it may be given an utility function that is used to weigh the different aspects of the execution. For example, one utility function may assign all weight to processing speed, choosing speed over energy usage any time, while another assigns equal weights to energy and speed. These aspects of the scheduling are accessible to the user of the application, who may in the configuration of an application choose his preference; energy conservation or processing speed.
3.1 Cyber Foraging Systems

Metrics

**Overheads and Mobility** Seeing as Chroma builds on Spectra it performs very much like Spectra with regards to the metrics listed in the beginning of Section 3.1. The overheads, initialisation and general usage alike, are the same as in Spectra, as is the hindrances with regards to mobility: no discovery mechanism, pre-installed services, no process/operation migration etc.

**Efficiency** Where Chroma differs from Spectra is in the efficiency of using available resources, and in the development support. By introducing the three modes of parallel execution, “fastest result”, “data decomposition”, and “best fidelity”, Chroma makes better use of available computing resources in over saturated environments. One obvious optimisation would be to introduce a fourth mode of parallel execution, where services within a single tactic was executed in parallel on multiple surrogates when possible. The possibility of doing this is reflected in the tactic syntax, where services separated by a comma are parallelisable, i.e., they have no interdependencies.

**Development** Regarding development Chroma introduced some great improvements compared to Spectra. For one, it is no longer the responsibility of the developer to invoke the RPCs constituting the chosen tactic; this is handled within Chroma by the operation executor. This means, than an application may now simply ask Chroma to perform an operation, and then wait for the results of performing that operation to come in. This process is described in detail in [6]. To ease the use of Chroma a stub generator has been created, and given a tactics file this generates the necessary methods that the developer needs to register and de-register the operation, to get the currently suggested fidelity of the operation, and to make Chroma perform the operation at the chosen fidelity.

3.1.3 Goyal and Carter

The cyber foraging system described by Goyal and Carter in [25] is unnamed, and will henceforth simply be referred to as “the system”. This system presents an entirely different approach towards cyber foraging, where virtual machines are used as the platform for executing “remote” code.

**System Description**

Using virtual machines as an “execution environment” in many ways simplify the process of cyber foraging; but this comes at a cost, as we shall see later on in this description. In such a system there is, naturally, a clear distinction between client and surrogates, as surrogates must be capable of supporting system virtual machines, which is a very heavyweight operation that is out of
reach for small mobile devices. This of course means, that the possibility of mobile devices co-operating to solve a task is lost—but this is not the focus of this system.

As with any other cyber foraging system, the first obstacle that need to be overcome, when a client needs the services of a surrogate, is that of surrogate discovery. In this system a central service registrar is used for this. Surrogates register themselves with the registrar upon start-up, and clients query this registrar when in need of a surrogate. Seeing as virtual machines are used, surrogates may be capable of offering multiple host operating systems to its clients—in the implementation described in the paper, virtual machines may be either Linux machines, of varying flavour, or BSD. When surrogates register themselves at the registrar they give information about which virtual machines they offer, so that this may be matched to client requests upon service discovery. After querying the registrar a client receives an IP-address of a possible surrogate. It is now up to the client to contact that surrogate, and establish a contract with the surrogate to provide the service for a specified amount of time. When this contract has been established, the surrogate initialises and boots a virtual machine image, and upon completion of boot the IP-address of the virtual machine is returned to the client. Now the virtual server is running a “clean” image of the specified OS, and it is now the responsibility of the client to install the required functionality onto the server. This may be done in one of two ways; either the client logs into the virtual server using ssh (the client has root access to the virtual server), or by invoking a simple service provided by the virtual server called a Sub Task Configuration Request. The virtual machine is running a small virtual server manager that offers this service to the client. The service is invoked by the client by sending an URL of a program that the client wants the surrogate to run; this would normally be a shell script that fetches the needed functionality from the Internet and installs it onto the virtual server. The entire process has been depicted in Figure 3.5.

![Figure 3.5](image.png)

Figure 3.5 Goyal and Carter’s cyber foraging system in action. This figure was taken directly from their original paper [25].

After all of this initial configuration has completed the client may start utilising the surrogate’s resources. How this is done is not the concern of this cyber foraging system—client application must themselves define how the actual cy-
ber foraging is performed; be it via installing a bunch of RPCs on the virtual machines, by relaying the entire application to the surrogate and then using some sort of desktop sharing mechanism such as VNC, or by some completely different approach.

**Metrics**

**Overheads** Any cyber foraging system based on the use of separate system virtual machine images per client, will have a substantial overhead when it comes to initialisation time. When a client decides that it needs a surrogate, it has to 1) contact the registrar, 2) receive an IP-address of a surrogate as response, 3) contact that surrogate and negotiate a service contract, 4) wait for a virtual machine image to be initialised and booted, 5) receive an IP-address of the virtual machine, and 6) ask that virtual machine to download and run a script that installs the needed functionality. This obviously takes a long time, i.e., in the order of multiple seconds if not minutes. Once the surrogate has been prepared in this way, the overhead of using cyber foraging may be kept small. This is of course application specific, but it would be reasonable to expect that all needed services would be installed at once onto the surrogate, and that shared storage between the surrogate and the client is in place, which would lead to a minimal overhead when invoking functionality.

**Mobility** Regarding mobility this system does not fare well. There is no “mobile” discovery mechanism, i.e., the clients need to know the address of the registrar beforehand. This will work if clients and surrogates only talk to each other over the Internet, which I suspect is the case in this system, but there are two big problems with that approach—network bandwidth and latency. Often when using cyber foraging a non negligible amount of data needs to be transferred between the client and its surrogate(s), e.g., when using speech recognition, image or video processing for augmented reality, etc. Using an Internet connection to transfer this data, with its comparably low bandwidth compared to using e.g., Wi-Fi to contact hosts on a local LAN, incurs a serious performance degradation. Even in the case where the amounts of data are small, the higher latency of a multi-hop Internet link will incur high costs, especially if many small calls are made to the surrogate.

One thing that does work well with regards to mobility is the use in unprepared environments. Surrogates need only supply some standard virtual machine images, and the client can then automatically install the needed functionality. Migration of work is not considered in this system, even though that would have been a trivial task considering that virtual machines are used. Using a virtual machine manager such as e.g., VMWare, it is possible to suspend a running virtual machine, whereafter it may be moved onto another surrogate and started there.
Chapter 3 Previous Cyber Foraging Research

**Efficiency** The system does not use any kind of intelligent scheduling; when the registrar chooses a surrogate for the client, load is not considered at all—it is mentioned that including this in the decision process would be a possibility. Once a surrogate is chosen no more scheduling is done on the system level, that is completely left to the client application. This means e.g., that the system has no built-in support for utilising multiple surrogates. Load-balancing within a single surrogate is included though, since this comes as standard in most virtual machine managers.

**Development** Development using this system is a very manual process. There are only a handful of functions in the cyber foraging API, that are needed only to locate a surrogate, negotiate a contract with it, and install some software on it. After that the application programmer is on his own.

**Security** One of the good things about the virtual machine approach is the added security of having interactions with different clients encapsulated within separate virtual machines. This means, that clients can not mess with the surrogates main operating system, and that they can not interfere with each other. Surrogates may still be used to launch attacks though, as there is no checking of what the client chooses to install and run on a surrogate.

3.1.4 Slingshot

Slingshot is another VM-based cyber foraging system that was presented by Su and Flinn in [77].

**System Description**

Slingshot has an entirely different approach towards VM-based cyber foraging than the system by Goyal and Carter, that was described in the previous section. While an Internet connection could be required on the surrogate in Goyal and Carter’s system, if it had to fetch the application specific software over the Internet, Slingshot requires an Internet connection on both the client and surrogate devices at all time. In Slingshot there is always one surrogate available; the *first-class replica*, which is an Internet connected machine controlled by the mobile user. When no local surrogates are available the mobile client utilises this first-client replica for performing heavy work, but this has obvious disadvantages compared to working with local surrogates because of the relatively low speed and high latencies of Internet links when compared to local Wi-Fi—especially with regard to upload speed from the mobile device to the surrogate. To alleviate this, Slingshot surrogates positioned at Wi-Fi hotspots are capable of receiving virtual machine images from the first-class replica, thus becoming *second-class replicas*. Such a Slingshot network, containing the first-class replica and two second-class replicas, is shown in Figure 3.6.
3.1 Cyber Foraging Systems

Without the local surrogates Slingshot is reminiscent of cloud computing, where the user is given control over a virtual machine hosted somewhere in “the cloud”, but by adding the local surrogates, or second-class replicas, the ailments of using cloud computing as I described in Section 1.1.3 are mitigated. In that respect Slingshot pursues the exact same goal as Satyanarayanan et al. [75].

**Using Surrogates** When the client device needs to perform some heavy task it asks all surrogates, both first- and second-class replicas, to perform the task. The first result returned is used—reminiscent of the fastest result optimisation used in Chroma. The fastest result will most probably come from one of the second-class replicas; mainly because these receive the input data faster, because of the higher bandwidth and lower latency of a local network link. Seeing as these replicas may not always be trusted, the result that comes in, at a later time, from the first-class replica is used to check whether the second-class replica returned a truthful answer. If the reply from a second-class replica differs significantly from the reply from the first-class replica, the co-operation with that second-class replica is terminated. All of this happens through the use of a Slingshot proxy installed on the client device. This proxy receives commands from its client applications and these are sent to the first-class replica and any second-class replicas currently in use, results are gathered, and the response is sent back to the application.

**Instantiating New Replicas** The instantiation of secondary replicas is done by the client by sending a migration request to the surrogate machine. The local surrogate then contacts the first-class replica and receives a copy of the current virtual machine image, i.e., the VM is suspended and check-pointed at
the first-class replica before it is shipped to the local surrogate. In this transfer only the volatile state of the virtual machine is sent to the local surrogate, the contents of the virtual hard disk is not sent. Instead a content-addressable storage system is used, where data is hashed and stored on disk at the first-class replica according to its hash value. Upon migration the chunk table, a table containing mappings from disk blocks to their hash values, is handed over to the new surrogate, and upon usage these blocks are transferred into a local block cache at the surrogate. Using this method the surrogate will initially fetch a lot of disk blocks into its cache, which will degrade its initial performance, but after a while it will have fetched the needed blocks and thus be ready to perform at full speed. The advantages of using such a system are 1) using content-addressable storage saves a lot of storage at the first-class replica, since many blocks will be identical when working with multiple virtual machine images (e.g., the entire operating system), and 2) only the parts of the disk that are currently in use are transferred to the surrogate, saving precious bandwidth when instantiating second-class replicas.

Transferring an entire VM image over the Internet is of course a lengthy operation. To alleviate this, Slingshot clients may carry with them snapshots of the first-class replica, that may be shipped to second-class replicas directly over the local Wi-Fi connection. Once such a snapshot has been booted, another lengthy process begins though: the second-class replica must now play back a potentially large event log, consisting of the events that has taken place since the snapshot was taken, so that it may become up to date.

**Metrics**

**Overhead**  Regarding overheads when performing cyber foraging, Slingshot of course has a major Achilles’ heel in its instantiation of secondary surrogates. When a second-class replica is instantiated the entire memory state of the virtual machine is initially transferred, closely followed by a large amount of disk blocks that are needed for general usage of the VM. Transferring such enormous amounts of data over the Internet will mean, that the time spent starting a new surrogate must be counted in tens of minutes or even more. Once the second-class replicas are in place the overhead of performing cyber foraging is application dependent, and can as such be kept very small. The Slingshot proxy forwards requests immediately to all available surrogates, and as soon as the first response arrives it is delivered to the application.

**Mobility**  Mobility is inherently very low in Slingshot because of the very large initialisation times; it is virtually impossible to be moving while using Slingshot. But, on the other hand, Slingshot does support full migration of tasks/state, which means that nomadic behaviour is very well supported, assuming that users spend enough time at the oases. Usage in unprepared environments is also very well supported; only a small Slingshot API and a virtual machine manager is needed on potential surrogates.
Efficiency  Slingshot has no kind of intelligent scheduling at all; it simply forwards all tasks to all surrogates and collects the results. Like it was the case in Goyal and Carter’s system, support for fairness and load-balancing within a surrogate is given through the use of virtual machine managers. Parallel usage of resources is indeed in play in Slingshot, but these resources are wasted rather that fully utilised, because the same task is shipped to all surrogates.

Development  The development process using Slingshot is not described in any detail in the paper.

3.1.5  Summary and Comparison

In the preceding sections a number of cyber foraging systems have been described. These are not the only relevant systems that I have examined, but they represent the most full-fledged, true cyber foraging systems available. Other systems, such as Riva’s Context Aware Migratory Services [69], using mobile code cyber foraging in a highly mobile setting, have also been very inspirational. As have some older texts on creating “mobile RPC systems” such as Kottmann et al. [37] and Bakre and Badrinath [3]—the latter of which actually has a very interesting migration of execution state.

Execution Model

The four full-fledged cyber foraging systems described; Spectra, Chroma, the system by Goyal and Carter, and Slingshot, fall quite naturally into two categories: RPC-based and VM-based. In Spectra and Chroma applications must adhere to a special development model: tasks (or operations in Spectra/Chroma jargon) must be defined as RPC services that must be manually installed onto surrogates, and the execution of tasks is then done by invoking these RPCs on remote servers—either manually as in Spectra, or automatically through the cyber foraging middleware as in Chroma. In the VM-based systems the actual process of cyber foraging was left up to the applications; especially in the system described by Goyal and Carter, where a surrogate simply offers root access to a clean virtual machine image, and leaves everything else to the application. In Slingshot there is an API that must be used when communicating with the proxy, which in turn communicates with the surrogates, but it seems that this proxy only relays messages to the virtual machines, and that these must themselves interpret the messages and act upon them; i.e., applications have to define their own cyber foraging protocols.

Scheduling

The scheduling abilities of the systems, and hence also their ability to efficiently utilise available surrogates, are very much linked to their execution model. The
VM-based approaches are not very adept at utilising multiple surrogates for efficiency, nor are they capable of using a mix of local and remote resources when that may be beneficial. In this respect the RPC-based approaches fare a lot better, with Chroma being able to utilise both local and remote resources very efficiently. The entire focus of both Spectra and Chroma is on intelligent scheduling, so these systems naturally excel in this respect.

**Mobility**

Regarding mobility none of the presented systems fare very well; the VM-based approaches are inhibited by their very large initialisation times, and the RPC-based approaches presented here by their need for pre-installed tasks and lack of migration support, which means that the peer is confined to stay within range of known surrogates. If, on the other hand, the mobility of the peer is confined to be within a restricted area, such as a university campus or an office building, systems such as Spectra and Chroma will perform very well. Of course neither Chroma nor Spectra have any discovery mechanism, but that could easily be added, and the pre-installed nature of the operations performed would mean, that surrogates could be used as soon as they were discovered. The VM-based systems are purely for nomadic behaviour, where peers move from one location to another and stay in this new location for an extended period of time before moving again. This is due to the very large initialisation times in VM-based systems; especially in the Slingshot system.

**Development**

With regard to the development process the VM-based and RPC-based approaches again have great differences. The two VM-based approaches are mainly concerned with creating a breeding ground, where cyber foraging may take place, by offering access to virtual machines. The RPC-based approaches, on the other hand, offer mobile middleware solutions that may be used by the developer to develop services and operations, and to utilise these in the execution of the applications. Slingshot does have some middleware support in that it uses a client proxy that is responsible for communicating with surrogates, but it still offers no controlled way in which cyber foraging applications may be developed. Of the systems described, Chroma is definitely the one that has the best developer support; it has a clearly defined way in which developers must 1) partition the application and thus define services that may benefit from remote execution, 2) define the operations that these services can be used within and the fidelities that they can be performed at, and 3) actually use the cyber foraging system on runtime. Especially with regard to the actual execution Chroma is superior to Spectra, simply because it performs all the RPC calls on behalf of the application—be they local or remote—which means, that the application does not have to worry about whether or not cyber foraging is currently used. Furthermore, when developing for Chroma a stub-generator is
3.2 Application Partitioning

Application partitioning is the process of dividing an application up into local code, that must always be performed by the client running the application, and remote executable tasks, that can be performed at a remote host. Such a partitioning must be performed in order to facilitate cyber foraging.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Spectra</th>
<th>Chroma</th>
<th>Goyal/Carter</th>
<th>Slingshot</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overheads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialisation</td>
<td>Low&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Low&lt;sup&gt;a&lt;/sup&gt;</td>
<td>High</td>
<td>Huge</td>
</tr>
<tr>
<td>Execution</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td><strong>Mobility</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discovery</td>
<td>No</td>
<td>No</td>
<td>Yes&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Yes (UPnP)</td>
</tr>
<tr>
<td>Unprepared usage</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Migration support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent scheduling</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Load-balancing</td>
<td>No&lt;sup&gt;c&lt;/sup&gt;</td>
<td>No&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallelism</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Development</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application CF API</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes&lt;sup&gt;d&lt;/sup&gt;</td>
</tr>
<tr>
<td>Task API</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

<sup>a</sup>Tasks are pre-installed onto surrogates.
<sup>b</sup>Centralised through registrar. How the registrar is discovered is not described.
<sup>c</sup>Since load on surrogates is considered during scheduling a kind of load-balancing takes place, because unloaded surrogates will be preferred.
<sup>d</sup>An API call is used to forward tasks to the client proxy, but it seems that it only forwards the request, i.e., it does not impose any structure on the process and hence has to be tailored specifically for each application.

Table 3.1 Comparison of selected metrics.

available, that can be used when creating new tasks; as described in [6].

Summary

To sum up a number of key metrics, that have been discussed when presenting the systems, have been listed in Table 3.1. Looking at this table it should become clear, that neither of the systems have all the desirable properties one could wish for in a highly mobile cyber foraging system. The VM-based approaches are able to operate in unprepared environments and migrate “tasks”, but their horrible performance when it comes to initialisation of surrogates render them useless in a mobile setting. The RPC-based approaches have very low initialisation time, but they lack the flexibility of the VM-based approaches when it comes to use in unprepared environments, which again renders them useless in highly mobile scenarios. In spite of these deficiencies, the presented systems all work very well in the scenarios that they were created for, and as such have been of great inspirational value for the cyber foraging frameworks designed within the Locusts project.

3.2 Application Partitioning

Application partitioning is the process of dividing an application up into local code, that must always be performed by the client running the application, and remote executable tasks, that can be performed at a remote host. Such a partitioning must be performed in order to facilitate cyber foraging.
In most cyber foraging systems this partitioning is a manual thing, i.e., the developer must manually specify which parts of the code can be offloaded to surrogates. This is of course an added burden that the developer must bear when developing cyber foraging enabled applications. To ease this burden some systems, such as Coign [32] by Hunt and Scott, try to automate this process.

### 3.2.1 Coign

Coign [32] is an automatic distributed partitioning system. What Coign is able to do, is to automatically partition applications into parts that may be executed remotely and parts that must be executed locally. The really interesting part is that all this is done without ever altering the source code—it is done directly on the binary application, i.e. without even knowing the source code.

The trick is, that Coign only works on applications built from COM objects. Such objects communicate through IPC interfaces, and Coign monitors these interfaces and intercept the messages. Coign thus works by 1) instrumenting the binary application so that messages may be intercepted, 2) profiling the application to discover which components may be moved to remote servers, 3) profiling the network to find the costs of communication between peers, and 4) instrumenting the application again to allow for remote execution. The profiling of IPC calls generates a call graph which is subjected to a min-cut graph cutting algorithm to select parts of the graph that may be moved.

When all profiling is done the application is instrumented in a way such that all component instantiations are caught by Coign, and these may then be executed on remote servers so that the COM objects reside on different machines. All communication between peers are done using DCOM.

Coign is interesting but it has some shortcomings. For one, it only works with applications written using the Microsoft COM model which is just not cross-platform enough for use in a real world setting of mobile devices. Secondly, the automatic partitioning of applications can never be as good as the manual approach. When preparing an application for cyber foraging a little domain knowledge goes a long way—it may for example make sense to bundle calculations into larger batches, parts of the application may be rewritten to run in parallel when more machines are in play, it may make sense to compress the data transferred between components when communication is now over a network link, etc.

While it is admirable that Coign is capable of distributing a binary application, I do not think that automating application partitioning to this degree is the best solution to the partitioning problem.
3.3 Application Fidelity

Application fidelity is sometimes referred to as application adaptability, i.e., it is concerned with an application's ability to adapt to variations in resource levels by changing the fidelity of its computations. Changing fidelity can, e.g., mean changing between fetching small, low resolution images in a mobile web browser when the available bandwidth is limited, and then changing to larger, full resolution images when better connectivity becomes available.

In this section, two systems offering support for this kind of switching between fidelities are presented, beginning with the Odyssey system in Section 3.3.1, which is in fact an integral part of the Spectra and Chroma systems described in Section 3.1. After that a similar system named Puppeteer is presented in Section 3.3.2.

3.3.1 Odyssey

Odyssey, while not an actual cyber foraging system, does introduce a very important concept that should be considered in any cyber foraging enabled application; namely application fidelity. In [55] fidelity is described as: “...the degree to which data presented at a client matches the reference copy at the server.”. Odyssey’s main focus is on supporting usage of remote services in a mobile setting, where there may be large variations in network connectivity over time. When high bandwidth network connections are available, results of a higher fidelity, which in their use-cases are more data intensive, may be transferred over the network link; but, when the connection quality is degraded to a degree such that the current fidelity may no longer be supported, the application should be notified of this, so that it may switch to a lower fidelity. This willingness to adapt to the current execution context makes it possible for the application to continue executing in spite of poor network connectivity. When higher bandwidth connections are available again, the application may adapt once more switching back into a high fidelity operating mode. The speed with which the system detects and responds to such variations in resource levels is called the agility of the system.

Odyssey’s goal is to create a framework for informed applications, that are always acutely aware of their current execution context. By providing this information to individual applications they can choose to adapt to suit the current resource availability. The resource measured in this way in the Odyssey system is network bandwidth, but extensions to include also network latency, disk cache space, CPU utilisation, battery power, and “money” are mentioned as a possibility. These resources are monitored by the Odyssey viceroy. Applications communicate their resource expectations by sending a resource descriptor to the viceroy using a system call. Such a resource descriptor contains the id of the resource, an upper and lower bound designating the amount of that resource needed to operate within the current fidelity, and the name of an upcall handler (callback function) that the viceroy may call if the resource availability...
strays out of the bounds given. When such an upcall handler is called, it is up to the application to send a new resource descriptor to the viceroy designating the bounds that the new fidelity operates within.

After reading the preceding description of Odyssey it may seem that changing application fidelity is a manual task, one that must be implemented anew for each application. This is not the case though; apart from the viceroy Odyssey supports a number of wardens that implement varying of fidelity for a number of data types. The concept of fidelity is tightly bound to the specific data types; changing fidelity for an image may be changing resolution or compression, for a video stream fidelity may be altered by changing the frame rate, compression, or individual frame quality, etc. This type-awareness is needed to implement changes in fidelity, and this is the responsibility of the wardens. A warden governs a single data type and is responsible for communicating with the remote services delivering such data. Seeing as the warden executes on the local machine, support for the changing of fidelity must sometimes also exist on the server-side; i.e., the remote service that the warden speaks to may be able to actually vary the fidelity of the data. This is especially true when the restricted resource is network bandwidth, in which case the server-side must be able to increase and decrease the size of the data moved over the network link.

For two example applications using Odyssey see Figure 3.7. In Figure 3.7(a) an image viewer application capable of browsing remote image libraries is shown. The application itself does not speak directly with the remote image servers; this is handled by the image warden. When high bandwidth network links are available high resolution, low compression images are sent to the client, but when the connection quality degenerates the server is asked to scale and compress images before sending them over the network link. The other example application, shown in Figure 3.7(b), is a speech recogniser, an example taken directly from the article by Noble et al. In this example, the server is not able to modify the fidelity, this change in fidelity is handled entirely by the warden. The local speech recogniser is capable of two things: 1) It can run the
first of several passes over the raw utterance given by the client application, which has modest CPU cost and brings down the size of the data with a factor of five, or 2) it may perform the entire recognition itself, which is very costly with regards to CPU and energy. The remote speech recogniser is also capable of two things: 1) It may do full speech recognition of a raw utterance, or 2) it may do recognition on an utterance that has been pre-processed by the client. Given these different ways to vary fidelity the application has three operating modes:

**High bandwidth**  Given high bandwidth the size of the input data is of less importance. The raw utterance is thus sent to the server for full recognition.

**Low bandwidth**  In a low bandwidth situation the size of the input is crucial. The raw utterance is thus pre-processed by the client, costing a small amount of CPU and energy, before the, now smaller, pre-processed utterance is sent to the server for recognition.

**Disconnected operation**  When no network connection is available the application may still function, albeit at a much slower pace. In this case the raw utterance is recognised in its entirety by the client.

Especially the speech recognition application in Figure 3.7(b) is very reminiscent of how a cyber foraging enabled application should work; apart from the fact that a fixed server is accessed. A very important aspect of a cyber foraging system is, that the cyber foraging applications should be able to function at all times—also when no surrogates are available.

Application adaption need not only occur when changes in bandwidth are detected, as in the Odyssey system. As mentioned earlier on, other resources such as CPU and battery power could also be measured, and applications could then for example be asked to switch to a less CPU intensive operation mode when multiple applications contend for the CPU, or when energy resources are low.

Application agility and fidelity are important concepts within cyber foraging; the availability of surrogates should augment an application, the application should not depend on it. The work done by Noble *et al.* is seminal in this respect identifying a very important application design paradigm that may be used when developing cyber foraging applications—and frameworks.

### 3.3.2 Puppeteer

The goals of Puppeteer, by de Lara *et al.* [15], is very similar to the goals of Odyssey in that it aims to adapt applications to changes in bandwidth availability. Even the scenarios that are presented resemble the ones used in the Odyssey article; they adapt by serving images of varying quality. The point where Puppeteer differs from Odyssey is in the way that adaption is done: in Odyssey the
adaption system offers an API and the applications thus have to be modified to use this API. In Puppeteer the applications must be component-based such that they expose an API. Instead of modifying the applications a “driver” for each application is written, which constitutes a Puppeteer proxy. This driver hooks into the application and is responsible for fetching data from the server, which it does through a dedicated Puppeteer server. The Puppeteer server must be aware of the structure of the data that it is serving, so that it may be able to version the data (i.e., to select between different fidelities of the data) and to use subsetting on the data (i.e., to serve only the needed subset of a document). For this adaption to be possible the application must provide an API that enables Puppeteer to view and modify the data the application operates on.

De Lara et al. presents the reader with a very nice overview (and a small taxonomy) of different kinds of application adaption. They make a distinction between data and control adaption, i.e., adapting the data or adapting the application’s control flow. They also make a distinction between system-based, application-based, component-based, and application-aware adaption. Application-based adaption is when the adaption is built into the application and only the application itself is aware of this adaption. Pure system-based adaption is when the applications are not aware of the adaption, meaning that the only kind of adaption possible is data adaption. Application-aware and component-based are both hybrids—they both offer the benefits of system-based adaption, e.g., the ability to load balance between multiple concurrent applications, and they both offer the possibility of control flow adaption through altering the way that the application acts.
Chapter 4

Towards Highly Mobile Cyber Foraging

One of the main objectives within the Locusts project has been the support of highly mobile cyber foraging, where highly mobile means that 1) the mobile device may be mobile while cyber foraging takes place, and 2) that this physical mobility takes the mobile device into unknown environments, where no previously known surrogates are available. We would like cyber foraging to be a viable computing technique when the user of the mobile device is moving throughout the environment at waking speed. In the following I will describe what that focus means when working with cyber foraging, and present some of the work we have done to achieve this goal. The concept of highly mobile cyber foraging and its consequences is the subject of [38].

To understand the impact of high mobility when performing cyber foraging the entire process of cyber foraging must be presented in detail; something Porras, Riva, and I did in [Paper II], where we identify a number of steps involved in any cyber foraging process. Here I will present some of these steps and discuss how high mobility influences the individual steps.

4.1 Surrogate Discovery and Environment Monitoring

In order to utilise surrogates available on the local network a peer or surrogate discovery mechanism must be in place. Using this discovery mechanism, mobile peers must be able to discover available surrogates, and, in some cases, surrogates should be able to discover each other. Whether or not surrogates are interested in knowing about other surrogates depends on the cyber foraging system. It is needed if the system wants to be able to support e.g., task migration, re-scheduling, or data sharing between surrogates.

When supporting high mobility this discovery process must be quite fast—if e.g., a user is walking past a Wi-Fi hot-spot, ten seconds may be all the time that is available to discover available surrogates, schedule a task, perform the task
at one of the surrogates, and receive the result before leaving range of the surrogate. Within these ten seconds the peer must thus not only discover available surrogates, it must also perform environment monitoring, i.e., collect information about the surrogates, so that it may make an informed decision when choosing amongst them.

Existing solutions such as UPnP, Bonjour, Jini, and Salutation could be used for this purpose, but as I describe in [39] these may be too slow to be useful in this highly mobile setting. The main problem with existing peer discovery mechanisms is that they are re-active, and in order to support highly mobile cyber foraging a pro-active approach is needed. If e.g., UPnP was used the following process would be performed in order to obtain information about available surrogates whenever a mobile peer entered a new network:

1. The mobile device would broadcast a discovery message.

2. The surrogates would all respond to this by sending a service announcement directly to the mobile device.

3. The mobile peer would then have to contact each surrogate one by one in order to get further information about the service they are offering.

This could be a lengthy affair if just a handful of surrogates were available. Apart from being re-active, another problem is that much of the communication is point-to-point, which means that all client devices have to connect to the available surrogates to ask for the same information. Furthermore, when scheduling in a cyber foraging setting, the mobile device needs up-to-date information about its environment, which means that it would have to ask all surrogates for this information not only upon entering a new network, but each and every time it performs a task.

The solution to these deficiencies of other discovery mechanisms is simple: have surrogates broadcast their service descriptions onto the local subnet instead of sending them point-to-point. Because broadcast is used all announcements need only be sent once. When a broadcast is needed is a design decision; either periodic broadcasts are used, or broadcasts could be sent out only upon client request or a significant change in resource levels. If the pro-active approach is taken, and surrogates thus broadcast a service description e.g., once per second, the discovery time would always be at most one second (disregarding packet loss). Using this approach, all peers on the network would always have up-to-date information about each other, and when new peers enter the network they are very quickly brought up to speed.

This is the approach taken in Presence; the service discovery mechanism we have built within the Locusts project. Periodically each Presence peer packs all its service descriptions into a single UDP packet and broadcasts this onto the local subnet. These announcements are very small, typically less than 100 bytes, so they incur very little overhead on the network.
Presence is described in detail in [39] and is available as open source software at http://code.google.com/p/presence-discovery/.

4.2 Scheduling for Mobility

Adding high mobility means a number of things to scheduling. For one the scheduling algorithm must be quite fast, so that as little time as possible is spent scheduling. Secondly, the high mobility means that scheduling must be done in unknown environments, i.e., in environments where the available surrogates are unfamiliar to the mobile device. This means that, to some extent, history based profiling of peers is inadequate as a means to choose between surrogates, so some other way of determining the most suitable surrogate must be found. Because of the physical mobility, the availability of surrogates may change rapidly, which means that scheduling must be a dynamic process, that may adapt itself to varying resource levels. Finally, if larger tasks is to be performed while the client device is physically mobile, some kind of task migration support is needed. That way a task may be partially solved by a surrogate and then moved either back onto the mobile device or to another surrogate.

Scheduling is the topic of Chapter 5 and the discussion of mobility’s impact on it is thus deferred to that chapter.

4.3 Mobile Task Execution

After having discovered and chosen amongst the available surrogates, the mobile device must have some means of actually performing its tasks on a remote host. As described in Section 3.1.5, there are amongst related systems two approaches to task execution: pre-installed RPCs or system virtual machines. For different reasons neither of these approaches are usable in a highly mobile setting; which is described in Section 4.3.1 and 4.3.2. In order to support highly mobile cyber foraging a mobile code approach is needed, such as e.g., the one used by Riva [69, 68] in her “Context Aware Migratory Services”. The mobile code execution system we developed within the Locusts project is described in Section 4.3.3, where I argue for its applicability in the field of highly mobile cyber foraging.

When considering task execution in a highly mobile setting three aspects are of key importance:

1. The time wasted between choosing a surrogate and the initiation of the actual task execution should be as small as possible—ideally it should be as small as the time spent transferring input data.

2. It must be possible to use any surrogate available, also ones that the mobile device has never worked with before. If surrogates need to be
Chapter 4 Towards Highly Mobile Cyber Foraging

“taught” how to solve a task, the overhead of this teaching should be as small as possible.

3. If the tasks being performed are long running, i.e., if their running time may exceed the amount time the mobile device is within range of the surrogate, some sort of task migration should be considered.

4.3.1 Pre-Installed RPCs

The advantage of working with pre-installed tasks is obvious—there is no need to install tasks on-demand, which means that the desired functionality is always immediately available. In the case of Spectra [4] and Chroma [5] the tasks are not only pre-installed, they are also continuously running on the surrogate, meaning that even the overhead of process creation is removed. This means, that the time between a client’s wish to utilise a surrogate till the task execution commences is minimal, and as such it could seem that pre-installed tasks would be the best possible solution for a highly mobile cyber foraging system.

The problem with pre-installed tasks is of course the fact that they need to be pre-installed onto all surrogates that the mobile device will be working with. In a highly mobile setting, where the mobile device may find itself in unknown environments, this restriction is too severe. It is unrealistic to believe that all tasks needed by the specific application that the mobile device is currently running, would be available on all surrogate machines throughout the world. In cases where the environment is restricted, e.g., when cyber foraging is confined to be performed within an university campus, a collection of pre-installed tasks could possibly be maintained. But even in that case, the maintenance work involved would be detrimental to the usefulness of the system.

Migration support is not immediately available when working with an approach such as the one in Spectra and Chroma. When tasks are running as operating system processes these could possibly be suspended, moved to another host, and resumed there using an eager-copy approach; as described by Richmond and Hitchens in [67]. The main problem with this approach is that it is 100% platform dependent—the hardware architecture and operating system of the receiving host must be identical to that of the sending host. This is not the case in a cyber foraging setting. A solution to this problem could be using a process virtual machine, like for example the Java virtual machine. There are other problems with that approach though; most notably the fact that the amount of data that must be moved when doing real process migration may be enormous because the entire process state needs to be migrated.

To sum up, the pre-installed RPC based approach yields very good performance in the face of physical mobility, but its mobility with regards to operating in unknown environments is in practise nonexistent.
4.3.2 System Virtual Machines

Using a system virtual machine, as it is done in Slingshot [77] and in the system by Goyal and Carter [25], the execution overhead is huge. The time that elapses from the point in time when a mobile device discovers a surrogate and until that surrogate is ready to perform a task is exceedingly long—i.e., it is measured in minutes, which is an eternity in a highly mobile system. This means that, at least in the form presented by those system, this approach is not feasible in a highly mobile context. Once a surrogate has been prepared to work with the mobile device, the overheads of performing tasks may be kept as low as in the pre-installed RPC case though.

One thing that a system level virtual machine does offer is seamless migration support. All major virtual machine managers offer suspend support, and when migration is needed the virtual machine can thus simply be suspended to disk and migrated to another host. This is a somewhat lengthy process though, and the migration of state can be extremely data intensive.

4.3.3 Mobile Code

Neither pre-installed tasks nor virtual machines have the desired properties for use in a highly mobile setting: The pre-installed tasks have a very small execution overhead but fall short when faced with unknown environments. The system virtual machine based ones are fully capable of working in unknown environments, but their initial cost of establishing an execution environment are so large as to be useless in a mobile setting.

In the Locusts project we have dealt with this problem by adopting a mobile code approach. Using mobile code one gains the benefits of both approaches:

- The initial execution overhead is as low as in the pre-installed scenario if the task is already installed on the surrogate, and if it is not the installation of the task is as fast as simply sending the code to the surrogate.
- The use in unknown environments is given—using mobile code the mobile device is able to dynamically push the needed functionality onto any surrogates it meets.

The mobile code approach taken in Locusts is reminiscent of the pre-installed approach in that mobile code is not always in use. Upon choosing a surrogate for executing a task it is checked whether the surrogate already offers the needed task, and only if it does not the task code is installed onto the surrogate. The overhead of this installation is negligible; in effect it amounts to transferring the code, which in most cases is only a few kilobytes in size, validating the code to check for security breaches, and writing the newly installed task to disk. After that the client may invoke the task any number of times; and other clients needing that tasks will be able to use it. The overhead of using out
mobile code approach is thus close to being as small as the execution overhead when using pre-installed tasks, while still retaining the possibility of operating in unknown environments.

There are of course a lot of security issues when working with mobile code; Chess [13] lists the following categories of security problems that must be addressed in a mobile code system:

- **Eavesdropping.** An attacker eavesdropping on a wire transmitting an agent to a surrogate may:
  - Change the code of the agent.
  - Change the state of the agent.

- **Malicious code (client side).** A malicious piece of mobile code may try to:
  - Perform DoS attacks on the surrogate (e.g., memory exhaustion, CPU hogging).
  - Use the surrogate as a platform for attacks on remote hosts.
  - Read secrets stored on the surrogate, or even destroy data on the surrogate.
  - Modify the code or state of other tasks executing on the surrogate.

- **Malicious hosts (surrogate side).** A surrogate executing a task may:
  - Alter the code of the task.
  - Change the state of the task before migration.
  - Return fabricated/false results.
  - Read any secrets stored in the code or in the state of the task.

As mentioned, security was not a main focus of the Locusts project, and we have thus not examined all of these issues. But, since we chose to implement our own mobile code execution environment from scratch using an insecure language, Stackless Python\(^1\), we have done a lot of work securing that execution environment. This is described in detail in [39].

Migration on an execution environment level is not something we have implemented. As will be described in Section 5 our migration model is one level above on the scheduling level. Seeing as we have based our execution environment on Stackless Python implementing task migration would be a possibility, as Stackless tasklets are *picklable*, which is Python jargon for serializable. It would thus be possible to implement task migration between surrogates by pickling a running tasklet, transferring its state to another host, and resume the execution there. We have not implemented this for a number of reasons: First off because there are some limitations to tasklet pickling, such as that a

---

\(^1\) An enhanced version of the Python programming language offering microthreading support. Homepage: [http://stackless.com](http://stackless.com)
pickled tasklet may not use any C functions, something that is used in many Python libraries, and furthermore the tasklet must not be holding any native resources such as e.g., files. Secondly, the size of a pickled tasklet can easily be huge because the entire tasklet state must be stored, and we thus deemed tasklet migration useless in a mobile setting.

4.4 Contribution

The contribution of our work within the Locusts project on highly mobile cyber foraging presented so far are twofold:

1. We have defined the concept of *highly mobile cyber foraging* and discussed what is needed to support such a setting; [38].

2. We have prototyped both a discovery mechanism and an execution environment supporting this setting; [39].
Chapter 5

Dynamic Scheduling

Scheduling is an important aspect of any cyber foraging system. The goal when choosing among a number of surrogates is to select the one that optimises some goal of the client—e.g., minimising execution time or the energy usage of performing the task. Because of the focus on highly mobile cyber foraging, we have chosen to design schedulers that strive to minimise total execution time, so as to minimise the likelihood that the mobile device has left range of the surrogate before the task has been performed.

In this chapter I provide an overview of the work on scheduling that has been done in the Locusts project. Section 5.1 presents the challenges faced when choosing a place to perform a single task in a cyber foraging setting. In this discussion of task placement the work done within the Scavenger cyber foraging framework is presented. In Section 5.2 the scheduling done in the Locusts cyber foraging framework is presented—where Scavenger sees scheduling as simple task placement Locusts uses workflows to represent tasks as consisting of multiple subtasks. In Section 5.3 the lessons learned while designing the two quite different schedulers of Locusts and Scavenger are presented, and Section 5.4 summarises the contributions of this work.

5.1 Task Placement

Cyber foraging may take place in any networked environment—be it an ad-hoc or a managed network. Figure 5.1 shows two network environments where cyber foraging could take place. In the design and development of Locusts and Scavenger, the two cyber foraging frameworks developed within the Locusts project, we have focused mainly on the setting shown in Figure 5.1(a); the managed Wi-Fi setting. We have chosen this focus because we find that this setting is the one most commonly found “in the wild” today. Users of smart phones and other Internet enabled devices may be using Internet connections over 3G
Figure 5.1 Two example network environments where cyber foraging could take place.

when they are mobile, but they also often roam open/public Wi-Fi networks to gain high-speed/free access. That we have had this focus does not mean that our techniques can not be applied to ad-hoc networking though; both Locusts and Scavenger could be employed in an ad-hoc network as the one shown in Figure 5.1(b).

Scheduling in a mobile, heterogeneous environment such as the one that must be catered for in cyber foraging is a complex matter. First off, there are no central scheduler that tasks can be submitted to, so the client itself must do the scheduling. Secondly, because of the ever changing environment, scheduling must be done at run-time acting on the information given at that point in time. This means, that the client is itself responsible for collecting the information needed to make an informed decision when performing tasks. Ideally the information available when scheduling a given task should be:

- The task in- and output size (bytes).
- Estimated running time when performed at any of the available surrogate machines (seconds).
- Bandwidth (bytes/second) and latency (seconds) information for all links in the network—both links between client and surrogate and links between surrogates.
- Data locality information. I.e., information about where the input data resides (it may be a result from a previous task stored at one of the surrogates), and whether or not the output data should be pushed back to the client.

Given this information an idealised scheduler is shown in Figure 5.2. In this figure \(I_{\text{size}}\) is the input size, \(I_{\text{loc}}\) the input location and, correspondingly, \(O_{\text{size}}\) and \(O_{\text{loc}}\) are the size and locality of the output. The bandwidth and latency between two hosts N and M are written as \(B_{N \rightarrow M}\) and \(L_{N \rightarrow M}\), and the running time of a task T on a machine M is \(T_M\).
5.1 Task Placement

```python
machines = surrogates + localhost
candidates = []
for M in machines:
    time = 0
    if not I_loc == M:
        time += I_size / B_M→I_loc + L_M→I_loc
    time += T_M
    if not O_loc == M:
        time += O_size / B_M→O_loc + L_M→O_loc
    candidates.append((time, M))
candidates.select_minimum()
```

Figure 5.2 Idealised scheduler.

Of course getting the information needed to implement such an ideal scheduler is not easy; if at all possible. In the following it will be described how this information has been approximated in Scavenger’s scheduler. Section 5.1.1 discusses how information about the network can be gathered, Section 5.1.2 is concerned with task specific information such as in- and output size, and Section 5.1.3 addresses the problem of collecting and working with peer specific information.

5.1.1 Network Information

The information needed about the network, i.e., bandwidth and latency measurements for all interconnecting links, are hard to come by, especially so if fluctuations due to peaks in traffic are to be taken into account. The problem is not measuring the current bandwidth and latency between two peers, this can be done quite easily. The problem is, that in order to measure it reliably a large amount of traffic is generated on the network, and seeing as these measurements must be done periodically between all peers in the network, an excessive amount of measurement traffic would be introduced to the network. The effect of this is especially detrimental in broadcast networks such as Wi-Fi. In Scavenger bandwidth and latency measurements have been substituted by a static media specification, where both clients and surrogates specify which kind of media they connect to the network with. These media specifications are then mapped to some expected bandwidth and latency values for that specific media. Consider again the network in Figure 5.1(a). This figure shows a common managed network consisting of a few wired and a number of wireless devices. In this environment the wired devices, the desktop PCs, would specify in their configuration that they are connected to the network using a 100 MBit connection, Laptops and UMPCs would probably be connected using IEEE 802.11g, and mobile devices perhaps using an IEEE 802.11b connection. During scheduling the client knows each peer’s specified bandwidth and when
considering transfer of data between two peers the smallest one of these two is chosen.

One possible solution to the problem of measuring bandwidth and latency is to measure whenever data must be transferred anyway; e.g., when input data is sent to a surrogate the bandwidth of the network link in use may be measured. We did initially implement this functionality in the Locusts framework, but there are some problems with this approach, which is why we have chosen to fall back to static definitions. One of the main problems with measuring bandwidth in this way is that, in order to get a reliable bandwidth measurement, a lot of data must be transferred. If for example the bandwidth is estimated based on a transfer of say ten kilobytes, the measurement would show a very low bandwidth, and if this information was subsequently used when scheduling a task with an input of two megabytes, the low bandwidth estimate could easily mean that the mobile device would try to perform the task itself even though remote execution would be much faster. The reason that data transfers are much faster for large amounts of data lies in the nature of the TCP connections used—the sliding window is gradually expanded to speed up the transfer, and when contending with others for the bandwidth, bandwidth is only allocated slowly to the new connection. As a small example consider Figure 5.3. In these tests a mobile device, a Nokia N900 smart phone,

![Figure 5.3 Bandwidth measurements. In these measurements a Nokia N900 was transferring chunks of varying size to a MacBook Pro over a IEEE 802.11b/g network. The MacBook Pro was connected to the router via an Ethernet cable while the N900 used Wi-Fi.](image)

is transferring chunks of data of varying size to a laptop computer, an early 2007 MacBook Pro, over a wireless link served by a Linksys WRT54G router. The laptop was connected to the router via an Ethernet cable while the mobile device used Wi-Fi. For each transfer size the mobile client sent that amount of data 50 times. After collecting the data about the 50 runs the top and bottom 10% are removed to remove any outliers, and the times reported here are the averages of the remaining 40 runs and their standard deviation. In the first test, depicted in Figure 5.3(a), the mobile device had to contend with others for the bandwidth—a third machine on the network was transferring large amounts
5.1 Task Placement

of data over the wireless link. In the second test, depicted in Figure 5.3(b), the network was quiet, i.e., it was only used to transfer the measurement data.

Looking at the data in Figure 5.3(a), it is clear to see that the measured bandwidth varies a lot depending on the size of the data transferred. In order to correctly assess available bandwidth in a noisy network such as this one, one would therefore have to store information about expected bandwidth related to transfer size. This is doable, and in fact reminiscent of how Scavenger handles variations in task execution time due to input variations, but it does have some problems. Consider now the plot shown in Figure 5.3(b) where the same link is measured in a quiet setting. In this measurement the effective bandwidth is almost the same for all transfer sizes—in fact the smaller transfer sizes on average get better bandwidth than the larger ones, albeit with a much larger standard deviation. This data does not correlate nicely with the measurements in a noisy setting.

In Scavenger a network link such as the one shown here would be given a fixed bandwidth value of 500 Kb/s which, looking at the data in Figure 5.3, does a good enough job of estimating the bandwidth that will be available on a slightly noisy network.

5.1.2 Task Specific Information

Task specific information in this regard is run-time information about the task being scheduled. This entails the size of in- and output data and data locality information.

Seeing as scheduling is done dynamically, i.e., at run-time, the task’s input is readily available at the time of scheduling. This means that information about task input is trivially given. The input may reside at other peers in the network, in which case the scheduling device will be holding a data handle instead of the actual input data. In Scavenger these data handles include information about the size of the data, so that this information may be used when scheduling regardless of the fact that the data is stored remotely.

Output data, on the other hand, is of course not given before the task has been performed. For some tasks it may be impossible to predetermine the size of the output, but for many regular tasks it is indeed possible to determine what the size of the output will be. Some tasks always return the same size output, while others have output of a size relative to input size or value. In Scavenger the developer may choose to designate the output size as a relation to the input size or value.

In Scavenger the developer may also choose whether to fetch the output and deliver it to the client application, or whether output data should be left at the surrogate and a data handle returned in its place. If data is left at a surrogate the overhead of transferring the output data is ignored when scheduling. The locality of the data is then considered when a new task is scheduling operating on it.
5.1.3 Peer Specific Information

The most complex part of task scheduling is assessing the running time of the task with the given input on any of the currently available peers. There is only one way of getting at such information and that is through profiling, which is also the approach taken by related systems such as Spectra [22] and Chroma [5].

One approach towards building a profile usable for task scheduling is creating a peer centric profile, where a history based profile containing information about the last \( n \) runs of a task on a specific peer is stored. Whenever that task is considered for execution on that specific peer, the profile may be consulted and the value found here can be used as an estimate of what the running time will be. How the history based profile is used is a design decision; in Scavenger the average of the profile data from the last ten runs is used as an estimate. Another approach would be to make a weighted average, where more recent profile data is given more weight.

There are some problems with this kind of peer centric profiling though: For one, it entails an assumption that a given task always has roughly the same running time. This is of course not true; for most tasks the running time will vary with input size or value. Furthermore, when working with highly mobile cyber foraging, the idea of having a peer centric profile, necessitating profile information about the specific peers that the client is currently within range of, works counter to the mobility of the system. In highly mobile cyber foraging it is more than likely that a given task has never been performed on the currently available surrogates. Both of these deficiencies of profile based scheduling in a cyber foraging setting have been considered in Scavenger, which is why Scavenger uses multidimensional profiles to reflect that a task’s running time may vary with input, and task centric profiles that may be used to reason about task running time on hitherto unknown surrogates.

**Multidimensional Profiles**

To approach the fact that running time varies with input size and/or value, Scavenger allows the developer to specify exactly which input parameters affect the running time of the task. This weight expression, which can be given as a regular Python expression referencing input parameters, is evaluated to yield a single weight value each time the task is scheduled. Using this value, which could e.g., be the size of an input file, Scavenger maintains a two-dimensional profile. After performing a task this profile is updated by inserting the collected profiling data into the “bucket” whose key most closely matches the given weight value. Updating the profile is done using the following simple algorithm:

1. If this is the first run simply create a bucket for the given weight value and insert the profile data there.

2. If buckets exist find the bucket closest to the given weight value and com-
pare the collected profile data to that bucket’s average:

(a) If the profile data differs less than a certain percentage insert it into this bucket.

(b) If the variation in profile data is too large create a new bucket for the data if, and only if, the weight values also differ more than a certain percentage.

By updating the profile in this way the profile data is capable of adjusting to variations in running time, while only maintaining as few buckets as possible. Consider the depiction in Figure 5.4. When the running time of the task increases rapidly many buckets will be created to reflect this, and when the increase is slow only few buckets are maintained. When doing lookups in the profile the bucket with the weight value closest to the current one is chosen, which can be done in $O(\log n)$ where $n$ is the number of buckets for that specific profile. Since $n$ will always be relatively small this lookup will be considered to have a constant running time.

![Figure 5.4 Two-dimensional profile. The coloured boxes are the buckets created, the dotted line is the predicted running time of the task, and the data points (dots) are actual profile data.](image)

Using this approach towards maintaining profiles solves the first of the deficiencies with profile based scheduling—now variations in running time due to input variations are correctly reflected in the profile.

**Task Centric Profiling**

As mentioned earlier on, the biggest problem with profile based scheduling in highly mobile cyber foraging is the peer centric profiles. Ideally, there should be some way of comparing the “strength” of different surrogate machines to
each other. What is needed is a measure such that the running time of a task on a machine of strength $x$, would be half the running time of that same task on a machine of strength $\frac{x}{2}$. No such perfect strength measure is available—multiple factors are in play when measuring a machines processing capabilities; factors such as CPU architecture, cache structure and speed, main memory speed, and even compilers all play a role in the performance of a modern CPU.

While nowhere near perfect, there are ways to compare the relative strengths of computers, most notably by benchmarking the machines by using a benchmarking suite. In Scavenger both surrogates and clients are benchmarked using the NBench\textsuperscript{1} benchmarking suite, and the score yielded by this benchmark is used as a strength value. In fact, NBench returns two scores for a system: an integer and a floating point performance score. To simplify matters these two scores are combined and the average is used as the peer strength in Scavenger.

This peer strength value gives a fairly good image of how a device performs when it comes to CPU intensive tasks, but it does have its problems. Different tasks may exercise different parts of the CPU, and some task may therefore perform much better on some architectures than on others. This can be alleviated by the use of dual-profiling, as will become clear shortly.

Knowing how strong peers are relative to each other we are able to build task centric profiles, i.e., profiles that are bound to a single task instead of to a (task, peer)-pair. Where the peer centric profiles could contain simple time measurements of earlier executions of the task, the task centric profile must contain some “task weight” that can be scaled by the peer strength. In Scavenger the information stored in these task centric profiles are the expected running time on a machine of strength one. When considering scheduling the task on a peer, this task weight is divided by the current peer strength to obtain an assessment of what the running time would be. The current peer strength referred to in the previous sentence, is the strength of the peer under its current load. The discovery packets periodically sent out by surrogates contain both their strength and their activity count, where the activity count is the number of tasks that are currently being performed within their execution environment. The current peer strength is then:

$$\text{Peer}_{\text{current\_strength}} = \frac{\text{Peer}_{\text{strength}}}{\text{Peer}_{\text{activity}} + 1}$$

It is thus assumed that active tasks share the CPU equally, and any other processes being performed by the operating system are ignored. Likewise, when the task weight is calculated the activity level of the surrogate while it was performing the task is considered.

Whether or not this task centric profile data is usable depends on the assumption described above, that e.g., a peer of strength ten will be able to perform a task twice as fast as a peer of strength five. That this assumption largely holds can be seen in Table 5.1.

\textsuperscript{1}http://www.tux.org/~mayer/linux/bmark.html
5.1 Task Placement

Table 5.1 Task weight measurements. Ideally values should be equal across rows.

<table>
<thead>
<tr>
<th></th>
<th>2 GHz G5</th>
<th>733 MHz G4</th>
<th>1 GHz Pentium 3</th>
<th>900 MHz Celeron M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness</td>
<td>50.980</td>
<td>50.188</td>
<td>54.600</td>
<td>52.755</td>
</tr>
<tr>
<td>Colour</td>
<td>54.012</td>
<td>53.605</td>
<td>57.223</td>
<td>54.249</td>
</tr>
<tr>
<td>Contrast</td>
<td>54.946</td>
<td>55.100</td>
<td>59.229</td>
<td>54.723</td>
</tr>
<tr>
<td>Sharpen</td>
<td>109.859</td>
<td>126.243</td>
<td>106.860</td>
<td>98.404</td>
</tr>
<tr>
<td>Blur</td>
<td>82.545</td>
<td>95.976</td>
<td>83.081</td>
<td>73.399</td>
</tr>
<tr>
<td>Invert</td>
<td>21.208</td>
<td>27.360</td>
<td>32.347</td>
<td>28.038</td>
</tr>
<tr>
<td>Scale</td>
<td>91.072</td>
<td>102.974</td>
<td>44.648</td>
<td>35.300</td>
</tr>
</tbody>
</table>

To produce the data in Table 5.1 four different machines performed seven image manipulation tasks and reported the weight they would assign to that task. All tasks were performed 50 times and the weight reported is the average of these runs. The machines in use are very different with regards to processor architecture; having both a PowerPC G4 and G5, an Intel Pentium 3, and an Intel Celeron M processor. Even with those differences in architecture, it can be seen that the assigned weights are quite similar, and when used in a history based profile these provide a good starting point for the scheduler. Notice though, that although the different machines tend to agree on the weight of most of the tasks, in some cases the architectural differences shine through. Consider for example the last row, where an anti-aliasing scale operation was performed. In these tests the PowerPC based (G4 and G5) machines reported weights that were twice or almost three times as high as the Intel based machines. This shows that while using this “task weight” based on benchmarking scores is no silver bullet, it does in most cases provide good results. And compared to having no knowledge at all about the estimated running time of task execution on unknown surrogates, it makes for a more informed scheduling in unknown environments.

### Dual Profiling

Using the task centric profile just described, gives a good starting point for the scheduler when scheduling in an unknown environment. But, as has been shown, the task centric profiles are not always precise, and peer centric information should therefore always be preferred if such information is available.

Based on this observation Scavenger’s scheduler works with dual profiles; as depicted in Figure 5.5. Whenever a task is performed two profiles are updated—a peer centric profile and a task centric profile. Both of these profiles are updated using the task weight measure, where the measured running time is scaled by both the surrogate’s strength and activity level. When a task is being considered for execution on a given surrogate, the peer centric profile is consulted first in order to give the most precise data precedence. If no peer specific information is available, which is likely in a mobile cyber foraging scenario, the task centric profile is consulted. The information stored here is likely to be less
Chapter 5 Dynamic Scheduling

Figure 5.5 Scavenger’s dual profiles. The peer centric profile contains information about the task on peers A, B, and C, whereas the task centric profiles holds information about the last five runs on any peer.

precise, but tests show that it is still quite effective at guiding the scheduling process.

In the current implementation, there is no pruning of the data within the peer centric profiles. In a real deployment one would have to work with an upper limit on how many peer centric profiles to store, maintaining a kind of peer centric profile cache. When pruning such a cache data about well-known and newly met peers should be allowed to stay—this would ensure that information about the surrogates that reside in the user’s regular environment was retained, while also leaving room for new surrogates that may become regulars.

Scavenger’s Scheduler

To sum up, Scavenger uses all of the information that was in use in the idealised scheduler presented in the beginning of this section. Of this information some is measured through profiling (estimated running time), some is statically defined by the participating peers (bandwidth), some is given at runtime (input size), and some is estimated by the developer of the task (output size). Using this information the scheduling algorithm shown in Figure 5.6 has been implemented. This figure presents the scheduling algorithm in simplified Python-like pseudocode that is very close to the actual implementation.

The running time of the scheduling algorithm is $O(n \log m)$ where $n$ is the number of peers currently available and $m$ is the maximum number of buckets within the two-dimensional profiles. In all practical use the number of buckets in the profiles will be a very small number; so small that the lookup in the profile may be considered to be of $O(1)$ running time, yielding a linear $O(n)$ running time to the entire scheduling algorithm. This modest running time means that the overhead of scheduling is very small, even on resource poor mobile devices.
5.2 Workflow Scheduling

In the preceding section it was shown how Scavenger schedules a single, atomic task within a heterogeneous environment. In the Locusts cyber foraging framework another approach to scheduling was used. In Locusts a task was represented as a directed acyclic graph—a workflow graph—as shown in Figure 5.7.

In this figure the “In” node represents the input to the task and the “Out” node is the task output. The nodes named $S_1$ to $S_5$ are subtasks\(^2\) and the edges connecting the nodes are output from one task that is being used as input to the next task. Some edges are thicker than others illustrating the amount of data that must be moved between the two subtasks. This representation was chosen for a number of reasons; all related to supporting highly mobile cyber foraging:

1. Large, atomic tasks are useless in highly mobile cyber foraging.

2. When there is a need to perform longer running tasks migration is a ne-

\(^2\)Notice that in the papers published about the Locusts cyber foraging framework these subtasks were called services. We stopped using this term to avoid confusion—we are not working with service oriented computing (SOA) as such.
Chapter 5 Dynamic Scheduling

In Figure 5.7 a task represented as a workflow graph of interconnected subtasks.

3. When running many small, interdependent tasks the overhead of moving in- and output back and forth between the client and the surrogate becomes very high.

Ad. 1 When working with highly mobile cyber foraging the amount of time available to perform a given task may be very limited. If for example the user of the mobile device is passing a Wi-Fi hot-spot at walking speed, the first couple of seconds are used joining the network and getting an IP address, and the following second is spent discovering surrogates, which perhaps leaves five to ten seconds to perform tasks before leaving the coverage of the Wi-Fi network. It thus makes sense to represent a task as a graph of small, interconnected subtasks of a more manageable size that can be performed before the client leaves the range of the surrogate.

Ad. 2 As described above, the time that a client is within range of a surrogate may be very limited. In order to be able to perform larger tasks, a way of migrating a running task is necessary. Given a suitable migration method, a task could be shipped to a surrogate, run there for a while, be transferred back to the client device, and, upon detecting another surrogate, shipped to a new surrogate for further processing. While doing this the amount of processing time that is wasted should be minimised. The graph representation lends itself perfectly to migration; at any point in time the task may be checkpointed by fetching any intermediate results, i.e., by fetching the input data that the currently running tasks are working on. If the client looses its connection to a
surrogate, the only work that is lost is that of the currently executing subtasks. Consider for example the running task in Figure 5.8. In this task subtasks $S_1$ through to $S_5$ have been performed, meaning that $S_6$ and $S_7$ may be currently running. When migrating this task only the data on the three marked edges, A, B, and C, must be fetched. After that the task may be submitted to another surrogate, who will be able to resume at subtasks $S_6$ and $S_7$. The only work that is lost by doing such a migration is thus the work of the currently running subtasks.

**Figure 5.8** Migration of a running task.

**Ad. 3** When performing many small, interconnected tasks, instead of a single large task, the overhead of transferring in- and output back and forth between the client and the surrogate may become enormous. By using the workflow graph representation doing coarse-grained scheduling, i.e., doing scheduling where groups of subtasks are handed over to a surrogate, becomes possible.

Both scheduling and migration using a workflow graph representation have been described in detail in [40].
5.3 Lessons Learned

In Section 5.1 the scheduling—or task placement—done in the Scavenger framework was described, and Section 5.2 presented the scheduling done in the Locusts framework. Where Scavenger is only concerned with single task placement, i.e., in finding the optimal place to perform a single, atomic task, Locusts sees the bigger picture and tries to be intelligent about where to place multiple subtasks in order to minimise the communication overhead.

By reading the above description of the two scheduling approaches, one could be tempted to deduct that the scheduling of Locusts was superior to that of Scavenger. This is not necessarily the case though, and in this section I will try explain why that is.

5.3.1 Scheduling of Interconnected Task

The main strength of the Locusts framework is its workflow representation of tasks that makes it possible to schedule tasks in a way that minimises inter task communication. Consider the task in Figure 5.9.

![Figure 5.9 Independent subtasks identified by the Locusts scheduler.](image)

This figure depicts a possible result of running the Locusts framework’s scheduling algorithm on the task shown earlier on in Figure 5.7. Within the task two independent tasks have been identified. These two tasks, each consisting of a number of subtasks themselves, may be handed over to a surrogate to be performed in their entirety. By performing such groups of subtasks at the same surrogate the communication overhead is brought down considerably. Take for example the leftmost task consisting of the three subtasks $S_1$, $S_2$, and
5.3 Lessons Learned

When performing those three subtasks the intermediate data flowing from $S_1$ and $S_2$ into $S_3$ never leaves the surrogate. Not having to waste time on this superfluous transfer of data yields a very big performance increase when performing fine-grained tasks consisting of many small subtasks, which is exactly the kind of tasks we would like to work with in highly mobile cyber foraging.

There are some problems with using this approach though, for example with regards to load balancing. Consider again the graph in Figure 5.9. Given two surrogates the leftmost task may be given to one while the rightmost may be given to the other. Now consider what happens if the execution time of $S_1$ on its surrogate is longer than the combined execution time of $S_3$ and $S_5$ on their surrogate. Once $S_3$ and $S_5$ are done, the surrogate that performed them is standing idle while the other surrogate is responsible for finishing $S_1$ and then performing $S_2$ and $S_4$. In this case it would be nice if $S_2$ could be performed by the idle surrogate, and the result of that computation shipped directly to the surrogate needing it.

As Scavenger’s scheduling has been described, it is concerned not so much with scheduling of larger tasks but more with simple task placement, i.e., with finding the current best place to perform a given atomic task. This does not mean that Scavenger is incapable of scheduling a fine-grained task as the one depicted in Figure 5.7—in fact is does that better than the Locusts framework even without full knowledge of the task structure. The reason for this is quite simple: remote data handles. Whenever a Scavenger task is performed the result of that task may be stored at the surrogate and a remote data handle returned in its place. When the next task in the task graph is scheduled, the information about data locality is considered so that the surrogate holding the data will with high probability receive the next task also—if this leads to the fastest execution of the task. Considering again the scheduling of the task in Figure 5.7 over two surrogates. The schedule using Scavenger could be like depicted in Figure 5.10. $S_1$ is given to the first surrogate, and, because that surrogate is now busy and thus has less CPU power to share, $S_2$ is given to the second surrogate. Now both surrogates are performing one task and Scavenger has to find out where to place $S_3$. Seeing as the input to $S_2$ and $S_3$ are the same, the second surrogate already holds the input for $S_3$ and considering this in the scheduling the second surrogate is chosen for $S_3$. After a while $S_2$ and

![Figure 5.10](image-url)
$S_3$ are finished and a data handle representing their result is returned to the client, who in turn schedules $S_5$ for execution on the second surrogate—the first surrogate is still working on $S_1$. When the first surrogate finishes $S_1$ it returns a data handle, and is thereafter given $S_4$ along with the two data handles representing its input. One of these handles it resolves to its local data store and the other it fetches directly from the second surrogate. Finally, when both $S_4$ and $S_5$ are done their results are returned as data handles to the client, which will probably choose to fetch the data because this ends the task.

In all of this the client assumes a bit more responsibility than it would in Locusts where a group of tasks are handed off at the time. The extra work done by the client is very modest though; it entails receiving small data handles when tasks finish, scheduling any dependent tasks, and sending the data handles to the chosen surrogates. Since scheduling is an $O(n)$ operation in Scavenger, where $n$ is the number of surrogates available this overhead is a very small price to pay for a more fine-grained control over the scheduling process.

5.3.2 Task Migration

Scavenger’s scheduling approach also have some advantages over Locusts’ approach with regards to task migration. In Locusts the client surrenders control of the fine-grained scheduling, and therefore also of all intermediate data, to the surrogates. Because of this task migration becomes a little harder to implement—the only one who knows that migration is needed is the mobile client itself, and without complete control over intermediate data that becomes very hard. In order to overcome this obstacle we implemented a callback mechanism in Locusts, so that the client would get a callback whenever a surrogate finished a subtask. Using this information the client could then request the intermediate data directly from the surrogate. This is in many ways reminiscent of simply returning a data handle, the main difference being that the ability to do more fine-grained, dynamic scheduling is lost.

Using Scavenger the client retains control of the task and its intermediate data. It may not actually have that data locally, but it knows exactly where to get it and has an estimate of how long that transfer would take. When mobile the client application may opt to fetch all intermediate data, so that it is always ready to migrate to another surrogate if a connection is lost. This does not mean that the data handles become superfluous; what a client application should do is to schedule dependent tasks using the remote data handles whenever possible, and then after scheduling the task the intermediate data should be fetched. In this way the intermediate data is fetched while the surrogate is working on it and no time is thus wasted waiting for the transfer.

The main lesson I have learned by working within the field of highly mobile cyber foraging is thus that removing all responsibilities from the mobile client is not necessarily a good idea. When I started out I thought only about how I could relieve the pressure on the small, mobile device as much as possible, and by handing off subtasks in large groups this was obtained. Doing this kind of
scheduling has its drawbacks though, as pointed out above. By keeping the
fine-grained scheduling at the client and working with remote data handles,
all of the advantages of Locusts’ scheduling approach are retained while its
ailments are to some extent cured. If I was to give advice on how to design
a cyber foraging system today I would therefore point solely to Scavenger’s
approach.

5.4 Contribution

With regards to scheduling the main contributions of my PhD work have been:

1. The design and development of a peer discovery mechanism that is suit-
able to the needs of a highly mobile cyber foraging system, i.e., that sup-
ports fast discovery and quick, dynamic change of the advertised inform-
ation for use in the scheduler; [39].

2. The design and development of an execution environment for mobile Py-
thon code. Within this work is the first steps in securing mobile Python
code without altering the Python interpreter; [39].

3. An examination of how a workflow representation could be used to fa-
cilitate scheduling in a mobile environment; [40].

4. The design, implementation, and thorough testing of a novel, adaptive,
dual-profiling scheduler for use in highly mobile cyber foraging. This
scheduler was initially described in [41] and later on in [Paper I] but it
was described in more detail in this thesis. As of writing this, I still have
unpublished results that will be submitted to conferences and/or journ-
als in the near future; [Paper V].
Chapter 6

Development Support

Developing a cyber foraging enabled application is by no means an easy task without a proper set of tools. As mentioned in the introduction to this thesis (see Section 1.1.2) cyber foraging is an involved computing technique, and a framework is clearly needed if it is to be applied outside of academia. Even given a framework, developing a cyber foraging enabled application can be quite demanding, as the developer will be working with distributed and possibly parallel computing—fields within computer science known to be hard for programmers to fully grasp.

When designing both Scavenger and the Locusts framework, development support was an important area of focus. In the Locusts framework the development process was made a bit cumbersome by the fact that the developer was responsible for building not only remote executable tasks (which we called services at that point in time), but also for building the workflow graph representation of the larger tasks. We did make some progress in that respect, which we have published in [Paper IV].

Scavenger, on the other hand, has another focus. Using Scavenger the developer is only responsible for creating small, remote executable tasks, and how these tasks are interconnected is of no real importance—this is handled entirely by the scheduler. The development model of Scavenger is described in some detail in [Paper I], but some more details will be given here.

6.1 Creating a Mobile Code Task

In related systems that provide developer support, the development process is divided into two separate parts: writing the application code and writing the remote executable tasks. This approach means that developers need to be acutely aware of the fact that cyber foraging is being employed in their application. In Spectra for example, the developer has to write remote executable
code as stand-alone programs that can be pre-installed onto surrogates for invoca-
tion. When a Spectra application then reaches a point where a task could be invoked, it must contact the local scheduler, receive information from the scheduler telling it whether or not to use remote execution, and if remote execution is chosen the application must itself invoke the task at the chosen surrogate. This process is eased somewhat in Chroma where the scheduler invokes the remote task itself returning only the result to the client application.

Scavenger tries to do as much as possible on the developers behalf, by handling scheduling, dynamic installation, invocation, and more. We do not aim at building a completely automated system, where the application developer is unaware of cyber foraging taking place; e.g., as was done in the Coign system. To quote Flinn et al. [22] we also believe that: “... a little application-specific knowledge can go a long way.”, meaning that developers should be given the possibility to tailor their applications to use in a cyber foraging environment, if they wish to do so. By fully automating the process a lot is lost, e.g., the possibility to work with delayed processing, i.e., queueing up tasks until surrogates become available.

In Scavenger we wanted an approach that:

1. Would unify the remote executable tasks and the local code so that the developer would only have to maintain one code base.
2. Would yield big performance benefits even for novice developers.
3. Would offer expert developers the opportunity to gain even better performance by offering optional methods to tweak—an example hereof would be working with remote data handles.

How we have achieved this with regards to development support is the focus of this section. As to the performance benefits the reader is referred to Chapter 7.

6.1.1 A Scavenger Task

A Scavenger task is a self-contained piece of Python code adhering to a simple interface; the details of this may be found in [39]. That the code must be self-contained means, that it must itself import all needed modules and define any needed classes and functions. Apart from that the task only needs to adhere to a very small interface: it must have a function named \texttt{perform} defined at the top level of its name space, this function must accept the task’s input as its arguments, and the task’s result must be returned by this function using the standard \texttt{return} keyword. Exceptions thrown within the \texttt{perform} function will be caught by the execution environment, sent to the client application, and re-raised there. A task adhering to that interface is shown in Figure 6.1.

A thing that was left out in the preceding description of Scavenger tasks, is the code validation process that takes place within the execution environment.
upon task installation. Whenever a new task is installed, the execution environment runs a two step validation process on the given task. In the first step it is checked whether any illegal module imports are done—only modules that are white-listed by the execution environment are allowed. The second step checks whether any illegal meta-programming functionality is used. If the task should fail any one of these tests, the task is not installed and the application is made aware of this through an exception. For more information about this validation see [39].

To use a manually created mobile code task, the developer has to ask for available surrogates, check whether the task is already installed on that surrogate and install it if necessary, and finally invoke the task at the surrogate. This approach is supported by Scavenger, as illustrated in Figure 6.2, but we do not encourage developers to use it. The Scavenger library is capable of completely automating all of this; which will be shown in the following section.

```python
from PIL import ImageEnhance as IE
def perform(image, factor):
    factor = 1.0 + factor
    return IE.Sharpness(image).enhance(factor)
```

Figure 6.1 A manually created Scavenger task.

```python
from scavenger import Scavenger
peers = Scavenger.get_peers()
if len(peers) != 0:
    if not Scavenger.has_service(peers[0], 'daimi.imaging.sharpen'):
        Scavenger.install_service(peers[0], 'daimi.imaging.sharpen',"
        from PIL import ImageEnhance as IE
        def perform(image, factor):
            factor = 1.0 + factor
            return IE.Sharpness(image).enhance(factor)
        "")
    output = Scavenger.perform_service(peers[0],
        'daimi.imaging.sharpen',
        'image':image, 'factor':1.0,
        timeout=60)
```

Figure 6.2 Manually performing a task at a surrogate.

The manual approach, using the low level Scavenger API, will not be described in any more detail here. The interested reader is referred to http://code.google.com/p/scavenger-cf/wiki/Tutorial for more information about the low level API. One thing to notice though, is the task naming convention: all tasks must have a unique name on the form org.app.task, where org is the organisation providing the application, app is the name of the application or module containing the task, and task is the name of the task. In
Figure 6.2 The name of the task is daimi.imaging.sharpen.

6.1.2 The Scavenge Decorator

Instead of using the low level API as shown in the preceding section, developers should use the scavenge decorator that has been defined within the high level Scavenger API. A Python function decorator is a function that is called before the function that it is decorating; so if the function foo is decorated with the function bar, then every time the foo function is invoked the bar function is invoked first and given the foo function as its input. Using this we have defined the scavenge decorator that can be used to decorate any function that could benefit of remote execution. Consider the example in Figure 6.3. In this example

```python
1 @scavenge
2 def sharpen(image, factor):
3     from PIL import ImageEnhance as IE
4     factor = 1.0 + factor
5     return IE.Sharpness(image).enhance(factor)
```

Figure 6.3 Using the scavenge decorator to create a sharpen task.

the same image sharpening function that was shown in Figure 6.1 is implemented using the scavenge decorator. The main difference here is, that this sharpen function is not defined in some separate file (or in a text string as in Figure 6.2), it is the same sharpen function as would be used for local execution. I.e., if the developer were not developing for cyber foraging, she would also have this function, the only difference being that @scavenge is added on the line above the function definition, and that module imports have been moved into the function. Having module imports within a function is no problem in Python—if the module is already imported it is simply treated as a no-op.

What happens behind the scenes when calling a function that has had the scavenge decorator applied to it is the following:

1. The decorator function is called with the decorated function as its input.
   (a) Using some meta-programming handles the source code of the function is fetched and the function is renamed to perform.
   (b) A unique task name is generated in the auto namespace based on the name of the module within which the function is defined and the MD5 sum of the function source code. An example name would thus be auto.imaging.421f64af195b9ccce0180cbbf24525fc.
   (c) Partial evaluation is used to specialise the scavenge_partial function on the task name, source, and some other arguments. This specialised function is returned by the decorator so that it may be called on the task input.

2. The scavenge_partial function does the scheduling.
6.1 Creating a Mobile Code Task

(a) If no surrogates are available the local function is simply called on the input.

(b) If surrogates are available the scheduling algorithm is run to find the most suitable place to perform the task.
   i. If the chosen place is the local device the local function is called on the input.
   ii. If a surrogate is chosen the following steps are performed:
      A. The task is installed onto the surrogate if necessary—it may already be installed within the execution environment of the surrogate.
      B. Task input is sent and the task is invoked on the surrogate.
      C. The result of the execution—be it a failure or a success—is returned to the scheduler who returns it to the application.

All of this is done in a way that is completely transparent to the developer using the scavenge decorator. When using the decorated function in the application no special provisions are needed, the function can be used as would any other Python function. If errors pertaining to the use of cyber foraging should occur in the execution of the task, these errors are in the current implementation thrown by the decorated function—which means that some extra error checking is indeed needed. This could be remedied easily by having the Scavenger library catch such errors, and revert to local execution upon doing so. That way there would be no visible difference between local and remote execution of a decorated function.

This development model, where the developer need only apply a simple decorator to resource intensive work, is in our view something unique within the field of cyber foraging. It maintains the benefits of a manual approach, in that the developer is asked to consider where and how remote execution is used, while being almost as easy to use as a fully automated approach.

Using the scavenge decorator as described above yields good results, but in order to optimise the scheduling of tasks and thus better utilise the available resources, the developer can choose to provide the decorator with some extra information about the decorated function. The decorator accepts three optional arguments: an output size relation, a task weight relation, and a boolean indicating whether to pro-actively fetch the output data. An example of the sharpen function using these arguments is shown in Figure 6.4. The three arguments

```python
@scavenge("len(image)", "image.width * image.height", True)
def sharpen(image, factor):
    from PIL import ImageEnhance as IE
    factor = 1.0 + factor
    return IE.Sharpness(image).enhance(factor)
```

**Figure 6.4** Passing arguments to inform the scheduling of a task.

shown in this figure is the used to inform the scheduling process, as it was described in Section 5.1. In the figure the output size of the sharpen function is
defined to be the length of the input image ($\text{len}(\text{image})$), because sharpening an image most likely does not alter the size of the image file significantly. The weight of performing the sharpening is related to the dimensions of the image; i.e., to the width and height in pixels ($\text{image}.\text{width} \times \text{image}.\text{height}$). And finally, if remote execution is used the function is asked to return a remote data handle instead of fetching the output data.

In order to take full advantage of the underlying cyber foraging system, the developer must add this extra information to each decorated function. Finding ways to automate this, so that no arguments are needed, is an interesting issue for future work within the field. Some of this information could e.g., be gathered through the use of profiling—but this is not something we have looked into in our current research.

### 6.2 Working with Larger Tasks

Creating larger tasks, consisting of multiple, interconnected subtasks on the form described in the previous section, is also possible using the simple, decorator-based approach when working with remote data handles. Consider for example the simple, linear task depicted in Figure 6.5. This figure shows the kind of linear task graph that would constantly arise in an image manipulation program such as e.g., AugIm—the demonstrator we have used as a test vehicle for both Scavenger and the Locusts framework.

When a small, resource poor device schedules such a task, it would most probably be interested in handing over the entire task to one surrogate, in order to minimise the communication overhead. Using the scheduling approach of the Locusts framework this would happen automatically, but even using the much simpler scheduling approach in Scavenger, where the scheduler would consider the three subtasks as completely unrelated tasks that must be scheduled, this kind of scheduling is easily obtained. By utilising information about data locality, the Scavenger scheduler would with a very high probability choose to perform the colour adjustment task at the same surrogate that performed the sharpen task, and in the same way the scheduling of the vertical flip task would favour the surrogate that performed the colour adjustment. The only thing that is needed to trigger this scheduling is thus working with data locality, i.e., working with remote data handles. The reason that the scheduling will only “with high probability” choose the “desired” schedule is in fact not

![Figure 6.5 A simple, linear task of the kind that occurs often in image manipulation software such as AugIm.](image-url)
due to a weakness but a strength of the scheduling approach. If for example the sharpen subtask is initially scheduled onto a given surrogate and, while it is performing that operation a much stronger surrogate becomes available, the scheduler may while scheduling the colour adjustment task see, that the overall execution time can be brought down by migrating the data between the surrogates and perform the remaining two tasks on the stronger surrogate. The step-by-step scheduling done by Scavenger thus adds more dynamism to the scheduling process, which is a good thing in a highly fluctuating environment such as the one offered in highly mobile cyber foraging.

In Figure 6.6 a complete code example implementing the task in Figure 6.5 is shown. Notice in the example that the only line of code, where the fact that remote data handles are in use is apparent, is in line 25 where the resulting image is fetched from the surrogate.

6.3 Contribution

Obtaining the kind of development support, that has been presented in this chapter, was an important part of our research objectives. To our understand-
ing, related work within the field have had nowhere near as easily usable a
development model. In fact, only a single system that we know of has focused
on the development model; that being the Chroma system whose development
model was presented by Balan et al. in [6]. In Chroma the developer had to
work with a very clear distinction between remote executable tasks and local
application code; a distinction brought forward by the fact that tasks had to be
written as stand-alone applications adhering to a special interface, that could
be pre-installed and started on surrogate machines. Some development tools
were made available, such as e.g., a stub generator that could create a task stub
for the developer to fill in.

In our development model such a stub generator is not needed—we have
reached an abstraction level where a mere annotation of already existing code is
enough. Apart from that, we have shown how larger tasks consisting of many
subtasks can be catered for efficiently by the use of remote data handles—an
approach that is especially suited to the highly dynamic environment used in
cyber foraging. These results have been published in [41] and [Paper I]. With
regards to developer support, some older results, reporting on our experiments
while working with the Locusts framework’s development model, have also
been published in [Paper IV].
This chapter presents some of the experimental results of my PhD work. As mentioned in Section 1.2, my research objectives have been divided into three parts: defining and examining the requirements of highly mobile cyber foraging, designing efficient scheduling algorithms for use in this setting, and designing a development model enabling novice programmers to develop cyber foraging enabled applications. The first of these objectives may be hard to evaluate other than to conclude that our approach performs nicely in a mobile setting—because of the very efficient peer discovery and monitoring, the scheduling capable of adapting to fluctuations in resource levels, and the mobile code approach. The other two objectives need to be evaluated though, so in Section 7.1 I present a subset of the extensive evaluation that has been done of Scavenger’s scheduling approach. This evaluates not only the scheduling done within Scavenger, it also shows off the benefits of using remote execution in terms of both execution time and energy usage. Section 7.2 presents an evaluation of the development support in Scavenger based on our experiences with building a complete mobile, cyber foraging application.

### 7.1 Scheduling

Scheduling, or task placement, is a very important aspect of any cyber foraging system. At the heart of cyber foraging lies the use of resources on stronger surrogate machines, and while doing so the mobile device must be able to decide 1) if remote execution is a viable option in the current environment, and if so 2) where to perform a specific task. The main differences between Scavenger and the related systems with regards to scheduling is the combination of task and peer centric scheduling and the use of adaptive profiles capable of adapting to variations in task execution time related to input variations. Task centric scheduling is used to support mobility with regards to physical environment, i.e., to make the scheduler capable of assessing task execution time
on unknown surrogates, and peer centric scheduling is used in order to more precisely estimate running times when working with known surrogates. Scavenger thus works with an adaptive, dual profiling scheduler and it is this scheduler that will be evaluated here.

An extensive benchmark study has been performed to validate the adaptive, dual profiling scheduler, and to compare it to other profiling scheduling approaches—i.e., comparing it to pure peer or task centric profiling. The testing has taken place in a highly heterogeneous environment, containing a number of computers and devices of differing architecture as described in Section 7.1.1.

This evaluation starts off by presenting some simple results showing that it is indeed fruitful to employ cyber foraging to support applications such as the AugIm demonstrator presented earlier in this thesis. This results of this evaluation is discussed in Section 7.1.2.

Many aspects are of importance when testing a cyber foraging scheduler. First and foremost, the scheduling overhead must be as low as possible to keep the number of CPU cycles wasted when doing local execution at a minimum. The scheduler being evaluated here utilises not one but two profiles, and these profiles are two-dimensional necessitating a logarithmic search before each look-up/update. In order to show what maintaining such profiles does to the scheduling overhead, a test has been performed comparing this overhead to those of a task centric and a peer centric scheduler. This test is described in Section 7.1.3.

One of the key benefits of using the task centric profile is that the scheduler should be able to quickly recognise the best possible surrogate for the job; even in environments where no previously known surrogates exist. This is tested in Section 7.1.4.

A part of the use case described in the introduction, in Section 1.1.1, was that heavy computing tasks could be queued up and then performed at a later point in time when surrogate machines become available. Section 7.1.5 presents a test of the kind of batch scheduling that this leads to.

In Section 7.1.6 the schedulers’ ability to adapt to varying running times caused by variations in input is tested.

And finally, in Section 7.1.7, some measurements are presented that show how energy may in fact be preserved by utilising cyber foraging through Scavenger to off-load resource intensive tasks.

7.1.1 Experimental Setup

Except otherwise stated, all experiments have been performed in a special test environment we have created that contains a mixture of machines varying in speed, architecture, and connectivity. The test environment is shown in Figure 7.1. The figure shows the network architecture, consisting of three desktop
7.1 Scheduling

PCs connected directly via cable to the Wi-Fi router, and two mobile devices, an Asus Eee 900 UMPC and a Nokia N800 Internet tablet, connected to the same router via Wi-Fi. On each device its architecture and CPU speed is listed along with its measured NBench strength rating; which is the strength measure that is used in the task centric profiling. The devices have been named A through E, and these names are used throughout this section. The operating systems used on the devices also differ: A is running Mac OS X 10.5, B is running Mac OS X 10.4, C is running Ubuntu Linux 9.04, D has Ubuntu 9.10, and E is running Maemo Linux 4.1.

7.1.2 Performance evaluation

For these experiments the client application was installed on E, and machines A, C, and D were available as surrogates. These surrogates are all fairly slow compared to modern machines, but when compared to the mobile device, they are lightning fast, and, as will be shown shortly, using even old machines like these as surrogates yields big benefits for the mobile devices utilising them.

For the first benchmark a linear string of three image operations, expressed as Scavenger tasks, are performed. The operations are an image sharpening, colour adjustment, and finally a contrast adjustment. These operations are performed both on the original five megapixel image and on a thumbnail version, roughly half a megapixel in size, which is the one shown in the UI of the mobile application. All tests have been run 50 times and the measurements reported here are the averages of these runs. The test was initially run without surrogates to measure the local execution time, and then with each surrogate turned on in turn. If all surrogates were turned on at the same time, the scheduler
would soon find that the PowerMac G5 was the strongest, and all tasks would be sent to that same surrogate. The results of the test are shown in Figure 7.2.

![Figure 7.2](image-url)  
**Figure 7.2** Results of the first benchmark. Each bar shows the total running time as experienced by the client when performing the task. Within three of the bars a lighter bar is depicted which illustrates the fraction of the time spent doing CPU bound work.

A number of things can be concluded looking at this figure. For one it is immediately clear that employing cyber foraging is essential when working on the original, five megapixel image. If the mobile device itself tries to perform the tasks it takes more than two and a half minutes—time in which the mobile device is left completely unresponsive because its CPU is 100% utilised. In fact, even performing the operations on the thumbnail version of the image is quite resource intensive for the mobile device, using around nine seconds to perform all three operations. This is detected by the scheduler, and these preview operations are thus also forwarded to surrogates when available. The bars depicting the running time of handling the original image when using a surrogate all have a lighter shaded bar within them. This illustrates the amount of CPU time spent actually performing the operations on the surrogate, and the rest of the running time can thus be accredited to network overhead. This network overhead of roughly eight to ten seconds cannot be reduced further by adding stronger surrogates, which sets a hard lower bound on the obtainable benefits when working with such relatively large in- and outputs. The client device, $E$, used an IEEE 802.11b connection, so in a setting where e.g., an IEEE 802.11g connection were in use, this network overhead would be brought down and stronger surrogates would be more beneficial.
7.1 Scheduling

7.1.3 Scheduling Overhead

In order to measure the scheduling overhead Scavenger’s code was instrumented to measure the time spent selecting a surrogate. This was measured for the task and peer centric schedulers as well as for the adaptive, dual profiling scheduler. The task was scheduled 50 times using six different input images of varying size. Input size is varied because marshalling of the input is a necessary step in the scheduling, in order to get the size of the input when transferred over the network. The task being performed is a brightness adjustment of the input image.

![Image Size vs Overhead](image)

**Figure 7.3** Scheduling overhead when using the adaptive, dual profiling scheduler. The task being performed is a brightness adjustment of an image. Each bar shows the percentage of the total running time that was spent scheduling. Over each bar the total running time in seconds is shown.

Figure 7.3 shows the result of running this benchmark using the adaptive, dual profiling scheduler. The plot shows the percentage of the total running time that is spent doing scheduling. For the first very small image, the overhead is a large percentage of the total running time; 16.6%. The total running time of this task is very low, so low in fact that local execution was chosen by the scheduler in all of the runs, meaning that the time spent scheduling is time wasted because no surrogate was chosen. This result is to be expected, others have found that tasks must be of a certain size, before they will benefit from cyber foraging [22]. As soon as the tasks get a little larger, i.e., already when the image has a size of 400x300 pixels, the scheduling overhead is very low compared to the total running time.
In order to compare the schedulers the overheads of the three different schedulers are plotted in Figure 7.4. The results shown in this figure are as expected: The task centric scheduler has the smallest overhead since its profile only contains one entry per task; the peer centric has an entry for each (peer, task)-pair, which makes its overhead slightly larger; and finally the combined scheduler that must consult both profiles is somewhat slower than the other two. The figure shows, that the cost of scheduling is somewhat larger for the more complex scheduler, but the increase in overhead stays within a reasonable 30% of the scheduler with the smallest overhead; the task centric scheduler. This benchmark shows, that even though multiple profiles are maintained, the cost of scheduling is not excessively high, and, as shown in Figure 7.3, when related to the total running time of the tasks the overhead is very low.

7.1.4 Learning Quickly

An important aspect of task centric profiling is, that it makes it possible for the scheduler to make informed task placement, even in cases where the currently available surrogates are unknown. In this benchmark, three tasks are performed in serial on the input image: the image is sharpened and then its brightness and contrast are adjusted. The benchmark is run ten times in the environment depicted in Figure 7.1, and before each test run the profile data of the client is emptied, to simulate performing the tasks in a previously unknown environment.
The best place to perform this task in the test environment is at surrogate $A$, so ideally this surrogate should be chosen as early as possible. The results of performing the test is shown in Table 7.1.

Initially all schedulers have no information about the task, meaning that they have no estimates of how long the task execution is going to take, and an execution time of zero seconds is then assumed on all devices. Because of this local execution is chosen by all three schedulers—the input data already resides at the local device, and thus this device is favoured by the scheduler. Now profiling information from a single run is available, and the task centric scheduler is able to employ this information to select the correct surrogate device in all succeeding runs. The peer centric scheduler unfortunately needs to perform the task at all available surrogates at least once, before it is able to select the correct surrogate device. It is thus not before the sixth run that the peer centric scheduler is capable of choosing the correct surrogate.

Note that the same surrogate is always chosen for all three tasks. This is because of the data locality factor that is present in all three schedulers. The result of the intermediate tasks, the sharpening and brightness adjustment, are left on the surrogate performing the task. This means, that when the next task is scheduled the input for that task already resides at that surrogate, and that surrogate will then naturally be preferred by the scheduler.

This small test shows that the physical mobility Scavenger aims at supporting actually works. Using the task centric scheduler when no other information is available means, that the appropriate execution plan is chosen much faster than when relying solely on peer centric data. It is important to note that using task centric profiling is not the same as simply choosing the strongest surrogate at all times. Using the profile makes it possible for the mobile device to choose local execution when that makes more sense.

Table 7.1 Results of performing the task placement benchmark.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Test run #</th>
<th>Distribution</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer centric</td>
<td>1</td>
<td>E, E, E</td>
<td>141.6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>D, D, D</td>
<td>29.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>20.4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>C, C, C</td>
<td>27.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>B, B, B</td>
<td>31.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>A, A, A</td>
<td>21.2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>A, A, A</td>
<td>20.0</td>
</tr>
<tr>
<td>Task centric</td>
<td>1</td>
<td>E, E, E</td>
<td>141.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A, A, A</td>
<td>20.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>23.3</td>
</tr>
<tr>
<td>Adaptive, dual-profiling</td>
<td>1</td>
<td>E, E, E</td>
<td>144.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A, A, A</td>
<td>19.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>22.7</td>
</tr>
</tbody>
</table>

7.1 Scheduling
7.1.5 Batch Scheduling

When no surrogates are available it is common for cyber foraging enabled applications to queue up tasks for later processing. When a client device having such queued up tasks enters an area with surrogates, it is interested in scheduling all of these tasks in parallel using the available surrogates. This scenario is tested in the benchmark covered here. The client device has a queue of 24 image operations that is to be applied to eight images; more precisely each of the eight images are sharpened, have their brightness adjusted, and finally have their contrast adjusted in that order. When the benchmark is started the first eight tasks are immediately scheduled, and as soon as one of these finish the task waiting on its output is scheduled. The benchmark has been run with six different images of varying size, each run has been repeated 50 times, and the results presented here are averages of these runs.

The results of this benchmark are shown in Figure 7.5 where the total running time of the three schedulers for the six different input image sizes are plotted. For such batch processing the schedulers have comparable performance, with the adaptive, dual profiling scheduler being slightly slower because of the added scheduling overhead. The comparable performance points at, that all three schedulers are equally good at selecting the right place to perform a given task in this environment. This means e.g., that the task centric scheduler, using its simplified view on task complexity and surrogate strength, is capable of creating schedules that are just as good as the ones selected by the

![Figure 7.5 Results from the batch scheduling benchmark. Eight parallel tasks, each consisting of three subtasks, are scheduled in an environment with four heterogeneous surrogates.](image-url)
more precise peer centric scheduler.

When doing batch scheduling it is important that tasks are distributed in a way such that the available surrogates are equally utilised. But this utilisation must be relative to the surrogate’s relative strength, so that stronger surrogates are assigned more tasks than weaker ones. Given the relative strength of the surrogate devices in the test environment in Figure 7.1, an ideal scheduler should give a bit more than 40% of the tasks to surrogate A, and approximately 20% each to surrogates B, C, and D. The actual distribution of tasks over the available surrogates that was obtained in the benchmarks is shown in Figure 7.6.

This figure shows the distribution of tasks in the benchmark using the largest input image. The first thing to notice in Figure 7.6 is the distribution of tasks using the task centric scheduler, which is as expected tied firmly to the strength ratings of the surrogates: surrogate A, being the strongest of the bunch, has received most of the tasks, B, being the weakest one, has received the least, and C and D, being of comparable strength, have received almost the same amount of tasks. Given a task of higher granularity, e.g., by scheduling more than eight tasks in parallel, the relation to the strength rating would become even more apparent.

In a scenario such as the one benchmarked here, the two remaining schedulers will both be using their peer centric profiles most of the time, and their choice of task distribution is thus expected to be alike. There is some small
variations in the chosen distribution between the two schedulers, and these should probably be accredited to the inexact nature of the task weight computation. When using per (task, peer)-pair profiles there is a greater chance of one of the profiles being skewed for some time, leading to a more diffuse scheduling. That the task weight computation is inexact is no problem for the task centric scheduler, since it uses the same value when considering execution on all peers.

A final thing that may be shown by this benchmark, is how efficient the schedulers are at using the information about network bandwidth and the locality of data. As in the benchmark presented in Section 7.1.3 the result of intermediate tasks are left at the surrogates performing those tasks, and the scheduler should thus be able to employ knowledge about that when scheduling dependent tasks. During each test run the client logged where each task was sent, and using that information the task schedules shown in Figure 7.7 have been created. Figure 7.7(a) shows the schedule chosen by the adaptive, dual profiling scheduler in one of the 50 runs using the image of size 400x300, and Figure 7.7(b) shows the schedule chosen by the task centric scheduler in one of the runs using the larger 2000x1500 pixel image.

![Schedules for an image of size 400x300 (a) and an image of size 2000x1500 (b). Task nodes are coloured to show where they have been performed. On each task node its position in the overall schedule is printed—i.e., the node with position 1 was scheduled before the node with position 2.](image)

The key difference between Figure 7.7(a) and 7.7(b) is the degree to which locality information is used. When scheduling the tasks using the smaller 400x300 pixel image, the cost of transferring the image between surrogates is relatively low—especially so between surrogates A, B, and C which are connected by a 100 MBit link. Notice though, that the tasks given to surrogate D are performed in serial on that surrogate. This is because surrogate D is
### 7.1 Scheduling

#### 7.1.6 Adaptive Profiling

The next benchmark presented here shows the benefits of working with an adaptive profile, i.e., a two-dimensional profile that can adapt to the fact that task running time varies with input. This benchmark performs a sequence of tasks that could easily occur in the use case described in the introduction. Initially five tasks adjusting the brightness of an image are scheduled on a small version of the input image, this simulates the user previewing operations on the mobile device, and then the same task is scheduled twice using the original, large version of the image, simulating that the user has chosen to apply the previewed operations to two images. The benchmarks has been performed using two schedulers, both of them dual profiling but only one working with adaptive profiles. Each test has been run 50 times and the results reported here are the averages of those runs. Table 7.2 shows aggregated results from the benchmark.

When working with a non-adaptive profile the first task, operating on the small input image, was scheduled at the client device and the second task was scheduled at surrogate A. Seeing as performing the task at the surrogate is way more expensive than local execution, the five next tasks are then scheduled at the client device. Two of those five tasks are operations on the large image, which are quite heavy to perform. This alters the weight of the task in the profile, and from that point on the task is considered heavy and therefore performed at the surrogate every time.

The adaptive scheduler on the other hand immediately detects that it is connected to the network using a Wi-Fi connection, and the transfer time for intermediate data is thus much higher for this surrogate. Initially all data must be sent from the client device to a surrogate over the clients Wi-Fi connection, which means that surrogate $D$ is just as likely to be chosen as any of the others.

In the schedule of the larger image in Figure 7.7(b) the cost of transferring intermediate data between surrogates becomes more significant. Because of this, dependent tasks are in most cases scheduled on the surrogate holding the data.

Both data locality and information about network bandwidth is thus used in a reasonable manner when scheduling using the schedulers defined within Scavenger.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Image</th>
<th>Surrogate A</th>
<th>Localhost</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual profile</td>
<td>small</td>
<td>246</td>
<td>4</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>98</td>
<td>2</td>
<td>8.73</td>
</tr>
<tr>
<td>Adaptive, dual profile</td>
<td>small</td>
<td>5</td>
<td>245</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>99</td>
<td>1</td>
<td>7.46</td>
</tr>
</tbody>
</table>

Table 7.2 The effect of adaptive profiling. Ideally the schedulers should perform all tasks on the small image locally and all tasks on the large image on surrogate A.
working with two very different task complexities, brought on by the difference in input image size. It therefore creates separate profiles for the two image sizes, meaning e.g., that the tasks working on the small image are almost always scheduled at the client device, while the tasks working on the larger image are only once scheduled on the client device.

This test shows, that using an adaptive profile makes it possible for a scheduler to adapt more quickly to variations in running time brought on by variations in input. Furthermore, in special cases such as the one shown in this benchmark, inefficient scheduling caused by continuous variations in input complexity can be avoided completely by having an adaptive profile.

7.1.7 Energy efficiency

Apart from the obvious benefit of having increased performance with regards to running time, initial studies conducted using Scavenger have also shown that energy is preserved by offloading resource intensive tasks. It is commonly believed that, in order to preserve precious energy resources, mobile devices should turn off their Wi-Fi interfaces as much as possible. The results presented here show that in some cases mobile devices should turn their Wi-Fi interface on to save energy—at least when turning it on means that CPU intensive work may be off-loaded.

The mobile device used in these tests is a Nokia N810 Internet Tablet with a 400 MHz TI OMAP 2420 (ARM1136) processor and 128 MB of DDR RAM. The surrogate in use in the benchmarks is a 2008-model MacBook with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of 1067 MHz DDR3 RAM running Mac OS X 10.5.8. The network media connecting the client and the surrogate was an IEEE 802.11g network served by a Linksys WRT54G router. Unless stated otherwise, the network was only in light use while tests were performed.

The application used for benchmarking was AugIm. The tasks of this application have been used in the following two experiments. The first experiment simulates that the user browses her images, selects an image for editing, previews three image operations on a 0.3 megapixel preview version of the image, and finally commits to these changes by applying them to the original 5 megapixel image. This is done for 15 images in each test run. The image operations performed are sharpening, brightness, and contrast adjustment, and all of these are targets for remote execution. All experiments have been performed six times, and the values reported here are the averages of these runs. The results of this experiment are listed in the first part of Table 7.3.

Performing such large tasks, i.e., working on the 5 megapixel images, is very resource intensive for the mobile device, while the surrogate can perform such operations in mere seconds. It thus comes as no surprise that the running time of the tests is brought down substantially by utilising cyber foraging. When the running time is reduced, so is the total amount of energy used. What is less obvious though, is that the immediate energy usage (energy
used per time unit) is also reduced; shown as the last column of the first part of Table 7.3. Turning on the Wi-Fi interface to use a surrogate thus brings the immediate energy consumption down by 30%, compared to turning off Wi-Fi and performing the task on the mobile device. This result is obtained because, on the chosen hardware platform, a fully utilised CPU consumes more energy than having Wi-Fi turned on and having an almost idle CPU; for more information see [Paper III].

The experiments performed in the first experiment asked the mobile device to perform very resource intensive tasks; tasks so large that a decent cyber for-aging application should refuse to perform them when no surrogates are avail-able, and rather queue them up for later processing when surrogates do become available. In the next experiment only operations on the smaller preview im-ages are performed. These operations can be performed by the mobile device in a few seconds, so local execution is feasible. But, if a surrogate is available, Scavenger’s scheduler soon finds out that the tasks may be performed faster by using the available surrogate. The test simulates that the user browses her images, selects an image, previews three image operations, and then queues the task for later execution. This is done for 75 images in each test run. Again, each test has been run six times and the numbers reported here are averages of these runs. The results of the second experiment are shown in the second part of Table 7.3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Experiment 1</th>
<th></th>
<th>Experiment 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Energy</td>
<td>Time</td>
<td>Energy</td>
</tr>
<tr>
<td></td>
<td>seconds</td>
<td>mAh</td>
<td>seconds</td>
<td>mAh</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy/mAh/s</td>
<td></td>
<td>Energy/mAh/s</td>
</tr>
<tr>
<td>No surrogates</td>
<td>2928±142</td>
<td>187±22</td>
<td>1666±27</td>
<td>58±2</td>
</tr>
<tr>
<td>Wi-Fi off</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>No surrogates</td>
<td>2868±90</td>
<td>206±18</td>
<td>1642±44</td>
<td>78±9</td>
</tr>
<tr>
<td>Wi-Fi on</td>
<td>98%</td>
<td>110%</td>
<td>99%</td>
<td>134%</td>
</tr>
<tr>
<td>Surrogate available</td>
<td>579±26</td>
<td>26±2</td>
<td>1135±54</td>
<td>44±3</td>
</tr>
<tr>
<td>Wi-Fi on</td>
<td>20%</td>
<td>14%</td>
<td>68%</td>
<td>76%</td>
</tr>
<tr>
<td>Surrogate available</td>
<td>1212±58</td>
<td>49±4</td>
<td>73%</td>
<td>84%</td>
</tr>
<tr>
<td>noisy network</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 7.3* In the first test, three image operations are applied to 15 image previews and then to the original images. In the second test, three image operations are applied to 75 of the smaller preview images.

Looking at the results of this experiment, a number of things become apparent. The first thing to notice is that again the total running time has been reduced by using remote execution, while also reducing the energy consumption. In this experiment however, the running time reduction is slight, because only a small fraction of the time is used actually performing the CPU intensive tasks—most time is spent idling while waiting for user input. Comparing the test with Wi-Fi on and no surrogates available to the one where a surrogate is available, the running time is brought down 30%, but the energy consumption
in the same tests is reduced by 43%. This is also reflected in the immediate energy consumption, shown in the last column, being lower when using a surrogate. The immediate energy consumption is, however, lowest in the test where the Wi-Fi interface is turned off, and the mobile device is thus performing all tasks on its own. The total energy consumption is still considerably higher though, using 32% more energy than when using remote execution, so using a surrogate is still preferable.

We have also performed test runs, where the network medium is in use by other computers. In this test the surrogate and another N810 device are continuously communicating, maintaining an approximately 800 kb/s rate of data transfer between them. Introducing this noise naturally made the cyber foraging perform slightly slower, thus using more energy in total. The immediate energy usage also became slightly larger in a noisy environment, but this may be accredited to the broadcast nature of Wi-Fi, where data sent between other peers in the network is overheard by all peers. What should be noted with regards to this test is, that the overall energy consumption is still 16% less than that when disabling Wi-Fi altogether, which is because the total running time is brought down by about 27%.

It is important to note, that while the necessary operations still take considerable time even when powerful surrogates are available, the user experience is completely different. When the mobile device is handling all operations, it is using its CPU 100% and is therefore sluggish and unresponsive. In contrast, when using a surrogate, the mobile device will be spending most of its time waiting for data and user input, and will therefore be responsive and capable of handling other tasks the user may decide upon.

7.1.8 Conclusion

In Scavenger a large number of schedulers have been implemented—far more than have been presented here. The schedulers yielding the best performance are the profile based ones, and this evaluation has presented three such profile based scheduling approaches: the task centric approach, where a profile is maintained for each task, the peer centric approach, where a profile is maintained for each (task, peer)-pair, and the adaptive, dual profiling approach that combines the task and peer centric profiles and furthermore adds adaptability to the profiles, so that they may adapt to changes in input complexity. These three scheduling approaches have been benchmarked and compared to see how they stack up to each other. All tests have been run in a heterogeneous computing environment using machines of differing architecture and operating system.

That using cyber foraging is able to bring down execution time of CPU intensive applications on mobile devices should come as no surprise; this has been shown by related systems numerous times. The tests presented here, in Section 7.1.2, that show this only adds two things to that bulk of data: 1) that Scavenger is a fully working cyber foraging system, and 2) that even when
using modern mobile devices offloading CPU intensive work to even old computers, such as the ones found in our test environment, yields big performance gains. Perhaps more surprisingly, using remote execution on these modern devices, with their relatively big, power hungry CPUs, also yields considerable energy savings, as was shown in Section 7.1.7.

In Section 7.1.3 the scheduling overhead of the three chosen schedulers was measured. This benchmark was meant to show what the overhead of maintaining multiple adaptive profiles was, when compared to maintaining only a single task or peer centric profile. The added scheduling overhead for maintaining these more complex profiles was roughly 30% when compared to the task centric scheduler, and roughly 15% when compared to the peer centric scheduler. The overheads are relatively low, e.g., the complex scheduler used 50 ms on average to devise a schedule, so on any reasonably sized computing task paying for this overhead can easily be outweighed by the benefits of utilising a stronger CPU. Given the benefits of working with the adaptive, dual profiling scheduler, paying the added cost for scheduling is considered worth it.

The key benefit of the task centric scheduler is its ability to make informed guesses at how a task will perform on a previously unknown surrogate device. This was tested in Section 7.1.4. As expected, the peer centric scheduler had to try all available surrogates in turn before it was able to select the best surrogate for the task, thus wasting a lot of time performing the task at slower surrogates. The task centric, and therefore also the dual profiling, scheduler were able to select the correct surrogate after a single execution of the task—as soon as a task weight is available an execution plan can be devised. It is important to note here, that the task centric scheduler does not merely select the strongest surrogate, which could be done without calculating the task weight. It needs the task weight to assess whether local execution would be faster than remote execution, and if remote execution is preferable it selects the surrogate that is currently most suited for the job, considering not only surrogate strength but also its current utilisation etc. When designing for mobile cyber foraging, where working with unknown surrogates is common, using a task centric profiling approach is therefore recommended.

Another important ability of a cyber foraging scheduler is being able to do batch scheduling, i.e., scheduling of multiple tasks onto currently available surrogates. In Section 7.1.5 a batch of eight image manipulation tasks, each consisting of three subtasks, were scheduled in a test environment with four very different surrogate machines. Running this benchmark showed that all three schedulers were capable of utilising multiple surrogates in parallel. Looking at the running time of the different schedulers it seems that the task centric scheduler was slightly better than the other two. This may be accredited to two things: first of the scheduling overhead is slightly smaller for this scheduler, and secondly its choice of execution place depends more heavily on current resource availability and less on the task weight.

The batch test also showed how nicely data locality is taken into consid-
eration during scheduling. Taking such a thing into consideration is trivial in a traditional task scheduler, where the placement of all tasks and subtasks is chosen before anything is scheduled. But in a mobile system such as Scavenger, the scheduling must be done dynamically at run-time, because resource availability may have changed since the last subtask was scheduled.

The benchmark presented in Section 7.1.6 showed the benefits obtained when working with an adaptive profile. This benchmark showed that the adaptive profile could greatly improve performance in cases where the same task was performed with varying input complexity.

This evaluation has shown of varying aspects of the adaptive, dual profiling scheduler used in the Scavenger cyber foraging framework. The scheduler has been shown to be able to work efficiently with the adaptive profiles, the locality of in- and output data, and the use of unknown surrogates through the task centric profile—all properties that makes it function well in the highly mobile setting it is designed for.

7.2 Developer Support

Returning to Engelbart’s ABC-model as introduced in Section 2.3, building applications using a framework is a B-level activity which supports A-level activities, i.e., user work. But at the same time, doing this B-level work evaluates the results of the C-level activities, i.e., the usefulness of the framework is evaluated. With regards to developer support, the focus in our research within the Locusts project has been on creating frameworks (C-level activity) to support easy development of cyber foraging enabled applications (B-level activity). The main focus in this part of the evaluation is therefore on the activity of developing a cyber foraging enabled application—the applications themselves are not evaluated in any way. That the applications benefit from using the framework was shown in detail in the previous section.

Grimm et al. [26] present four criteria for evaluating a framework from a developer’s point of view: completeness, complexity, performance, and utility.

**Completeness** is an assessment of whether the framework is powerful enough to build useful applications. Are the primitives offered by the framework complete enough to enable the developers to create different kinds of applications?

**Complexity** refers to how hard it is to write code using the framework. Ideally the framework should make it easy to develop cyber foraging enabled applications.

**Performance** refers to the performance of the applications built using the framework. In the case of a cyber foraging framework the developer will expect that utilising remote resources will increase the performance of the application.
Utility is a measure of whether or not the framework has been successful in allowing developers to build cyber foraging enabled applications. As Grimm et al. say: “After all, a system architecture is only as useful as the programs running on top of it.”.

All but one of these criteria will be adopted here when evaluating the Scavenger framework. The criteria that is left out is performance; that there may be performance increases when utilising Scavenger, both with regards to running time and energy usage, was shown in detail in Section 7.1 and will not be discussed further here.

Throughout the development of both Scavenger and the Locusts framework, we have developed small and larger prototype applications. The main prototype is the Augmented Image Manager, called AugIm, which implements the first use-case described in Section 1.1.1; i.e., the use-case where a tourist is editing images taken with a smart phone directly on the phone itself. This demonstrator exists in two versions—one created for Nokia N800/N810 Internet Tablets using the Locusts framework, and one created for Nokia N900 smart phones using the Scavenger framework. When talking about AugIm the focus in this section is on the latter version. This evaluation is presented in Section 7.2.2.

The section on AugIm evaluates the use of the high level Scavenger API, where cyber foraging is applied by adding a simple decorator to resource intensive functions. The low level API has also been evaluated; both by the fact that the high level API is built using the functionality offered by the low level API, and through the work done by a number of master’s students. This evaluation is presented in Section 7.2.1.

### 7.2.1 Evaluating the Low Level API

The low level Scavenger API, that can be used to gain manual access to cyber foraging capabilities, has been evaluated in a number of ways. The API consists of only a small handful of functions enabling the developer to:

1. Ask for a list of currently available surrogates. This contains up-to-date information about each surrogate with regards to strength, utilisation, and network bandwidth.
2. Check whether a task is installed on a given surrogate.
3. Install a task onto a surrogate.
4. Perform a task on a surrogate.
5. Create and resolve remote data handles at a surrogate.

The first evaluation of this framework is the fact that the high level API, offering fully automated cyber foraging, is built from these simple constructs.
The low level API is thus complete enough to implement all the functionality of the high level API with regards to scheduling, error handling, remote execution etc. In order to properly evaluate completeness of a framework it is not enough to implement a single application though; and furthermore it is preferable if the developer using the API is not the same person that designed it. Fortunately, multiple applications have been built using the low level API by master’s students who have been using Scavenger to experiment with scheduling [2, 33] and security models for cyber foraging [1].

The work done by Andersen and Schurmann [2] and Junge and Jakobsen [33], where Scavenger was used as a testbed for experimenting with task scheduling, led to the conclusion that the version of Scavenger they were working with was missing one important feature in order to support some scheduling approaches. The missing feature was the ability to preemptively push input data onto a surrogate and have a remote data handle returned. Using such functionality high level schedulers, that have a view of the entire task as a graph of interconnected subtasks, may be able to prime a surrogate that is already performing a task on its behalf, by pushing input for the next task while it is performing the previous one. By doing this the task initialisation overhead of the next task may be brought down dramatically. This functionality was therefore added to the low level API, and apart from that the overall experience of working with Scavenger was positive. In particular the decision to catch, serialise, ship back, and the re-raise exceptions occurring in task code was appreciated by the developers; this gave them immediate feedback when something was wrong in their task code. There is one minor annoyance with regards to this error reporting though and that is the reported line numbers. Consider the example in Figure 7.8.

![Figure 7.8 A remote executable task as seen in the application and in the execution environment.](image)

What this figure shows is two things: first off, it is apparent that the line numbers that the task resides on differ between the application and the remote executable task. So if the same error occurs locally and on the remote host
different line numbers will be reported. Secondly, when the mobile code task is installed onto the surrogate it is not presented to the surrogate in the exact same form as seen by the developer in the application. During the validation of the task code a monkey patching header is appended in order to make sure that no malicious built-in functions are called, and to make sure that the task may only access certain parts of the file system. Adding this monkey patching header means that the line numbers reported in error messages do not even match with the line numbers within the task—had they done so the developer would have had a slightly better chance of quickly identifying where the error occurred.

Apart from the work presented above, there also is some ongoing work being done with Scavenger at the moment. Jesper Jakobsen is currently writing his master’s thesis on the subject of opportunistic, dynamic cluster computing, where he is using Scavenger and the low level API to experiment with building a complete cluster solution for scientific computing using scavenged resources on a local area network. Ubbe Welling, a PhD student within computer science at Aarhus University, is currently using Scavenger to experiment with programming models for distributed, parallel processing. And lastly, Janne Parkkila at Lappenranta University of Technology is currently employed by Jari Porras to work on various aspects of cyber foraging using Scavenger.

As has been described here, the low level API of Scavenger has been in use by a little more than a handful of developers. What Scavenger offers to these developers is a turn-key solution to dynamic, adaptable, distributed processing. By installing the Presence and Scavenger daemons on a machine reachable over the local area network this machine is immediately ready to perform tasks, and all the developers need to do is import the scavenger library and start using these remote resources.

**Conclusion**

With regards to completeness, the low level API has proved to be versatile enough to support the development of schedulers suited for both mobile and static environments, as a testbed for distributed, parallel processing, and for building an opportunistic cluster for scientific computing. The API is small and simple, but the few handles offered seem to be enough to facilitate diverse usage patterns. There is one functionality that is missing, which would increase the scalability of applications built using Scavenger when operating in larger networks. Scavenger was designed to work in a mobile cyber foraging setting, where the expected number of surrogates available is relatively small, and where a client would normally be using only a handful of surrogates at the same time. When used on larger local area networks, e.g., in the opportunistic cluster for scientific computing, a client may wish to work with a large number of surrogates at the same time in order to solve some large computational problem. In order to use Scavenger more efficiently in such a setting an asynchronous version of the API would be needed, so that the client would not
need to keep TCP connections to all active surrogates.

The complexity of the API is very low—the developer need only acquaint himself with five functions, an object representing a surrogate/peer in the system, and the few restrictions that are posed on task code (described in detail in Section 6.1). All developers working with the framework have understood these simple constructs almost immediately.

The utility of the low level API has been proved by the successful applications developed using Scavenger. The students that have worked with it have all been capable of creating working prototype systems.

7.2.2 Evaluating the High Level API

AugIm is the only fully-fledged application that has been built using Scavenger’s high level library, i.e., using only the Python decorators and none of the low level functions. Instead of porting AugIm from the N800/N810 version to the N900 I deliberately chose to implement it entirely from scratch, so as to be able to evaluate the development process more precisely. The older, Locusts based version of AugIm was written using Python and PyGtk, whereas the newer, Scavenger based version is built using PyQt and Python—a change in platform that was in part chosen to avoid the temptation of copy-pasting large parts of the code base. When implementing the Scavenger based version I chose to initially implement it without the use of cyber foraging. After implementing this fully-fledged mobile image manager without the use of cyber foraging, the use of cyber foraging through Scavenger was added.

The Application

AugIm is an image manager that operates in two modes: browse mode where the user is able to browse the images she has taken with the built-in camera (depicted in Figure 7.9), and edit mode where the user may perform a number of adjustments to a chosen image (depicted in Figure 7.10). The entire application interface has been custom made to operate on a touch screen display, meaning that very few standard widgets have been used—this of course added a bit to the development time. On the other hand, something that detracted a lot from the development time was the fact that a similar application had been implemented before, and this experience of course made the development process much easier.

Using AugIm the user initially browses her images in the browse mode, as shown in Figure 7.9. One thing to notice while browsing is, that the images being browsed are much larger than the display, and they are therefore scaled as needed when being displayed. These scaled down images are stored on disk, so that the scaling need only be done once. When the user finds an image that she would like to edit, she presses an on-screen button and enters the editor, as shown in Figure 7.10. Within the editor the user may choose between
eight different image operations: sharpen, blur, brightness, colour, contrast, horizontal and vertical flip, and scale. These are but a small selection of the operations that could be added to such software—more elaborate examples could be red-eye reduction or facial recognition. When the user chooses an operation it is applied only to the small, preview version of the image and the result is displayed right away. Once the user is satisfied and leaves the editor, she is prompted about whether to save or discard the changes done to the image. If she chooses to save the changes the chosen operations are applied to the original image in the background while the user is free to browse other images.

The entire application, without the use of cyber foraging, was built in approximately a week and a half—give or take a day. Had I not had the experience from implementing the prior version of AugIm I would estimate the development time to be at least twice that; e.g., three to four full workweeks.

Adding Cyber Foraging Support

Given a finished application that did not utilise cyber foraging, the real test of our development framework could begin. From having worked with the application and having run it without cyber foraging on the mobile device, I had a good idea of where cyber foraging would be needed:

Applying image operations to original images One obvious place where cyber foraging would be beneficial is in performing the image operations on the original, five megapixel images. These operations are far too heavy for the mobile device to handle on its own.
Applying image operations to preview images  One thing that became apparent by using the application was, that for some operations even applying them to the small 800 × 480 preview images took multiple seconds on the mobile device. Perhaps something could be gained by utilising cyber foraging here as well?

The initial scaling of images  Another thing that became apparent when using the finished application, was that the initial scaling of images into thumbnails was heavy enough to be an annoyance when trying to quickly browse through new images.

All targets for cyber foraging listed above pertain to the image operations being performed. These eight operations were implemented as eight separate functions—which would be normal practise when the functionality is needed in more than one place. These functions needed to have the scavenge decorator applied to them. Adding the decorator made the functions into Scavenger tasks, which immediately activated the use of cyber foraging and yielded big performance benefits.

There was one small problem though, that made the addition of cyber foraging slightly more complex. AugIm uses the Python Imaging Library for the image operations, and sadly the Image objects that this library works on are not picklable—which is Python jargon for serialisable, i.e., that the data type is capable of being packaged for storage or transmission over the network. This was no problem in the non-cyber foraging enabled version, since all processing was done within the same process. Because of this small nuisance, we had to rewrite the tasks a bit, so that they would operate on raw image data instead of Image objects. Another nuisance was the fact that Image objects can only be
created from files, and not from in-memory data as is needed in AugIm. An example task rewritten to operate on raw image data is shown in Figure 7.11.

```python
@scavenge('len(#0)', 'len(#0)', True)
def sharpen(image_data, factor = 1.0):
    from PIL import ImageEnhance, Image
    from StringIO import StringIO
    sio = StringIO(image_data)
    pil_image = Image.open(sio)
    new_image = ImageEnhance.Sharpness(pil_image).enhance(1.0 + factor)
    sio = StringIO()
    new_image.save(sio, 'JPEG', quality=95)
    return sio.getvalue()
```

Figure 7.11 The sharpen task as it is implemented in AugIm.

Now that the functions operate on raw image data, an Image object must be created within the function instead. Because of the limitation with regards to creating new instances of the Image class, the StringIO class is used to wrap the in-memory data and present it like a file to the Image class. Creating the Image instance is done in lines 4–6. Once the sharpen operation has finished the function must return the raw image data of the resulting image, and again the only way of getting this data out of an Image object is by writing it to a file. To avoid using expensive disk I/O this is also done using the StringIO wrapper, as is shown in lines 8–10. Six out of ten lines of code are thus there only because of this deficiency of the Image class, i.e., the lack of serialisation support. Given a better imaging library, operating on a serialisable data type, the task could have been written as shown in Figure 7.12 where the only difference between the regular and the Scavenger-enabled versions of the function is the decorator and the added import statement in line 3.

```python
@scavenge('len(#0)', 'len(#0)', True)
def sharpen(image, factor = 1.0):
    from PIL import ImageEnhance
    return = ImageEnhance.Sharpness(image).enhance(1.0 + factor)
```

Figure 7.12 The sharpen task as it should have been—given a serialisable image data type.

Having to change data type meant that not only the eight functions doing image operations had to be altered, the callers of these functions also had to call them in a different way, using raw image data instead of Image objects.

The final change that needed be done, to finish integrating Scavenger into AugIm, was adding a step that would fetch the result of the last image operation in a chain of operations, so that the image could be stored or displayed locally. Adding all of this was done in about two hours, and given a more cooperative imaging library I believe this could be brought down to less than one hour.
Conclusion

Concluding on the completeness of the high level API is not easy to do given that only a single complete application has been built using it. I have throughout the past year developed numerous small demonstrators using the API, but only once have I taken the time to built an entire application using it. What can be concluded is this: the high level API was complete enough to implement AugIm; an application making use of many facets of Scavenger as a cyber foraging system.

As for the complexity of the API, adding cyber foraging support to an application such as AugIm was easily done—even in the face of problems introduced by the use of the Python Imaging Library. The development time of the application itself was done in approximately eight days, and the addition of cyber foraging was done in a mere two hours. These time measurements should be seen in the light that it was I, the designer and developer of the cyber foraging framework, that implemented it, but even so the results are quite impressive I think. In a more real-to-life situation it would be the developer of an application that added cyber foraging support to his application. This developer would have intimate insight into the application itself, meaning that he or she would know exactly where to look if any complications should arise, as was the case with the unpicklable Image objects in AugIm. Given this expert knowledge of the application the addition of cyber foraging through Scavenger should be easy, and in most cases it should entail no more than 1) importing the scavenger module, 2) moving module imports into the chosen functions, 3) adding the scavenge decorator, and 4) optionally considering how remote data handles could be employed.

Before adding cyber foraging support the AugIm code base contained 1759 lines of source code. Adding cyber foraging in the simplest possible way, by merely adding the decorator and moving module imports inside the eight image manipulation operations, would add only approximately 25 lines of code—adding a mere 1.4% of extra code. But AugIm does not use Scavenger in the simplest fashion—in order to fully utilise Scavenger it works with remote data handles, which adds a little bit of extra complexity. Adding that complexity means that 49 lines of code were added and 13 changed; meaning that 62 lines of code was affected in total which means that 3.4% of the code base pertains to the use of cyber foraging. There was one problem with AugIm though, and that was the unpicklable Image objects. Because of this problem 58 extra lines of code had to be introduced, yielding a grand total of 1876 lines of code whereof 120 was touched because of the use of Scavenger. So in the end 6.4% of the code base has been added/ altered with the introduction of cyber foraging. This number will of course vary greatly depending on the application—I would expect it to normally be much smaller, especially if working with picklable data types.

Comparing our results with regards to developer support to the only other cyber foraging frameworks that have developer support as a stated goal (Spec-
tra [22] and Chroma [6]), I think it is safe to say that we have made a great improvement to the overall usability of the system. In those two systems the developer adding cyber foraging support had to move the task code out of the application, use a stub generator to create code stubs for the RPCs, retrofit the task code into one of these stubs, and manually install that RPC onto a surrogate. Within the application the call to the original function would then be replaced by a calls to numerous functions that would register the task upon application start-up, signal to the system that the task would be called soon, send individual arguments, call the task, cleanup after performing the task—all of these are separate function calls that the developer must insert in the correct place and order. Using Scavenger it is in most cases enough to add a decorator and move some import statements. The task code is kept in place within the application, which gives the developer a cleaner application structure to cope with—from the developers point of view the functionality is in fact placed where it is used. Furthermore, the fact that there is no need to manually install the task onto surrogates means that not only development but also deployment is made a lot easier in Scavenger.
Chapter 8

Comparison to Related Work

Having described both my research and the evaluation thereof, I in this chapter return to the related work in order to clarify how my work relates to existing cyber foraging systems. The focus in my work has been on supporting high mobility and providing a development model allowing for easy access to cyber foraging capabilities. In Section 8.1, I present a comparison on the mobility of Scavenger and the related systems, and in Section 8.2 the development support offered by Scavenger is related to that of the related systems.

8.1 Mobility

What makes my work stand apart from the related systems is the focus on mobility both with regards to physical mobility and operation in unknown environments. In Chapter 3 four related cyber foraging systems were described and their proficiencies with regards to different metrics important to mobility were assessed. This was summed up in Table 3.1 which for comparison is replicated here in Table 8.1 with the addition of Scavenger in the last column. In the following I will go through the metrics mentioned in this table one by one describing how Scavenger measures up the other systems.

Overheads

*Initialisation overhead* is the time elapsed from the client’s initial wish to perform a task at a surrogate till the execution commences. In Spectra [22] and Chroma [5], the two systems that use pre-installed, always-running RPCs, this initialisation overhead is small since it consist only of scheduling, i.e., selecting which surrogate to perform the task at. After having selected a surrogate everything else is ready and running. In the VM-based systems system the initial overhead of task execution is enormous; in Goyal and Carters system [25]
Table 8.1 Comparison of selected metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Spectra</th>
<th>Chroma</th>
<th>Goyal/Carter</th>
<th>Slingshot</th>
<th>Scavenger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overheads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialisation</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Huge</td>
<td>Low</td>
</tr>
<tr>
<td>Execution</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Mobility</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discovery</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Unprepared usage</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Migration support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent sched</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Load-balancing</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Parallelism</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application CF API</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task API</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Services are pre-installed onto surrogates.
- Centralised through registrar. How the registrar is discovered is not described.
- Since load on surrogates is considered during scheduling a kind of load-balancing takes place, because unloaded surrogates will be preferred.
- An API call is used to forward tasks to the client proxy, but it seems that it only forwards the request, i.e., it does not impose any structure on the process and hence has to be tailored specifically for each application.
- In Scavenger tasks are defined within the application itself.

a new virtual machine image must be stared and the needed functionality installed, and in Slingshot [77] a second-class replica of a virtual machine must be transferred over the Internet. When working with known surrogates, i.e., surrogates that have previously been used to perform the given task, the initialisation overhead of Scavenger is same as in Spectra and Chroma; but when working with unknown surrogates the overhead is slightly larger because Scavenger needs to install the task code onto the surrogate. This installation of task code is quickly done though; the code is typically only a few kilobytes in size and the installation process consists of only a few steps: 1) the code is transferred to the surrogate, 2) the surrogate validates the code, which is done by scanning the code for import statements and use of illegal keywords, and 3) if accepted the code is written to disk on the surrogate. For all but very small tasks the overhead of this installation is negligible compared to the overall task running time. Scavenger thus delivers performance comparable to that of Spectra and Chroma in the static network setting they are designed for, while also supporting the added flexibility of the VM-based systems in unprepared environments by adding only a small installation overhead—an overhead that is orders of magnitude smaller than that of the VM-based systems.

Remote execution overhead: Once the cost of initialisation has been paid and a surrogate has been chosen the execution may commence. The execution overhead is the overhead that lies within each call to the surrogate. Transfer of input (and possibly output) data is obviously a part of the execution overhead, a fact
that is factored into the scheduling done in both Spectra, Chroma, and Scaven-
ger. Spectra and Chroma uses a Coda [74] file system shared by clients and surrogates alike for input data, and when considering a surrogate for execu-
tion of a task the Coda cache state of the surrogate is taken into consideration. Scavenger clients and surrogate have no shared file system, but the surrogates do have a cache that can store results of previously performed tasks or for stag-
ing input data. When scheduling in Scavenger this locality of the data is con-
sidered—reminiscent of how it is done in Spectra and Chroma. When it comes
to output data Scavenger is smarter than Spectra and Chroma: when schedul-
ing Scavenger knows whether or not the output data is immediately needed by the client, and it is thus able to employ this information when choosing a surrogate.

Apart from the overheads induced by the transfer of in- and output data none of the systems have any significant execution overheads. Spectra, Chroma,
and Scavenger gain an advantage over the VM-based approaches by support-
ing data staging which in some cases may save them the cost of in- and output transfer.

Mobility

*Discovery:* Does the systems provide the means to efficiently discover new sur-
rogates? In Spectra and Chroma there are no means of discovery; they rely on static definitions of surrogates written down in a file on the client. The two VM-
based approaches do have peer discovery; Goyal and Carter’s system using a central registrar and Slingshot using UPnP. In Scavenger Presence [39] is used.
Using Presence Scavenger can guarantee that new surrogates are discovered within a second, and that the context information available when scheduling is always up-to-date to within a second.

*Usage in unprepared environments:* Spectra and Chroma are unusable in un-
known environments because of their reliance on pre-installed tasks. The VM-
based systems have been designed to enable such use, but their large initial-
isation overheads render them useless in a mobile setting. Scavenger, on the other hand, has been tailor made to function in such environments, and it does so by adding only a slight initialisation overhead when installing a mobile code task. Scavenger is thus the only cyber foraging system supporting mobile use in unprepared environments.

*Migration support:* The only system that support true migration is Slingshot. In Slingshot all actions are performed at all replicas, meaning that the first-class replica should always be up-to-date and migration of a running “task” is thus what happens whenever a new second-class replica is instantiated. In Scaven-
ger there is no support for true task migration—it is impossible to move a running task from one surrogate to another. This could be implemented by using tasklet pickling which is supported by Stackless Python, but some practical re-
strictions, e.g., when using native resources and C-based libraries, meant that we have opted against it.
That true task migration is not supported in Scavenger does not mean that the concept of task migration is not supported altogether; it depends on what one looks on as a task. In Scavenger a task is a small, atomic piece of functionality that may be run on an input and yield an output, but in many cases many of these small tasks are composed together in order to form a larger task. This is reminiscent of e.g., Chroma’s tactics that are composed of a number of services. Taking this view on a task as consisting of many smaller interconnected subtasks, as was done in the Locusts framework, task migration becomes manageable even without the ability to migrate running subtasks. How this works within Scavenger has been described in Section 5.3.2.

Scheduling

Intelligent scheduling: Of the related systems only Spectra and Chroma supports scheduling of tasks. Both of these use task centric profiling; Spectra using the bogomips\(^1\) rating of the CPU as a strength rating, and Chroma using the CPU clock speed. Both of these measures are poor measures for CPU performance when comparing across different CPU models and architectures. In Scavenger the measure for CPU strength is a real CPU benchmark—a number calculated by a benchmarking suite that by design is meant to be comparable across different CPUs.

Apart from the task centric profile, that enables Scavenger to apply its profiled information in unknown environments, a more precise peer centric profile is also maintained. This dual profiling, where both task and peer centric profiles are maintained, is unique to Scavenger. Using it leads to more precise scheduling when working with known surrogates, since the peer centric profile may be employed, and in cases where no such information is available the task centric profile may be used.

Spectra, Chroma, and Scavenger are all capable of adapting to variations in complexity brought on by changes in e.g., input size or value. In Spectra and Chroma this is done using a complex resource predictor \([54, 53]\) using machine learning techniques to derive the complexity of a task by consulting logs of previous executions. In Scavenger this has been simplified greatly by introducing the adaptive profiles that dynamically create a two-dimensional profile when fluctuations in task running time exceeds a given threshold—a process that is described in detail in Section 5.1.3.

Load-balancing: When considering a cyber foraging system load-balancing may mean two things: 1) balancing the load of currently executing tasks on a surrogate machine, and 2) balancing the load of all tasks over all available surrogates. The first kind of load-balancing is supported by the VM-based approaches since that functionality comes as standard in most virtual machine managers. The RPC-based systems, Spectra and Chroma, use operating system processes for their tasks and thus leave this kind of load-balancing to the

\(^1\)A measurement of CPU speed made by the Linux kernel when it boots.
8.2 Development support

Apart from Scavenger the only other related systems that have focused on development support are Spectra and Chroma. Both of these systems offer APIs for both task and application development.

The main difference between Scavenger and those systems is that in Scavenger’s execution environment ensures that tasks get a fair share of the processor by using Stackless’s support for preemptive scheduling. This scheduling is done intra-process and only ensures that tasks share their resources equally; as for system wide load-balancing Scavenger relies on the operating system in the same way as Spectra and Chroma.

Considering the distributed load-balancing, the VM-based approaches have no such support. Spectra, Chroma, and Scavenger on the other hand are capable of distributed load-balancing simply by the fact that surrogate utilisation is considered when scheduling a task.

**Parallelism:** Does the framework support use of multiple surrogates in parallel, and if so how does it use this parallelism? Scavenger supports parallel execution of tasks by 1) being completely thread safe, allowing the developer to perform multiple tasks at once, and 2) by updating the local context information when scheduling tasks to immediately reflect the change in available resources at a surrogate. These local updates, which are of course overwritten when new information arrives from the surrogate, mean that multiple tasks submitted simultaneously will be load-balanced over the available surrogates. The results of this is clearly seen in Section 7.1.5.

Some of the related systems, Chroma and Slingshot, use parallel processing as well. In Slingshot a client working with multiple surrogates will submit all tasks to all surrogates, and the first answer to be returned may then be used. If operating in an untrusted environment the value of subsequently returned results, especially those coming from a trusted surrogate such as the first-class replica, may be used to check whether the first result was correct. Chroma also works with parallel execution of one task using three models: *fastest result* is reminiscent of Slingshot’s approach, *data decomposition* asks the user to provide functionality to split up input data and combine output data and then creates multiple smaller tasks that may be performed at different surrogates, and *best fidelity* asks different surrogates to perform the task at different fidelities and then returns the highest fidelity result to be returned within a given time limit.

Scavenger does not offer these kinds of parallel execution—but a technique such as *fastest result* would be simple to add. Data decomposition was a part of the Locusts cyber foraging framework, but tests showed that in many use cases the cost of splitting up input and combining output data was too large, especially given the relatively small expected running time of the tasks scheduled in a mobile cyber foraging setting.
venger the distinction between application code and task code is removed. In Scavenger tasks are ordinary Python functions residing within the application code, that have been annotated with the scavenge decorator. When local execution is performed that exact function is called, and when doing remote execution Scavenger automatically creates a mobile code task out of the annotated function and handles scheduling and execution. That task code is kept in place within the application gives the developer a cleaner application structure to cope with—from the developer’s point of view the functionality is in fact placed where it is used. In Spectra and Chroma tasks have to be defined as stand-alone applications, adhering to a special task interface, that could be pre-installed and started on surrogate machines. This division between application and task code forces the developer to redesign his application when adding cyber foraging support, whereas Scavenger only asks the developer to annotate some functions and adhere to a few simple rules about module import and use of meta programming.

Another benefit of Scavenger’s approach of using in-line tasks is that when task code exists within the application, the application does not depend on external functionality—which is especially important when local execution is chosen. In Spectra and Chroma clients must be running the application, the cyber foraging client, the cyber foraging server, and all of the pre-installed tasks in order to perform tasks locally. On small mobile devices this could incur a serious resource overhead. In Scavenger the clients are not running the daemon software; all cyber foraging related activity takes place in the Scavenger API which is a single Python module that must be imported, and local task execution is done as described above.
This thesis has presented the research I have done during my PhD studies, which I have conducted within the field of pervasive computing—or more precisely within the area of cyber foraging. My research has had three intertwined goals: 1) to understand, describe, and present designs for highly mobile cyber foraging, 2) to design, develop, and experiment with scheduling algorithms for use in such a highly mobile setting, and 3) to create a development model encompassing all the requirements of highly mobile cyber foraging while still being easy for developers to use. These three research objectives have been described in detail in this thesis in Chapters 4, 5, and 6 respectively where the contributions of my work have been clearly stated.

I have taken an experimental approach to research, meaning that I have preferred designing, implementing, and evaluating real systems instead of focusing on mere theoretical discussions. E.g., when I have talked about scheduling in this thesis this has not merely been a discussion of different scheduling algorithms, but rather a complete discussion of the hows and whys of designing and implementing such scheduling in a highly mobile cyber foraging setting. I have presented and discussed a complete solution to the kind of dynamic scheduling needed to support highly mobile cyber foraging, and provided two distinct implementations of such systems freely available for the reader to experiment with.

9.1 Addressing the Research Theses

The working theses of the Locusts project, which were stated in Chapter 2, were:

**System support** It is possible to build a cyber foraging system supporting efficient use of remote resources in a mobile setting where 1) the user is physically mobile while execution of tasks takes place, and 2) this phys-
ical mobility takes the user into unknown territory necessitating a mobility of the tasks themselves.

**Performance gains** Even with having to cope with such a high level of mobility, such a system will be usable on small, mobile devices and will yield significant performance improvements enabling the mobile devices to take on tasks otherwise deemed unsuitable on such platforms.

**Development support** Such functionality can be presented to developers in a form such that even novices with little or no experience with distributed computing can build efficient, cyber foraging enabled applications.

**Energy efficiency** By using such a cyber foraging framework mobile devices will be able to save energy.

Through the design of both the Locusts framework and the Scavenger framework, we have shown by example that the goal of highly mobile cyber foraging is indeed reachable. This is achieved by utilising efficient, pro-active discovery, light weight tasks, and a flexible mobile code based approach towards task installation and execution. In related work the systems would either be good at physical mobility (Chroma and Spectra) or at usage in unprepared environments (Goyal and Carter’s system and Slingshot); Scavenger is the first to successfully combine these two types of mobile cyber foraging.

That the performance of a mobile device with regards to processing power may be improved by utilising cyber foraging is nothing new; this has been shown by all related systems. What Scavenger does that is special in this regard is to make this functionality available on small, mobile devices such as smart phones. By keeping task code within the mobile application, the need for running a cyber foraging daemon/server on the mobile device is removed—all the client application need do is to import the Scavenger module. This module is light weight, spawning only a single thread that is used for discovery and environment monitoring, and any device capable of running Python would be capable of using this module. The only other resource intensive operation within the Scavenger library is the scheduling. That this scheduling incurs only an acceptable overhead on small mobile devices was shown in Section 7.1.3.

With Scavenger we have simplified the development of cyber foraging enabled applications to a degree where no experience with distributed computing is needed. When using Scavenger in the simplest fashion all that is needed to apply cyber foraging to a function is decorating it with a single keyword. In order to utilise the full strength of Scavenger a little more is needed; e.g., the use of remote data handles where applicable, but this is not strictly necessary, and great performance improvements can be had by just decorating a handful of functions.

The final thesis, that energy could be preserved by utilising cyber foraging, has been shown by the experiments presented in Section 7.1.7. In these tests it is shown that energy may be preserved on a modern mobile device by offloading CPU intensive tasks; even when compared to turning the Wi-Fi interface off
and doing the computation locally. These tests have recently been replicated by Janne Parkkila on a Nokia N900 smart phone, and preliminary results show the same kind of behaviour on that platform.

9.2 Realising the Use-Cases

In the introduction three use-cases employing cyber foraging were presented. The first of these, the image manipulation use-case, was realised through AugIm, the image manager application described in Section 7.2.2. Using AugIm the user is capable of performing image operations that would normally be out of reach for a mobile device.

The second use-case is that of continuous voice recognition. In this use-case a doctor is able to enter information into an electronic patient journal by talking into a small headset. Given the speech recognition software, accessible in Python, implementing this use-case would be well within the reach of Scavenger. In the use-case the application queues up utterances when no surrogates are available, and submits these as recognition tasks when surrogates are found. This kind of queueing is not built into the Scavenger decorator, and the developer would thus have to use the low level API to check for surrogate availability.

The final use-case is concerned with augmented reality and talks of a wearable computing device capable of doing scene interpretation, facial recognition, and voice synthesis in order to guide an Alzheimer’s patient by providing information about his environment and the people therein. At the bottom of this application lies an endless stream of images that must be interpreted, and texts that must be converted to voice. Again this can be implemented using Scavenger.

Using Scavenger it is thus possible to implement all the visions set forth in the introduction—but there is one important thing missing before this will be a reality: A security model enabling clients and surrogates to trust each other. Some of this work has been begun by Kristian Andersen [1], who prototyped a surrogate authentication mechanism within Scavenger based on asymmetric cryptology and certification, but more work is needed before Scavenger’s approach towards cyber foraging becomes feasible in real deployment.

Adding that final component, the security model, would enable users to use their smart phones as strong computing devices capable of solving any task at hand. And, through Scavenger’s focus on mobility, this functionality would be available to the user regardless of location. One could imagine a future where ordinary users would not need to own a stationary computer or even a laptop, but where a small tablet device such as an iPad would be sufficient for all use. When in need of processing power—or just in order to save energy—Scavenger could be employed, offloading resource intensive work to nearby surrogates.
These surrogates could for example be a PlayStation 3\textsuperscript{1} or other gaming console found in the user’s home, and when on the move surrogates could be offered by service providers in the same way that Wi-Fi access is today. One could easily imagine Wi-Fi routers sporting a larger CPU in order to be usable as a surrogate for the mobile devices connected to its network. We are not quite there yet, but in the near future this could become a reality, and Scavenger has shown by example how it could be done.

Summary

In my research I have contributed to the field of cyber foraging in a number of ways. Together with Niels Olof Bouvin I have defined the term highly mobile cyber foraging, and examined what would be needed to support it. I have, in part together with Jari Porras, designed, implemented, and evaluated a novel adaptive, dual-profiling scheduler for use in such a setting. I have developed an entire cyber foraging system that has been released as open source, so that other researchers and developers may use it to experiment with cyber foraging, or to benchmark their own systems against it. Finally I have created a development framework that eases the implementation of cyber foraging enabled applications, while still being powerful enough to support efficient cyber foraging.

This thesis has presented the fruits of my three-year foray into the world of academia. The years have been spent diving into the subject matter—reading up on related systems and technologies—and, because I have taken an experimental approach towards research, on designing, developing, and experimenting with highly mobile cyber foraging systems. These activities have lead to the publication of one technical report [39], three peer reviewed workshop papers [38, 40, 44], two peer reviewed conference papers [43, 41], one published journal paper [14], one journal paper that has just been conditionally accepted, and a book chapter [61].

During my studies I have also spent some time on another university, working together with professor Jari Porras and his staff at Lappeenranta University of Technology (LUT) in Lappeenranta, Finland. This lead to the book chapter being written by Jari and I and a third author named Oriana Riva, and to an as of yet unpublished article on dynamic, profile based scheduling that will be submitted to a conference or journal in the near future.

Scavenger as a platform for mobile remote execution and cyber foraging has been used by a number of students during the past couple of years. We have had master’s students doing projects using Scavenger, as well as some master’s theses based on work done using Scavenger as a platform for experimenting with dynamic cluster scheduling. Even though the Locusts project has already ended, and my PhD scholarship is nearing its end, the work on

\footnote{\textsuperscript{1}The Scavenger daemon has been tested and found to run nicely on the PS3 platform.}
Scavenger continues. We currently have one master’s student working on his master’s thesis using Scavenger, one PhD student using Scavenger for his studies of parallel programming models, and a Finnish developer, employed by Jari Porras at LUT, working on developing applications using Scavenger and on experimenting with dynamic scheduling in a mobile environment.
Chapter 10

Future Work

Cyber foraging as a computing technique contains many aspects that could be venues for future work. I have in my work concentrated mainly on application partitioning and scheduling, but because of my experimental approach I needed to implement a complete system, which means that I have been working with all aspects of cyber foraging. Some of my ideas for future work thus involve aspects of cyber foraging that have been neglected because of my restricted area of focus.

Discovery and Monitoring

In order to have an efficient means of peer discovery and monitoring, the Presence service discovery framework was developed. For a number of reasons, all described in detail in [39], the existing service discovery solutions were found lacking, and a new protocol and implementation was therefore created.

Presence uses pro-active announcements, meaning that a Presence peer offering some service will announce this periodically by broadcasting a single UDP packet on its local subnet. In cases with no or very little activity on the network, i.e., when no client are in need of cyber foraging, these packets are “wasted” because they deliver information that nobody is currently interested in. But being pro-active also means, that when a client is in need of cyber foraging, it already knows everything it needs to know and may therefore begin using remote resources immediately.

An alternative approach would be to use re-active announcements, where a peer only sends out announcements when another peer asks for that information. These announcements may still be broadcast, so that all peers may update their information about the sending peer. Using re-active announcements would introduce a delay to each invocation of a remote executable task, because the scheduler would have to ask all surrogates on the network to send
new announcements, receive those announcements, and then consider the in-
formation therein when scheduling. This approach will not scale to situations
where a lot of activity is going on, but on the other hand resources will be saved
when no cyber foraging is going on. Another problem with this approach is the
introduced delay.

Even though there are some obvious problems with using the re-active ap-
proach, it also has some benefits with regards to operation in low activity en-
vvironments. It would therefore be interesting to investigate exactly how the
re-active approach would perform under different work loads, and perhaps to
see if a hybrid approach could be designed, where surrogates would switch
between pro- and re-active announcements as needed. In order to test these
things a simulation would be needed.

Task Centric Profiling

One of the main contributions of the Scavenger framework is its adaptive, dual-
profiling scheduler that contains both peer and task centric profiling. The task
centric profiler builds a history based profile of a task’s running times that is
weighed by the “strength” of the peers performing it. The strength of a peer
in this respect being the peer’s rating using the NBench benchmarking suite.
The validity of this approach relies on the assumption that “if task A takes one
second on a surrogate of strength ten, it will take a surrogate of strength one
ten seconds to perform that same task”. Because of the highly heterogeneous
nature of the surrogates this assumption does not always hold, but as has been
shown in Chapter 5 it is a good approximation.

In the work done validating the task centric profiling we have been working
mainly with image manipulation tasks, because this was the metier of our main
demonstrator AugIm. When working with these diverse image manipulation
tasks the assumption about strength relations held to an acceptable level, but it
would be interesting to perform a more exhaustive test of this assumption by
measuring the “task weight” estimate of a large number of different tasks (and
inputs) on a large number of machine types. We have done some of this work
already, but a little more coding and testing is needed before publishing these
results. The draft [Paper V] that I am currently working on together with Jari
Porras has been written to demonstrate the strength of the task centric sched-
uler, and in working with that we are currently investigating which weights
are assigned to a number of tasks (encryption, decryption, compression, de-
compression, fractal rendering, and ten image manipulation tasks) by twelve
different machines of varying CPU size and architecture. Some of these meas-
urements seem to point at, that the task centric scheduler could be made even
more precise by working with two task weights; one scaled by the NBench in-
teger rating and one by the floating point rating. By collecting both weights
and then e.g., preferring to use the set that has the smallest standard deviation,
the scheduler would be able to make a distinction between integer or floating
point heavy tasks, which would make the running time estimates a little more precise. Investigating how such a scheduler could be designed and evaluating the effect of these changes is an interesting venue for future work. In related systems bogomips and CPU clock rate are used as a CPU strength measure. It would be interesting to set up experiments measuring the accuracy of these measures compared to using a benchmarked score.

Developer Tests

One of our foci throughout the Locusts project was developer support. We wanted to build a cyber foraging solution that even novice programmers could use, and as described in Chapter 6 we think that we have achieved this goal. What is still missing though, is doing rigorous user tests where our framework is handed over to programmers—much in the same way as Balan et al. [6] did when testing their development model. This kind of large scale user test has not been done yet for a number of reasons—but mainly because of time and resource constraints.

We have had a number of students working with Scavenger in their master’s thesis and smaller projects. All of those students have been impressed with the ease of use of the system, but they have been working with the lower level API, and we thus do not have a proper evaluation of working with the high level development API that should enable novice developers to build cyber foraging enabled applications. Performing such an evaluation would therefore be an obvious next step.

Security

One aspect that has been all but completely neglected in my research is that of security. As the Scavenger prototype is right now nobody would ever want to use it in real life because of the lack of security measures. The only security measures that have been taken within Scavenger are the following:

- Task code is checked for maliciousness by black-listing a number of keywords that give access to e.g., meta programming in Python. Given that task code does not import untrusted modules this should keep the surrogate safe from task intruding on its private data.

- Tasks are killed after a fixed amount of time in order to avoid denial of service (DOS) attacks. This does not mean that DOS attacks can not be launched and have an effect, it merely means that when the malicious peer orchestrating the attack stops, the surrogate is able to recuperate. It also means that badly written code containing endless loops are weeded out eventually.
• Module imports are white-listed, meaning that only imports of approved Python modules are all allowed when installing new task code on a surrogate.

These security measures are all centred around protecting the surrogate from malicious clients. What is completely missing is protection of the client from a malicious surrogate. The client must e.g., be able to trust that the surrogate returns truthful results, and that it does not misuse any data it is entrusted. In that respect it would be interesting to investigate the use of secure multiparty computation in cyber foraging.

As an effect of the module white-listing it is impossible for tasks to work with modules that have not been white-listed by the surrogate. In a real world deployment one would have to design a secure but flexible solution to the problem of dynamic module installation.

There are many more aspects to securing a system such as Scavenger than I have hinted at here, and this is of course a very interesting venue for future work that could be a PhD study in itself. It is of crucial importance that these problems be solved if cyber foraging is to be usable in untrusted environments.
Appendix A

Installing and Using Presence

Presence is the peer discovery framework I developed to be able to prototype a cyber foraging system. What makes Presence differ from existing discovery systems is described in detail in [39]. In this appendix I will describe how to download, build, and install Presence onto a surrogate—clients need no such installation since there is a consumer-only version of Presence available, that is written entirely in Python and as such may be distributed along with the Scavenger library.

A.1 Getting the Source Code

Presence has been released as open source software and the most recent version can always be fetched at its Google Code homepage: http://code.google.com/p/presence-discovery/. At the front page there are source snapshots available, but the easiest way to get at the most recent version is by simply checking out the Subversion repository. This is done with the following command:

svn checkout http://presence-discovery.googlecode.com/svn/trunk/ presence

A.2 Building the Binary

Presence in its current implementation has been built using Nokia’s Qt C++ libraries. In order to build Presence these Qt libraries and some related development tools must be installed. Depending on the platform you are building on the installation method may vary—have a look at http://qt.nokia.com for further information about installing Qt development tools on your build system. The only two Qt libraries that are strictly necessary to build Presence is QtCore and QtNetwork, and the only development tool that is strictly required is qmake.
Having installed the Qt development tools the process of building the Presence binary is as such—assuming that \texttt{DIR} is the directory that the Presence source code resides in:

\begin{verbatim}
cd \texttt{DIR/daemon}
qmake
make
\end{verbatim}

Note that, in order to build it using GNU Make in Mac OS X you need to pass in the argument \texttt{-spec macx-g++} to qmake. If you do not use this argument an XCode project is built instead, and you need to build that using XCode.

A.3 Running Presence

Now that the binary has been built for the desired platform, the next step is to actually start the Presence daemon. Presence runs as as a daemon; i.e., it is supposed to run in the background at all time, so adding it to an init script is a possibility. Presence accepts the following command line arguments:

\begin{itemize}
  \item \texttt{-n name} Sets the announced name of the Presence peer. The default is a randomly generated name such as \texttt{presencenode-479899323}.
  \item \texttt{-p port} Sets the announcement port. This port must be the same on all Presence peers that are supposed to receive each others announcements. The default is 12345.
  \item \texttt{-c port} Sets the client port. This is the internal port number that Presence clients, i.e., applications using the Presence library, connects to. Default is 2000.
  \item \texttt{-i interval} The interval between pings in seconds. The default is 1, meaning that peers announce their services once every second.
\end{itemize}

A.4 Installing the Presence Library

In order for applications to be able to communicate with the local Presence daemon, the Presence Python library must be installed on the peer acting as a surrogate as well. The easiest way to install this is by using one of the distributable packages on the Presence Google Code page. But, for those who want to build their own from the source repository, the process needed to build such a distributable is described below:

\begin{verbatim}
cd \texttt{DIR/build/python-lib}
./make_dist.sh x
\end{verbatim}
This creates an archive called *py-presence-x.tar.gz*. Using that archive (or one downloaded from the homepage) the library can be installed in the following way:

```
tar xfvz py-presence-x.tar.gz
cd py-presence-x
python setup.py build
python setup.py install
```

Notice that the final step must be performed by a user that has sufficient access rights to the `site-packages` directory of the chosen Python interpreter.
Appendix B

Installing and Configuring the Scavenger Daemon

The Scavenger daemon must be installed on all peers that wish to make their resources available to others on the local network. It depends on Presence—which must be installed as described in Appendix A.

B.1 Getting the Source Code

Just as Presence, Scavenger has also been released as open source software and is hosted as a Google Code project on this page http://code.google.com/p/scavenger-cf/. There are some pre-built Scavenger packages on the homepage that can simply be downloaded, unpacked, and started, but in this Appendix I will describe how to manually install and configure a Scavenger daemon. The first thing to do is fetch the source code:

```
svn checkout http://scavenger-cf.googlecode.com/svn/trunk/ scavenger
```

Apart from the Scavenger source the source code for the NBench benchmarking suite should also be fetched—this is needed in order to benchmark the surrogate to obtain a strength rating. NBench can be fetched here: http://www.tux.org/~mayer/linux/bmark.html.

B.2 Building and Running NBench

To get the best results when doing task centric scheduling, the surrogate should be benchmarked using NBench. This step is optional—if no NBench rating is given the surrogate will benchmark itself using a simple “bogomips”-like measure upon initial start-up.
Building NBench is done as shown below:

```
tar xfvz nbench-byte-2.2.3.tar.gz
cd nbench-byte-2.2.3
make
```
If you are building under Mac OS X you need to add an additional step; before running `make` you must edit the `Makefile` uncommenting lines 77 and 78:

```
# for a Mac with OsX and the Darwin environment
CC = cc
CFLAGS = -O3 -DOSX
```
Once the `nbench` executable has been built you can invoke it directly from the command line, which will perform the benchmark in a couple of minutes. While the benchmark is running some status information is printed, but the only information that is important to Scavenger is the final “Original Bytemark Results” score:

```
==========================ORIGINAL BYTEMARK RESULTS======================
INTEGER INDEX : 85.225
FLOATING-POINT INDEX: 67.530
Baseline (MSDOS*) : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
```
Given this information the Scavenger surrogate strength rating is the average of the integer and the floating-point rating; e.g., this machine would be rated \((85.225 + 67.530)/2 \approx 76\). This rating must be written into the Scavenger daemon’s configuration file, which will be described shortly.

### B.3 Installing Stackless Python

Stackless Python is a prerequisite of the Scavenger daemon so you need to fetch and install it from this homepage [http://stackless.com](http://stackless.com). There are binaries available for Mac and Windows, but in this section I will describe how to build it from source on a Linux or Mac machine.

The first thing to do is to fetch the source code for the latest 2.x-branch of Stackless Python from the homepage. Assuming that the fetched archive is called `stackless-2xy-export.tar.bz2` the install procedure is as follows:

```
tar xfvj stackless-2xy-export.tar.bz2
cd python-2.x.y-stackless
mkdir $HOME/opt $HOME/bin
./configure --prefix=$HOME/opt
make
make install
ln -s $HOME/opt/bin/python $HOME/bin/stackless
```
B.4 Configuring the Scavenger Daemon

Notice that when building on a Mac an additional argument is needed by configure in order to successfully compile Stackless Python—you must add the argument `--enable-stacklessfewerregisters`. This will build and install Stackless Python in a folder named `opt` in your home directory and make a symbolic link called stackless in `$HOME/bin` that may be used to invoke the interpreter. By adding `$HOME/bin` to your `$PATH` you gain the ability to invoke Stackless Python (using the `stackless` command) without losing the ability to call your standard Python installation (by using the `python` command). This version of Stackless Python will then be used to run the Scavenger surrogate software, so any needed Python libraries—e.g., the Presence library needed by the Scavenger daemon and the Python Imaging Library\(^1\) that is needed by the AugIm application—should be installed using this interpreter. This means, that the `stackless` command should be used when invoking the `setup.py` installation files of the Python libraries.

B.4 Configuring the Scavenger Daemon

Assuming that the Scavenger source code has been checked out into the directory `DIR`, the configuration for the daemon will be placed in `DIR/daemon/scavenger.ini`. Note that if the configuration file does not exist, it will be auto-generated upon initial launch. The configuration file has currently only a few values that can be set: the network bandwidth, the number of CPU cores to use, and the strength of a single core. An example auto-generated configuration file is shown below.

```
[network]
speed = 500000
```

```
[cpu]
cores = 1
strength = 88.85326
```

If no configuration is given Scavenger defaults to an effective bandwidth of 500 Kb/s, which is approximately what you get on an IEEE 802.11b network, a single core is dedicated to the daemon, and the surrogate strength is approximated by a simple internal benchmark. This internal benchmark is not as precise as NBench, so an NBench rating should be preferred—e.g., the estimate in this example is \( \approx 89 \) on the same peer that in Section B.2 received an NBench score of \( \approx 76 \).

\(^1\)http://www.pythonware.com/products/pil/
Appendix B Installing and Configuring the Scavenger Daemon

B.5 Running the Scavenger Daemon

Now that all dependencies have been built and installed all that remains is to start the daemon. Just like Presence, Scavenger is also meant to be run as a daemon on the peer acting as a surrogate, so adding it to an init script is a possibility. Given that the Presence daemon is already running, and that all of the above steps have been successfully performed, starting a Scavenger daemon is as simple as issuing the following commands:

```
cd DIR/daemon
PYTHONPATH=../ stackless main.py
```

Now the Scavenger daemon is running, announcing its availability via Presence, and ready to receive tasks from clients.

Note that Scavenger is prototype software, and currently it should be treated as such. While it is very stable crashes may occur, so it is advisable to keep it running in a terminal so that stdout and stderr are visible and ready to receive debug output.
Appendix C

Using the Scavenger Library

The Scavenger library resides in the same source code repository as the Scavenger daemon at http://code.google.com/p/scavenger-cf/. The easiest way to obtain and install the library is by downloading a distributable package from the homepage—it is the py-scavenger-x.y.z.tag.gz packages that contain the Python library. In this section I will describe how to build and install this library from source. I will assume that the Scavenger source code has already been fetched, according to the instructions given in Section B.1, and that it resides in DIR.

C.1 Building a Distributable Package

Reminiscent of how it was done in Presence, the Scavenger Python library can be built by going to DIR/build/lib. The process of building a distributable package is as shown below:

```
  cd DIR/build/lib
  ./make_dist.sh x
```

This creates an archive called py-scavenger-x.tar.gz just like the ones that can be downloaded from the homepage.

C.2 Installing the Library

When installing the library it is important that the Python interpreter that will be used to run Scavenger-enabled applications is used when invoking the setup script. This need not be a Stackless Python interpreter—any regular Python interpreter of version 2.5 and upwards will do. The installation of the library is done as follows:
Appendix C  Using the Scavenger Library

tar xfvz py-scavenger-x.tar.gz
cd py-scavenger-x
python setup.py build
python setup.py install

Now the scavenger module is available in the chosen Python interpreter and applications using Scavenger may thus be run.

C.3 Installing Dependencies

The Scavenger library depends on the Presence library—either the full fledged library that comes with the Presence daemon, whose installation is described in detail in Section A.4, or the pure Python library that has no dependency to the Presence daemon. Building and installing the pure Python library is done in the same way as when building the ordinary Presence library, the only difference being that this library’s build script resides in DIR/build/pure-python-lib.
Part II

Papers
Scavenger: Transparent Development of Efficient Cyber Foraging Applications

Mads Darø Kristensen, Niels Olof Bouvin

Abstract

This paper presents Scavenger—a cyber foraging system supporting easy development of mobile cyber foraging applications, while still delivering efficient, mobile use of remote computing resources through the use of a custom built mobile code execution environment and a new adaptive, dual-profiling scheduler.

One of the main challenges within cyber foraging is that it is very difficult to develop cyber foraging enabled applications. An application using cyber foraging is working with mobile, distributed and, possibly, parallel computing; fields within computer science notoriously hard for programmers to grasp. In this paper it is shown by example how Scavenger approaches this challenge.

1 Introduction

Cyber foraging was introduced by Satyanarayanan [73] in 2001 and further refined by Balan et al. [4] in 2002. The term covers the opportunistic use of available computing resources by small, mobile devices, and as such is applicable for all kinds of resources that surrogate computers, i.e., the stronger computers offering up their services, may be in possession of. This could be resources such

as CPU power, storage, network connectivity, display capabilities, printers etc. By transparently utilising all available resources and thus blurring the boundaries between the individual machines, pervasive computing [78] comes closer to realisation. In this paper, and in the Scavenger system presented herein, the focus is on foraging for CPU power.

Mobile computing devices are increasingly common—in a matter of years almost everyone will be carrying some kind of smart phone. While these devices are becoming more powerful with regards to processing power, the limiting factor remains battery capacity. A modern smart phone may be fully capable of performing quite resource intensive tasks, e.g., simple image manipulations such as sharpening an image, but doing so is still quite slow and as it drains the battery, it is problematic in a mobile setting. Such tasks can be solved efficiently by utilising cyber foraging to offload the CPU intensive work to nearby surrogates.

Consider the following use-case: A tourist is sitting in a café going through the pictures she has taken earlier in the day. The pictures were taken using the multiple megapixel camera in her smart phone, and she is browsing them to select the ones that she wants to upload to her online image sharing service, so that her friends and family may see them. Before uploading them she applies some filters to them—some need sharpening, others red-eye reduction, and yet others may need their brightness/colour/contrast adjusted. All of these operations are applied only on small previews of the photographs on her smart phone (well within its capabilities), but when she presses the “apply” button, cyber foraging is used to apply the image operations to the actual, much larger, images. Her mobile device has already scanned its environment, located a couple of surrogates provided by the café and some laptop computers owned by other customers, assessed their respective capabilities, and can therefore quickly offload both the image operations and the uploading of the resulting pictures to these devices, leaving her phone free for her to use—and, more importantly, leaving her phone’s battery at an acceptable level so that she may use it for the rest of the day. If no surrogates are available at the café, her mobile device will ask her to choose between performing the operations locally or queueing the operations until later on as surrogates become available.

In a scenario such as this, cyber foraging is not only a benefit—it is almost a necessity. As will be shown in the evaluation in Section 6, asking a current smart phone to apply just three image operations to a single five megapixel image takes more than two and a half minutes, whereas performing the same operations using cyber foraging can be done in less than 17 seconds, and that result is obtained using relatively resource poor surrogates.

As noted by Porras et al. [61], a number of software components must be in place in order to realise a scenario such as the one described above. Firstly, a discovery mechanism must be in place, so that the mobile devices may discover and receive information about available surrogates. Secondly, a way of partitioning an application into local code and remote executable tasks must be devised; preferably in a way that minimises the burden on the application
developer. Thirdly, a scheduler capable of doing cost assessment is needed so that the tasks are performed in the right place—that place being either locally, or at one or more of the available surrogates. Finally, some way of actually performing the identified tasks on a remote host is needed; and preferably this should work on any surrogate regardless of architecture and software availability. I.e., some kind of mobile code approach should be employed.

This paper presents the Scavenger system and discusses how it meets these challenges. The key contributions of Scavenger presented in this paper is 1) the cyber foraging system as a whole, 2) a novel dual-profiling scheduler enabling informed use of resources even in unknown/unprepared environments, 3) a mobile code approach towards task distribution and corresponding execution environment for mobile Python code, and 4) a development model that makes the process of creating cyber foraging enabled applications transparent to the applications programmer.

The benefits of having a cyber foraging system capable of performing all of these complex tasks are nil, if there is not an easy way for application programmers to utilise the system. Cyber foraging entails distributed, and in many cases parallel, computing; two fields within computer science that are known to be hard for programmers to fully grasp. Furthermore, in order for cyber foraging to be really usable, the system should be capable of using mobile code so that functionality may be pushed onto surrogates as needed—thus adding another level of complexity to application development. In previous cyber foraging systems \cite{5, 22, 77, 25} the burden put on the application programmer is quite heavy. Scavenger, on the other hand, is designed to be easy for programmers to use, and in this paper it will be shown by example how highly mobile, distributed, parallel applications may be built, without putting a heavy burden on the application programmer.

The structure of this paper is as follows: In Section 2 related work is discussed. Section 3 presents the Scavenger cyber foraging system with special focus on the benefits of using a mobile code approach, and on how scheduling is done using the dual-profiling scheduler. Section 5 demonstrates how easily a cyber foraging enabled application may be developed using Scavenger, Section 6 shows some experimental results using the developed prototype to show that the application does indeed work satisfactorily, to show off some of the aspects of the scheduler, and to assess the impact of energy consumption. Finally, the paper is concluded in Section 7.

2 Related Work

In the years that have passed since the introduction of the term cyber foraging, a number of different systems have been developed within academia. These systems have had very different foci with regards to e.g., level of mobility, development model, and deployment method—some using per client virtual machines \cite{77, 25}, others relying on pre-installed services reachable via RPC \cite{5, 22},
and yet others using a more flexible mobile code approach [50].

Some of the earliest examples of cyber foraging systems were the twin systems, Spectra [22] and Chroma [3], developed at CMU. Both of these systems used pre-installed RPCs to perform remote tasks, meaning that the mobility of the client devices were restricted to areas where the required functionality was already installed. The mobile code approach in Scavenger alleviates this need to prepare surrogates beforehand. Spectra and Chroma both have schedulers capable of choosing between available surrogates by continuously updating and consulting a history based profile that is maintained for each known surrogate—an approach that works well considering the limited mobility of their system. In this paper, a novel dual-profiling history based scheduling approach is presented, and it is described how this approach yields good performance in both known and unknown environments. Another difference is that Spectra and Chroma seem to be working with only one profile per task, whereas Scavenger works with two-dimensional profiles to account for the fact that task complexity may vary with input size or value. The developer creating applications using Spectra and Chroma must manually partition the application into remote executable RPCs (to be pre-installed on the surrogates), and on run-time the scheduler will tell the application where to perform the different tasks. The amount of code rewriting needed in Scavenger is kept at a bare minimum; all that is needed is that remote executable functions, or tasks, are annotated as such—the rest, both installation and invocation, is then handled by the Scavenger library.

Some systems, e.g., the system by Goyal & Carter [25] and Slingshot by Su & Flinn [77], have experimented with using OS-level virtual machines (VMs) on surrogates, giving a client access to an entire VM hosted at a surrogate. While this approach is as flexible as using mobile code when it comes to deployment, the mobility of such systems is naturally rather low, seeing as an entire VM must be spawned and any needed functionality fetched and installed before any cyber foraging can take place. Scheduling of tasks is non-existent in these systems; as soon as the client application receives access to the virtual machine, it is left up to the application itself, how this resource should be utilised. The same goes for development; the systems provide access to a VM, and the development model, i.e., how the application communicates with and uses this VM, is application specific.

When it comes to developer support related systems have had different approaches; ranging from completely automatic approaches that study an application’s execution history graph and offloads parts of the application based on this. This is the approach taken by Hunt & Scott in the Coign system [32] and by Messer et al. [50]. Other systems, as described above, are at the opposite extreme, offering nothing more that access to a virtual machine. Finally, some systems have taken the middle ground offering some middleware support for cyber foraging, where the developers have to adhere to a specific API. This approach is taken by Spectra, Chroma, and Scavenger. This is done because, as noted by Flinn et al., “a little application-specific knowledge can go a long
way." [22]; meaning that although it is in some cases possible to employ cyber foraging without rewriting an application, the addition of distributed and parallel capabilities may very well mean that some parts of the application can be solved more efficiently by making very small rewrites, e.g., by unrolling a loop into a bunch of parallel tasks.

As noted by Paluska et al. in [58], applications operating in a pervasive computing environment must be ready to adapt to the ever changing resource levels. This adaptation must happen at run-time, since the application programmer cannot be expected to predict all possible resource combinations that the application will have to function in. The approach taken by Paluska et al. is to create a high-level decision making system where the developer of the application states the goals of a given task, and then a number of techniques capable of reaching that goal by using available resources as provided. The set of techniques are extendable at run-time, so that new kinds of resources may be used. In Scavenger, working in the limited domain of offloading CPU intensive tasks, a mobile code approach has been taken, meaning that resources may be “taught” how to perform a task, alleviating the need for resource specific adaption.

Cloud computing is an area of research that is closely related to cyber foraging. Using cloud computing computationally intensive work can be pushed over the Internet into “the cloud”, where the tasks may be scheduled in a data centre using known cluster computing techniques. This removes the burden of resource discovery and scheduling from the client device and thus simplifies the process of using remote execution considerably. Cloud computing is not always a good solution to the problems that cyber foraging aims to solve though, as noted by Satyanarayanan et al. [75]. Because of the high latencies of a WAN link when compared to a wireless LAN connection, and because of the fact that the bandwidth of a wireless LAN is typically two orders of magnitude higher than the wireless Internet bandwidth available to a mobile device, cloud computing is not always feasible. In an application such as the one described in Section 1, the high latency of a WAN link would mean that a considerable delay would be incurred if applied to the operations on the smaller preview versions of the images. And, if applied to the larger, original versions of the images, the bandwidth limitations would mean that the time spent transferring data would by far outweigh the benefits of using remote execution.

3 The Scavenger Cyber Foraging System

Scavenger consists of two independent software components: the daemon running on surrogates enabling them to receive and perform tasks, and the library used by client applications. A device can run both, thus enabling the use of cooperative cyber foraging, where devices work together for the greater good.

Both the client library and the daemon are written in Python, and the execution environment, which is written using Stackless Python, is capable of
performing mobile Python code. Stackless Python is used exclusively on surrogates, so client devices need not support Stackless. Regular Python support is needed on client devices, which is supported on all major PC platforms and also by a large number of mobile platforms. We currently have Scavenger demonstrators running on Nokia’s Maemo based devices, Apple’s iPhone, HTC’s Android phones, Windows, MacOS X, Linux etc.

Figure 1 A high-level view of Scavenger’s architecture.

3.1 Architecture

A high-level view of Scavenger’s architecture is shown in Figure 1. A surrogate must run the Scavenger daemon, and in order to use such resources, an application needs to include the Scavenger library and adhere to some simple rules that are described below.

The daemon consists of a small front-end offering remote access to the mobile code execution environment through some RPC entry points. This front-end is also responsible for device discovery, which it does by using the Presence service discovery framework [39], that has been developed especially for use in cyber foraging settings, where up-to-date information about available resources is critical. Discovery is used not only by client devices; surrogates also collect information about other surrogates available in the environment. For now, this information is used when surrogates fetch intermediate results from each other, but in the future this information could be used for re-scheduling and task migration, enabling the surrogates to hand over tasks to other surrogates. As a part of the Scavenger library, clients also use Presence to discover available surrogates.

All client applications must use the Scavenger library. This library offers two ways of working with cyber foraging: 1) a manual mode, where the application may itself ask for a list of available surrogates, install code onto these surrogates, and invoke this code; and 2) a fully automated mode, where the
application programmer only needs to annotate candidate remote executable functions, and then Scavenger will take care of the scheduling. The focus of this paper is on development using the automatic mode, and on the kind of scheduling that is done when using this mode.

3.2 Mobile Code

Behind the Scavenger front-end lies the mobile code execution environment that offers dynamic installation and execution of Python code. Using this, task deployment in Scavenger becomes the simple task of 1) checking whether the needed functionality is already installed on the surrogate, and 2) installing it before invoking it if necessary. This is handled automatically by the Scavenger library. If true mobility is to be supported, using mobile code is a necessity. Using pre-installed tasks only works in the limited cases where all needed tasks have already been installed on all potential surrogates, and using virtual machines is far too heavyweight, if it is to be used in a mobile environment, where the user may only be within range of a surrogate for as few minutes at a time. By using a trusted mobile code environment, as in Scavenger, it is possible to use pre-installed functionality, if such exists, and if the needed functionality is unavailable, it may simply be installed on-demand by the mobile clients.

Scavenger’s execution environment is designed to fully utilise multi-core machines. A Scavenger daemon can be configured to use any number of cores, and for each core, a core scheduler is spawned that handles tasks performed on that core. Tasks are, in the current implementation, submitted to these core schedulers in a round robin manner. That the number of cores to utilise is a configuration parameter means that a user running a Scavenger daemon is capable of limiting the amount of resources dedicated. Thus, a laptop with two cores can be offering up its services as a surrogate, and at the same time be completely usable by its local user because only one core will be used by Scavenger.

While the execution environment supports installation of code, it is not a mobile code environment in the sense that code must always be given along with task input in order to perform a task. In many cases a client using a surrogate will be interested in invoking the same task multiple times, using different input data for each invocation. Furthermore, if multiple client devices are running the same application, the tasks defined within will be the same, and it thus makes sense for the surrogates to store these tasks for future use once they have been installed. This is handled in Scavenger by giving each task a unique identifier, and upon invoking a task the client initially asks the surrogate whether that task is already installed. These task identifiers may be defined manually by the task developer, but are in most cases autogenerated by the Scavenger library.

Working with mobile code raises a lot of security concerns—especially in a cyber foraging scenario, where untrusted clients must be allowed to install and run their own code. To ensure that malicious clients cannot access the private
data of a surrogate computer, or use a surrogate to launch attacks on other computers on the network, a number of measures has been taken in Scavenger to secure the execution environment. Before mobile code is accepted into the execution environment, it is validated using both a black-listing and a white-listing approach. Built-in language features that are deemed dangerous, such as many of the reflective features of Python, are black-listed, and module imports are white-listed, thus only allowing code that imports trusted modules from the standard library. More measures are in place to secure the execution environment, but further details are outside of the scope of this paper; see [39] for more information.

(a) An ad-hoc network environment.  
(b) A managed network environment.

Figure 2  Two networked environments where cyber foraging could take place.

4 Dynamic Scheduling

Scheduling in a mobile, heterogeneous environment such as the one that must be catered for in cyber foraging is a very complex matter. First off, there is no central scheduler that tasks can be submitted to, so the client must do the scheduling itself. Secondly, due to the ever changing environment, scheduling must be done at run-time acting on the information available at that point in time. Thus, the client is solely responsible for collecting the information needed to make an informed decision when performing tasks. Consider the pervasive computing environment depicted in Figure 1.2(a). An ad hoc network is shown in this figure—such as one that may emerge between the devices of passengers in a train. As this example shows, there is a great heterogeneity of devices with three different brands of smart phones, two laptop computers, and a PDA. Scavenger, utilising the cross platform deployable Python language, runs on all of these platforms. All devices in this scenario may act as both surrogates and clients in a pure peer-to-peer fashion, but in most cases users of small devices such as smart phones and PDAs will not be willing to share resources because of severe battery life limitations.

A cyber foraging scheduler must be able to assess such environments in an instant, gathering the information needed in a dynamic, pro-active fashion. Ideally, the information available when scheduling a task should be:
• The task in- and output size (bytes).
• Estimated running time when performed at any of the available machines (seconds).
• Bandwidth (bytes/second) and latency (seconds) information for all links in the network—both links between client and surrogate and links between surrogates.
• Data locality information. I.e., information about where the input data resides (it may be a result from a previous task stored at one of the surrogates), and whether or not the output data should be pushed back to the client.

Given this information, an idealised scheduler is shown in Figure 3. In this figure $I_{\text{size}}$ is the input size, $I_{\text{loc}}$ the input location and, correspondingly, $O_{\text{size}}$ and $O_{\text{loc}}$ are the size and locality of the output. The bandwidth and latency between two hosts N and M are $B_{N \rightarrow M}$ and $L_{N \rightarrow M}$, and the running time of a task T on a machine M is $T_M$.

```
1 machines = surrogates + localhost
2 candidates = []
3 for M in machines:
4     time = 0
5     if not $I_{\text{loc}}$ == M:
6         time += $I_{\text{size}}$ $B_{M \rightarrow I_{\text{loc}}}$ + $L_{M \rightarrow I_{\text{loc}}}$
7     time += $T_M$
8     if not $O_{\text{loc}}$ == M:
9         time += $O_{\text{size}}$ $B_{M \rightarrow O_{\text{loc}}}$ + $L_{M \rightarrow O_{\text{loc}}}$
10    candidates.append((time, M))
11 candidates.select_minimum()
```

Figure 3 Idealised scheduler.

Getting the information needed to implement such an ideal scheduler is not easy; if even possible. In the following, it is discussed how this information can approximated in a cyber foraging setting.

4.1 Network Information

The network characteristics, i.e., bandwidth and latency measurements for all interconnecting links, are hard to determine, especially if fluctuations due to peaks in traffic are to be considered. The problem is not measuring the current bandwidth and latency between two peers as this is straightforward. However, in order to reliably measure network characteristics, a large amount of network traffic must be generated, and seeing as these measurements must be done periodically between all peers in the network, an excessive amount of measurement traffic would be introduced to the network. The effect of this is especially detrimental in broadcast networks such as Wi-Fi. Therefore, Scavenger eschews pro-active bandwidth and latency measurements in favour of
a static media specification, where both clients and surrogates specify which kind of media they connect to the network with. These media specifications are then mapped to some expected bandwidth and latency values for that specific media. Consider the network in Figure 1.2(b). In this figure, a common managed network consisting of a few wired and a number of wireless devices is shown. In this environment, the wired devices, i.e., the desktop PCs, would specify in their configuration that they are connected to the network using a 100 MBit connection, Laptops and UMPCs would most likely be connected using IEEE 802.11g, and mobile devices perhaps using IEEE 802.11b. During scheduling, the client knows each peer’s specified bandwidth and when considering transfer of data between two peers the smallest common denominator is chosen.

These static bandwidth specifications can be augmented by real measurements if and when a given link is traversed. Following the examples of Spectra [22] and Chroma [5] by instrumenting the RPC primitives used, Scavenger is capable of collecting information about a link whenever it is in use by the cyber foraging system, e.g., when installing or performing a task. These measurements can be used to augment the static specifications by collecting a history based profile of bandwidth measurements and then assigning a weight to these measurements and the static specification. Assigning a large weight to the static specification would cushion the effects of inaccurate bandwidth measurements, whereas a smaller weight puts more trust into the measured values.

4.2 Task Specific Information

Task specific information in this regard is run-time information about the task being scheduled. This entails the size of in- and output data and data locality information.

Seeing as scheduling is done dynamically, i.e., at run-time, the task’s input is readily available at the scheduling device. This means that information about task input is trivially given. The input may reside at other peers in the network, in which case the scheduling device will have a data handle instead of the actual input data. These data handles include information about the size of the data, so that this information may be used when scheduling whether the data is stored locally or not.

Output data, on the other hand, is of course not given before the task has been performed. For some tasks, it may be impossible to predetermine the size of the output, but for many regular tasks it is indeed possible to determine what the size of the output will be. Some tasks always return the same size output, while others have output of a size relative to input size or value. The developer may in Scavenger designate the output size as a relation to the input size or value. How this is done is shown in Section 5.

When developing with Scavenger it is also possible to choose whether to
pro-actively fetch the output of a finished task and deliver it to the client application, or whether output data should be left at the surrogate and a data handle returned in its place. If data is left at the surrogate the overhead of transferring the output data is ignored—and its locality is taken into consideration if it is later to be used as input to another task.

4.3 Peer Specific Information

The most complex part of task scheduling is assessing the running time of the task with the given input on any of the currently available peers. There is only one way of getting at such information and that is through profiling.

One approach towards building a profile usable for task scheduling is creating a peer centric profile, where a history based profile containing information about the last $n$ runs of a task on a specific peer is stored. Whenever that task is considered for execution on that specific peer, the profile may be consulted and the value found herein can be used as an estimate of what the running time will be. How the history based profile is used is a design decision; in Scavenger the average of the profile data from the last ten runs is used as an estimate. Another approach would be to make a weighted average, where more recent profile data is given more consideration.

There are some issues with this kind of peer centric profiling: For one, it assumes that a given task always has roughly the same running time. This is not true; for most tasks the running time will vary with input size or value. Furthermore, when working with highly mobile cyber foraging, the idea of having a peer centric profile, necessitating profile information about the specific peers that the client is currently within range of, works counter to the very mobility of the system. In highly mobile cyber foraging it is more than likely that a given task has never been performed on the currently available surrogates. Both of these deficiencies of profile based scheduling in a cyber foraging setting have been considered in Scavenger, which is why Scavenger uses multidi-dimensional profiles to reflect that a task’s running time may vary with input, and task centric profiles that may be used to reason about task running time on hitherto unknown surrogates.

A final concern about using peer centric profiles in a highly mobile setting, is the sheer volume of profiling data that must be stored. Profile data must be stored for each (task, peer)-pair and in a highly mobile setting, where the mobile device will be working with a large number of surrogates, this will lead to an enormous amount of profile data that must be stored—and searched through upon scheduling. Avoiding this problem is done by treating the peer centric profiles as a cache with a defined maximum size and being careful about which entries to evict when the time comes to reduce the size. The cache eviction policy favours two things: old friends and new acquaintances; i.e., information about peers that the mobile device has worked with a large number of times is always kept, and information about the handful of surrogates most recently used is stored as well. In this way, the profile data on surrogates
available in the user’s regular environment, such as her home or workplace, is retained, and new information about the surrogates currently surrounding her is also kept for the time being, so that it may be used for scheduling in the near future.

**Multidimensional Profiles**

To reflect that running time often varies with input size and/or value, Scavenger works with multidimensional profiles where profiling data is stored in separate batches depending on their input. The task developer may specify which input parameters affect the running time of the task, and using this information a single key value is calculated. If the developer does not specify this, the input size in bytes is used as default. Using this value, computed at run-time every time a task is invoked, Scavenger maintains a two-dimensional profile. After performing a task the profile is updated by inserting the collected profiling data into the “bucket” whose key most closely matches the given key value. Updating the profile is done using the following simple algorithm:

1. If this is the first run simply create a bucket for the given key value and insert the profile data there.

2. If buckets exist find the bucket closest to the given key value and compare the collected profile data to that bucket’s average:
   
   (a) If the profile data differs less than a certain percentage insert it into this bucket.

   (b) If the variation in profile data is too large create a new bucket for the data if, and only if, the key values also differ more than a certain percentage.

By updating the profile thus, the profile data is capable of adjusting to variations in running time, while only maintaining as few buckets as possible. Consider Figure 4. When the running time of the task rises quickly, many buckets will be created to reflect this, and when the increase is low, only very few buckets are maintained. When doing lookups in the profile, the bucket with the key value closest to the current one is chosen, which can be done in $O(\log n)$ where $n$ is the number of buckets in the profile.

Using this approach towards maintaining profiles solves the first of the deficiencies with profile based scheduling—now variations in running time due to input variations are correctly reflected in the profile.

**Task Centric Profiling**

As mentioned earlier on, the biggest problem with profile based scheduling in highly mobile cyber foraging is the peer centric profiles. Ideally, there should
be some way of comparing the “strength” of different surrogate machines to each other. What is needed is a measure such that the running time of a task on a machine of strength $x$, would be half that of the same task on a machine of strength $\frac{x}{2}$. Unfortunately, no such perfect strength measure is available—multiple factors are in play when measuring a machine’s processing capabilities; CPU architecture, cache structure and speed, main memory speed, and even compilers all play a role in the performance of a modern computer.

While nowhere near perfect, there are ways to compare the relative strengths of computers, most notably by benchmarking the machines by using a benchmarking suite. Both surrogates and clients are in Scavenger benchmarked using the NBench\(^1\) benchmarking suite, and the score yielded by this benchmark is used as a strength value. This peer strength value gives a rough estimate of how a device performs when it comes to CPU intensive tasks, but it does have its problems. Different tasks may exercise different parts of the CPU, and some task may therefore perform much better on some architectures than on others. This is in Scavenger alleviated by the use of dual-profiling, as will become clear shortly.

Knowing how strong peers are relative to each other we are able to build task centric profiles, i.e., profiles that are bound to a single task instead of to a (task, peer)-pair. Where the peer centric profiles could contain simple time measurements of earlier executions of the task, the task centric profile must contain some “task weight” that can be scaled by the peer strength. The information stored in these task centric profiles are the expected running time on a machine of strength one. When considering scheduling the task on a peer, this task weight is divided by the current peer strength to obtain an assessment of what the running time would be. The current peer strength referred to in the previous sentence, is the strength of the peer under its current load. The discovery packets periodically sent out by surrogates contain both their strength.

\(^1\)http://www.tux.org/~mayer/linux/bmark.html
and their activity count, where the activity count is the number of tasks that are currently being performed within their execution environment. The current peer strength is then estimated as:

\[ Peer_{current\_strength} = \frac{Peer_{strength}}{Peer_{activity} + 1} \]

It is thus assumed that active tasks share the CPU equally, and any other processes being performed by the operating system are ignored. This is not a fully realistic assumption, but it does deliver a usable estimation of the current load. The problem with reading the actual CPU load or load average, using e.g., a call such as `getloadavg`, is one of granularity; reading the current load makes little sense since it fluctuates far too much, reading the average over some time period, like the one, five, and 15 minute averages given by `getloadavg`, is also imperfect—especially so in a use setting such as cyber foraging where task execution is most often done within mere seconds. Using the activity count within the execution environment makes more sense because of a few observations we have made: 1) typical laptop or desktop computers consume an almost constant amount of processing power when in regular use, which means that the CPU intensive tasks performed in the execution environment will almost always be able to share a constant amount of resources, 2) modern surrogates have multiple CPU cores, and the execution environment will therefore most likely run on its own on a single core, in which case the amount of available computing resources are constant, and 3) the Stackless Python based scheduler within the execution environment makes sure that tasks get an equal share of the CPU resources.

Whether or not the task centric profile data is usable depends on the assumption described above, that e.g., a peer of strength ten will be able to perform a task twice as fast as a peer of strength five. That this assumption largely holds can be seen in Table 1.

<table>
<thead>
<tr>
<th>Task</th>
<th>G5 2 GHz</th>
<th>G4 733 MHz</th>
<th>Pentium 3 1 GHz</th>
<th>Celeron M 900 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness</td>
<td>51</td>
<td>50</td>
<td>55</td>
<td>53</td>
</tr>
<tr>
<td>Colour</td>
<td>54</td>
<td>54</td>
<td>57</td>
<td>54</td>
</tr>
<tr>
<td>Contrast</td>
<td>55</td>
<td>55</td>
<td>59</td>
<td>55</td>
</tr>
<tr>
<td>Sharpen</td>
<td>110</td>
<td>126</td>
<td>107</td>
<td>98</td>
</tr>
<tr>
<td>Blur</td>
<td>83</td>
<td>96</td>
<td>83</td>
<td>73</td>
</tr>
<tr>
<td>Invert</td>
<td>21</td>
<td>27</td>
<td>32</td>
<td>28</td>
</tr>
<tr>
<td>Scale</td>
<td>91</td>
<td>103</td>
<td>45</td>
<td>35</td>
</tr>
</tbody>
</table>

**Table 1** Task weight measurements. Ideally values should be equal across rows.

To produce the data in Table 1, four different machines performed seven image manipulation tasks and reported the weight they would assign to that task. All tasks were performed 50 times and the weight reported is the average of these runs. The machines in use are very different with regards to processor architecture; having both a PowerPC G4 and G5, an Intel Pentium 3, and an
Intel Celeron M processor. Even with those differences in architecture, it can be seen that the assigned weights are quite similar, and when used in a history based profile, these provide a good starting point for the scheduler. Notice, that even though the different machines tend to agree on the weight of most of the tasks, the architectural differences shine through in some cases. Consider for example the last row, where an anti-aliasing scale operation was performed. In these tests the PowerPC based (G4 and G5) machines reported weights that were twice or almost three times as high as the Intel based machines. This shows that while using this “task weight” based on benchmarking scores is no silver bullet, it does in most cases provide good results. And compared to having no knowledge at all about the estimated running time of task execution on unknown surrogates, it makes for a more informed scheduling in unknown environments.

![Peer Centric Profile
Task Centric Profile](image)

**Figure 5** Scavenger’s dual profiles. The peer centric profile contains information about the task on peers A, B, and C, whereas the task centric profiles holds information about the last five runs on any peer.

**Dual Profiling**

Using the task centric profile described above gives a good starting point for the scheduler when scheduling in an unknown environment. But, as has been shown, the task centric profiles are not always precise, and peer centric information should therefore always be preferred if such information is available.

Based on this observation, Scavenger’s scheduler works with dual profiles; as seen in Figure 5. Whenever a task is performed, two profiles are updated—a peer centric profile and a task centric profile. Both of these profiles are updated using the task weight measure, where the measured running time is scaled by the surrogate’s strength and activity level. When a task is being considered for execution on a given surrogate, the peer centric profile is consulted first in order to give the most precise data precedence. If no peer specific information is available, which is likely in a mobile cyber foraging scenario, the task centric profile is consulted. The information stored here is likely to be less precise, but our tests show that it is still quite effective at guiding the scheduling process.
Scavenger’s Scheduler

To sum up, Scavenger uses estimates of all the information that was in use in the idealised scheduler presented in the beginning of this section. Using this information the scheduling algorithm shown in Figure 6 has been implemented. This figure presents the algorithm in simplified Python-like pseudocode very close to the actual implementation.

```
peers = get_surrogates()
if peers == []:
    return localhost
peers.append(localhost)
input_size = len(dumps(input))
output_size = eval(output_size_expression)
complexity_interval = eval(complexity_expression)
task_centric_profile = get_tc_profile(task)
candidates = []
for P in peers:
    time = 0
    if input.location != P:
        bw = min(P.bandwidth, input.location.bandwidth)
        time += input_size / bw
    peer_centric_profile = get_pc_profile(task)
    current_strength = P.strength / (P.activity + 1)
    if peer_centric_profile:
        time += peer_centric_profile / current_strength
    else:
        time += task_centric_profile / current_strength
    if output.location != P:
        bw = min(P.bandwidth, output.location.bandwidth)
        time += output_size / bw
    candidates.append((time, P))
return select_minimum(candidates)
```

Figure 6 Scavenger’s scheduler in simplified Python-like pseudocode.

The running time of the scheduling algorithm is $O(n \log m)$ where $n$ is the number of peers currently available and $m$ is the maximum number of buckets within the two-dimensional profiles. In all practical use the number of buckets in the profiles will be a very small number; so small that the lookup in the profile may be considered to be of $O(1)$ running time, yielding a linear $O(n)$ running time to the entire scheduling algorithm. This modest running time means that the overhead of scheduling is very small, even on resource poor mobile devices.

5 Development Using Scavenger

The application that will receive cyber foraging capabilities in this section is the image browser/editor for mobile devices described in Section 1. Operations such as the ones supported by this application are quite heavy for modern mobile devices to perform, but they are capable of performing them on their own, when no surrogates are available. It will in the following be shown, how to add cyber foraging to an existing application when using Scavenger. It is therefore
assumed that the application has already been developed and is running on the
mobile platform of choice, and what will be shown here is merely the changes
needed to enable cyber foraging.

As mentioned above, the Scavenger library has two operating modes:
manual and fully automatic. In this example only the automated mode will
be demonstrated. The automated mode works by the use of Python function
decorators. Python decorators are higher order functions that wrap the func-
tion they are attached to, so that when the decorated function is called, the
decorator is called first. Using this built-in language feature adding cyber for-
aging can be as simple as adding a decorator to a function that may benefit
from remote execution. There are a few rules though, that must be adhered to
for the remote execution to work:

1. The function must be self-contained, i.e., it must not call other functions
   or methods defined elsewhere in the application.

2. Modules used from within the function must be imported within the
   function itself, so that these modules may also be imported and used at
   the surrogate.

```python
@scavenge
def sharpen(image, factor):
    from PIL import ImageEnhance as IE
    factor = 1.0 + factor
    return IE.Sharpness(image).enhance(factor)
```

**Figure 7** A simple task defined using a Python decorator.

A small example function adhering to these rules is shown in Figure 7. In
this figure the function “sharpen” is decorated using the “scavenge” decorator.
This decorator enables use of automatic cyber foraging whenever the sharpen
function is called. What the decorator does is to 1) fetch the function source, 2)
generate a unique id for the function, 3) ask the scheduler to schedule the task
immediately. The scheduler then 4) checks whether surrogates are currently
available, 5) chooses the most eligible surrogate (which may in fact be the local
device), 6) installs the function code onto the chosen peer if needed, and 7)
performs the task on the peer returning the result.

All of these steps are completely automated by using the scavenge decor-
arator, leaving the developer free to worry about other, non-distributed parts
of the application. The functions used must be self-contained, and this is reflec-
ted in the example as import statements are done within the function body,
whereas the regular Python coding style would be to have these import state-
ments in the top of the source file. Also the function body does not call external
functionality, apart from the functionality imported from the Python Imaging
Library (PIL), which is a standard module found on all surrogates. That the
function must be self-contained does not mean that developers may not define
new functions and classes—as long as these functions and classes are defined within the function body of the decorated function it will work just fine.

The scavenge decorator can be used in this simple manner, but in order to give the scheduler optimal working conditions, some arguments should be given to the decorator: an expression relating the output size to the size and/or value of the input parameters, and an expression designating which input parameter(s) are determining for the complexity of the task. These arguments can in Scavenger be given directly to the decorator as two strings. Within these strings the input parameters of the function may be referred to by position; e.g., the expression `len(#0)` would refer to the length of the first input parameter. Anything expressible in a Python expression can be written into these strings. This gives a very powerful mechanism for expressing how output size and task complexity relates to the size, value, or any other property of the input parameters.

Apart from these arguments one more optional argument is accepted by the decorator. This argument is a simple boolean value; if set to true the result data from the task is left at the surrogate, and only a remote data handle is returned. If set to false, the result data is fetched and returned to the client application. These remote data handles may be used as input parameters for future task invocations, and the scheduler will then take data locality into account when scheduling the task. Working with remote data handles is completely transparent to the developer. Wherever a task input is needed a remote data handle may be given in its place. The Scavenger library will detect this automatically and schedule accordingly. Using these arguments the full implementation of the sharpen task is shown in Figure 8.

```python
@scavenge('len(#0)', 'len(#0)', True)
def sharpen(image, factor):
    from PIL import ImageEnhance as IE
    factor = 1.0 + factor
    return IE.Sharpness(image).enhance(factor)
```

Figure 8 The actual source code of the sharpen task in the image editor.

In the image editor a chain of image operations are often applied to the same image. The last parameter of the decorator has therefore been set to true, to keep the result data at the surrogate for use when the next task in the chain is scheduled.

It is thus simple to add cyber foraging capabilities to an application using Scavenger’s automated mode. The client application is completely oblivious to whether or not remote execution is performed, it just uses a self-contained function to do its resource intensive work, and the Scavenger library takes care of the rest. As for parallel usage of resources, a client application should handle that in the usual way, by spawning worker threads to handle multiple concurrent tasks. The Scavenger library is completely thread safe, and is capable of scheduling multiple tasks in parallel over multiple surrogates. If no surrogates
are available, the heavy work will be done by the worker threads on the client device.

Using the scavenge decorator, a cyber foraging enabled image editor prototype for Nokia Internet Tablets has been developed. All that needed to be changed in the application to incorporate Scavenger was to add the decorator to twelve image operations, other than that the application is left unchanged. A screen shot of this application is shown in Figure 9.

The source code for this demo application can be found alongside the Scavenger source. The next section offers a brief look at the performance benefits that can be reaped by using Scavenger.

Figure 9 The Augmented Image Manager (AugIM) demonstrator.

6 Experimental Results

Extensive benchmarks and experiments have been performed to measure the performance increase gained when using Scavenger, and also to validate the scheduling approach when compared to a handful of other schedulers. A minute dissection of all of these results are outside of the scope of this paper; what will be presented here is a select few tests that show of some of the more important aspects of Scavenger as a cyber foraging system. The benchmarks reported on are all done within the mobile image editor presented above, and the test environment is the one shown in Figure 10.

The figure shows the network architecture, consisting of three desktop PCs connected directly via cable to the Wi-Fi router, and two mobile devices, an Asus Eee 900 UMPC and a Nokia N800 Internet tablet, connected to the same router via Wi-Fi. On each device its architecture and CPU speed is listed along with its measured NBench strength rating; which is the strength measure that is used in the task centric profiling. The devices have been named A through D, and these names are used throughout this section. The operating systems used
6.1 Performance evaluation

For these experiments the client application was installed on $E$, and machines $A$, $C$, and $D$ were available as surrogates. These surrogates are all fairly slow compared to modern machines, but when compared to the mobile device, they are lightning fast, and, as will be shown shortly, using even old machines like these as surrogates yields big benefits for the mobile devices utilising them.

For the first benchmark a linear string of three image operations, expressed as Scavenger tasks, are performed. The operations are an image sharpening, colour adjustment, and finally a contrast adjustment. These operations are performed both on the original five megapixel image and on a thumbnail version, roughly half a megapixel in size, which is the one shown in the UI of the mobile application. All tests have been run 50 times and the measurements reported here are the averages of these runs. The test was initially run without surrogates to measure the local execution time, and then with each surrogate turned on in turn. If all surrogates were turned on at the same time, the scheduler would soon find that the PowerMac was the strongest, and all tasks would be sent to that same surrogate. The results of the test are shown in Figure 11.

A number of things can be concluded looking at this figure. For one it is immediately clear that employing cyber foraging is essential when working on the original, five megapixel image. If the mobile device itself tries to perform the tasks it takes more than two and a half minutes—time in which the mobile device is left completely unresponsive because its CPU is 100% utilised. In fact, even performing the operations on the thumbnail version of the image is quite resource intensive for the mobile device, using around nine seconds to perform all three operations. This is detected by the scheduler, and these
Figure 11 Results of the first benchmark. Each bar shows the total running time as experienced by the client when performing the task. Within three of the bars a lighter bar is depicted which illustrates the fraction of the time spent doing CPU bound work.

preview operations are thus also forwarded to surrogates when available. The bars depicting the running time of handling the original image when using a surrogate all have a lighter shaded bar within them. This illustrates the amount of CPU time spent actually performing the operations on the surrogate, and the rest of the running time can be accredited to network overhead. This network overhead of roughly eight to ten seconds cannot be reduced further by adding stronger surrogates, which sets a hard lower bound on the obtainable benefits when working with such relatively large in- and outputs. In a setting where e.g., an IEEE 802.11g connection were in use, this network overhead would of course be brought down and stronger surrogates would be more beneficial.

6.2 Batch Scheduling

The second benchmark discussed here investigates the possibility of using multiple surrogates in parallel; again without having to do anything special on the client side, as this is handled entirely by the Scavenger scheduler. When no surrogates are available, it is common for cyber foraging enabled applications to queue up tasks for later processing. When a client device having such queued up tasks enters an area with surrogates, it is interested in scheduling all of these tasks in parallel using the available surrogates. This scenario is tested in the benchmark covered here. The client device has a queue of 24 image operations that is to be applied to eight images; more precisely each of the eight images are sharpened, have their brightness adjusted, and finally have their contrast adjusted in that order. When the benchmark is started the first eight
tasks are immediately scheduled, and as soon as one of these finish the task waiting on it is scheduled. The input images used in this test were three megapixel (2000×1500 pixels), each run has been repeated 50 times, and the results presented here are averages of these runs.

When doing batch scheduling it is important that tasks are distributed in a way such that the available surrogates are equally utilised. This utilisation must be relative to the surrogate’s relative strength, so that stronger surrogates are assigned more tasks than weaker ones. Given the relative strength of the surrogate devices in the test environment in Figure 10, an ideal scheduler should give a bit more than 40% of the tasks to surrogate A, and approximately 20% each to surrogates B, C, and D. The actual distribution of tasks over the available surrogates that was obtained in the benchmarks is shown in Figure 12. Figure I.12(a) shows the average distribution of tasks over the available surrogates, whereas Figure I.12(b) shows an example of a single schedule of the tasks. On each task node in Figure I.12(b) its position in the overall schedule is printed—i.e., the node with position 1 was scheduled before the node with position 2.

![Figure 12](image)

(a) Average distribution of tasks to surrogates in percentage.

(b) An example distribution from a single run. Task nodes are coloured to show where they have been performed.

**Figure 12** Distribution of tasks in the batch scheduling experiment.

The results shown in Figure I.12(a) yielded the expected behaviour. Most tasks are performed by the stronger surrogate, A, whereas surrogates B, C, and D share the remainder of the tasks amongst them. Surrogate D is given a few percent more than B and C because it has a slightly stronger CPU.

Another thing that may be shown by this benchmark, is how efficient the scheduler is at using the information about network bandwidth and the locality of data. In this benchmark, the result of intermediate tasks are left at the surrogates performing those tasks, and the scheduler is thus able to employ knowledge about that when scheduling dependent tasks. During each test run the client logged where each task was sent, and using that information the task
schedules shown in Figure 1.12(b) was created. Looking at the coloured task graph, it is seen that the scheduler very often chooses to perform a dependent task on the surrogate already holding the data. A handoff between surrogates $A$, $B$, and $C$ is seen twice because such a handoff is cheap given the 100 Mb/s network connecting them, whereas a handoff from one of these surrogates to $D$ is only seen once, because of its relatively slow IEEE 802.11g connection.

6.3 Energy efficiency

Apart from the obvious benefit of having increased performance with regards to running time, initial studies conducted using Scavenger have also shown that energy is preserved by offloading resource intensive tasks.

The mobile device used in these tests is a Nokia N810 Internet Tablet with a 400 MHz TI OMAP 2420 (ARM1136) processor and 128 MB of DDR RAM. The surrogate in use in the benchmarks is a 2008-model MacBook with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of 1067 MHz DDR3 RAM running Mac OS X 10.5.8. The network media connecting the client and the surrogate was an IEEE 802.11g network served by a Linksys WRT54G router. Unless stated otherwise, the network was only in light use while tests were performed.

The application used for benchmarking was AugIM, shown in Figure 9. The tasks of this application have been used in the following two experiments. The first experiment simulates that the user browses her images, selects an image for editing, previews three image operations on a 0.3 megapixel (MP) preview version of the image, and finally commits to these changes by applying them to the original 5 MP image. This is done for 15 images in each test run. The image operations performed are sharpening, brightness, and contrast adjustment, and all of these are targets for remote execution. All experiments have been performed six times, and the values reported here are the averages of these runs. The results of this experiment are listed in the first part of Table 2.

Performing such large tasks, i.e., working on the 5 MP images, is very resource intensive for the mobile device, while the surrogate can perform such operations in mere seconds. It thus comes as no surprise that the running time of the tests is brought down substantially by utilising cyber foraging. When the running time is reduced, so is the total amount of energy used. What is less obvious though, is that the immediate energy usage (energy used per time unit) is also reduced; shown as the last column of the first part of Table 2. Turning on the Wi-Fi interface to use a surrogate thus brings the immediate energy consumption down by 30%, compared to turning off Wi-Fi and performing the task on the mobile device. This result is obtained because, on the chosen hardware platform, a fully utilised CPU consumes more energy than having Wi-Fi turned on and having an almost idle CPU; for more information see [44].

The experiments performed in the first experiment asked the mobile device to perform very resource intensive tasks; tasks so large that a decent cyber foraging application should refuse to perform them when no surrogates are avail-
able, and rather queue them up for later processing when surrogates do become available. In the next experiment only operations on the smaller preview images are performed. These operations can be performed by the mobile device in a few seconds, so local execution is feasible. But, if a surrogate is available, Scavenger’s scheduler soon finds out that the tasks may be performed faster by using the available surrogate. The test simulates that the user browses her images, selects an image, previews three image operations, and then queues the task for later execution. This is done for 75 images in each test run. Again, each test has been run six times and the numbers reported here are averages of these runs. The results of the second experiment are shown in the second part of Table 2.

<table>
<thead>
<tr>
<th>Description</th>
<th>Experiment 1</th>
<th></th>
<th></th>
<th>Experiment 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Energy</td>
<td>Energy</td>
<td>Time</td>
<td>Energy</td>
<td>Energy</td>
</tr>
<tr>
<td></td>
<td>seconds</td>
<td>mAh</td>
<td>mAh/s</td>
<td>seconds</td>
<td>mAh</td>
<td>mAh/s</td>
</tr>
<tr>
<td>No surrogates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wi-Fi off</td>
<td>2928±142</td>
<td>187±22</td>
<td>0.0642</td>
<td>1666±27</td>
<td>58±2</td>
<td>0.0350</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>No surrogates</td>
<td>2868±90</td>
<td>206±18</td>
<td>0.0719</td>
<td>1642±44</td>
<td>78±9</td>
<td>0.0475</td>
</tr>
<tr>
<td>Wi-Fi on</td>
<td>98%</td>
<td>110%</td>
<td>112%</td>
<td>99%</td>
<td>134%</td>
<td>136%</td>
</tr>
<tr>
<td>Surrogate</td>
<td>579±26</td>
<td>26±2</td>
<td>0.0452</td>
<td>1135±54</td>
<td>44±3</td>
<td>0.0390</td>
</tr>
<tr>
<td>available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wi-Fi on</td>
<td>579±26</td>
<td>26±2</td>
<td>0.0452</td>
<td>1135±54</td>
<td>44±3</td>
<td>0.0390</td>
</tr>
<tr>
<td>Surrogate</td>
<td>68%</td>
<td>14%</td>
<td>70%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no noisy</td>
<td>1212±58</td>
<td>49±4</td>
<td>0.0401</td>
<td>1212±58</td>
<td>49±4</td>
<td>0.0401</td>
</tr>
<tr>
<td>network</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 In the first test, three image operations are applied to 15 image previews and then to the original images. In the second test, three image operations are applied to 75 of the smaller preview images.

Looking at the results in Table 2, a number of things become apparent. The first thing to notice is that again the total running time has been reduced by using remote execution, while also reducing the energy consumption. In this experiment however, the running time reduction is slight, because only a small fraction of the time is used actually performing the CPU intensive tasks—most time is spent idling while waiting for user input. Comparing the test with Wi-Fi on and no surrogates available to the one where a surrogate is available, the running time is brought down 30%, but the energy consumption in the same tests is reduced by 43%. This is also reflected in the the immediate energy consumption, shown in the last column, being lower when using a surrogate. The immediate energy consumption is, however, lowest in the test where the Wi-Fi interface is turned off, and the mobile device is thus performing all tasks on its own. The total energy consumption is still considerably higher though, using 32% more energy than when using remote execution, so using a surrogate is still preferable.

We have also performed test runs, where the network medium is in use by other computers. In this test the surrogate and another N810 device are continuously communicating, maintaining an approximately 800 kb/s rate of
data transfer between them. Introducing this noise naturally made the cyber foraging perform slightly slower, thus using more energy in total. The immediate energy usage also became slightly larger in a noisy environment, but this may properly be accredited to the broadcast nature of Wi-Fi, where data sent between other peers in the network is overheard by all peers. What should be noted with regards to this test is, that the overall energy consumption is still 16% less than that when disabling Wi-Fi altogether, which is because the total running time is brought down by about 27%.

It is important to note, that while the necessary operations still take considerable time even when powerful surrogates are available, the user experience is completely different. When the mobile device is handling all operations, it is using its CPU 100% and is therefore sluggish and unresponsive. In contrast, when using a surrogate, the mobile device will be spending most of its time waiting for data and user input, and will therefore be responsive and capable of handling other tasks the user may decide upon.

6.4 Scheduling Overhead

One of the differences between Scavenger and the related systems is, that we encourage developers to make liberal use of cyber foraging by using the “scavenge” decorator. The scheduler is fully capable of determining whether or not remote execution is a viable choice at the given point in time, and if it is not the overhead of the scheduling is negligible. There are three cases to consider with regards to this overhead. The first case is when no surrogates are available, and in this case the overhead of using the decorator is a single function call and a check to see whether a dictionary is empty. In the second case surrogates are available and remote execution is chosen, and in the third surrogates are again available but local execution is chosen. In the second and third cases actual scheduling is done and a benchmark of these cases have been performed. In this experiment the task was scheduled 50 times using six different input images of varying size. Input size is varied because marshalling of the input is a necessary step in the scheduling, in order to get the size of the input when transferred over the network. The task being performed is a brightness adjustment of the input image. The results of this test can be seen in Figure 13.

The first bar of Figure 13 depicts an extremely small task; it is in fact so small that local execution is chosen, meaning that the time spent scheduling is time wasted. In this case the overhead of the scheduling is 16.6%, which amounts to 0.04 seconds. In all the other bars remote execution is used, and the cost of doing scheduling is negligible—even when considering the relatively low running time yielded by using remote execution.
Figure 13 Scheduling overhead in percent relative to the total running time of the task. Over each bar the total running time of the task in seconds is shown.

7 Conclusion

This paper has presented Scavenger, a cyber foraging system with a new approach towards task distribution and scheduling. Scavenger’s mobile code approach has been presented, as has its novel dual-profiling scheduler using adaptive history-based profiling. The scheduler works with multiple factors when selecting where to place tasks; considering both data locality, network capabilities, device strength, and task complexity. The entire source code of Scavenger is released as open source, and practitioners within the field of pervasive computing are encouraged to fetch it and experiment with it in their own computing environments. The source may be found at: http://www.interactivespaces.net/projects/Locusts/

The paper has also shown how cyber foraging enabled applications may be created using the Scavenger system. The process of writing highly distributed, parallel cyber foraging applications has in this system been reduced to adding a single decorator to resource intensive code that may benefit of remote execution. This is a big step forward for the usability of cyber foraging as a general pervasive computing technique—to some degree answering one of the original questions posed by Satyanarayanan in [73] while defining the term cyber foraging.

Finally, this paper has shown through experiments that 1) even for modern mobile devices applying cyber foraging to common tasks such as simple image manipulations yields very large speedups—even when the surrogates are relatively old computers as was the case in these experiments, 2) that Scavenger as a cyber foraging system is capable of greatly increasing the performance of mobile applications, and that 3) considerable energy savings can gained using cyber foraging.

Future work within the Scavenger system entails mainly experimentation with the dual profiling scheduler to further prove the viability of the task-centric profiling approach. Also further experimentation concerning the en-
ergy usage of the mobile clients will be performed—perhaps augmenting the scheduler such that energy usage may become a factor when task placement is chosen.
1 Introduction

Mobile devices such as PDAs and mobile phones are rapidly advancing to become full-fledged personal computing devices. In particular, besides supporting phone calls, mobile phones nowadays provide storage, computing, communication, and multimedia capabilities thus to be considered the primary personal computing devices of the future [7]. However, although relatively powerful, mobile devices will always be constrained in terms of physical size, thus leading to limitations in their computing and communication capabilities, battery lifetime as well as screen and keyboard size. These constraints inhibit mobile devices from fully supporting increasingly demanding mobile applications. Furthermore, although processing capabilities have followed Moore’s law for the last 30 years, the more critical resource on mobile devices is battery energy density, which has shown the slowest trend in mobile computing [59].

Current trends in the field of mobile and ubiquitous computing, such as advances in sensor technology, wireless sensor networks, and mobile ad hoc networks, enable and promote the usage of networked resources to augment resource-constrained mobile devices. According to the ubiquitous computing vision, embedding computation into the surrounding environment enables
people to exploit available computing capabilities in an unobtrusive manner, so that ubiquitous computing systems ultimately become an invisible technology and interactions with computers become natural [78]. Computing utilities of such ubiquitous environments, often called smart spaces, include traditional desktop devices, wireless mobile devices, digital assistants, game devices, wrist watches, clothing, sensors, RFIDs, cars, consumer electronics (e.g., TV, microwave), etc. Mobile devices entering smart spaces probe their surroundings to look for devices offering resources such as processors and storage repositories, or input/output devices (e.g., displays, microphones, and video cameras). They opportunistically use such resources and, every time any of such devices becomes unavailable or new ones appear, they adapt accordingly. This process is usually referred to as cyber foraging defined as "living off the land" [73].

Cyber foraging is not the only possible approach to accomplish dynamic resource management. Both parallel and distributed computing systems have used similar approaches. Parallel computing environments take advantage of several processing elements by partitioning the computational problem and executing multiple parts simultaneously. In traditional parallel computing, the execution environment is usually fixed although dynamic processes such as dynamic scheduling, load balancing, and process migration are commonly used to make efficient use of parallel resources. Distributed or grid computing represents a special type of parallel computing which assumes the availability of actual machines with CPU, memory storage, power, network interface, etc. connected to a wired network, instead of multiprocessors connected to a single computer bus. Distributed systems aim to unify multiple system resources (e.g., servers, storage systems, and networks) into a single large system and hide the distributed nature of the environment to the end user. In contrast to traditional parallel and distributed approaches, cyber foraging primarily targets wireless-enabled mobile devices operating in dynamic mobile environments.

This chapter starts with a description of example scenarios where cyber foraging can be applied. It then gives an overview of the cyber foraging process with a special focus on the challenges that arise in every step of the process. The discussion continues with an overview of some existing prototype systems that support cyber foraging.

2 Scenarios

Cyber foraging can be applied in several mobile computing scenarios. This section outlines three examples with a special focus on the type of problems cyber foraging can solve.

Wearable computing: In wearable computing systems, a common goal is usually to minimize the size of the computing equipment while retaining the necessary device functionality. An example scenario, taken from [38], is a doctor wearing a small microphone while doing home visits. Using the micro-

Paper II Dynamic Resource Management and Cyber Foraging
phone the doctor is able to enter information about his patients into an electronic journal. To enable this functionality the microphone must be able to do speech recognition and send the recognized sentences over a secure connection to some central server. While the latter is indeed possible the former of these operations is not—a mobile device of such small dimensions will simply not have the raw processing power to perform proper speech recognition. On the other hand, if a more powerful computing device is currently available in its vicinity, the task of speech recognition can be forwarded to such a machine. If no powerful device is currently available, the microphone will store the recorded sound for later processing, when a computing server becomes available.

**Image processing:** During crowded events such as a political convention or a football match at the Olympic stadium, policemen can use mobile cameras to identify suspicious entities. Face recognition algorithms usually rely on relatively powerful computing capabilities and on the availability of large databases of facial images. Policemen’s cameras collect photos and can directly process part of the collected images through a verification application where an algorithm verifies that a certain face corresponds to a claimed identity by using a locally available database of a small number of mugshots. Alternatively, in the presence of powerful computing servers, photos of unverified identities can be processed by more advanced algorithms that are capable of identifying unknown faces by relying on much larger mugshot databases.

**Region monitoring:** After an earthquake it is necessary to monitor the state of precarious buildings in the immediate proximity of a disaster area, so as to help with the rescue operations. Using the sensors of computing devices situated within the disaster area, a monitoring client can carry out visual or sensorial analysis of buildings and streets in the affected region. A mobile ad hoc network can be formed on-the-fly using all available devices and observations can be provided to the remote monitoring client. In spite of device mobility and battery exhaustion of the devices hosting the monitoring task, the remote client would like to receive a continuous stream of observations. Consequently, after a certain device has left the region of observation or has failed, a new device capable of hosting the monitoring task should be quickly discovered and the task migrated to the new device.

These usage scenarios demonstrate how cyber foraging can be useful in many different situations and for different purposes. In the first scenario, cyber foraging permits augmenting the limited speech recognition capabilities of a small wearable microphone with the processing capabilities of a powerful server. In the second scenario, it allows policemen equipped with small portable camera devices to carry out advanced face recognition by occasionally interacting with larger databases of mugshots and the available infrastructure. Finally, the remote monitoring client presented in the last scenario is able to collect observations from the immediate proximity of a disaster area by delegating the monitoring task to a network of collaborating devices currently located in the disaster area.
3 The Cyber Foraging Process

Cyber foraging aims to dynamically augment the capabilities of client applications running on resource-constrained mobile devices. Cyber foraging can be defined as the opportunistic use of resources and services provided by computing devices available in the surrounding environment. These nearby computing devices are generally referred to as surrogates.

As Figure 1 illustrates, to fully accomplish cyber foraging a multi-step process needs to successfully take place. First of all, client applications that want to delegate computing tasks to external resources must discover surrogates available in their vicinity. Once the surrogate discovery has completed, the application needs to be partitioned into locally executable tasks and remotely executable tasks that may be assigned to surrogates. The best execution strategy specifying “where” to execute “which” tasks is determined by trading off cost and performance of the cyber foraging process if performed in the current execution environment. In addition, an important requirement of many applications and surrogates is that before cyber foraging can occur a trust relationship must be established among the two parties, for instance, to prevent malicious code from harming the surrogate and vice versa. Once all prerequisites are guaranteed, cyber foraging can occur. The actual execution can take different forms. For example, application tasks can be pre-installed on surrogates or can be migrated on-demand. The communication paradigm between client and surrogates (e.g., message-based, publish-subscribe, client-server paradigm) can also vary a lot depending on the execution environment and the type of task. Fi-
nally, as surrogates available in mobile environments can quickly appear and disappear as well as change their offered resources, clients must constantly monitor changes of their execution environment and adapt the cyber foraging process accordingly. Consequently, client applications that rely on cyber foraging are also required to integrate a certain degree of adaptability and be able to adjust their execution to the current level of resource availability.

Depending on the execution conditions, all or some steps of the cyber foraging process may need to be repeated multiple times. For instance, when a more powerful surrogate appears in the environment the cyber foraging tasks can be moved from the old surrogate to the new one. This implies that the steps of application partitioning, cost assessment, and trust establishment must be re-executed. Another possibility is that at some point a currently employed surrogate becomes overloaded thus decreasing the quality of the provided services. In this case, a new application partitioning strategy can move some of the tasks to another surrogate while the other tasks continue uninterrupted on the current surrogate.

In this section, we describe the process of cyber foraging and highlight the challenges that arise in each step of such a process together with possible solutions.

3.1 Surrogate Discovery

The first step of any cyber foraging process is the discovery of surrogates. Client applications need to discover surrogates available in the surrounding environment that are capable of providing the necessary resources. Surrogates thus need to be discoverable by means of some of the available networking communication protocols. Some technologies (e.g., Bluetooth) permit limiting the discoverability of the device in order to avoid misuses of the device’s resources. In the context of this chapter, we assume that devices are discoverable through the used networking technology. However being discoverable does not mean that the surrogate is willing to share its resources. A surrogate willing to make some of its resources available to others needs to advertise its availability and capabilities as well as provide means to be accessed.

Several middleware architectures and protocols for service discovery exist [66, 49]. These can be employed to support surrogate discovery as well. In static environments, cyber foraging can employ centralized discovery protocols designed for fixed local area networks where the number of participants is limited and the devices are relatively static. Examples of these types of protocols include Salutation\(^1\) and Jini\(^2\). Surrogates can register with the protocol registry by specifying the offered resources. A client can then contact the service discovery server and submit its request for surrogates. The service discovery server matches the client’s request against the registered surrogates and, in case

\(^1\)http://www.salutation.org/
\(^2\)http://www.jini.org/
of many available surrogates, the best matching surrogate is selected based on surrogate-specific attributes. The surrogate’s address together with other qualifying properties are then returned to the client that can therefore contact and use its resources. A good example of centralized discovery model is the Web services approach. Web services use the Universal Description, Discovery and Integration (UDDI) standard\(^3\) for registering and discovering services.

A large part of the solutions proposed for fixed networks rely on centralized registries. This makes the surrogate discovery process dependent on the availability of the central directory, which constitutes a bottleneck. To achieve larger scalability, distributed approaches such as VIA [11] permits sharing data among several discovery domains. Decentralized solutions such as Universal Plug and Play (UPnP)\(^4\) and Service Location Protocol (SLP) [28] represent a better fit for dynamic environments such as mobile ad hoc networks, where mobility and failures are common and it is therefore not possible to rely on any centralized server. Some proposed solutions are Bluetooth SDP [9], GSD [12], Konark [29], and SSD [71]. Peer-to-peer technologies have also been used for distributed resource discovery. Hoschek has proposed a peer-to-peer based approach for distributed databases that was applied to his distributed Web Service Discovery Architecture [30, 31]. Another Distributed Web Service Discovery Architecture is presented by Sapkota et al. in [72].

There exist also hybrid solutions that integrate both infrastructure-based and infrastructure-less approaches. PeerHood [60] combines different networking technologies (e.g., Bluetooth, WLAN, GPRS) and discovery protocols under one interface thus providing a unified view of the available surrogates and offered services. Clients use PeerHood to discover surrogates either in their close proximity (using Bluetooth and WLAN connectivity) or further on in the network (using fixed service registries accessible using GPRS). Clients may also register their own sharable resources to PeerHood thus allowing others to use them.

A first challenge with existing service discovery protocols is interoperability. Even though these protocols share the same basic principles, they all have different origins and employ different technologies. Due to incompatible data representation and communication formats, service discovery protocols do not interoperate with each other. Hence, in general, clients are able to discover only services that are advertised with the protocol(s) they support. Since it is very unlikely that in the future one service discovery protocol will dominate or that device manufactures will offer service discovery technologies on low-cost devices, proxy-based and middleware architectures to enable service discovery interoperability have been proposed [27, 64, 23].

A second challenge to be considered is mobility. In highly dynamic environments, it is hard to constantly maintain up-to-date information on the number and location of surrogates as well as on the type and quality of provided

\(^3\)http://www.oasis-open.org/committees/uddi-spec/
\(^4\)http://www.upnp.org/
resources. Update mechanisms are usually built using a proactive or reactive approach. In the reactive approach, updates are exchanged only when an event occurs, for instance, a surrogate leaves the network. In the proactive approach, update messages are constantly exchanged and a consistent view of the network is maintained. In addition, mobility information can be used to adjust the surrogate advertisement rate and the range of dissemination [12, 63].

A third challenge of the discovery process is how to describe the offered resources. Surrogates need to provide sufficient information describing their resources so that clients can select the most suitable surrogates for their execution. Web services use the Web Services Description Language (WSDL) and Resource Description Framework (RDF) to describe services and resources. Sihvonen presents in his PhD thesis [76] an approach called Personal Service Environment (PSE) that changes its configuration depending on the available resources or capabilities. Capabilities are described with Composite Capability Preference Profile (CC/PP) and RDF. In the field of grid computing, Liny and Raman [47] present the Classad mechanism that allows description of both resource and task requirements. This approach supports the matchmaking procedure needed in the placement decision.

### 3.2 Application Partitioning

To enable the use of cyber foraging an application has to be split into *locally executable code*, such as the application GUI and its model data, and *remotely executable code* that may benefit from remote execution. An application may contain multiple tasks that are good candidates for remote execution depending on the amount of computing, storage, and communication resources they require to be performed. As these resources are relatively constrained on small mobile devices, such as PDAs and mobile phones, the more resource-intensive a task is, the more beneficial the remote execution process will be. On the other hand, as we will explain in the next subsection, cyber foraging implies some overhead, for example in terms of the communication bandwidth required to migrate tasks, transfer control messages, and receive task responses. Therefore, in principle, a task should be executed remotely only when this overhead can be amortized by the gain brought by utilizing cyber foraging.

A very important question to answer is whether the application partitioning should be a manual process performed by the application developer, or whether it is possible to automate such a process. There exist several tradeoffs in deciding between manual and automatic task partitioning. An automatic partitioning system sounds alluring, since it implies less work on the application developer’s side. An example of such a system is Coign [32]. Coign makes partitioning and distribution of tasks possible without altering the source code of the application. Coign works on applications consisting of distributable COM components. It constructs a graph model of the application’s inter-

---

\[5\] http://www.w3.org/TR/wsdl

\[6\] http://www.w3.org/RDF/
component communication where the nodes represent the COM components, the edges represent the communication between such components, and the edge weights represent the amount of communication. Given this graph a graph-cutting algorithm is employed to partition the application by cutting at the edges with the smallest weights. This information paired with a network profile may then be used to make positioning decisions at runtime.

While automated partitioning has its merits, in many cases, manual partitioning can be more effective because, as also noted by Flinn et al. in [22], a little application-specific knowledge can go a long way when preparing an application for distribution. The inclusion of distribution into the original application logic often alters entirely the actual program structure. For instance, it may become more convenient to execute an initially linear program as a number of parallel tasks—thus better utilizing the available surrogates. Such optimizations would be hard to detect for an automated distribution algorithm. Furthermore, an automated partitioning approach will always solve tasks to the same fixed degree. It thus does not offer any variability in the task execution which, as we will discuss shortly, can be of high importance in a cyber foraging setting. Ideally, a cyber foraging framework should cater for both kinds of distribution, falling back to automatic distribution when no instrumented version of an application is available.

3.3 Placement Decision and Cost Assessment

Once an application has been partitioned into locally and remotely executable tasks, the basis for utilizing cyber foraging has been built. The next step is to decide on an execution plan. The execution plan specifies where a given task is to be executed. Execution plans are dynamic in nature since they must take factors such as the current network environment, device resource levels, and surrogate availability into consideration.

For example, the network depicted in Figure 2 shows a number of possible network settings that a mobile device may encounter during its operation. In the left side of the image, there is a corporate network where a number of computers (surrogates) connected to the wired network are offering several types of services and resources. These surrogates may be discovered and accessed by a mobile client through the corporate network infrastructure. In the middle of the image, there is a personal computer in a home network connected to the Internet by an ADSL connection. Its surrogate services may be reached through a direct Bluetooth connection. Finally, on the right side of the image, there is a WLAN hotspot, such as the ones found in many cafés today. Using this WLAN mobile clients can act as surrogates for each other and perform tasks in co-operation. The heterogeneity of the environment depicted in Figure 2 points out many of the factors that the task placement and cost assessment processes need to take into account when selecting an execution plan. Some of these factors are the following:
Network bandwidth and latency between:
- the client and the surrogate(s),
- the surrogate(s) and the Internet, if the remotely executed task requires Internet connectivity, and
- the client and the Internet.

Full capacity and current utilization of resources such as battery, CPU, and storage of:
- the client device, and
- the surrogate(s).

Resources consumed by:
- data transfer over the available wireless links, and
- CPU-intensive tasks.

Task-specific properties:
- Resource demands—either as defined by the author of the task or based on history-driven profiling,
- input and output data/code sizes, and
- adaptability in face of changing resource levels.

These factors interrelate in different ways. For instance, when considering whether to remotely execute a given task, the time and energy necessary for
transferring input and output data between the client and its surrogate must be taken into account. This computation involves task-specific properties (sizes of input and output data), information on the execution environment (current network latency and bandwidth), and device-specific properties (energy consumed by wireless communication). Taking all these factors into consideration is not straightforward. However, there are systems [22, 4] that have made progress into this direction.

In choosing an execution plan, a scheduler needs to decide on what should be optimized: Is the execution time the most important aspect? Or is the energy consumption more relevant? Or should both factors be equally weighted in selecting the most appropriate plan? If cyber foraging is employed primarily to reduce the overall execution time, then the decision of remotely executing a task should be based entirely on the latency estimation. If, instead, reducing energy consumption has higher priority it may be necessary to remotely execute some tasks at the cost of a higher latency.

3.4 Security and Trust

Security and trust are essential requirements to enable sharing of resources in cyber foraging scenarios. When a client’s task is remotely executed on a surrogate, the task code needs to be protected from a malicious surrogate and the surrogate needs to be protected from a malicious client’s tasks. More specifically, a surrogate needs to prevent a malicious client’s task from making an excessive use of its resources. For example, a malicious task may exhaust the energy of the hosting surrogate or corrupt some of its data, launch an attack on the device, or infect other tasks running on the surrogate. A potential solution to this problem is to perform admission control of incoming tasks and require the migrating tasks to specify upper bounds on the amount of necessary resources.

On the other hand, a malicious surrogate may alter the data of a migrating task and compromise its correct execution. Or it may even alter the task’s code and propagate viruses to other surrogates and to the client itself. This can be solved by transferring the task’s code to surrogates in an encrypted form. In this case, each client and surrogate will need to carry a pair of public/private keys [25]. Another security threat is represented by the situation in which a malicious surrogate may attract client’s tasks by pretending to possess false resources (i.e., fake surrogates). In a distributed setting, this issue can be partly solved by using redundancy—when operating in an untrusted environment the same task could be placed on multiple surrogates. Otherwise, if available, a trusted third party can certify that surrogates claim to have resources that they actually own. Redundancy could also be employed to hamper the damages possibly caused by hostile surrogates. The same task is executed on more than one surrogate. When the results are collected, if a surrogate returns a result different from the results provided by the majority of the surrogates, this surrogate is deemed as hostile and will not be used in any further execution.
This approach is employed in the Slingshot system [77].

Integrating support for trust in a cyber foraging system is generally a complex task. If a centralized authority is available, this can authenticate clients and surrogates, monitor the environment, and detect malicious entities. If a centralized authority is not present or simply too expensive to be maintained, as in the case of mobile ad hoc networks, an entity can establish trustworthy relationships by relying on its direct experiences with the same entities as well as by relying on others’ recommendations. A recommendation is generally defined as “the perception that a node creates through past actions about its intentions and norms” [52]. To isolate malicious entities, reputation mechanisms such as [51, 46, 79] can be used.

The issues met when considering security and trust in a cyber foraging setting vary depending on the code distribution approach that is used. This is closely related to the execution strategy adopted by the cyber foraging system, and is therefore discussed in the following subsection.

### 3.5 Task Execution

Once it has been decided which surrogates are available (surrogate discovery), which tasks can be executed remotely (application partitioning), and what cost is associated with every possible distribution strategy (cost assessment), it is the responsibility of the execution phase to decide on 1) how the tasks should be physically allocated to surrogates, 2) how the communication with surrogates should be implemented, and 3) how the task executions on multiple surrogates should be co-ordinated.

In the distributed computing field, Corba and Java RMI offer a means for supporting remote execution in distributed computing environments. Corba uses object references to hide the location of the executing service. Java RMI implements methods for remote procedure calls and thus enables the use of distributed resources. Both approaches present some weaknesses when applied to mobile environments such as failure detection, high latency, and communication overhead. Another approach to distributed computing is offered by cluster computing and grid computing. Both originating from the parallel computing field, they are solely based on the use of shared resources. Although earlier approaches exist, Beowulf\(^7\) clustering and the SETI@home\(^8\) applications started an era in which extra-computing resources are efficiently used for common purposes. However, in these distributed computing approaches the discovery of resources is usually based on static structures, e.g. host files, centralized registers, resource brokers, and portals. The use of static structures hinders their applicability to a mobile environment, even though current trends in grid computing have started focusing also on more dynamic behavi-

---

\(^7\)http://www.beowulf.org/
\(^8\)http://setiathome.berkeley.edu/
ors by looking at the dynamic service creation and discovery issues\(^9\).

In the mobile context, the selection of the best mechanism for physically allocating tasks over the available surrogates depends on the execution environment. If it is the case of mobile devices that perform cyber foraging in known environments, such as within a corporate network, tasks may be pre-installed on fixed surrogates and clients can then use simple RPC to migrate tasks. If mobile devices are performing cyber foraging in unknown, and thus not previously configured, environments a solution based on mobile code where task code is pushed onto surrogates for execution may be more feasible.

The advantages of using pre-installed tasks in a trusted environment are obvious: 1) there is a smaller overhead involved in performing tasks remotely, 2) surrogates can trust the code that they are executing since only their administrators may install tasks, and 3) clients can trust the results returned by surrogates. The disadvantage of such a scheme is the lack of flexibility and in particular the poor support for mobility—a mobile device using this kind of cyber foraging is bound to a specific physical location, and when this known environment is left behind all tasks must be performed locally. Approaches based on mobile code require surrogates to only offer a generic cyber foraging service to clients. Clients can use this service to directly install their tasks and remove them once the execution is completed. However, by using mobile code in unknown environments some serious trust and security issues arise—issues that are well-known within the field of mobile agents, see e.g., [13]. A more in-depth discussion of the different execution possibilities can be found in Section 4.

3.6 Environment Monitoring and Application Adaptability

Pervasive computing environments present a variable level of resource availability [73], such as computing power, storage capabilities, and network bandwidth. Applications need to exploit computing, storage, and communication opportunities whenever available, and must be able to survive when such resources are not available anymore. In addition, with portable wireless devices, disconnections and device failures have to be treated as part of normal operation. Disconnections can occur either accidentally due to loss of wireless connectivity or voluntarily to save battery or reduce connection costs. Wireless communication can also be temporarily degraded due to signal interference and thus cause packet losses, variable bandwidth, and high error rates.

Therefore, applications should constantly monitor their execution environment while consuming almost no power [18]. Every time the execution environment changes, the change must be detected and if it is permanent enough to trigger a reconfiguration, then the behavior of the application must change accordingly. Adapting the application’s fidelity to fluctuating resource levels has been shown to be effective in coping with such dynamism [55, 22, 15, 6, 5].

\(^9\)http://www.akogrimo.org/
Fidelity defines the degree to which service results returned to the application matches the expected service quality. For instance, fidelity can be temporarily degraded to the minimal level acceptable by the user in order to minimize the resource consumption in a resource-poor environment.

How the application fidelity can be varied strongly depends on the data the application manages. For instance, in the case of an application performing speech recognition, when no surrogates are available the lowest level of fidelity is provided: the application simply stores the speech audio files for later processing. When a “weak” surrogate is available a medium level of fidelity is provided: the application uses a “1.5-way” task-directed speech recognition where the speaker adheres to a certain grammar and only utterances within that grammar can be recognized. Finally, when “powerful” surrogates are available the highest level of fidelity is provided using a more resource-consuming approach: the application uses a “2-way” general speech recognition where the recognizer tries to recognize every single word without assuming any grammar.

In addition, learning about the resource usage of an application, predicting resource impoverishment, and anticipating changes of the application’s requirements are all important challenges to fully accomplish efficient reconfiguration.

3.7 Summary

In this section, we have illustrated the steps that a system needs to implement in order to successfully perform cyber foraging. Table 1 summarizes these steps and includes a description of their main characteristics.

4 Cyber Foraging Approaches

Several approaches have been proposed to accomplish cyber foraging in mobile computing environments. In this section we review three well-known mechanisms that have been used in this domain and present the corresponding implemented systems that integrate such approaches. As depicted in Figure 3, cyber foraging mechanisms can be based on remote procedure call (RPC), virtual machine (VM) techniques, or mobile code.

RPC-based approaches assume the environment to be pre-configured. RPC functions are pre-installed on available surrogates such that mobile clients entering the computing environment can remotely invoke functions offered by the surrogates. Due to the need to pre-configure the environment, RPC-based approaches do not provide high flexibility and therefore suit only static environments. VM-based approaches can provide higher flexibility by allowing clients and surrogates to install arbitrary code on the surrogate machines. Approaches based on mobile code provides the highest flexibility. The applica-
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| Surrogate discovery  | • Surrogates must be discoverable by means of some wireless network technology.  
  • Services and resources offered by each surrogate device must be described and advertised.                                                |
| Application partitioning | • Applications have to be partitioned into locally executable tasks and remotely executable tasks.  
  • Both automatic and manual partitioning schemes may be considered.                                                                                     |
| Cost assessment       | • Before placing tasks on external surrogates the cost of using remote execution must be assessed.  
  • A placement decision aims to minimize an application-specific cost function assessing power, communication, and processing resources consumed by cyber foraging.  
  • Local execution should always be a possibility so that the application is not dependent on the availability of surrogates. |
| Trust establishment  | • Clients must be able to trust surrogates and vice versa.  
  • Clients must be sure that a surrogate does not: 1) alter the code or state of their tasks, 2) return false results, 3) access private data, and 4) pretend to own resources that it does not own.  
  • Surrogates must be sure that clients do not: 1) perform DoS attacks on the surrogate, 2) use the surrogate to launch attacks on other peers, 3) access private data stored on the surrogate, 4) modify the code or state of other tasks currently performed on the surrogate. |
| Task execution        | • It must be decided how tasks are physically allocated to surrogates (pre-installed RPCs, mobile code, virtual machines).  
  • A means for communicating between clients and surrogates must be chosen (Java RMI, Corba, etc.)  
  • Optionally, the possibility of utilizing multiple surrogates in parallel – for improving the execution’s performance or security reasons – should be considered too. |
| Environment monitoring | • Monitoring of the execution environment is necessary to dynamically adapt to changes in the level of resource availability.  
  • Several resource parameters such as CPU utilization, power consumption, and network latency/bandwidth need to be monitored.  
  • The surrogate availability must also be monitored.                                                                                     |

Table 1 Summary of challenges posed when designing a cyber foraging system
latency, computing resources, reliability, etc. On the other hand, the overhead for migrating and installing the task on the surrogate node can affect the application’s responsiveness depending on the system implementation.

4.1 RPC-based

Two examples of RPC-based cyber foraging approaches are Spectra [4, 22] and Chroma [5]. In these systems client applications are partitioned into locally executable code and a number of remotely executable tasks. These tasks are pre-installed on surrogate computers that then offer RPC services for invocation. Apart from requiring the pre-installation of tasks a shared Coda [74] file system is also used to exchange data between clients and surrogates. This means that Spectra and Chroma only function in preconfigured environments that support specific functions required by each applications.

In Spectra both mobile clients and surrogates are running Spectra servers along with some resource monitors. When a client is within the range of a surrogate it decides, based on both the monitored resource levels of the current environment and the execution history of the tasks, where to place a task; i.e., whether to locally execute the task or invoke the corresponding RPC function at the surrogate. Spectra uses the Coda file system for exchanging data between
clients and surrogates. Input and output files of the RPC functions are placed in this shared file system.

Chroma is an advanced version of Spectra. Chroma seeks to support computationally intensive interactive applications, such as speech recognition, natural language translation, and augmented-reality applications, which can operate at multiple fidelities. Since these interactive applications generally require minimal user distraction, users specify in advance their high-level preferences and then the system autonomously decides at runtime how to execute applications. Chroma has been designed to be effective in applying different cyber foraging strategies depending on the resource conditions and the level of fidelity to be guaranteed.

Chroma particularly targets the problem of application repartitioning in dynamic mobile computing environments. It makes use of tactics that are high-level declarative specifications of application-specific knowledge to determine optimal application partitioning. Each tactic describes one way of combining RPCs to execute a certain task. In selecting the best tactic, Chroma measures the available resources and selects the tactic that maximizes certain user-specific utility functions. Utility functions express the user’s preferences in one or more fidelity attributes.

PeerHood is another example of this type of an approach. When a PeerHood-enabled device enters a new environment it proactively scans the neighborhood and looks for PeerHood-enabled surrogates. PeerHood-enabled surrogates advertise their services and thus clients can find out about the available RPC service functions. Both mobile devices and ordinary servers may behave as PeerHood-enabled surrogates. Kallonen et al. present in [35] an image processing application on top of the PeerHood middleware. The surrogate carries out image processing and returns the results to the mobile device. PeerHood also provides a remote monitoring service that allows the transfer of monitoring actions from one surrogate to another surrogate. In this way, a mobile client can be constantly informed about changes occurring in its vicinity while saving battery.

4.2 VM-based

Where the RPC-based approaches presented required fully prepared environments, both with regards to individual application support and the use of a shared file system, VM-based approaches provide increased flexibility. The usage of virtual machines makes it possible to allow users to “install” their own functionality on the surrogates on-demand. Apart from the flexibility gained by utilizing virtual machines VM-based approaches also allow clients to fairly share resources of surrogate machines by using the load balancing functionality of virtual machine managers. In addition, they enable easy clean-up once the client’s execution has completed as all execution has taken place within an easily replaceable virtual machine image. Two systems using VM-based mechanisms to support cyber foraging are Slingshot [77] and the system described
by Goyal and Carter in [25].

Slingshot replicates services on surrogate machines located at hotspots. A primary replica of each service runs on a remote server owned by the mobile user. Secondary replicas are instantiated on surrogates at or near the hotspot where the user is currently located. A proxy running on the resource-constrained mobile device broadcasts each service request to all replicas. The first received response is passed to the application. The advantage is that secondary replicas located at the user’s hotspot can improve the response time while the primary replica serves as a stable repository for the application state in case of surrogate crashes. Each replica runs within its own VM, which encapsulates all-application specific state such as a guest OS, shared libraries, executables, and data files. The surrogate machines simply consist of the host OS (i.e., Linux), the VM monitor (i.e., VMware), and Slingshot.

In the infrastructure proposed by Goyal and Carter [25], surrogate managers maintain root partition images for each available OS. In response to a client request, the manager initializes a pre-allocated root partition with the appropriate root image and starts a new virtual server. The IP address of the virtual server is returned to the client. To submit a task to the surrogate, the client sends a request to the virtual server manager running on the server. The request consists of a URL pointing to the program the server is requested to run on the client’s behalf. The server manager can download and install the necessary software packages from such a URL. Once the execution is completed or the client’s allocated time slot expires, the server removes the installed software and restores the original clean state.

4.3 Mobile code

A third approach to supporting cyber foraging is based on mobile code. Due to its flexibility, this approach particularly suits highly mobile environments where clients and surrogates move and surrogates change their resource capabilities over time. This is the case of an ad hoc network of collaborating devices, where nodes dynamically join and leave the environment and fail or voluntarily switch themselves off to save resources.

Context-aware migratory services [69] is a framework designed to support service execution in highly volatile mobile ad hoc networks. A migratory service is a service capable of migrating to different nodes in the network in order to effectively accomplish its task. The service executes on a certain surrogate node as long as it is able to provide semantically correct results to the client; when this is not possible anymore, it migrates through the network until it finds a new surrogate node where the execution can be resumed.

For instance, if we consider the region monitoring scenario previously described (see Section 1), the migratory service monitors the disaster area by executing on a surrogate device available in such a region. When such a device moves away from the region of observation, the service migrates to a new sur-
rogate device currently located in the region of interest. The monitoring service periodically transfers observations of the disaster area to the remote client. There are two main advantages of using migratory services in implementing such a scenario. First, when the current surrogate node becomes unsuitable for hosting the monitoring service, the client does not need to perform any surrogate discovery because the current service can autonomously migrate to a new surrogate that is qualified for accomplishing the current task. Second, the migratory service incorporates all the state information necessary to resume the interaction with the client upon the migration to a different surrogate has completed.

The service migration occurs transparently to the client, and except for a certain delay, no service interruption is perceived by the client. Although a migratory service is physically located on different surrogates over time, it constantly presents a single virtual end-point to the client. Hence, a continuous client-service interaction can be maintained.

The migratory services model incorporates three main mechanisms. The first monitors the dynamism of interacting entities (client and surrogates) by assessing context parameters characterizing their state of execution and available resources. The second specifies, through context rules, how the service execution is influenced and should be modified based on the variations of those context parameters. The third makes the service capable of migrating from one surrogate to another and of resuming its execution once migrated.

To support the migratory services model, the migratory services framework needs to run on each node willing to co-operate in the ad hoc network. This framework was built on top of the Smart Messages [10, 36] distributed computing platform which provides support for execution migration, naming, routing, and security. Smart Messages are similar to mobile agents, which also use migration of code in the network. A mobile agent can be seen as a task that explicitly migrates from node to node. However, mobile agents typically name nodes by fixed addresses and know the network configuration a priori. Instead, Smart Messages are responsible for their own routing at each node in the path between two nodes of interest. This feature makes approaches such as Smart Messages capable of more quickly adapting to changes that may occur in the network topology and resource distribution.

Another system based on mobile code is Locusts [38]. In this system mobile code is used to enable higher client mobility. As discussed earlier, using mobile code makes a cyber foraging system more flexible, because it makes it possible for a client to utilize cyber foraging in unknown environments, as long as the cyber foraging framework itself is installed on the surrogates. Using tasks specified as graphs of interconnected, mobile-code based services the Locusts framework is capable of dynamically installing and performing tasks in unprepared environments.

Finally, an important requirement for systems based on mobile code is portability. The more independent the approach is from the underlying hardware
and software platform the more usable it is. On the other hand, guarantee-
ing portability may affect the overall performance of the system. The Smart Messages platform, for instance, was initially implemented by modifying the Java virtual machine running on the surrogate nodes in order to provide efficient migration. Later on, the Portable Smart Messages [65] platform capable of running on unmodified virtual machines was proposed. Performing migration without having access to the VM internals required a lightweight migration approach based on Java bytecode instrumentation. This made the implementation portable, but costlier than the first implementation in terms of execution time.

4.4 Summary

This section introduced three different approaches towards realizing cyber foraging: RPC-based, VM-based, and mobile code based. The advantages and disadvantages of the different approaches are summarized in Table 2.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC-based</td>
<td>+ Small overhead when performing tasks.</td>
<td>∞ Not very mobile – tasks must be pre-installed.</td>
</tr>
<tr>
<td></td>
<td>+ Security and trust is easier to establish.</td>
<td>∞ No obvious support for task migration.</td>
</tr>
<tr>
<td></td>
<td>+ Portable – can be language agnostic.</td>
<td></td>
</tr>
<tr>
<td>VM-based</td>
<td>+ Flexible – no preparation of surrogates needed.</td>
<td>∞ Very heavyweight with regard to initialization.</td>
</tr>
<tr>
<td></td>
<td>+ After initial setup the overhead when performing tasks can be kept small.</td>
<td>∞ Not very mobile because of the high cost of initialization.</td>
</tr>
<tr>
<td></td>
<td>+ Portable – Completely language agnostic.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ Migration of the entire state is trivial.</td>
<td></td>
</tr>
<tr>
<td>Mobile code</td>
<td>+ Very flexible – no preparation of surrogates needed.</td>
<td>∞ Not always portable – typically the mobile code has to be expressed in a specific language.</td>
</tr>
<tr>
<td></td>
<td>+ High mobility – low initialization times means that unprepared surrogates can be utilized almost immediately.</td>
<td>∞ Security issues with regards to using mobile code must be addressed.</td>
</tr>
<tr>
<td></td>
<td>+ Migration of tasks is possible – even autonomous migration if needed (mobile agents).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ Migration can be done quickly – with very little overhead.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Advantages and disadvantages of the different cyber foraging approaches

As Table 2 shows, each approach works well within a specific usage scenario. If, for example, cyber foraging is employed within the confines of a single
organization/home, an RPC-based approach may be a viable solution. This would offer fast access to cyber foraging with the needed security measures already given by the existing computing infrastructure, and the maintenance of surrogates and installation of tasks may be performed by the administrative staff. In a less static setting, where the user of the cyber foraging service must be able to perform cyber foraging in unprepared environments, using a VM-based or mobile code based approach is necessary. In this case the rate of mobility becomes important; if the user is static for long periods of time (hours instead of minutes) a VM-based approach may be viable. The main drawback of a VM-based approach is the very high setup time; when the client initially enters a new area, it will take on the order of minutes to establish a “connection” to a surrogate, and therefore the VM-based approach is mainly useful for low mobility. If, on the other hand, the client is highly mobile, a mobile code approach may be suitable. Using mobile code, the “setup” time between client and surrogate is minimal and migration of running tasks (e.g., a task is moved from one surrogate to another because of client mobility) becomes possible—even in the face of high client mobility.

5 Summary and Outlook

This chapter has introduced cyber foraging and showed how it can be employed to support mobile applications running on resource-constrained mobile devices. The core principle behind cyber foraging is to make opportunistic use of resources and services provided by nearby computing devices, called surrogates. The first part of the chapter has focused on describing the cyber foraging process consisting of six main processes: surrogate discovery, application partitioning, cost assessment, trust establishment, task execution, and environment monitoring. Problems and solutions to accomplish each single step have been described. The second part of the chapter presented three different approaches to supporting cyber foraging and described existing systems that implement such approaches.

There are still numerous challenges that need to be solved to make cyber foraging a widespread technique. First of all, security and trust issues can hinder the applicability of cyber foraging to unknown environments. This has been demonstrated by several systems based on code mobility which did not manage to go beyond small-scale research prototypes. Pre-configuring computing environments to support cyber foraging is a viable solution but lack the flexibility required by mobile users. Another challenge that current research is facing is how to provide reasonable performance in highly volatile environments such as ad hoc networks. In these environments not only nodes are mobile, but they also offer heterogeneous resources and are subject to frequent failures.

Another major challenge that must be considered to make cyber foraging a viable solution outside of research laboratories, is ease of the development
of cyber foraging enabled applications. It is a well-known fact that application developers find it hard to develop parallel programs. Cyber foraging applications are not only parallel but they are also distributed and must work in very unstable environments. Hence adequate frameworks that provide abstractions for distribution, task placement, concurrency, error-correction, etc. are needed. In principle, a developer should only be responsible for defining the tasks that the application would like to have performed and the rest should be handled transparently by the cyber foraging system.
Using Wi-Fi to Save Energy via P2P Remote Execution

Mads Darø Kristensen  
Niels Olof Bouvin

Abstract

Mobile devices are becoming increasingly powerful with regards to processing speed, networking capabilities, storage capacity etc. While these improvements open up new possibilities on handheld devices such as smart phones, the battery remains a limiting factor. Using the powerful CPUs of modern mobile devices depletes the energy resources very rapidly; as does using wireless networking.

Cyber foraging is a pervasive computing technique where small mobile devices offload resource intensive tasks to stronger computers in the vicinity to improve performance and battery life. This comes at the cost of spending energy on wireless networking, and it is commonly assumed that wireless networking is too costly on small mobile devices. We investigate this claim in this paper, and show that it does not always hold, and that energy may indeed be preserved by utilising cyber foraging—even for relatively small tasks.

This paper presents energy measurements of a modern mobile computing device, showing that utilising the relatively large CPU of such a device is very expensive—even more so than using Wi-Fi. Experiments performed with the Scavenger cyber foraging system are presented; a system enabling easy development of lightweight, highly mobile cyber foraging applications. It is shown how the energy usage of performing large tasks can be brought down drastically, and that even for relatively small tasks the total energy usage can be almost halved.

Published as: Kristensen, M. D. and Bouvin, N.O., Using Wi-Fi to Save Energy via P2P Remote Execution. In Sixth International Workshop on Mobile Peer-to-Peer Computing (MP2P’10), Proceedings of Pervasive Computing and Communications Workshops, 2010. Eighth Annual IEEE International Conference on Pervasive Computing (PerCom’10), (Mannheim, Germany, April 2nd, 2010). IEEE.
1 Introduction

Portable computing devices are ubiquitous, almost all Wi-Fi enabled, and constrained by their battery. We present in this paper work on how offloading computation heavy processes to more capable computers through cyber foraging can increase battery life and improve the user experience.

The term cyber foraging was introduced by Satyanarayanan [73] in 2001 and refined by Balan [4] in 2002. It covers the opportunistic use of available computing resources by small mobile devices, and is a technique that allows resource poor mobile devices to offload some of their heavy work to nearby computers over a (wireless) network connection.

There are a number of advantages of using locally available resources in an ad-hoc, P2P fashion compared to offloading to a server on the Internet. If the system is designed to always rely on external computation, it is useless on its own without network access. Similarly, if the remote services are temporarily unavailable, the devices is likewise affected. The device’s bandwidth and latency will be best connected to nearby computers, rather than with machines placed elsewhere on the Internet. Furthermore, utilising local, already available resources will be more energy-efficient, as it involves fewer components.

Consider the following use-case: A tourist is sitting in a café going through the pictures that she has taken earlier in the day. The pictures were taken using the megapixel camera in her smart phone, and she is browsing them to select the good ones that she wishes to upload to her social networking account for her friends and family to see. Prior to uploading them, she edits the pictures—cropping, applying filters—some need sharpening, others red-eye reduction, and yet others may need their brightness/colour/contrast adjusted. All of these operations are applied only on small previews of the photographs on her smart phone, but when she presses the “apply” button, cyber foraging is used to perform the image operations on the actual images. Her mobile device automatically scans its environment, identifying a couple of surrogates provided by the café and some other customers' laptops, and quickly offloads both the image operations and the uploading of the resulting picture to these devices, leaving her phone free for her to use—and, more importantly, leaving her phone’s battery at an acceptable level so that she may use it for the rest of the day. If no surrogates are available at the café, her mobile device will prompt her to choose between performing the operations locally, or postponing the operations until surrogates become available.

The mobile ad-hoc networking research community has long been rightfully concerned with energy-efficiency, and have devised systems and protocols to extend battery-life by minimising and optimising network utilisation, as this plays a major role in the energy consumption of e.g., small, wireless sensors (see e.g., [19, 42]). We take the lessons learned to heart, but our focus in this paper is different: We wish to extend usable battery life on interactive devices, not embedded sensors with expected life expectancies measured in
months or years. Compared to wireless sensors, our devices have powerful processors (yet still dwarfed by ordinary computers), and high-speed network connections (typically 802.11g). As we show in this paper, the CPU associated energy costs outweigh those of the network interface, and we therefore advocate—in the appropriate situation—that offloading as much as possible is a good strategy, even if it entails more network traffic.

The paper is structured as follows: Related work is discussed in Section 2, followed by a short introduction to the Scavenger cyber foraging framework in Section 3. Section 4 presents the experiments performed and discusses the results obtained in these experiments. Finally, Section 5 concludes the paper, and Section 6 gives directions for future work.

2 Related Work

Power saving through offloading on portable computers has been a topic at least since the late nineties, with authors such as Othman [57] and Rudenko [70] studying the consequences of load balancing between battery powered portable devices and mains powered stationary machines. While the wireless bandwidth available at the time (in the Othman study, 9.6–100 kbps) was much less than today, making transferring data a lengthier and costlier proposition, the studies found that offloading was in many cases beneficial both to battery life and the user experience, as tasks completed earlier.

A later, very comprehensive study was carried out by Flinn et al. [21], where the power consumption of several different applications were studied in high detail. The purpose of the Flinn study was to investigate the combination of fidelity adaptation with offloading for the purposes of extending battery life of mobile devices using the Odyssey [55] platform. The study was performed using the PowerScope [20] probing tool, which utilises two computers and a digital multimeter to precisely measure the rate of power consumption.

In comparison, our measurements of the remaining power is much cruder, as we rely on the device in question to self-report. However, as outlined in Section 4, the measurements match the behaviour of the device in a predictable manner, and were thus deemed sufficient for our purposes.

Since the above mentioned studies, mobile technology has progressed considerably, not only in terms of bandwidth and processing power, but also in terms of dynamically throttling or sleeping components to conserve power. This progress in the platforms available does however not render offloading obsolete, as desktop machines have become faster still, while portable devices can conserve power when idling.
3 The Scavenger Cyber Foraging System

Scavenger is a cyber foraging system developed within the Locusts\(^1\) research project at Aarhus University. The aim of the Locusts project is to design and develop highly mobile cyber foraging systems, and in doing so, creating a platform for experiments with distributed, mobile computing. The main foci in this respect have been on experimenting with task placement and scheduling, and on providing the easiest possible development model for cyber foraging enabled applications.

A high-level view of Scavenger’s architecture is shown in Figure 1. All that is needed for a device to serve as a surrogate is to install the Scavenger daemon. In order to use such resources as are available, an application merely needs to include the Scavenger library and adhere to some simple rules. A device may run both software components at the same time, enabling the device to utilise resources at other devices, while simultaneously sharing its own resources for cyber foraging.

\(^{1}\)http://www.interactivespaces.net/projects/Locusts/

![Figure 1: A high-level view of Scavenger’s architecture.](image-url)
3.1 The Scavenger Daemon

The daemon consists of a small front-end providing remote access to a mobile code execution environment through RPC entry points. This front-end is also responsible for device discovery through the Presence service. The Presence discovery framework [39] was developed especially for use in a cyber foraging setting, where up-to-date information about available resources is critical. Discovery is used by clients searching for surrogates, and by surrogates collecting information about other nearby surrogates. This information is used when surrogates fetch intermediate results from each other.

Scavenger takes a mobile code approach towards task distribution to achieve the maximum level of mobility. When a client device has a task to perform, it can check whether currently available surrogates have the necessary task code installed within their execution environments, and if not it may install the task dynamically at run-time. Thus, cyber foraging is available wherever there are devices running the Scavenger daemon; there is no need for pre-installation of tasks, which would confine the use of remote resources to pre-defined areas. The mobile code execution environment accepts tasks written in Python\(^2\), and the environment itself is written in a special Python variant called Stackless Python\(^3\). Stackless Python was chosen due to its extremely light-weight threading model, and because it yields full control over the scheduling of these threads, which is needed in an execution environment that is executing untrusted code.

Scavenger’s execution environment is designed to fully utilise multi-core machines. A Scavenger daemon can be configured to use any number of cores, and for each core, a core scheduler is spawned that handles tasks performed on that specific core. Tasks are, in the current implementation, submitted to these core schedulers in a round robin fashion. By letting the number of cores used by Scavenger be a configuration parameter, we enable a user to limit the amount of resources to dedicate to Scavenger. Thus a laptop with two cores can be contributing its services as a surrogate, and at the same time be completely usable by its local user, as only one core is used by Scavenger.

Working with mobile code has a lot of implications to security—especially in a cyber foraging scenario, where untrusted clients must be allowed to install and run their own code. A number of measures have been taken in Scavenger to secure the execution environment ensuring that malicious clients cannot access the private data of a surrogate computer, or use a surrogate to launch attacks on other computers on the network. This is outside the scope of this paper—for more in-depth information about the mobile code environment developed for Scavenger, see [39].

\(^2\)http://python.org
\(^3\)http://www.stackless.com
3.2 Developing Scavenger Applications

The Scavenger library grants applications cyber foraging capabilities. In Scavenger, cyber foraging can be enabled by simply annotating a function with a small Python decorator. The function must be self-contained, i.e., it must contain everything needed to perform its work, including module imports, defined classes etc. An example of a function adhering to this is shown in Figure 2.

```python
@scavenge
def sharpen(image, factor):
    from PIL import ImageEnhance as IE
    factor = 1.0 + factor
    return IE.Sharpness(image).enhance(factor)
```

Figure 2 The source code of a function performing an image sharpening. The scavenge decorator is used to enable cyber foraging for this function.

In this figure a function to perform image sharpening is shown. This function relies on the Python Imaging Library (PIL) which it imports on line 3. Apart from having the import statement in the function body, as opposed to in the header of the source file, this function body is similar to any regular Python function. The only real difference is the use of the scavenge decorator in line 1. The decorator accepts optional arguments, used to inform the scheduler about the expected output size, the task complexity etc., but these have been omitted here for clarity. The scavenge decorator enables use of automatic cyber foraging, whenever the sharpen function is called.

When a function has been decorated, the Scavenger library handles the creation of a mobile code task and scheduling of the same, whenever the function is invoked. When performing this scheduling, Scavenger considers factors such as in- and output size, network bandwidth, and task complexity when deciding where to perform a particular task. Task complexity is, among the above mentioned factors, the most difficult to (automatically) obtain. Scavenger uses an adaptive, dual-profiling scheduler maintaining both peer and task centric profiles for each task. The peer centric profile contains information about the execution time of the task when performed on the specific peer, while the task centric profile contains a task weight that may be used to obtain an estimated running time on any peer. Task complexity is dependent on input size and/or value, and to reflect that, all profiles, both the peer and task centric ones, are two-dimensional. The developer can then designate which input parameters are responsible for varying the complexity of the task, so that profile data may be collected in a way that reflects this relation between input size and/or value and task complexity. The inner workings of the scheduler are of course far more complex than hinted at here, but further details are out of the scope of this paper. For a more in-depth discussion of these aspects of Scavenger see [41].

Scavenger’s scheduler optimises solely towards execution time; i.e., it chooses the execution plan that minimises the total execution time as perceived
by the client. What is shown in this paper is, that even though the focus of the scheduler is on execution time, because of the energy consumption characteristics of the mobile device, this also means that energy is saved—even though the Wi-Fi interface must be active for cyber foraging to work.

## 4 Experiments

The mobile device used in these tests is a Nokia N810 Internet Tablet with a 400 MHz TI OMAP 2420 (ARM1136) processor and 128 MB of DDR RAM. This device has been chosen for a number of reasons; mainly because of its Linux based operating system, which allows for easy access to system information such as e.g., battery status. The N810 is also considered representative of the current generation of smart phones in that it uses the same CPU as many of these smart phones, and, apart from its rather large display and lack of any cellular networking, its specifications are similar to those of current smart phones.

The surrogate in use in the benchmarks is a 2008-model MacBook with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of 1067 MHz DDR3 RAM running Mac OS X 10.5.8. The network media connecting the client and the surrogate was a IEEE 802.11g network served by a Linksys WRT54G router. Unless stated otherwise, the network was only in light use while tests were performed.

Before we describe the tests of using Scavenger to offload resource intensive tasks, we must establish a baseline. For this baseline the energy consumption of the N810 has been measured for one hour under differing circumstances. Prior to each test, the battery of the N810 has been fully charged, ensuring that the observed behaviour is not due to characteristics of different regions of the battery. For this test the energy consumption of the N810 is measured in five different settings, the first four being the combinations of having the Wi-Fi interface turned on or off with either an idle or an 100% utilised CPU. In the fifth test, the Wi-Fi interface is turned on and a large amount of data is transferred over the interface during the test. In all tests the LCD display was turned on but dimmed down to the lowest brightness setting. The results of these five tests are shown in Figure 3.

The results of these baseline tests show a number of interesting things. It can be seen that having the Wi-Fi interface turned on does consume a considerable amount of energy. When the system is idle for one hour with the Wi-Fi interface turned off, it consumes 84 mAh, yielding a battery lifetime of 18 hours, but when the Wi-Fi interface is turned on, the energy usage increases by 46% to 123 mAh, bringing the battery lifetime down to around 12 hours. It should be noted that, in order to keep the Wi-Fi interface turned on, a very small amount of traffic was generated by having the N810 continuously ping the router. When a large amount of data is transferred the cost of using Wi-Fi is even greater. In the test where network traffic was generated, the N810 would alternate between sending and receiving data at a rate of approximately 800 kb/s. Sending and receiving data at this rate increased the energy consump-
tion by 150% compared to having the Wi-Fi interface turned off, and with 70% compared to simply having Wi-Fi turned on.

It is thus clear that using the Wi-Fi interface is a costly affair on a mobile device. But, as the measurements also show, fully utilising the CPU is even more expensive. The two tests where the CPU was 100% utilised have the worst performance energy-wise; i.e., fully utilising the CPU while Wi-Fi is off uses more energy than even heavy use of Wi-Fi. Comparing e.g., the cases with no Wi-Fi and an idle CPU vs. a fully utilised CPU, using the CPU increased the energy consumption by 180%. This tells us, that there is a basis for obtaining energy savings through the use of Wi-Fi reliant cyber foraging.

We have within the Locusts project developed a demonstrator application that realises the use-case presented in the introduction of this paper. The tasks of this application have been used in the following two experiments. The first experiment simulates that the user browses her images, selects an image for editing, previews three image operations on a 0.3 megapixel (MP) preview version of the image, and finally commits to these changes by applying them to the original 5 MP image. This is done for 15 images in each test run. The image operations performed are sharpening, brightness, and contrast adjustment, and all of these are targets for remote execution. All experiments have been performed six times, and the values reported here are the averages of these runs. The results of this experiment are listed in Table 1.

Performing such large tasks, i.e., working on the 5 MP images, is very resource intensive for the mobile device, while the surrogate can apply such operations in mere seconds. It thus comes as no surprise that the running time of the tests is brought down substantially by utilising cyber foraging. When
Experiments

Table 1 Results of the first experiment. Here three image operations are applied to 15 image previews and then to the original images.

<table>
<thead>
<tr>
<th>Description</th>
<th>Time $\sigma$ seconds</th>
<th>Energy $\sigma$ mAh</th>
<th>Energy/Time $\sigma$ mAh/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>No surrogates, Wi-Fi on</td>
<td>2868 90</td>
<td>206 18</td>
<td>0.0719</td>
</tr>
<tr>
<td>No surrogates, Wi-Fi off</td>
<td>2928 142</td>
<td>187 22</td>
<td>0.0642</td>
</tr>
<tr>
<td>Surrogate available</td>
<td>579 26</td>
<td>26 2</td>
<td>0.0452</td>
</tr>
</tbody>
</table>

Looking at the results in Table 2, a number of interesting things become apparent. The first thing to notice is that once again the total running time has been brought down by using remote execution, which has also reduced the energy consumption. In this experiment however, the running time reduction is slight, because only a small fraction of the time is used actually performing the CPU intensive tasks—most time is spent idling while waiting for user input. Comparing the test with Wi-Fi on and no surrogates available to the one where a surrogate is available, the running time is brought down 30%, but the energy consumption in the same tests is reduced by 43%. This is also reflected in the fact that the immediate energy consumption, shown in the last column of Table 2, is lower when using a surrogate. The immediate energy consumption is, however, lowest in the test where the Wi-Fi interface is turned off, and the mobile device is thus performing all tasks on its own. The total energy
<table>
<thead>
<tr>
<th>Description</th>
<th>Time σ seconds</th>
<th>Energy σ mAh</th>
<th>Energy/Time σ mAh/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>No surrogates, Wi-Fi on</td>
<td>1642 44</td>
<td>78 9</td>
<td>0.0475</td>
</tr>
<tr>
<td>No surrogates, Wi-Fi off</td>
<td>1666 27</td>
<td>58 2</td>
<td>0.0350</td>
</tr>
<tr>
<td>Surrogate available</td>
<td>1135 54</td>
<td>44 3</td>
<td>0.0390</td>
</tr>
<tr>
<td>Surrogate available, noisy network</td>
<td>1212 58</td>
<td>49 4</td>
<td>0.0401</td>
</tr>
</tbody>
</table>

**Table 2** Results of the second experiment. In these tests three image operations are applied to 75 of the smaller preview images.

consumption is still considerably higher though, using 32% more energy than when using remote execution, so using a surrogate is still preferable.

We have also performed test runs, where the network medium is in use by other computers. In this test the surrogate and another N810 device are continuously communicating, maintaining an approximately 800 kb/s rate of data transfer between them. Introducing this noise naturally made the cyber foraging perform slightly slower, thus using more energy in total. The immediate energy usage also became slightly larger in a noisy environment, but this may properly be accredited to the broadcast nature of Wi-Fi, where data sent between other peers in the network is overheard by all peers. What should be noted with regards to this test is, that the overall energy consumption is still about 17% less than that when disabling Wi-Fi altogether, which is because the total running time is brought down by about 27%.

5 Conclusion

It is commonly known within our field, that Wi-Fi interfaces use large amounts of energy, and as such should be turned off as much a possible in order to preserve precious energy resources. This paper has presented measurements done on a modern mobile device, a Nokia N810 Internet Tablet, that supports this assumption—using a Wi-Fi interface is indeed expensive. But, as the measurements also show, using the relatively large CPUs on modern mobile devices is even more resource demanding. Seeing as CPU intensive work is more resource demanding than using Wi-Fi, using remote execution for CPU intensive tasks through a sufficiently lightweight cyber foraging system, such as the Scavenger system presented here, yields considerable energy savings.

This paper has presented experiments done using a cyber foraging enabled mobile image editor, and it has been shown how the energy usage of performing image operations, on even quite small images, can be reduced considerably,
leading to a longer battery lifetime for the mobile device. Energy savings of 86% for very large tasks and 24% for relatively small tasks have been obtained in the experiments.

These results suggest, that keeping wireless interfaces turned off whenever they are not in direct use, may not yield the best energy efficiency after all. In order to maximise the battery lifetime of a mobile device, Wi-Fi should be turned off whenever it is not in use, but, as has been shown in this paper, it may make sense to turn it on before doing CPU intensive tasks, if turning it on means that cyber foraging can be used to offload some of the resource demanding work.

6 Future Work

The experiments presented here have all been performed on a single mobile device. In the near future we will be replicating the experiments on other mobile platforms, such as the Apple iPhone and the Nokia N900 smart phone.

The results obtained in this paper point to that it sometimes makes sense to choose remote execution even in the case where this will yield a longer running time. It would thus be interesting to experiment with adding energy profiling to the scheduling process of Scavenger in order to experiment with scheduling towards energy efficiency rather than running time.

Acknowledgements

This paper has been funded by a research grant from the Danish Research Council for Technology and Production Sciences. The authors would like to thank Nokia for providing us with mobile hardware. Furthermore, the authors would like to thank the anonymous reviewers for their insightful comments.
Developing Cyber Foraging Applications for Portable Devices

Mads Darø Kristensen    Niels Olof Bouvin

Abstract

This paper presents the Locusts cyber foraging framework. Cyber foraging is the opportunistic use of computing resources available in the nearby environment, and using such resources thus fall into the category of distributed computing. Furthermore, for the resources to be used efficiently, parallel computing techniques must also be employed. Distributed and parallel computing are two concepts that are both notoriously known for being very hard for developers to grasp. Because of this one might think that techniques such as cyber foraging would have a hard time surviving outside of research environments. In this paper a framework is presented that has special focus on making cyber foraging accessible for all developers.

1 Introduction

Mobile computing devices, such as the current line of high-end mobile phones, are rapidly advancing to become the personal computing device of the future. Besides supporting phone calls, this new line of “smart” phones provide display, input, communication, storage, and processing capabilities that rival modern PDAs. However, while these devices are relatively powerful, mobile

devices will always be constrained in terms of physical size—which in turn leads to constraints in energy capacity and processing power.

One defining thing about these new mobile computing devices is that they all sport high bandwidth, wireless network interfaces, and are thus capable of communicating with other devices in their vicinity. This local network connectivity can be used by the mobile devices to alleviate their constraints. By opportunistically utilising nearby computing resources when available, through techniques such as remote execution, the mobile devices may 1) perform computing tasks faster, 2) perform tasks that would normally not be feasible to perform on small devices, and 3) preserve precious energy. This opportunistic use of available computing resources on surrogate computers is called cyber foraging [73, 4], and is defined as “living off the land”.

For an example of a mobile application that could benefit from cyber foraging see e.g., [62], or consider the following use-case:

A tourist is sitting in his hotel room relaxing after a long day of sightseeing. Using the high quality camera in his smartphone he has taken a large number of pictures during the day, and now he is going through them to find the ones that he would like to keep. When he finds a picture that he likes he applies a couple of filters/effects to the picture until it fits his liking. Whenever he applies a filter his picture browsing application checks the local network, and if stronger machines are available the filtering process is forwarded to one of these surrogate machines. If no surrogates are available the filtering is done by the local device—which takes a long time on such small devices. When he is satisfied with the picture he marks it for keeping, which means that it will be sent to his online storage account, and again this operation will be performed by a surrogate machine if available. When he has gone through all of the pictures and thus finishes the reviewing process the pictures are deleted from the phone and he is ready for another day of sightseeing.

The realisation of cyber foraging in mobile applications, as described above, is no trivial task. The process of cyber foraging entails, but is not restricted to, the following five inter-related steps 1) peer (surrogate) and service discovery, 2) application partitioning into local and remotely executable code, 3) intelligent scheduling of tasks to make sure that tasks are only executed remotely when it is beneficial, 4) monitoring of the local environment (network links and computing machinery) to inform the scheduling process, and 5) establishment of security and trust.

An application developer should not need to be concerned with all of these intricate details of the cyber foraging process—otherwise no cyber foraging enabled applications would ever be developed outside of research environments. We present in this paper the Locusts framework, which enables developers to easily create cyber foraging applications. The paper is structured as follows:
We present the Locusts framework in Section 2, describe how applications are developed in Section 3, describe an demonstration application in Section 4, present related work in Section 5 and the paper is concluded in Section 6.

2 The Locusts Framework

The focus of this paper is on application development using Locusts and the internal complexities of the framework will therefore not be described in great detail. The architecture of Locusts is described in Section 2.1, and Section 2.2 introduces Locusts’ notion of tasks and services.

2.1 Architecture

As outlined in the previous sections, cyber foraging relies on opportunistically offloading resource intensive work to nearby, more powerful machines (known as surrogates). If no surrogates are available, the work must be handled locally. Thus, Locusts must accommodate defining work to be done, identifying potential surrogates, invoking tasks on remote machines (which of course includes moving input and output back and forth), if possible and else performing them locally, as well as executing tasks submitted from other machines. Locusts has been designed in a component-based manner to handle all these tasks in a manner, that is easily maintainable as well as extensible.
The overall architecture can be seen in Figure 1. All participating machines run an instance of the Locusts daemon, so they can all share the associated responsibilities.

The cornerstone of cyber foraging is discovering available services, and this is in Locusts handled by the Presence daemon. This daemon utilises UDP broadcasts over WiFi to quickly detect nearby surrogates. The discoveries made by the Presence daemon are passed on to the Locust Daemon, which handles all core cyber foraging tasks: It keeps track of accessible surrogates, assess their suitability and services, they offer; it offloads jobs to other Locusts instances; it performs jobs on behalf of local clients as well as other Locusts instances; and it monitors its own available resource levels in order to inform other Locusts instances of its suitability as a surrogate.

The central actor in this is the Scheduler, which, based on available information, decides whether to do jobs locally or remotely. Regardless of where jobs are to be performed, the Scheduler creates an execution plan, which describes the order of things to be done, and these are then performed locally or remotely by the Execution Environments.

The application developer need not worry about the intricacies of the Locusts architecture—as described in Section 3, a developer can invoke tasks (using the API described in Section 3.1) and expect them to be executed without having to bother with where and how. The Locust Daemon communicates using XML-RPC, which allows for client applications to be written in any language. The AugIM application described in Section 4 is written in Python, but any language platform supporting XML-RPC would be suitable for a Locusts application.

2.2 Tasks and Services

If a client application developer is to invoke jobs to be executed remotely, the jobs must first be defined and implemented. Jobs in Locusts are defined as tasks—which are directed graphs of services. Services are single, small operations that can be performed quickly, and combined into tasks. Examples of services might be an image manipulation operation, or uploading a file to a server.

Services are advertised by the Locusts instances. Given a palette of available services, a task can be arranged, connecting the output and input of the various services together in a graph, an example of which can be seen in Figure 2.

Our approach of modelling tasks as directed graphs of services affords us some advantages. It allows Locusts to start parallel service invocations to several surrogates at once (if more are available) for increased performance, and as the task graph itself can be exported to other surrogates (quite likely to the surrogate, that will perform most of the computations), the client will only have to transmit the initial task input and later receive the final output, rather than
having to send intermediate results back and forth. This increases performance dramatically, especially in bandwidth starved environments. Task graphs also fit well within a mobile cyber foraging scenario: If a task must be aborted, because the client is moving out of reach, we can checkpoint the task by collecting the generated input and noting the associated vertexes, Later, it is trivial to resume and complete the task. This is much simpler than having to develop services that can be check-pointed arbitrarily, and thus puts less of an onus on the developer.

3 Developing Applications Using Locusts

As described above, Locusts works with the dual concepts of tasks and services. An application developer need to fully understand these concepts, as they are the fundament of the framework. The process of developing an application that utilises Locusts consists of the following steps:

- Identify the operations within the application that may benefit from remote execution and implement them as Locusts services. See Section 3.3.

- Using the services defined in the preceding step, and possibly some of the built-in services already available in Locusts, define tasks that can be handed over to surrogates. This may be done beforehand or it may be done dynamically on run-time. See Section 3.2.

- Of course, to be able to use the services and tasks defined the application needs to include and work with the Locusts API. See Section 3.1.

3.1 Working with the Locusts API

An application needs to import and work with the Locusts API to use the services offered by the Locusts daemon. Working with the Locusts daemon is inherently asynchronous. When a task is shipped to the daemon, the API registers a callback listener that receives all information about the task; be it status
messages, error indications, or returned output. This asynchronous behaviour can be hidden from the developer, if desired. Many developers find it easier to develop using blocking (synchronous) calls and Locusts supports this while still letting more experienced developers use the more powerful asynchronous operations.

See Code Listing 1 for an example of a simple, synchronous use of the Locusts API. Perusing the code, we find the following: The first line imports the

```python
1 from Locusts import Locusts, load_task
2 daemon = Locusts()
3 task = Locusts.load_task('mytask')
4 try:
5    result = task.perform(42, 'foo')
6 except Locusts.LocustsException, e:
7    print 'Daemon failed to perform task.'
8 except Locusts.ServiceException, e:
9    print 'A service has raised an exception.'
```

Code 1: A sample synchronous task being executed using the Locusts daemon.

Locusts class, which represents a connection to the Locusts daemon, and a function called `load_task`, which is used to load pre-defined tasks from files. In the second line, a Locusts daemon object is created. The third line creates a `Task` object by loading the task definition from a file. In line five, the task is performed and the result is stored in `result`—unless of course an error occurs. If an error occurs within the daemon, a `LocustsException` is raised, and if an error occurs within one of the services within the task a `ServiceException` is thrown.

Thus, the synchronous approach to task execution in Locusts is quite simple. The more powerful asynchronous approach can be handled through the use of callbacks. To receive callbacks, a listener object must be provided that contains the methods `status`, `error`, and `done`. These methods all take a single argument which for `status` and `error` are a descriptive string, as for done the argument is the result of performing the task, i.e., the task output. The class `ServiceListener` implements these methods, so that it may be sub-classed when only one or two of the methods are needed. An example performing the same task as in Code Listing 1 is shown in Code Listing 2.

### 3.2 Defining Tasks

In the preceding section, it was shown how simple it is to work with tasks within Locusts. What was left out of that description was how the tasks themselves are defined. As described in Section 2, a Locusts task is a composition of services; or, more precisely, a directed graph of interconnected services. This graph is currently described using an XML representation of both tasks and services. The document type declaration (DTD) of tasks is shown in Code Listing 3.
from Locusts import Locusts, load_task
demon = Locusts()
task = Locusts.load_task('mytask')

class MyListener(ServiceListener):
    def __init__(self):
        ServiceListener.__init__(self)
    def done(result):
        print "The result was", result

task.perform_async(MyListener(), 42, 'foo')

Code 2: An asynchronous execution of a Locusts task.

As can be seen in the DTD, a Locusts task consists of one or more interconnected services and one output definition. For each use of a service, the input parameters of the service are bound to either input parameters of the task, or output parameters of other services in the task. In the case, where a parameter is bound to a task input value, the class of the value is input, and if bound to an output from another service, the class is result and the attribute sourceId pinpoints the service whose output is referenced. For both task input and service results, it is so that if multiple values exist, i.e., if the task has multiple input or the service has multiple output, the specific value needed can be pinpointed with the valueId attribute. The output of the task is defined within the output section, where values are referenced in the same way as within services. A simple example task that sharpens and rotates an image is shown in Code Listing 4.

3.3 Creating Services

Tasks, as described in Section 3.2, require a number of available services. These services may be pre-installed services, that are a part of the Locusts framework, but, to achieve maximum flexibility with regards to mobility and usage in un-
known environments, Locusts also caters for a mobile code approach towards services. Besides the code of the service a service description is needed. The DTD for service descriptions is shown in Code Listing 5 and an example service using this DTD can be seen in Code Listing 6.

Services must, in the current implementation, be implemented in the Python programming language. For Locusts to be able to perform the services they must adhere to a specific interface, but, to keep the process of creating Locusts services as simple as possible, this interface has been kept to a minimum. The only thing that a service must implement is a function called perform that must be placed at the top-level of the module describing the service. This function can take any number of inputs, but the names of the input parameters must correspond to the names specified in the service description.

When the function is done, i.e., when it is ready to return an output, it may do so in the regular fashion using the return keyword. If the service has a single
output any picklable (the Python jargon for marshalling) type may be returned. If, on the other hand, the service has multiple outputs these outputs must be packed inside a Python dict (a hash map) having string keys pointing to picklable values. The names of the keys in the output dict must correspond to the output identifiers in the service description. Uncaught exceptions thrown from within the service code are caught by Locusts and passed on to the application as error messages. Using this interface, the rotate service described in Code Listing 6 could be implemented as simply as shown in Code Listing 7. This code assumes that the input image is an Image object from the Python Imaging Library (PIL)—if it is not an exception will be thrown. It may seem odd to use

```
def perform(image, angle):
    return image.rotate(angle)
```

remote execution for a task that consists of a single line of code, but of course the main body of code behind this example lies within PIL.

4 Demonstration

Using the simple API described in the preceding section a small demonstrator application, that supports the use-case presented in the introduction, has been created. AugIM\(^1\), or the Augmented Image Manager, is an image manipulation application, capable of browsing a collection of images and applying a number of simple operations on these images. Using the Locusts cyber foraging framework AugIM tries to relocate the execution of all image handling tasks to stronger surrogates in the vicinity.

The use-case in Section 1 discussed smartphones, and AugIM could well have been implemented on such a device, but we have chosen to develop AugIM on a Nokia N800. The Nokia Internet tablet line with its large touch-screen interfaces, wireless network interfaces (both Wi-Fi and Bluetooth) is very repres-

\(^1\)AugIM and a Windows installable version of the Locusts daemon can be obtained from the following URL http://www.daimi.au.dk/~madsk/augim/.

---

```xml
<service name="daimi.imaging.rotate">
  <input>
    <value id="image" />
    <value id="angle" />
  </input>
  <output>
    <value id="image" />
  </output>
</service>
```

Code 6: A service definition of the rotate service.

Code 7: Implementation of the rotate service.
Entuitive of the next line of smartphones that are arriving; phones such as the Nokia Aeon and Apple’s iPhone. AugIM was implemented in Python using the GTK+ widget set, and it was tested on a Nokia N800 device.

The user selects in AugIM a number of operations to be applied to an image. These operations are previewed on the mobile device by performing them locally on a small preview image. When the user is satisfied with an image and either chooses to save it or send it to an online web-album, the actual, larger image is processed. This means that a Locusts task consisting of a string of image filters is dynamically built while the user is playing around with the preview image, and this task is then performed, possibly on a remote surrogate, when the user is done with the image. A screen shot of AugIM in action can be seen in Figure 3.

![Figure 3 The AugIM application.](image)

Using Locusts in an application such as AugIM was extremely easy. To keep the UI responsive, the heavy task of performing image manipulation was split into worker threads, so all that was needed to employ Locusts was letting these worker threads 1) build task definitions, 2) send the task and the input image to an available surrogate, and 3) use the synchronous perform call to wait for the result to arrive. The services were all quite simple as well—in fact the rotate service shown in code listing 7 is the actual implementation of the rotate operation within AugIM.

The focus of this paper has been on development using Locusts, but the raison d’être of Locusts is for small, mobile devices to achieve better performance—be it with regards to energy usage or processing speed. To show that it is in fact the case, that using Locusts in an application such as AugIM increases performance, a small extract of an extensive benchmark study is presented in Table 1.

The single service performed in the task consisting of only one service is a blur operation. In the four-service task the image is blurred, sharpened, in-
5 Related Work

Cyber foraging is a part of pervasive computing. Whereas pervasive computing at one point seemed like science fiction [78], we are gradually seeing it realised (though not necessarily as originally envisioned) with the ubiquitous access to computing and network outlined above.

There have been a number of cyber foraging systems over the years, such as Spectra by Flinn et al. [22], Chroma by Balan et al. [5], the system by Goyal & Carter [25], the system by Kalapriya et al. [34], Slingshot by Su & Flinn [77], and TJam by Riva et al. [69]. The authors behind these systems have identified a number of issues that are common for cyber foraging systems, many of which are fundamental challenges for mobile computing.

Of these systems only one, Chroma, has, to our knowledge, dealt directly with the problem of simplifying the process of developing cyber foraging enabled applications. The results of their findings have been described by Balan et al. in [6]. While Chroma is based purely on pre-installed RPC functions that clients may call on designated surrogate computers, Locusts has a much more mobile approach, where there is no distinction between client and surrogate, and where mobile code is employed to allow for maximal freedom of movement.

6 Conclusion and Future Work

This paper has presented the Locusts cyber foraging framework and has shown how applications using this framework can be built. The process of developing an application was described in detail, and a demonstrator application, an
augmented image manager, was presented. A selection of benchmarks providing proof of AugIM’s improved performance when using cyber foraging was shown.

Locusts has two main goals: 1) to be easily accessible for developers, and 2) to provide support for highly mobile cyber foraging; for more information about the term highly mobile cyber foraging see [38]. To address the first issue, we will be performing user studies in the near future, where masters students will do projects using Locusts. A feature planned for future releases that address the high mobility of Locusts, is the introduction of a more powerful task definition language; one that, among other things, introduces looping and conditional constructs. Introducing such a language will make it possible to work with even smaller services, which in turn makes task migration more seamless.

Acknowledgement

This paper has been funded by a research grant from the Danish Research Council for Technology and Production Sciences. The authors would also like to thank Nokia for providing us with the N800 devices that we are using for our current prototyping of the system.
Profile Based Scheduling of Compute Intensive Tasks in Mobile Networks

Mads Darø Kristensen       Jari Porras

Abstract

Cyber foraging is a pervasive computing technique where small mobile devices offload some of their more resource intensive tasks to stronger computing machinery in the vicinity. A very important aspect in any cyber foraging system is scheduling or task placement, which is the process of choosing the correct place to perform a given task at any given moment in time, whilst the user of the mobile client device is moving through the environment. This paper presents the Scavenger cyber foraging system, focusing on how the scheduling is done within this system. The problem of scheduling in a highly mobile environment is described, and an adaptive, dual-profiling scheduler, addressing this problem in a completely decentralised way, is presented along with an extensive benchmark study comparing it to regular profile based schedulers.

When working with profile based scheduling, a profile is built for each (task, peer)-pair, collecting information about the performance of the given machine when performing that specific task. Such profiles may only be used to say something about the performance of a task on a specific machine. In highly mobile cyber foraging scenarios, it is likely that a client will only work with a given surrogate machine a handful of times, making such machine-specific profiles less useful. Scavenger therefore introduces a new take on task-specific profiles—profiles that can be used to make an informed guess about the performance of a task on any surrogate machine.

Unpublished draft: Kristensen, M. D. and Porras, J., Profile Based Scheduling of Compute Intensive Tasks in Mobile Networks.
1 Introduction

The use of mobile computing devices is increasing. Today a very large percentage of ordinary consumers are carrying around smart phones, mobile gaming devices, and UMPCs. These mobile devices are becoming increasingly powerful but are still quite resource constrained when it comes to resources such as processing power and energy capacity, rendering them useless when it comes to performing tasks that entail heavy processing. In cases where a mobile device actually has the CPU power to perform a heavy task, the delimiting factor will be the energy capacity, because performing a heavy task, i.e., fully utilising the CPU for a while, uses a lot of energy.

Cyber foraging is a pervasive computing technique that tries to alleviate these deficiencies in mobile devices. The term “cyber foraging” was coined by Satyanarayanan [73] and further defined by Balan et al. [4] and is defined as the opportunistic use of available computing resources by small mobile devices. More precisely, it is a technique that allows small mobile devices to offload some of their resource intensive work to stronger surrogate machines in their vicinity.

Consider the following use-case: A tourist is sitting in a café going through the pictures she has taken earlier in the day. The pictures were taken using the megapixel camera in her smart phone, and she is now browsing them to select the good ones that she wishes to upload to her online storage account for her friends and family to see. Before uploading them she applies some filters to them—some need sharpening, others red-eye reduction, and yet others may need their brightness/colour/contrast adjusted. All of these operations are applied only on small previews of the photographs on her smart phone, but when she presses the “apply” button, cyber foraging is used to perform the image operations on the actual images. Her mobile device automatically scans its environment, finding a couple of surrogates provided by the café and some other customers’ laptops, and quickly offloads both the image operations and the uploading of the resulting pictures to these devices, leaving her phone free for her to use—and, more importantly, leaving her phone’s battery at an acceptable level so that she may use it for the rest of the day. If no surrogates are available at the café, her mobile device will ask her to choose between performing the operations locally, or postponing the operations until surrogates become available.

To implement a cyber foraging system capable of realising the above use-case, a lot of software components must be in place: a discovery mechanism must be in use so that surrogates can be discovered, a scheduler capable of choosing the right surrogate for the task at hand is needed, and, for the system to be truly flexible, a mobile code execution environment is needed, so that the clients may push new functionality to the surrogates. Because of the heterogeneous nature of cyber foraging—client devices are very heterogeneous as are the potential surrogates—the scheduler is an important component. The scheduler must be capable of always selecting the best surrogate for the given task;
considering factors such as relative surrogate strength, client device strength, network speeds and latencies, task complexity, surrogate utilisation levels etc. Some of these factors, such as surrogate and client strength and task complexity, are hard to measure, making the scheduler a very complex piece of software. This paper presents the Scavenger cyber foraging framework, with a special focus on the scheduler employed in that system, showing how all these factors may be taken into consideration when dynamically scheduling tasks in a mobile environment. A novel dual-profiling approach to task scheduling is presented along with experimental results from an actual implementation of the system showing the viability of the approach.

The remainder of this paper is structured as follows: Section 2 discusses what is needed in order to realise a use case such as the one described above. Section 3 presents the Scavenger cyber foraging framework, and in Section 4 the scheduling done in this system is described in detail. In Section 5 experiments using Scavenger are presented and their results discussed, and the paper is concluded in Section 7 with some pointers to future work in Section 8.

2 Realising the Use Case

Developing a cyber foraging enabled application without some system support is a very challenging task. Using cyber foraging entails mobile, distributed, and in some cases parallel, programming—all fields within computer science known to be hard for developers to fully grasp. A framework handling most of these challenges is thus needed if cyber foraging is ever to be a general programming technique, and not merely a research topic.

Any cyber foraging system faces a large number of challenges in order to work in an ever changing, mobile, distributed environment. First of, surrogate computers must be discovered, so that a client device may know where to perform a given task. It is of course not enough to merely know of the existence of a surrogate computer, some additional information such as current utilisation, relative strength, and network bandwidth may also be needed, in order to make an informed decision when scheduling a task.

Once discovery and surrogate monitoring is in place, the system must have a way of assessing the cost of performing any given task at the available surrogates, so that the execution plan yielding the current best performance may be chosen. This is done by the scheduler.

After scheduling has taken place, trust must be established between the client device and the chosen surrogate. The client must be able to trust that the surrogate will perform its task, that it will not share any sensitive data with other parties, and that it will return truthful results. The surrogate, on the other hand, must be able to trust that the client will not be misusing its resources, e.g., to launch attacks on other peers on the network, and that it will not try to access sensitive data stored at the surrogate.
Provided that trust has been established, the client now needs a way to perform the actual task execution on the surrogate. This may be done by invoking pre-installed remote procedure calls (RPC), or it may be done more flexibly by using mobile code, pushing functionality and data to the surrogate for processing.

Finally, for any of this to actually work, the cyber foraging system must define a way to partition applications into local code, run entirely on the mobile device, and remote executable code or tasks.

For a more thorough discussion of the components needed to implement a cyber foraging system see Porras et al. [61].

3 Scavenger

Scavenger is a cyber foraging framework being developed within the Locusts project at Aarhus University. It is written entirely in Python and is freely available as open source at http://code.google.com/p/scavenger-cf/.

Scavenger implements all components needed for a fully fledged cyber foraging system, but the main focus throughout the development of the system has been on task execution, scheduling, and developer support. This section presents Scavenger as a system, giving an overview of its architecture and the development model that it provides.

3.1 Architecture

An coarse grained overview of Scavenger’s architecture can be seen in Figure 1. This figure depicts the major components that Scavenger is comprised of.

In the preceding section, where the components needed to realise a cyber foraging scenario were described, the first components mentioned were peer discovery and monitoring. This is in Scavenger handled by a purpose-built discovery system called Presence. Presence has been designed with highly mobile cyber foraging in mind, and as such has a fairly aggressive, pro-active approach towards service discovery. One does not merely need peer discovery in cyber foraging—after discovering a peer the client needs to monitor that peer, collecting information such as current CPU load and availability, so that this information may be used later on in the scheduling process. Because of this need for continuously updated information about available peers, the most efficient way of doing service announcements in a cyber foraging setting is using simple subnet broadcasts. The Presence daemon thus periodically broadcasts any registered services on the local IP subnet, and the corresponding Presence library collects these broadcasts and builds a context table of available peers. Presence was developed within the Locusts project and has been released as open source software at http://code.google.com/p/presence-discovery/.
To achieve a high level of flexibility a mobile code approach is needed, so that mobile users may dynamically install tasks onto surrogates, and thus always be able to perform their tasks wherever they may be. The mobile code language in Scavenger is Python—an interpreted language supported on a myriad of platforms. A mobile code execution environment capable of fine grained scheduling over multiple CPUs/cores has been implemented using Stackless Python. When working with mobile code security becomes paramount, and Python has no built-in security measures. Because of this the execution environment implements its own measures using a black-listing/while-listing approach, where dangerous language features are masked using black listing, and module imports are allowed through white listing. For more information about this execution environment and its security measures see [39]. Using this mobile code execution environment client applications can, by adhering to a very minimal interface, dynamically push functionality onto Scavenger surrogates, and, if no surrogates are available, the exact same code may be performed by the client device.

Clients communicate with surrogates by use of RPC. This is handled entirely by the Scavenger library which is offering simple hooks such as `install_task`, `perform_task`, etc. All communication between client and surrogate goes through the Scavenger front-end, which acts as a small layer between

---

**Figure 1** Architectural overview of Scavenger.
the execution environment, the data store, and the context information collected by Presence. When performing a task at a surrogate, the client may choose to leave the output of the task on the surrogate, so that it may be used as input for later tasks. Instead of the output data the client is then given a data handle, that may be used to fetch the data or as input to a later task invocation. These data handles are considered in the scheduling of tasks so that data locality may be taken into account when choosing amongst multiple surrogates. When a surrogate receives a task for execution it checks whether any of the task inputs are data handles. If so it either fetches the data from the local data store or, if the data is stored on another surrogate, it checks its context table for information about the surrogate and using that information it fetches the data.

One component that may seem to be missing in the figure is the scheduler. Scheduling in a cyber foraging system must be done in a dynamic, decentralised manner. This means that scheduling must be done by the client device, and scheduling in Scavenger is thus part of the Scavenger library. How the scheduler works is described in detail in Section 4.

3.2 Development Model

Scavenger goes to great lengths to alleviate the application programmer of the pains of developing cyber foraging applications. In this section Scavenger’s development model is briefly described.

A task in Scavenger must adhere to a few simple rules:

1. It must contain a perform function which must accept the task’s input as its arguments and return the task’s output.

2. No black-listed keywords may be used within the task. These are keywords such as open, eval and any reflective programming handles.

3. Only white-listed modules may be imported. This means that only modules considered safe can be imported.

Apart from these three simple rules anything goes. The task may e.g., raise exceptions, which will be handled gracefully at the surrogate and re-raised on the client side. Once functionality has been expressed as a task, it may be dynamically installed onto and invoked at a surrogate.

To ease the process a little, Scavenger introduces a function decorator that may be used to completely automate it. Using the decorator a developer can indicate that this function may be a viable choice for remote execution. The decorated function can be used as any other function—the developer need not consider that it may be performed on a remote host. When the decorated function is invoked on runtime, the Scavenger library creates a task and schedules the task considering all available surrogates. If local execution is chosen, the local function is simply invoked, otherwise the task is automatically installed.
onto the chosen surrogate prior to invocation. An example task using the function decorator is shown in Figure 2.

```python
#scavenge
def sharpen(image, factor):
    from PIL import ImageEnhance as IE
    factor = 1.0 + factor
    return IE.Sharpen(image).enhance(factor)
```

**Figure 2** A sample task using the decorator.

At line 1 in this figure the `scavenge` decorator is used to annotate the `sharpen` function. The contents of the decorated function must be self-contained, i.e., modules used in the function must be imported within the function itself; which is done for the PIL module in line 3. This simple function sharpens an input image and returns the sharpened image as output. Using a function decorator that automates task creation, installation, and invocation, means that application partitioning is not really necessary—or at least that it happens without the developer having to worry about it.

There are more than one decorator available in Scavenger, some of them accepting arguments that may be used to further inform the scheduler about the task to improve the scheduling. An example of such an argument could be output size designation where the developer may tell the scheduler what the expected output size will be. This can be given as an expression relating input size or value to output size. This will be elaborated on in Section 4 where the scheduling algorithms are described.

### 3.3 Demonstrator

Using the Scavenger library a demonstrator application has been developed that realises the use case described in the introduction. This application is called The Augmented Image Manager, or simply AugIM. The application has been written for the Nokia N810 Internet Tablet, is written entirely in Python, and is available as open source at the Scavenger project site. A screen shot of the application in action is shown in Figure 3.

The only parts of this application that are “special” because of the use of cyber foraging are the functions performing the actual image operations which are prefixed by a Scavenger decorator. Adding cyber foraging to an application such as this one, on a relatively resource poor device such as the N810, greatly improves its performance—as will be shown in Section 5. And, because of the simple development model, the amount of work needed to incorporate cyber foraging into the application is minimal.
4 Dynamic Scheduling

Scheduling in a mobile, heterogeneous environment such as the one that must be catered for in cyber foraging is a very complex matter. First off, there are no central scheduler that tasks can be submitted to, so the client itself must do the scheduling. Secondly, because of the ever changing environment, scheduling must be done at run-time acting on the information given at that point in time. This means, that the client is itself responsible for collecting the information needed to make an informed decision when performing tasks. Ideally the information available when scheduling a given task should be:

- The task in- and output size (bytes).
- Estimated running time when performed at any of the available machines (seconds).
- Bandwidth (bytes/second) and latency (seconds) information for all links in the network—both links between client and surrogate and links between surrogates.
- Data locality information. I.e., information about where the input data resides (it may be a result from a previous task stored at one of the surrogates), and whether or not the output data should be pushed back to the client.

Given this information an idealised scheduler is shown in Figure 4. In this figure $I_{\text{size}}$ is the input size, $I_{\text{loc}}$ the input location and, correspondingly, $O_{\text{size}}$ and $O_{\text{loc}}$ are the size and locality of the output. The bandwidth and latency between two hosts N and M are $B_{N \rightarrow M}$ and $L_{N \rightarrow M}$, and the running time of a task T on a machine M is $T_M$. 

Figure 3 The Augmented Image Manager.
Of course getting the information needed to implement such an ideal scheduler is not easy; if at all possible. In the following it will be described how this information can approximated in a cyber foraging setting.

4.1 Network Information

The information needed about the network, i.e., bandwidth and latency measurements for all interconnecting links, are hard to come by, especially so if fluctuations due to peaks in traffic are to be considered. The problem is not measuring the current bandwidth and latency between two peers, this can be done quite easily. The problem is that in order to measure it a large amount of traffic is generated on the network, and seeing as these measurements must be done periodically between all peers in the network, an excessive amount of measurement traffic would be introduced to the network. The effect of this is especially detrimental in broadcast networks such as Wi-Fi. In Scavenger bandwidth and latency measurements have been substituted by a static media specification, where both clients and surrogates specify which kind of media they connect to the network with. These media specifications are then mapped to some expected bandwidth and latency values for that specific media. Consider the network in Figure 5.

This figure shows a common managed network consisting of a few wired and a number of wireless devices. In this environment the wired devices, the desktop PCs, would specify in their configuration that they are connected to the network using a 100 MBit connection, Laptops and UMPCs would probably be connected using an IEEE 802.11g, and mobile devices perhaps using an IEEE 802.11b connection. During scheduling the client knows each peer’s specified bandwidth and when considering transfer of data between two peers the smallest one of these two is of course chosen.
4.2 Task Specific Information

Task specific information in this regard is run-time information about the task being scheduled. This entails the size of in- and output data and data locality information.

Seeing as scheduling is done dynamically, i.e., at run-time, the task’s input is readily available at the scheduling device. This means that information about task input is trivially given. The input may reside at other peers in the network, in which case the scheduling device will be holding a data handle instead of the actual input data. In Scavenger these data handles include information about the size of the data, so that this information may be used when scheduling regardless of the fact that the data is stored remotely.

Output data, on the other hand, is of course not given before the task has been performed. For some tasks it may be impossible to predetermine the size of the output, e.g., a task that fetches a file over the Internet. But for many regular tasks it is indeed possible to determine what the size of the output will be. Some tasks always return the same size output, while others have output of a size relative to input size or value. In Scavenger the developer may designate the output size as a relation to the input size or value. This is done by giving the function decorator an argument containing a string representation of a Python expression. In this expression \( #n \) will be substituted by the \( n \)’th input argument before the expression is evaluated. For example the expression \( \text{len}(#0) \times 2 \) would mean that the output data is expected to be of twice the size as the first input argument.

The function decorator also accepts an argument telling it whether to fetch the output and deliver it to the client, or whether output data should be left
4 Dynamic Scheduling

at the surrogate and a data handle returned in its place. If data is left at the surrogate the overhead of transferring the output data is ignored.

4.3 Peer Specific Information

The most complex part of task scheduling is assessing the running time of the task with the given input on any of the currently available peers. There is only one way of getting at such information and that is through profiling, which is also the approach taken by related systems such as Spectra [22] and Chroma [5].

One approach towards building a profile usable for task scheduling is creating a peer centric profile, where a history based profile containing information about the last $n$ runs of a task on a specific peer is stored. Whenever that task is considered for execution on that specific peer, the profile may be consulted and the value found here can be used as an estimate of what the running time will be. How the history based profile is used is a design decision; in Scavenger the average of the profile data from the last ten runs is used as an estimate. Another approach would be to make a weighted average, where more recent profile data is given more weight.

There are some problems with this kind of peer centric profiling though: For one, it entails an assumption that a given task always has roughly the same running time. This is of course not true; for most tasks the running time will vary with input size or value. Furthermore, when working with highly mobile cyber foraging, the idea of having a peer centric profile, necessitating profile information about the specific peers that the client is currently within range of, works counter to the mobility of the system. In highly mobile cyber foraging it is more than likely that a given task has never been performed on the currently available surrogates. Both of these deficiencies of profile based scheduling in a cyber foraging setting have been considered in Scavenger, which is why Scavenger uses multidimensional profiles to reflect that a task’s running time may vary with input, and task centric profiles that may be used to reason about task running time on hitherto unknown surrogates.

Multidimensional Profiles

To approach the fact that running time varies with input size and/or value, Scavenger introduces an additional argument to the function decorator. Using this argument the developer may specify exactly which input parameters affect the running time of the task. This “complexity expression” can be given as a Python expression referencing input parameters in the same way as it was described for the output size designation. This complexity expression is evaluated to yield a single “complexity value”, and using this value, which could e.g., be the size of an input file, Scavenger maintains a two-dimensional profile. After performing a task this profile is updated by inserting the collected profiling data into the “bucket” whose key most closely matches the given complex-
ity value. Updating the profile is done using the following simple algorithm:

1. If this is the first run simply create a bucket for the given complexity value and insert the profile data there.

2. If buckets exist find the bucket closest to the given complexity value and compare the collected profile data to that bucket’s average:

   (a) If the profile data differs less than a certain percentage insert it into this bucket.

   (b) If the variation in profile data is too large create a new bucket for the data if, and only if, the complexity values also differ more than a certain percentage.

By updating the profile in this way the profile data is capable of adjusting to variations in running time, while only maintaining as few buckets as possible. Consider the depiction in Figure 6. When the running time of the task rises fast many buckets will be created to reflect this, and when the increase is slow only very few buckets are maintained. When doing lookups in the profile the bucket with the complexity value closest to the current one is chosen, which can be done in $O(\log n)$ where $n$ is the number of buckets for that specific profile. Since $n$ will always be relatively small this lookup will be considered to have a constant running time in the remainder of this paper.

![Figure 6](image)

**Figure 6** Two-dimensional profile. The coloured boxes are the buckets created, the dotted line is the predicted running time of the task, and the data points (dots) are actual profile data.

Using this approach towards maintaining profiles solves the first of the deficiencies with profile based scheduling—now variations in running time due to input variations are correctly reflected in the profile.
Task Centric Profiling

As mentioned earlier on, the biggest problem with profile based scheduling in highly mobile cyber foraging is the peer centric profiles. Ideally, there should be some way of comparing the “strength” of different surrogate machines to each other. What is needed is a measure such that the running time of a task on a machine of strength $x$, would be half the running time of that same task on a machine of strength $\frac{x}{2}$. Of course no such perfect strength measure is available—multiple factors are in play when measuring a machines processing capabilities; factors such as CPU architecture, cache structure and speed, main memory speed, and even compilers all play a role in the performance of a modern CPU.

While nowhere near perfect, there are ways to compare the relative strengths of computers, most notably by benchmarking the machines by using a benchmarking suite. In Scavenger both surrogates and clients are benchmarked using the NBench\(^1\) benchmarking suite, and the score yielded by this benchmark is used as a strength value. In fact, NBench returns two scores for a system: an integer and a floating point performance score. To simplify matters these two scores are combined and the average is used as the peer strength in Scavenger.

This peer strength value gives a fairly good image of how a device performs when it comes to CPU intensive tasks, but it does have its problems. Different tasks may exercise different parts of the CPU, and some task may therefore perform much better on some architectures than on others. This is alleviated by the use of dual-profiling, as will become clear shortly.

Knowing how strong peers are relative to each other we are able to build task centric profiles, i.e., profiles that are bound to a single task instead of to a (task, peer)-pair. Where the peer centric profiles could contain simple time measurements of earlier executions of the task, the task centric profile must contain some “task weight” that can be scaled by the peer strength. In Scavenger the information stored in these task centric profiles are the expected running time on a machine of strength one. When considering scheduling the task on a peer, this task weight is divided by the current peer strength to obtain an assessment of what the running time would be. The current peer strength referred to in the previous sentence, is the strength of the peer under its current load. The discovery packets periodically sent out by surrogates contain both their strength and their activity count, where the activity count is the number of tasks that are currently being performed within their execution environment. The current peer strength is then:

$$Peer_{current\_strength} = \frac{Peer_{strength}}{Peer_{activity} + 1}$$

It is thus assumed that active tasks share the CPU equally, and any other processes being performed by the operating system are ignored. Likewise,

\(^1\)http://www.tux.org/~mayer/linux/bmark.html
when the task weight is calculated the activity level of the surrogate while it was performing the task is considered.

Whether or not this task centric profile data is usable depends on the assumption described above, that e.g., a peer of strength ten will be able to perform a task twice as fast as a peer of strength five. That this assumption largely holds can be seen in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>2 GHz</th>
<th>733 MHz</th>
<th>1 GHz</th>
<th>900 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>G5</td>
<td>50.980</td>
<td>50.188</td>
<td>54.600</td>
<td>52.755</td>
</tr>
<tr>
<td>G4</td>
<td>54.012</td>
<td>53.605</td>
<td>57.223</td>
<td>54.249</td>
</tr>
<tr>
<td>Contrast</td>
<td>54.946</td>
<td>55.100</td>
<td>59.223</td>
<td>54.723</td>
</tr>
<tr>
<td>Colour</td>
<td>109.859</td>
<td>126.243</td>
<td>106.860</td>
<td>98.404</td>
</tr>
<tr>
<td>Blur</td>
<td>82.545</td>
<td>95.976</td>
<td>83.081</td>
<td>73.399</td>
</tr>
<tr>
<td>Invert</td>
<td>21.208</td>
<td>27.360</td>
<td>32.347</td>
<td>28.038</td>
</tr>
<tr>
<td>Scale</td>
<td>91.072</td>
<td>102.974</td>
<td>44.648</td>
<td>35.300</td>
</tr>
</tbody>
</table>

Table 1  Task weight measurements. Ideally values should be equal across rows.

To produce the data in Table 1 four different machines performed seven image manipulation tasks and reported the weight they would assign to that task. All tasks were performed 50 times and the weight reported is the average of these runs. The machines in use are very different with regards to processor architecture; having both a PowerPC G4 and G5, an Intel Pentium 3, and an Intel Celeron M processor. Even with those differences in architecture, it can be seen that the assigned weights are quite similar, and when used in a history based profile these provide a good starting point for the scheduler. Notice though, that although the different machines tend to agree on the weight of most of the tasks, in some cases the architectural differences shine through. Consider for example the last row, where an anti-aliasing scale operation was performed. In these tests the PowerPC based (G4 and G5) machines reported weights that were twice or almost three times as high as the Intel based machines. This shows that while using this “task weight” based on benchmarking scores is no silver bullet, it does in most cases provide good results. And compared to having no knowledge at all about the estimated running time of task execution on unknown surrogates, it makes for a more informed scheduling in unknown environments.

**Dual Profiling**

Using the task centric profile just described, gives a good starting point for the scheduler when scheduling in an unknown environment. But, as has been shown, the task centric profiles are not always precise, and peer centric information should therefore always be preferred if such information is available.

Based on this observation Scavenger’s scheduler works with dual profiles; as depicted in Figure 7. Whenever a task is performed two profiles are up-
Figure 7 Scavenger’s dual profiles. The peer centric profile contains information about the task on peers A, B, and C, whereas the task centric profiles holds information about the last five runs on any peer.

dated—a peer centric profile and a task centric profile. Both of these profiles are updated using the task weight measure, where the measured running time is scaled by both the surrogate’s strength and activity level. When a task is being considered for execution on a given surrogate, the peer centric profile is consulted first in order to give the most precise data precedence. If no peer specific information is available, which is likely in a mobile cyber foraging scenario, the task centric profile is consulted. The information stored here is likely to be less precise, but tests show that it is still quite effective at guiding the scheduling process, as will be shown in Section 5.

Scavenger’s Scheduler

To sum up, Scavenger uses all of the information that was in use in the idealised scheduler presented in the beginning of this section. Of this information some is measured through profiling (estimated running time), some is statically defined by the participating peers (bandwidth), some is given at run-time (input size), and some is estimated by the developer of the task (output size). Using this information the scheduling algorithm shown in Figure 8 has been implemented. This figure presents the scheduling algorithm in simplified Python-like pseudocode that is very close to the actual implementation.

The running time of the scheduling algorithm is $O(n \log m)$ where $n$ is the number of peers currently available and $m$ is the maximum number of buckets within the two-dimensional profiles. In all practical use the number of buckets in the profiles will be a very small number; so small that the lookup in the profile may be considered to be of $O(1)$ running time, yielding a linear $O(n)$ running time to the entire scheduling algorithm. This modest running time means that the overhead of scheduling is very small, even on resource poor mobile devices.
peers = get_surrogates()
if peers == []:
    return localhost
peers.append(localhost)
input_size = len(dumps(input))
output_size = eval(output_size_expression)
complexity_interval = eval(complexity_expression)
task_centric_profile = get_tc_profile(task)
candidates = []
for P in peers:
    time = 0
    if input.location != P:
        bw = min(P.bandwidth, input.location.bandwidth)
        time += input_size / bw
    peer_centric_profile = get_pc_profile(task)
    current_strength = P.strength / (P.activity + 1)
    if peer_centric_profile:
        time += peer_centric_profile / current_strength
    else:
        time += task_centric_profile / current_strength
    if output.location != P:
        bw = min(P.bandwidth, output.location.bandwidth)
        time += output_size / bw
    candidates.append((time, P))
return select_minimum(candidates)

Figure 8 Scavenger’s scheduler in simplified Python-like pseudocode.

5 Results

An extensive benchmark study has been performed to validate the adaptive, dual profiling scheduler, and to compare it to other known profiling scheduling approaches. The testing has taken place in a highly heterogeneous environment, containing a number of computers and devices of differing architecture as shown in Figure 9.

The figure shows the network architecture, consisting of three desktop PCs connected directly via cable to the Wi-Fi router, and two mobile devices, an Asus Eee 900 UMPC and a Nokia N800 Internet tablet, connected to the same router via Wi-Fi. On each device its architecture and CPU speed is listed along with its measured NBench strength rating; which is the strength measure that is used in the task centric profiling. The devices have been named A through D, and these names are used throughout this section. The operating systems used on the devices also differ: A is running Mac OS X 10.5, B is running Mac OS X 10.4, C is running Ubuntu Linux 9.04, D has Ubuntu 9.10, and E is running Maemo Linux 4.1.

Many aspects are of importance when testing a cyber foraging scheduler. First and foremost, the scheduling overhead must be as low as possible to keep the number of CPU cycles wasted when doing local execution at a minimum. The scheduler presented in this paper utilises not one but two profiles, and these profiles are two-dimensional necessitating a logarithmic search before
each lookup/update. In order to show what maintaining such profiles does to the scheduling overhead, a test has been performed comparing this overhead to those of a task centric and a peer centric scheduler. This test is described in Section 5.1.

One of the key benefits of using the task centric profile is that the scheduler should be able to quickly recognise the best possible surrogate for the job; even in environments where no previously known surrogates exist. This is tested in Section 5.2.

A part of the use case described in the introduction was that heavy computing tasks could be queued up and then performed at a later point in time when surrogate machines become available. Section 5.3 presents a test of this batch scheduling.

Finally, in Section 5.4 the schedulers’ ability to adapt to varying running times caused by variations in input is tested.

5.1 Scheduling Overhead

In order to measure the scheduling overhead Scavenger’s code was instrumented to measure the time spent selecting a surrogate. This was measured
for the task and peer centric schedulers as well as for the adaptive, dual profiling scheduler. The task was scheduled 50 times using seven different input images of varying size. Input size is varied because marshalling of the input is a necessary step in the scheduling, in order to get the size of the input when transferred over the network. The task being performed is a brightness adjustment of the input image.

![Image Size vs Overhead](image.png)

Figure 10 - Scheduling overhead when using the adaptive, dual profiling scheduler. The plot shows the percentage of the total running time that was spent scheduling. Over each bar the total running time in seconds is shown.

Figure 10 shows the result of running this benchmark using the adaptive, dual profiling scheduler. The plot shows the percentage of the total running time that is spent doing scheduling. For the first two very small images, the overhead is a large percentage of the total running time; 34.7% and 16.6% respectively. The total running times of these two tasks are very low, so low in fact that local execution was chosen by the scheduler in all of the runs, meaning that the time spent scheduling is time wasted because no surrogate was chosen. This result is to be expected, others have found that tasks must be of a certain size, before they will benefit from cyber foraging [22]. As soon as the tasks get a little larger, i.e., already when the image has a size of 400x300 pixels, the scheduling overhead is very low compared to the total running time.

In order to compare the schedulers the overheads of the three different
Figure 11 Scheduling overhead for the task centric scheduler, peer centric scheduler, and adaptive dual profiling scheduler that combines the two.

Scheduling overhead for the task centric scheduler, peer centric scheduler, and adaptive dual profiling scheduler that combines the two. The figure shows, that the cost of scheduling is somewhat larger for the more complex scheduler, but the increase in overhead stays within a reasonable 30% of the scheduler with the smallest overhead; the task centric scheduler. This benchmark thus shows, that even though multiple profiles are maintained, the cost of scheduling is not excessively high.

5.2 Learning Quickly

An important aspect of task centric profiling is, that it makes it possible for the scheduler to make informed task placement, even in cases where the currently available surrogates are unknown. In this benchmark, three tasks are performed in serial on the input image: first the image is sharpened and then its brightness and contrast are adjusted. The benchmark is run ten times in the environment depicted in Figure 9, and before each test run the profile data of the client is emptied, to simulate performing the tasks in a previously unknown environment.

The best place to perform this task in the test environment is at surrogate A, so ideally this surrogate should be chosen as early as possible. The results of performing the test is shown in Table 2.

Initially all schedulers have no information about the task, meaning that they have no estimates of how long the task execution is going to take, and an execution time of zero seconds is then assumed on all devices. Because of
Table 2 Results of performing the task placement benchmark.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Test run #</th>
<th>Distribution</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer centric</td>
<td>1</td>
<td>E, E, E</td>
<td>141.6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>D, D, D</td>
<td>29.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>20.4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>C, C, C</td>
<td>27.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>B, B, B</td>
<td>31.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>A, A, A</td>
<td>21.2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>A, A, A</td>
<td>20.0</td>
</tr>
<tr>
<td>Task centric</td>
<td>1</td>
<td>E, E, E</td>
<td>141.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A, A, A</td>
<td>20.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>23.3</td>
</tr>
<tr>
<td>Adaptive,</td>
<td>1</td>
<td>E, E, E</td>
<td>144.8</td>
</tr>
<tr>
<td>dual-profiling</td>
<td>2</td>
<td>A, A, A</td>
<td>19.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>A, A, A</td>
<td>22.7</td>
</tr>
</tbody>
</table>

Table 2 Results of performing the task placement benchmark.

this local execution is chosen by all three schedulers—the input data already resides at the local device, and thus this device is favoured by the scheduler. Now profiling information from a single run is available, and the task centric scheduler is able to employ this information to select the correct surrogate device in all succeeding runs. The peer centric scheduler unfortunately needs to perform the task at all available surrogates at least once, before it is able to select the correct surrogate device. It is thus not before the sixth run that the peer centric scheduler is capable of choosing the correct surrogate.

Note that the same surrogate is always chosen for all three tasks. This is because of the data locality factor in the schedulers. The result of the intermediate tasks, the sharpening and brightness adjustment, are left on the surrogate performing the task. This means, that when the next task is scheduled the input for that task already resides at that surrogate, and that surrogate will then naturally be preferred by the scheduler.

5.3 Batch Scheduling

When no surrogates are available it is common for cyber foraging enabled applications to queue up tasks for later processing. When a client device having such queued up tasks enters an area with surrogates, it is interested in scheduling all of these tasks in parallel using the available surrogates. This scenario is tested in the benchmark covered here. The client device has a queue of 24 image operations that is to be applied to eight images; more precisely each of the eight images are sharpened, have their brightness adjusted, and finally have their contrast adjusted in that order. When the benchmark is started the first
eight tasks are immediately scheduled, and as soon as one of these finish the task waiting on it is scheduled. The benchmark has been run with seven different images of varying size, each run has been repeated 50 times, and the results presented here are averages of these runs.

The results of this benchmark are shown in Figure 12 where the total running time of the three schedulers for the seven different input image sizes are plotted. For such batch processing the schedulers have comparable performance, with the adaptive, dual profiling scheduler being slightly slower because of the added scheduling overhead. The comparable performance points at, that all three schedulers are equally good at selecting the right place to perform a given task. This means e.g., that the task centric scheduler, using its simplified view on task complexity and surrogate strength, is capable of creating schedules that are just as good as the ones selected by the more precise peer centric scheduler.

When doing batch scheduling it is important that tasks are distributed in a way such that the available surrogates are equally utilised. But this utilisation must be relative to the surrogate’s relative strength, so that stronger surrogates are assigned more tasks than weaker ones. Given the relative strength of the surrogate devices in the test environment in Figure 9, an ideal scheduler should give a bit more than 40% of the tasks to surrogate A, and approximately 20% each to surrogates B, C, and D. The actual distribution of tasks over the available surrogates that was obtained in the benchmarks is shown in Figure 13.
Figure 13 The distribution of tasks when performing the batch scheduling benchmark.

This figure shows the distribution of tasks in the benchmark runs using the largest input image. The first thing to notice in Figure 13 is the distribution of tasks using the task centric scheduler, which is as expected tied firmly to the strength ratings of the surrogates: surrogate A, being the strongest of the bunch, has received most of the tasks, B, being the weakest one, has received the least, and C and D, being of comparable strength, have received almost the same amount of tasks. Given a task of higher granularity, e.g., by scheduling more than eight tasks in parallel, the relation to the strength rating would become even more apparent.

In a scenario such as the one benchmarked here, the two remaining schedulers will both be using their peer centric profiles most of the time, and their choice of task distribution is thus expected to be alike. There is some small variations in the chosen distribution between the two schedulers, and these should probably be accredited to the inexact nature of the task weight computation. When using per (task, peer)-pair profiles there is a greater chance of one of the profiles being skewed for some time, leading to a more diffuse scheduling. That the task weight computation is inexact is no problem for the task centric scheduler, since it uses the same value when considering execution on all peers.

A final thing that may be shown by this benchmark, is how efficient the schedulers are at using the information about network bandwidth and the locality of data. As in the benchmark presented in Section 5.1 the result of intermediate tasks are left at the surrogates performing those tasks, and the sched-
uler should thus be able to employ knowledge about that when scheduling dependent tasks. During each test run the client logged where each task was sent, and using that information the task schedules shown in Figure 14 have been created. Figure V.14(a) shows the schedule chosen by the adaptive, dual profiling scheduler in one of the 50 runs using the image of size 400x300, and Figure V.14(b) shows the schedule chosen by the task centric scheduler in one of the runs using the larger 2000x1500 pixel image.

![Schedules for an image of size 400x300 (a) and an image of size 2000x1500 (b). Task nodes are coloured to show where they have been performed. On each task node its position in the overall schedule is printed—i.e., the node with position 1 was scheduled before the node with position 2.](image)

The key difference between Figure V.14(a) and V.14(b) is the degree to which locality information is used. When scheduling the tasks using the smaller 400x300 pixel image, the cost of transferring the image between surrogates is relatively low—especially so between surrogates A, B, and C which are connected by a 100 MBit link. Notice though, that the tasks given to surrogate D are performed in serial on that surrogate. This is because surrogate D is connected to the network using a low speed Wi-Fi connection, and the transfer time for intermediate data is thus much higher for this surrogate. Initially all data must be sent from the client device to a surrogate over the clients Wi-Fi connection, which means that surrogate D is just as likely to be chosen as any of the others.

In the schedule of the larger image in Figure V.14(b) the cost of transferring intermediate data between surrogates becomes more significant. Because of this, dependent tasks are in most cases scheduled on the surrogate holding the data.

Both data locality and information about network bandwidth is thus used
5.4 Adaptive Profiling

The final benchmark presented here shows the benefits of working with an adaptive profile, i.e., a two-dimensional profile that can adapt to the fact that task running time varies with input. This benchmark performs a sequence of tasks that could easily occur in the use case described in the introduction. Initially five tasks adjusting the brightness of an image are scheduled on a small version of the input image, this simulates the user previewing operations on the mobile device, and then the same task is scheduled twice using the original, large version of the image, simulating that the user has chosen to apply the previewed operations to two images. The benchmarks has been performed using two schedulers, both of them dual profiling but only one working with adaptive profiles. Each test has been run 50 times and the results reported here are the averages of those runs. Table 3 shows aggregated results from the benchmark.

When working with a non-adaptive profile the first task, operating on the small input image, was scheduled at the client device and the second task was scheduled at surrogate A. Seeing as performing the task at surrogate is way more expensive than local execution, the five next tasks are then scheduled at the client device. Two of those five tasks are operations on the large image, which are quite heavy to perform. This alters the weight of the task in the profile, and from that point on the task is considered heavy and therefore performed at the surrogate every time.

The adaptive scheduler on the other hand immediately detects that it is working with two very different task complexities, brought on by the difference in input image size. It therefore creates separate profiles for the two image sizes, meaning e.g., that the tasks working on the small image are almost always scheduled at the client device, while the tasks working on the larger image are only once scheduled on the client device.

This test shows, that using an adaptive profile makes it possible for a scheduler to adapt more quickly to variations in running time brought on by variations in input. Furthermore, in special cases such as the one shown in this

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Image</th>
<th>Surrogate A</th>
<th>Localhost</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual profile</td>
<td>small</td>
<td>246</td>
<td>4</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>98</td>
<td>2</td>
<td>8.73</td>
</tr>
<tr>
<td>Adaptive,</td>
<td>small</td>
<td>5</td>
<td>245</td>
<td>0.11</td>
</tr>
<tr>
<td>dual profile</td>
<td>large</td>
<td>99</td>
<td>1</td>
<td>7.46</td>
</tr>
</tbody>
</table>

Table 3 The effect of adaptive profiling. Ideally the schedulers should perform all tasks on the small image locally and all tasks on the large image on surrogate A.
benchmark, inefficient scheduling caused by continuous variations in input complexity can be avoided completely by having an adaptive profile.

6 Related Work

The past decade a number of cyber foraging systems have been proposed. These systems have varied greatly in their approach towards designing systems to support the visions of Satyanarayanan, who coined the term cyber foraging in his “Pervasive Computing: Visions and Challenges” [73] article from 2001. While the term cyber foraging was introduced in 2001, earlier work using the same techniques and general ideas have laid the ground for the field, e.g., the Odyssey system by Noble et al. [55] and M-RPC by Bakre and Badrinath [3].

One of the main challenges when designing a cyber foraging system, is deciding on how to do the actual distribution and execution of tasks. Some systems, such as Spectra by Flinn et al. [22] and Chroma by Balan et al. [5], rely on pre-installed functionality invokable by means of RPC, meaning that surrogate computers must be prepared beforehand to cater for the needs of the mobile clients. Other systems, such as Slingshot by Su and Flinn [77] and the system by Goyal and Carter [25], go to the opposite extreme and offer an entire virtual machine to the mobile client. In Slingshot this virtual machine is a shallow clone of a central virtual machine operated by the user of the mobile client, whereas in Goyal and Carter’s system a fresh virtual machine image is offered to the mobile client, and the client can then install any needed functionality. With regards to task distribution and execution Scavenger places itself in between these two extremes by taking a mobile code approach. In Scavenger mobile Python code may be distributed to surrogates on demand, thus obtaining the high mobility of the virtual machine based systems in that tasks do not need to be pre-installed onto surrogate machines, while avoiding the extremely large overhead of establishing an entire virtual machine on a surrogate before it can be used.

With regards to scheduling, the systems that are closest in nature to Scavenger are the aforementioned Spectra and Chroma. These RPC-based systems use task centric, history based profiling to choose amongst available surrogates, as described by Narayanan et al. [54]. Having a task centric profile means that they also employ a device “strength” measure, to be able to relate profiled task running times to a running time on a specific surrogate. In Spectra and Chroma, both systems that only run under Linux, they use the “bogomips” rating reported by this operating system for a CPU. Scavenger uses a more precise measurement of CPU performance, namely an actual benchmark of the CPU’s capabilities under different circumstances. In this paper an alternative approach towards profile based scheduling, a dual-profiling history based scheduler, is presented, and it is described in detail how it works and which factors it takes into consideration. Spectra and Chroma’s profiling seems to be one profile per task, whereas Scavenger works with two-dimensional profiles
to account for the fact that the running time of a task may vary with input size or value.

7 Conclusion

This paper has presented the Scavenger cyber foraging system. Scavenger aims at supporting highly mobile cyber foraging—meaning that cyber foraging is performed while the user is physically mobile, and that this mobility may find the user in environments where no known surrogates are available, necessitating system support for dynamic task distribution. It has been described how Scavenger approaches this mobility by providing fast discovery of surrogates, mobile code tasks that may be pushed onto surrogates on demand, and a scheduler that is capable of selecting the current best surrogate for a given task. By being intelligent about task placement, i.e., trying to select the surrogate that will solve the task the fastest, the risk of the client leaving range of the surrogate while the task is being performed is minimised.

The focus in this paper has been on the schedulers found within the Scavenger system. In Scavenger a large number of schedulers have been implemented—far more than have been presented here. The schedulers yielding the best performance are the profile based ones, and this paper has presented three such profile based scheduling approaches: the task centric approach, where a profile is maintained for each task, the peer centric approach, where a profile is maintained for each (task, peer)-pair, and the adaptive, dual profiling approach that combines the task and peer centric profiles and furthermore adds adaptability to the profiles, so that they may adapt to changes in input complexity. These three scheduling approaches have been extensively benchmarked and compared to see how they stack up to each other. All tests have been run in a heterogeneous computing environment using machines of differing architecture and operating system.

In Section 5.1 the scheduling overhead of the three chosen schedulers was measured. This benchmark was meant to show what the overhead of maintaining multiple adaptive profiles was, when compared to maintaining only a single task or peer centric profile. The added scheduling overhead for maintaining these more complex profiles was roughly 30% when compared to the task centric scheduler, and roughly 15% when compared to the peer centric scheduler. The overheads are relatively low, e.g., the complex scheduler used 50 ms on average to devise a schedule, so on any reasonably sized computing task paying for this overhead can easily be outweighed by the benefits of utilising a stronger CPU. Given the benefits of working with the adaptive, dual profiling scheduler, paying the added cost for scheduling is considered well worth it.

The key benefit of the task centric scheduler is its ability to make informed guesses at how a task will perform on a previously unknown surrogate device. This was tested in Section 5.2. As expected, the peer centric scheduler had to
try all available surrogates in turn before it was able to select the best surrogate for the task, thus wasting a lot of time performing the task at slower surrogates. The task centric, and therefore also the dual profiling, scheduler were able to select the correct surrogate after a single execution of the task—as soon as a task weight is available an execution plan can be devised. It is important to note here, that the task centric scheduler does not merely select the strongest surrogate, which could be done without calculating the first task weight. It needs the task weight to assess whether local execution would be faster than remote execution, and if remote execution is preferable it selects the surrogate that is currently most suited for the job, considering not only surrogate strength but also its current utilisation etc. When designing for mobile cyber foraging, where working with unknown surrogates is common, using a task centric profiling approach is therefore recommended.

Another important ability of a cyber foraging scheduler is being able to do clever batch scheduling, i.e., scheduling of multiple tasks onto currently available surrogates. In Section 5.3 a batch of eight image manipulation tasks, each consisting of three subtasks, were scheduled in a test environment with four very different surrogate machines. Running this benchmark showed that all three schedulers were capable of utilising multiple surrogates in parallel. Looking at the running time of the different schedulers it seems that the task centric scheduler was slightly better than the other two. This may be accredited to two things: first of the scheduling overhead is slightly smaller for this scheduler, and secondly its choice of execution place depends more heavily on current resource availability and less on the task weight.

The batch test also showed how nicely data locality is taken into consideration during scheduling. Taking such a thing into consideration is trivial in a traditional task scheduler, where the placement of all tasks and subtasks is chosen before anything is scheduled. But in a mobile system such as Scavenger, the scheduling must be done dynamically at run-time, because resource availability may have changed since the last subtask was scheduled.

The last benchmark presented in Section 5.4 showed the benefits obtained when working with an adaptive profile. This benchmark showed that the adaptive profile could greatly improve performance in cases where the same task was performed with varying input complexity.

8 Future Work

The results presented in this paper suggest that using a task centric profile based on a CPU benchmark such as NBench yields good results. The experiments conducted for this paper have all been focused on performing image manipulation tasks, in order to support the use case described in the introduction and implemented in the AugIm demonstrator application. It would be interesting to perform experiments with other kinds of compute intensive tasks, to see if the chosen strength measure holds up to other kinds of tasks. Further-
more, it would be interesting to do experiments with other strength ratings; such as e.g., the Linux “bogomips” rating or other CPU benchmark suites.

The bandwidth and latency values used in Scavenger right now are statically defined at the device. This of course does not reflect the variations in network performance that are experienced in actual use. This deficiency means that the scheduler will often underestimate the time needed to transfer in- and output data. As mentioned, actively monitoring network bandwidth introduces too much noise to the network, but measurements could be performed on data that must be transferred anyway; i.e., when in- and output data is transferred these transfers could be timed and thus a network profile could be built. Whether this would enhance the scheduling is not immediately obvious, and further experimentation would therefore be beneficial.

While many experiments have been performed using Scavenger, yet more experimentation could be advantageous. For example the results seem to show that the best scheduler would be a task centric scheduler using adaptive profiles; a scheduler that has not been implemented in Scavenger yet. Implementing this scheduler and testing it against the adaptive, dual profiling scheduler is something that will be done within the near future.

Acknowledgements

This paper has been funded by a research grant from the Danish Research Council for Technology and Production Sciences. Thanks also goes to Nokia for providing us with the mobile hardware used in the experiments and prototype implementation.
Bibliography


235


