Saving Energy in Network Hosts With an Application Layer Proxy: Design and Evaluation of New Methods That Utilize Improved Bloom Filters

by

Miguel Jimeno

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Kenneth Christensen, Ph.D.
Rafael Perez, Ph.D.
Miguel Labrador, Ph.D.
Stephen Suen, Ph.D.
Rahul Tripathi, Ph.D.

Date of Approval:
December 11, 2009

Keywords: Data structures, energy efficiency, performance evaluation, proxying, network applications

© Copyright 2010, Miguel Jimeno
To God, my wife, and my family.
Acknowledgements

I would like to thank my advisor at USF Dr. Kenneth Christensen. His support has been important for me to grow as a researcher during the 4 years I have stayed in Tampa. I express my sincere thanks to Dr. Rafael Perez, Dr. Miguel Labrador, Dr. Stephen Suen, and Dr. Rahul Tripathi for serving on my committee. I would also like to thank Bruce Norman, for his help on the topic of power management of network hosts. His work and support has been a guide in my research. I thank Dr. Allen Roginsky for his assistance on the mathematical part through the sections of Chapter 4. I appreciate the help provided by Beverly Caggiano, from the USF Library, who provided a list of 2 million book titles used in Chapter 4 to evaluate search algorithms presented. I express also my deepest gratitude to my wife for her constant support in this journey even in the troublesome moments.
List of Tables iii

List of Figures iv

Abstract vii

Chapter 1: Introduction 1
  1.1 Background 1
    1.1.1 Energy Use of ICT Equipment 1
    1.1.2 Energy Savings Opportunities 3
  1.2 Motivation for Research 4
  1.3 Contributions Made in This Research 6
  1.4 Organization of This Dissertation 7

Chapter 2: Background and Literature Review 9
  2.1 Overview of Power Management for ICT Hosts 9
    2.1.1 Power Management in Wireless Devices 13
  2.2 Induced Power Consumption in Network Hosts 14
    2.2.1 Client-Server Applications as a Cause of Induced Energy Use 15
    2.2.2 P2P Applications as Cause of Induced Energy Use 20
  2.3 Overview of the Gnutella P2P File Sharing Protocol 22
    2.3.1 Query-Based Protocol 23
    2.3.2 Tasks of a Leaf Peer and Ultradee 24
  2.4 Existing Approaches to Save Energy Wasted Due to Induced Use 25
  2.5 Network Connectivity Proxying (NCP) for Power Management 28
    2.5.1 Early Work in Prooing of ARP 30
    2.5.2 Prooing for UPnP 31
    2.5.3 Current and Recent Efforts to Define and Standardize Prooing 32
  2.6 String Search Algorithms and Data Structures 33
    2.6.1 Hash Tables 33
    2.6.2 Bloom Filters 35
    2.6.2.1 Applications of Bloom Filters 38
    2.6.3 Algorithms for String Searching 41
  2.7 Chapter Summary 43

Chapter 3: Proxying as a Means of Reducing Induced Energy Use 44
  3.1 General Requirements for a Network Connectivity Proxy (NCP) 45
  3.2 Covering TCP Connections 48
    3.2.1 Design of gSOCKS 49
    3.2.2 Prototype Implementation 52
    3.2.3 System Evaluation 54
  3.3 Covering Application Protocols – SIP 57
    3.3.1 Design of SIP Catcher 57
    3.3.2 Prototype Implementation 61
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Summary of Best-of-N Method Improvement</td>
<td>75</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Best-of-N Method (for N=100) Against Power-of-2 Improvement</td>
<td>75</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Numerical Results for SAHT</td>
<td>92</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Numerical Results for Bloom Filter</td>
<td>92</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Experimental Results for SAHT</td>
<td>93</td>
</tr>
<tr>
<td>Figure 1.1.</td>
<td>2006 US Electricity Usage (Not to Scale) [101]</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.1.</td>
<td>Power Consumption Versus Utilization</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.2.</td>
<td>Yahoo Messenger IM System Architecture</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.3.</td>
<td>Magic Jack Device for VoIP Phone Calls</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.4.</td>
<td>Components and Start of a Communication with SIP [117]</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.5.</td>
<td>Timing Diagram for the Start of a SIP Call</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.6.</td>
<td>Timing Diagram for File Querying in a P2P Gnutella Network</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.7.</td>
<td>Magic Packet Format</td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.8.</td>
<td>Verdiem Surveyor Infrastructure [137]</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2.9.</td>
<td>Operation of an NCP</td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.10.</td>
<td>Hash Table Algorithms to Add a String and to Find a String</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.11.</td>
<td>An Example of a Bloom Filter [20]</td>
<td>36</td>
</tr>
<tr>
<td>Figure 2.12.</td>
<td>Bloom Filter Algorithms to Add a List of Strings and Test a String</td>
<td>37</td>
</tr>
<tr>
<td>Figure 2.13.</td>
<td>Power-of-2 Algorithms From Lumetta and Mitzenmacher [86]</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3.1.</td>
<td>System View of the NCP</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.2.</td>
<td>Packet Processing by the NCP</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.3.</td>
<td>Architecture of the NCP</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.4.</td>
<td>Architecture of Power State Signaling</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.5.</td>
<td>The Target System: Linksys WRT54G</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.6.</td>
<td>Test-Bed to Evaluate gSOCKS</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3.7.</td>
<td>Results From IM Experiment</td>
<td>56</td>
</tr>
<tr>
<td>Figure 3.8.</td>
<td>Architecture of SIP Catcher</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.9.</td>
<td>Timing Diagram for a SIP Call Initiation Proxied by the SIP Catcher</td>
<td>60</td>
</tr>
<tr>
<td>Figure 5.2.</td>
<td>Sleeping Host With P2P Application Proxied by a Stateless Proxy</td>
<td>105</td>
</tr>
<tr>
<td>Figure 5.3.</td>
<td>Main Programs for Host and Proxy</td>
<td>106</td>
</tr>
<tr>
<td>Figure 5.4.</td>
<td>Functions and Processes Common to Host and Proxy</td>
<td>107</td>
</tr>
<tr>
<td>Figure 5.5.</td>
<td>Processes Specific to Host or Proxy</td>
<td>107</td>
</tr>
<tr>
<td>Figure 5.6.</td>
<td>Netburner Development Kit</td>
<td>108</td>
</tr>
<tr>
<td>Figure 5.7.</td>
<td>Results From Query Forwarding Experiment</td>
<td>111</td>
</tr>
<tr>
<td>Figure 5.8.</td>
<td>Results From Query Hit Experiment</td>
<td>111</td>
</tr>
<tr>
<td>Figure 5.9.</td>
<td>Sleeping Host With P2P Application Proxied by a P2P Stateful Proxy</td>
<td>113</td>
</tr>
<tr>
<td>Figure 5.10.</td>
<td>Architecture of the P2P Stateful Proxy</td>
<td>115</td>
</tr>
<tr>
<td>Figure 5.11.</td>
<td>Gnutella Message Processing by the P2P Catcher</td>
<td>117</td>
</tr>
<tr>
<td>Figure 5.12.</td>
<td>The Test-Bed for P2P Stateful Proxy</td>
<td>119</td>
</tr>
<tr>
<td>Figure 5.13.</td>
<td>Test-Bed Used to Test the P2P Stateful Proxy</td>
<td>121</td>
</tr>
<tr>
<td>Figure 5.14.</td>
<td>Query Hit Response Sent by the Awake Host</td>
<td>122</td>
</tr>
<tr>
<td>Figure 5.15.</td>
<td>Query Hit Response by Proxy</td>
<td>122</td>
</tr>
<tr>
<td>Figure 5.16.</td>
<td>Messages Sent by the Host if Proxy Does Not Discard Messages</td>
<td>123</td>
</tr>
<tr>
<td>Figure 6.1.</td>
<td>Future Directions for Network Connectivity Proxying</td>
<td>127</td>
</tr>
</tbody>
</table>
Saving Energy in Network Hosts With an Application Layer Proxy: Design and Evaluation of New Methods That Utilize Improved Bloom Filters

Miguel Jimeno

ABSTRACT

One of the most urgent challenges of the 21st century is to investigate new technologies that can enable a transition towards a society with a reduced CO$_2$ footprint. Information Technology generates about 2% of the global CO$_2$, which is comparable to the aviation industry. Being connected to the Internet requires active participation in responding to protocol messages. Billions of dollars worth of electricity every year are used to keep network hosts fully powered-on at all times only for the purpose of maintaining network presence. Most network hosts are idle most of the time, thus presenting a huge opportunity for energy savings and reduced CO$_2$ emissions.

Proxying has been previously explored as a means for allowing idle hosts to sleep yet still maintain network presence. This dissertation develops general requirements for proxying and is the first exploration of application-level proxying. Proxying for TCP connections, SIP, and Gnutella P2P was investigated. The TCP proxy keeps TCP connections open (when a host is sleeping) and buffers and/or discards packets as appropriate. The SIP proxy handles all communication with the SIP server and wakes up a sleeping SIP phone on an incoming call. The P2P proxy enables a Gnutella leaf node to sleep when not actively uploading or downloading files by handling all query messages and keyword lookups in a list of shared files. All proxies were prototyped and experimentally evaluated.

Proxying for P2P lead to the exploration of space and time efficient data structures to reduce the computational requirements of keyword search in the proxy. The use of pre-computation and hierarchical structures for reducing the false positive rate of a Bloom filter was explored. A Best-of-N Bloom filter was developed, which was shown to have a lower false positive rate than a standard Bloom filter and the Power-
of-2 Bloom filter. An analysis of the Best-of-N Bloom Filter was completed using Order Statistics to predict the false positive rate.

Potential energy savings are shown to be in the hundreds of millions of dollars per year assuming a modest adoption rate of the methods investigated in this dissertation. Future directions could lead to greater savings.
Chapter 1: Introduction

One of the most urgent challenges of the 21st century is to investigate new technologies that can enable a transition to a sustainable society with a reduced CO$_2$ footprint. This is key, given that results from most recent studies show that accumulation of greenhouse gases in the atmosphere is growing faster than initially predicted [125]. This dissertation addresses solutions to save energy in PCs, the major contributors of CO$_2$ for Information and Communication Technology (ICT).

1.1 Background

It is important to quantify how much energy ICT is consuming and also how much it is contributing to total CO$_2$ emissions. With this information, the energy waste of PCs can be calculated and the reasons for it determined. This quantification helps measuring the potential energy savings of proposed solutions for energy savings.

1.1.1 Energy Use of ICT Equipment

The CO$_2$ footprint of the IT industry has been estimated to be 2% of the global CO$_2$ footprint, which is roughly the same as that of the aviation industry [46]. Figure 1.1 shows electricity use in the U.S. for 2006 [101]. Calculations state that ICT and consumer electronic devices will consume close to 200 TWh in 2009. But forecasts estimate that this number will double by year 2030 [62]. Network hosts consume a large and growing amount of energy. Data centers and servers consumed an estimated 1.2% of total U.S. electricity consumption in 2006 for a total cost of about $4.5 billion [78], [116]. However, even greater is the electricity consumed by network hosts not in data centers, as has been calculated by several studies. The most updated source [101] calculates it as more than the 2% shown in Figure 1.1.

The work presented in [99] was the first publication where measurements of power consumption of PCs were presented. The authors took measurements in offices located in Canada. This publication did not consider them as network hosts, and did not forecast power consumption of all PCs in the country. The first
An authoritative source on energy use of all the network hosts was that published by Lawrence Berkley National Laboratory (LBNL) in a 2001 report [75]. It stated that office and network equipment consumed 74 TWh/year at that time. That represented about 2% of the total electricity consumption of the U.S. About 9% of the electricity consumed by commercial buildings is from office equipment – much of this equipment being network-connected office PCs [118]. A single PC consuming 100 W (a not unreasonable power consumption for a desktop PC) powered-on at all time for one year is equal to [51]:

- 0.88 metric tons of CO₂
- 0.12 passenger cars for one year
- 77.3 gallons of gasoline consumed
- 0.09 homes for one year

The last item means that the addition of a single PC powered-on at all time will add about 9% to the utility bill of a typical U.S. household (based on 11,965 kWh/year, according to [51]). Beyond PCs are the growing number of commercial and consumer devices – such as set-top boxes and game consoles – that are connected to the Internet and have become network hosts. It is expected that in the future even more types of devices will become network hosts.

Much of the electricity consumed by network hosts is wasted. Being connected to the Internet requires some active participation. When hosts fail to do this, they “fall off the network” and applications fail. Billions of dollars worth of electricity every year are used to keep network hosts fully powered-on at all times.
times only for the purpose of maintaining network connectivity or “presence” [100]. If not for the necessity of network connectivity most of these hosts could be asleep the majority of the time, with resulting in energy savings. It is the need to maintain network connectivity that contributes to the disabling of existing power management features in many PCs, game consoles, and other PC-like devices.

1.1.2 Energy Savings Opportunities

Why are the majority of office and home PCs left powered-on even when not in active use, such as overnight and during weekends? Surveys have found that between 50% and 60% of office desktop PCs are left on continuously [109], [115]. Specifically the survey in [109] explains why employees leave computers on all night. Around 31% of the surveyed employees in the U.S. said that they leave computers on during the night for reasons that induce computers to consume power when they do not use them. They answered, among other responses, that they forget to turn their PCs off (13% of the employees responded so), leave it on for updates to be installed during the night (9%), or also because it is a policy of the company (9%). The following two major reasons for leaving PCs on during the night can be extracted from this:

1. The annoyance to a user of having to wait for a PC to wake-up out of the sleep state, and
2. The need to maintain network connectivity at all times to allow for remote access and/or for network-centric applications to maintain their state.

The first reason is becoming less significant as PCs become able to wake up faster [4], while the second reason is becoming more significant as more applications and protocols rely on persistent Internet connectivity. The need to maintain network connectivity at all times is becoming even more urgent to address as the number of PC-like devices, including set-top boxes, game consoles, and so on proliferate. All of these devices are connected to the Internet and many require full connectivity at all times to provide the services they are intended to deliver. To maintain network connectivity, a host must be able to support a number of application and protocol primitives, including:

1. Maintain host-level reachability by responding to periodic ARP requests in order to not age-out of the last-hop router ARP cache,
2. Maintain its IP address by generating periodic DHCP lease requests in order to not lose its IP address (if using DHCP to obtain an IP address),
3. Maintain its manageability by responding to ICMP packets, such as ping,
4. Support NetBIOS name resolution by responding to NetBIOS name queries as appropriate (if running NetBIOS protocol and applications),

5. Maintain application-level reachability by responding to TCP SYN packets sent to open (listening) ports,

6. Maintain or preserve application state (e.g., current user workspace and data) for any applications with open long-term TCP connections,

7. Maintain or preserve application state by responding to any number of application-level messages including heartbeat messages and specific requests for service.

If a host can remain reachable even when sleeping and keep its NIC powered-on, it can then be woken-up from the network by “trigger” packets that the NIC recognizes. The Magic Packet was invented for this purpose in the mid-1990s [8]. Most NICs today support both Magic Packet and the ability to pattern match for specific packets to trigger a wake-up. However, wake-up does not completely solve the problem of network connectivity. Wake-up triggered on pattern matching can cause both missed and unnecessary host wake-ups resulting in applications that fail and/or reduce energy savings.

Estimating the exact cost of the network connectivity problem is difficult. One study suggests that half of all electricity consumed by PCs is wasted [99]. Energy consumption and waste of network hosts is examined later in this dissertation. Clearly, solving the network connectivity problem must be of urgent interest as a research problem.

1.2 Motivation for Research

Saving the wasted energy due to the need to maintain network connectivity can be done by 1) redesigning network protocols and applications, or 2) encapsulating the intelligence for maintaining network presence in an entity other than the core of the networked devices. The second option has been termed Network Connectivity Proxying (NCP) [25], [28], [53], [100]. An NCP is an entity that maintains full network presence for a sleeping network host [100]. The NCP concept was first proposed in the late 1990s in the context of an ARP proxy [25] and defined further in the early 2000s [23], [53]. Currently, a basic NCP functionality is being standardized by Ecma [34] and is also already part of the EPA Energy Star Specification for Computers [112]. This current work addresses items 1) to 5) in the list in the previous section. Current and previous work related to the NCP concept is described later in this dissertation.
is lacking in all current and previous work is a focus on network applications that drive induced energy use—items (6) and (7) in the list in the previous section. Induced energy use is formally defined as the increment for higher power state of a device needed to maintain network connectivity. There are two motivating topics that this dissertation addresses:

1. Network applications driving energy waste in PCs and PC-like devices, and
2. The need for more efficient methods of directory and keyword searching so that P2P applications can be proxied.

Network applications require connectivity to maintain presence and, in some cases, to preserve the local and remote state. Two examples can be given to demonstrate the need for network connectivity:

1. VoIP phones relaying on the SIP communication protocol are left on for long periods of time for the sole purpose of receiving calls. In the U.S., energy wasted is close to 4 TWh/year due the use of these applications, and
2. P2P applications that force users of hosts running them to leave them on during long periods of time to keep connected to the P2P network. In the U.S., the energy wasted is about 2 TWh/year due to the sole purpose of keeping hosts connected to the P2P networks.

These applications and the protocols they use need to be able to respond to incoming messages at all times. Thus, these applications effectively induce energy consumption even when not in active use. In many enterprises, PCs are left powered-on overnight so that they can be remotely managed (e.g., for installing software updates). Problems of remote management of sleeping PCs are already well addressed in previous work (e.g., by commercial tools such as [59], [137], [60] and [14]). This, however, targets only office PCs. Although many of the ideas can be used for home PCs, the decentralized and heterogenic aspect of networks of home PCs allows further room for research.

The NCP as being defined and standardized by Ecma [100], does not consider applications with long-lived TCP connections such as telnet, SSH, IM, and others. It has generally been considered too difficult, except in very specialized cases related to high-performance servers [79], to hand-off TCP connections between entities. Can TCP connections be proxied without requiring full hand-off of connections between the host and proxy? There are other applications that rely on incoming connection requests to provide some type of service. This is the case for low-usage web servers, that provide service for file requests, and VoIP
phones that accept new connections to initiate phone calls over the IP network. Can low-usage web servers and hosts running VoIP phones sleep and be woken up only to provide their service? And what is necessary to allow these hosts to sleep for long periods of time?

A key class of applications that is driving increased energy use of desktop PCs is P2P file sharing. A P2P node must remain fully connected to the network at all times while it is sharing files. A P2P node must be able to forward Query messages and directly respond to query messages. A P2P node must respond to all query messages that have keywords matching files stored (and shared) by the node. A key question addressed in this dissertation is: can P2P applications be proxied using the NCP concept?

P2P applications place a large demand on the host processor to implement string search for keywords in directory lists (of shared files). P2P nodes may share thousands to millions of files, each file identified with a file name string that can be many words in length (e.g., the title of a song or name of a movie). Each incoming query message generates a search of the directory list for keywords. To implement directory search in an NCP would currently require significant processing capability (and thus large power consumption). Are there ways in which directory searching can be simplified – made less computationally complex – to enable implementation on small, low-power devices with relatively small processors?

A key observation is that the response to a P2P query does not need to be precise – a rare “false positive” is acceptable (and would be resolved manually by a user searching for files). This suggests the possibility of using a probabilistic data structure such as a Bloom filter [15] for implementing directory lists. Bloom filters have been well studied for implementation in hardware, but less so for software. Can Bloom filters be both improved and applied to the directory search problem to make proxying of P2P applications feasible? It is key to explore how efficient and suitable this data structure is when used by a type of proxy, in terms of space and time usage.

1.3 Contributions Made in This Research

This research addressed the open problems described in the previous section. In doing so, the following contributions were made:

1. The first exploration of application-level proxying as a means to reduce energy consumption of network hosts. The research in this contribution area focused on system-level prototyping and evaluation of a TCP connection proxy, a SIP proxy, and a Gnutella P2P proxy.
2. The first use of pre-computation and hierarchical structures for reducing the false positive rate of a Bloom filter. The research in this contribution area focused on algorithm design, implementation, and analysis, including a study of a signature-based hash table compared to a Bloom filter.

3. One of the first applications of Bloom filters to measurably reduce the computational requirements of keyword search, in this case applied to a Gnutella P2P proxy. The research in this contribution area focused on system-level prototyping and performance measurement in the scope of the Gnutella P2P proxy from contribution (1).

Contribution 1) was published in [69], contribution 2) was published in [70], [71], [68]. Contribution 3) was published in [67].

1.4 Organization of This Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 contains background and a literature review. The chapter describes current work in power management and proxying, and reviews Bloom filters and string search methods.
- Chapter 3 describes the design, implementation, and evaluation of an NCP for proxying TCP connections for SSH and IM, and SIP for IP phones.
- Chapter 4 describes the design, implementation, and evaluation of new search algorithms that could be used to improve energy efficiency of a proxy for a P2P application. The algorithms are: the Best-Of-N and the two-tier Bloom filter, which are search algorithms for probabilistic Boolean responses. Also, a data structure and a search algorithm that allow information retrieval are presented. They are: Signature Array Hash Table (SAHT) and Bloom filter based keyword search, respectively.
- Chapter 5 describes the design, implementation, and evaluation of two types of a proxy for a P2P application. The first is the P2P Stateless proxy, which covers for a P2P application running on a sleeping host. This proxy requires TCP connections to be closed and re-established each time the host changes from a sleep to an awake state and vice versa. The second is the P2P Stateful proxy, which covers for a P2P application by keeping the TCP connections established by the host open for as long as the host sleeps. The Stateful proxy combines the ideas of TCP connection proxying in Chapter 3 and uses the Bloom filter based keyword search algorithm from Chapter 4.
• Chapter 6 concludes the dissertation by describing benefits of the contributions presented, including calculations of potential energy savings. It also describes possible directions for future work.
Chapter 2: Background and Literature Review

To understand how energy can be saved, it is necessary to identify what is driving power consumption in network hosts. This is necessary to fully understand how the ideas in the following chapters contribute to reducing energy consumption of network hosts.

2.1 Overview of Power Management for ICT Hosts

Power management in hosts is the addition of hardware and software features to reduce their power consumption when not being used [104]. An early version of this concept was used in [99], where Newsham and Tiller proposed the use of stickers pasted on top of office computers to persuade users to save energy by turning off computers when they leave the offices. Power management policies have been proposed through different approaches. Those that have had the most impact are proposed by efforts or groups, usually composed by government agencies and companies with high influence. Solutions to improve energy savings through power management have been also proposed by independent researchers.

During the last two decades several groups involving industry, government and academia have been formed to propose ideas for power management of ICT hosts. The Advanced Power Management (APM) specification was the first attempt of computer manufacturers to enable power management capabilities in PCs [2]. The first version of this specification was created in 1992, and the last one in 1996. This specification was an effort between Intel and Microsoft, and the idea was to develop an interface between the operating system and the BIOS to achieve power management. This development includes the definition of power states, which are the basis of the power states used today in power management. An improved specification was released in 1996 under the name of Advance Configuration and Power Interface Specification (ACPI) [1]. In this specification, a group of power states are defined, from which states S0 to S5 can be found in almost any device today.
The most important states are as follows:

- **State S0** is for Working (when the power consumption is the highest).
- States S1 through S4 are sleeping states, differentiated by the components that remain powered-on. In S3 (Suspend), main memory is still powered-on, while in S4 (Hibernation), main memory content is saved on a disk, and all the components are powered-off.
- **State S5** is called Soft Off (when the power consumption should be the closest to zero).

In states S1 through S4, power consumption should be the closest to that in S5. This specification describes the structures and mechanisms necessary for the power management to be directed by the operating system. Hardware and software in a computer use this specification to comply with power management policies. If a device is on (or in S0 state according to the ACPI power states), the operating system running on a device with power management policies enabled will always log time-stamps of the last time a user input was detected. It will start a timer and if during a certain period no new input is received, it will move to a sleep state. This period of time is called power management delay time by the authors of [76]. A technique that focuses on shortening the power management delay time (or saturating it) was proposed in that publication. To shorten the delay time means that the PC will wait less time to go to sleep. The authors of the publication calculated potential energy savings by shortening power management delay times in office PCs, printers, copiers and displays. To do this, delay times were varied between 5 minutes and 60 minutes. The energy savings achieved in periods during which the devices stayed in power saving states were about 4 times more when the delay time was set to 5 minutes instead of 60 minutes. However, one of the findings of the study is that many users got annoyed by small delay times and decided to set delays of about 60 minutes. The authors suggested the study of the use of technologies that could diminish the user annoyance.

There are several public agencies and laboratories dedicated to proposing and implementing power management policies. The U.S. government through the Department of Energy created the Energy Information Administration (EIA) and the Lawrence Berkeley National Laboratory (LBNL). The EIA has been in charge of publishing many forecasts and analysis regarding energy use in the country and its impact on the economy [38]. LBNL is one of the laboratories that have been more dedicated to the promotion of power management in computers. The report in [104] is a clear attempt to promote power management in PCs and monitors. Other reports like that in [103] and [114] have calculated potential savings by the
aggressive use of power management policies in office equipment. However, LBNL presented another report in 2001 where the authors found that despite these efforts, only around 44% of computers were turned off at nights [138] during informal walk-throughs in offices. The U.S. government also created the Environmental Protection Agency (EPA) in 1970. The EPA created the Energy Star program in 1992 as a program where manufacturers of electronic devices could voluntarily label their products as energy efficient [37]. The program started with the issue of an Energy Star logo for PCs and monitors manufacturers that comply with given specified requirements. These requirements issued by Energy Star for PCs are currently in specification version 5.0 [112]. The specification classifies the PCs according to the hardware parts that compose them. According to the classification, computers must consume less than a posted amount of kWh a year to be issued the Energy Star logo. The logo is stamped on these devices, and it is promoted by EPA with monthly lists of the devices that bear that logo. The results of this program have been evaluated in a report [119]. This report evaluated all devices that bear that logo and made calculations of energy savings since the first specification for each type of device appeared.

Private groups and laboratories from academia have also been involved in efforts to increase energy savings of the Internet, and especially PCs. University of South Florida and University of California Santa Barbara among many others, have contributed with their projects [131], [132]. Groups like the Climate Group have focused on reducing the carbon print of business and government in general. Reports from that group, such as “Smart2020”, have given a broad idea of how ICT can contribute to decrease the energy use of each country [125]. Another group is the Climate Savers Computing group, founded by Google and Intel in 2007 [30]. This group and others like the Green Computing Impact Organization [50], are private efforts by consumers and business to reduce the power consumption of enterprises.

A key reference in power management is the work by Gupta and Singh in [54]. They addressed the power consumption of the networking devices that connect the Internet and proposed different solutions. They started by dividing the energy use by all the components: hubs, switches, and routers. They described how devices spent too much time being idle, thus being energy inefficient for the whole network. They proposed the modification of the protocols used by the Internet to enable more power management policies in the devices that are connected to it. To decrease energy use of the devices, they proposed modifying
them in such a way that some components can be turned off when not in use, or at least put in power saving states. This leads to energy use rates more proportional to the load of the devices.

The case of energy proportional computing in servers was studied in [12]. The authors proposed the modification of how servers are designed, such that they consume energy proportionally to how much they are used. They suggested that manufacturers should focus on building components which have energy proportionality in mind. This is key to saving energy during periods where the utilization is low, but the power consumed is already high. The idea of energy proportional computing is the final goal of power management. Figure 2.1 shows power consumption versus utilization, where the typical curve depicting the power consumption as the utilization increases is compared with the idealistic one (the straight line). Ideally, a PC should consume no power when it is not being used, very low power when it is barely used, and in general, a proportional amount of power to utilization. Today, PCs consume marginally lower power when idle or barely used compared to when fully used. As stated in [12], the system components are still not achieving energy proportional computing together. In the hardware level this is hard to achieve, as the system component manufacturers are different. PCs should consume power proportionally as they are utilized, and solutions can be explored that go beyond what the components of the systems can do, and this is the goal of this dissertation.

Dynamic Power Management (DPM) is a design methodology that configures an electronic device to be able to provide certain services with the use of the minimum necessary components, acting at its
minimum possible load [13]. This is key to consider, given that systems do have different workloads during different periods of time. A survey presented in [13] describes power-managed systems by first defining and giving examples of power-manageable components of a system, and how they interact to configure the system to be power-managed. At the same time, the authors of the survey developed a mathematical framework to highlight advantages and disadvantages of DPM techniques. They also described different DPM techniques for system components, and how they were currently implemented.

### 2.1.1 Power Management in Wireless Devices

Power management in networks composed by wireless hosts has been studied more thoroughly than networks of wired hosts. Most of the wireless hosts are mobile hosts and as such do not have an endless source of power (like wired hosts which are plugged to power outlets) but instead relay on batteries. This makes power management a mandatory topic of research. The research community has long been interested in methods to reduce power consumption in order to increase battery life. According to [72], research has been focused on improving the energy efficiency of hardware components of wireless hosts, but of interest is what can still be done to allow software components to contribute to the energy efficiency of the system. The authors describe energy efficient network protocols through the different layers of the protocol stack. They describe, for example, how different protocols and standards in the MAC layer were designed having the energy efficiency of the hosts in mind. In the case of the 802.11 standard [57], it was proposed what could be done in case the host wanted to save energy. In the transport layer, the protocols include modifications of the TCP protocol to make it aware of the wireless environment.

The design of applications usually pays no attention to the power needs of the host when the application is running. Critical energy resource in mobile computing is drawing the attention of researchers towards how applications are not aware of power consumption and resources available. The authors of [36] proposed that operating systems should be designed to allow applications participate in the power management policies of the device. The publication presented an application developed for a Palm Pilot. The intention was to establish what kind of information the operating system could give to the software to improve the energy efficiency of the device. The study was based on power consumption measurements when the application performed different tasks. Although the target system was a mobile device, this idea could be used in all types of devices.
One approach that has been explored is to have a hierarchy of hardware components ranging from high-power and high-function at the bottom, to low-power and low-function at the top. The low-power components do work for the high-power lower tiers, and thus allow the high-power components to sleep. One example of this is “Wake on Wireless” where a low-power out-of-band wireless component is used to wake up a (relatively) high-power PDA [122]. Another example is a laptop with three levels of hardware where the lower-power components perform simple network tasks (such as keeping email up to date) allowing the higher power components to sleep [126]. The term “proxy-based architecture” is used in [126]. The work in Chapter 3 of this dissertation is more general in scope than that of [122] and [126].

In general, aggressive power management ideas common for wireless hosts are not usually found in wired hosts. One reason for this could be that the wireless community is relatively new, which allowed designs of protocols and standards to take into account issues like power consumption. The adoption of new designs in already deployed networks of wired hosts, such as those that compose the backbone of the Internet might not be an easy to adopt approach.

2.2 Induced Power Consumption in Network Hosts

Increasingly, networked applications have been found to “induce” energy use in PCs and other networked devices. Induced energy use occurs when devices are required to remain fully powered-on – even when no user is present and network access is only sporadic or incidental – in order to respond to network protocol messages [102], and has been considered as a driver of energy use in hosts [26]. One example of an application producing induced energy use is peer-to-peer (P2P) file sharing. A desktop PC, or host, participating in a P2P network must be fully powered-on at all times in order for it to make its files available to other P2P hosts. This is the case even if the actual time during which files are downloaded from a P2P host is very small. The participative notion of this application forces hosts to remain connected to contribute to the network. Some applications that use the client-server model also induce energy use. Users may prefer to leave applications running all the time to receive updates, notifications and/or services.

Many application messages flow over TCP connections, a fact that makes network connectivity harder to maintain, because TCP uses sequence numbers. For a third party to maintain a TCP connection open, it would have to know the sequence number of the last packet. If the right number is not used, the connection will be dropped by the other side. Not only will the application layer require constant connectivity, for
messages like keep-alives, but in some cases the transport layer will also require it because TCP is used. The main purpose of TCP is to assure delivery of bytes in both sides of a connection. For that reason the protocol requires both sides to be responsive at all times. Otherwise retransmissions, back-off times and eventually connection drops will occur. Some application messages flow over UDP, in which case the network connectivity problem is at the application layer only.

2.2.1 Client-Server Applications as a Cause of Induced Energy Use

Three examples of client-server applications that drive the waste of induced energy use are described here. Secure Shell (SSH) is an application protocol used for secure remote login and other services over an insecure network [141]. It is used as a secure telnet replacement for remote access to a terminal console. SSH uses a TCP connection. If the TCP connection is dropped, the application state is lost in the remote computer. In addition, a lengthy re-login process is required to reconnect. As a result, users with SSH connections typically disable power management in their local (client) hosts. During an active SSH connection, messages can flow from the remote computer to the console. For a host that has a SSH session open to be able to sleep, the TCP connection must be preserved and any messages queued for later display on the console window in the host (that is, for when the host wakes up). Otherwise, the host will have to remain awake.

Another application is Instant Messaging (IM), a communication that is established in real time. It is often used in multi-way chat sessions. IM uses a TCP connection from IM client to an IM server, so it uses the typical client-server architecture. A study shown in [65] provided an overview of the communication architectures used by different IM clients. They studied the technologies used by AIM, MSN Messenger, and Yahoo Messenger. In order to allow millions of IM clients to be connected to the servers at the same time, the architecture has to be designed to be scalable. This leads, according to the study, to two types of approaches: a symmetric and an asymmetric approach. In a symmetric architecture, each server performs the same functions so that it is irrelevant to the client to which server it connects to. In an asymmetric architecture, each server performs a different function, like logging in, discovering other clients on the network, maintaining a chat room, or forwarding a message from one client to another one. From the IM architectures studied in the paper, the simplest is the one used by Yahoo Messenger, shown in Figure 2.2. The client performs all the functions over persistent TCP connections established with one of the Yahoo
servers. Three types of messages are sent through those connections: 1) login messages, 2) control messages and 3) IM (chat) conversation messages. If a client TCP connection is dropped, the client is removed from the multi-way chat and any IM messages sent during the disconnected period will not be received by the disconnected client. For an IM client to be able to sleep and not lose incoming messages during its sleep period, the TCP connection must be preserved and any messages queued for later display on the IM client application in the host, similar to the above described SSH case.

The third example of applications that drive induced energy use is Voice-over-IP (VoIP) phones. VoIP is a term used for a group of technologies that deliver voice communication over IP networks instead of the common phone system. Soft-phones and hard-phones are used to deliver the communication. Soft-phones are software applications that implement VoIP using a host. VoIP hard-phones are specialized telephones for VoIP use only. VoIP hard-phones (or just IP phones) are replacing conventional telephones in companies. VoIP phones represented about 27% of the total installment of phone lines in 2005, according to [111]. About 25 million IP phones were sold that year worldwide, according to the same source, and IP phones were being sold more than conventional telephones. Only recently has the environmental impact of using IP phones as a replacement for landlines been studied. An example is the analysis shown in [24] where the authors explained the issues to be taken into account when moving from conventional landlines to VoIP. One of the issues was power consumption. Calculations were made for the whole VoIP infrastructure, consisting of a VoIP gateway, a router, a call manager and the hard-phones used. Energy requirements for three scenarios were calculated. The first two scenarios were a mix of IP phones and two types of conventional telephones, and the third one used only IP phones. It was shown that the third one consumed 2.5 times more power than the first scenario and 2 times more power than the second scenario.
The interest of this dissertation is on the IP phones only, not on the core of the VoIP network. IP phones manufacturers include power saving states in the phones so that they consume less power when idle. Still, with these states, the estimated average of 6 W per IP phone can be considered to include periods of idleness or actual use.

The power consumption of IP phones can be calculated as follows. An IP phone consumes an average of 6 W when active, according to a report from a private company that measured and compared Cisco and ShoreTel IP phones power usages [133], and from a white paper from Cisco [22]. This average is likely to increase because phones are being designed with features that consume more energy. Assuming that companies apply policies to turn off phones during nights and weekends, IP phones can still be powered-on about 12 hours a day during weekdays. There is no precise source to determine actual use of IP phones (that is when used to specifically make or receive calls). It can be assumed that phones are in active use for only 5% of the time. So, if one assumes phones are used a 1/2 hour per day, the energy that IP phones are wasting is then the energy used during 95% of the time (when not used for calls). According to the same white paper from Cisco [22], there is no significant difference in power usage when being active or when being idle. A single IP phone then consumes 18.7 kWh a year, which costs $2 dollars at $0.10 dollars per kWh (based on 12 hours a day being powered-on during 5 days a week). The power consumption could be calculated for the phones of a company. It can be assumed that a company has 10,000 IP phones. A total of 187 MW are needed to power up the phones and the company spends around $18,000 dollars a year to pay for that energy. The annual worldwide shipment of IP phones (around 25 million according to [111]) consumes about 467 GWh a year. This clearly shows the impact of the power consumption of IP phones. Because IP phones are idle most of the time, most of this energy can be saved.

Besides the power consumption of IP hard-phones, the soft-phones also drive the power consumption of the hosts on which they are running. For a call to reach the soft-phone, the host has to be running all the time. There are cases like Magic Jack [87] where even the soft-phone disables power management options of the host. Magic Jack is a physical device (shown in Figure 2.3) that can be plugged into a PC, and runs a soft-phone. It uses USB port to connect to one of the USB ports of the PC. It also has a port to connect a conventional telephone handset. When the device is connected to a PC, it installs a soft-phone that implements the SIP protocol for call session control. The hardware inside of the device creates an interface
between the software installed in the PC and the conventional telephone. The manufacturer of Magic Jack provides interconnection with the telephone network at the same time, which allows the device to be used to make and receive calls.

The VoIP phones use several session control and communication protocols. Typical protocols used to control the call session are H.323 (standardized by the ITU-T), SIP (standardized by IETF), and the Skinny Call Control Protocol (SCCP) (proprietary protocol from Cisco). The common protocol used by IP soft-phones and hard-phones is the SIP protocol. H.323 and SCCP are not implemented in soft-phones. Session Initiation Protocol (SIP) is an application layer protocol used to establish, modify and terminate a communication session (or call) between two or more participants [117]. The participants are called User Agents (UA) and use SIP service providers as intermediates. UAs register with SIP registrars located at the domain of the provider and a SIP proxy server (located at the same domain) will route requests to/from that UA. The SIP protocol is based on an HTTP-like request/response transaction model, where a UA participates as client and another UA participates as server. A SIP communication between two UAs (Alice and Bob) is shown in Figure 2.4 and is established in the following way. First Bob has to register to a SIP service provider (server.com), step (1). The registrar stores Bob’s information in a Location Service (2). If Alice wants to contact Bob, Alice will have to know the address of Bob. Alice sends the request to the proxy server of server.com (3). The proxy queries the Location Service to find the exact location of Bob (4) and (5), then sends the request to Bob directly (6). After the communication is successfully established, Alice and Bob will establish a Peer-To-Peer communication using protocols to transport voice and video like Real-time Transfer Protocol (RTP). The SIP session is established and remains so until one of the ends terminates the communication. It is important to note that although the registrar and the proxy server of the
A typical SIP message contains message header, method, status code, and other fields. The message header is used to provide details about the SIP session. Details include content type, length, language, identification of the session, and so on. The method is a function the UA likes to invoke in the SIP proxy server. Typical methods are REGISTER, INVITE, and BYE. The REGISTER method is used by a UA to register to the SIP registrar of a certain SIP provider. An INVITE is sent by an UA when it wants to establish a communication with another UA. A BYE method is used to terminate a communication. The status code contains the correct status of a call. Possible statuses are grouped as Informational, Client-error, and Server-error. If no error occurs, statuses are just of the Informational type, and include TRYING, and RINGING. A TRYING status is set in the first SIP message sent by Bob when it is contacted (it receives the INVITE). A RINGING status is set when Bob’s phone is actually ringing. The initialization of a SIP call (a Caller to the IP phone) is shown in Figure 2.5. This figure includes the last hop router the IP phone is connected to.

One of the available fields for the message header of a SIP message is the expiration field (Expires) that establishes the relative time after which the message (or content) expires. The meaning of this

Figure 2.4. Components and Start of a Communication with SIP [117]
expiration is method dependent. In the case of a registration, it is used by the UA that wants to register (establish a binding) with a certain SIP registrar. The Expires is set to negotiate the expiration time of the binding to be established. The SIP registrar then agrees with the requested expiration time or sets it to a new value. The SIP protocol defines a way for SIP registrars to remove unused bindings from their system. The Expires field of the header is used for this purpose. So each IP phone with a binding at the registrar will have to re-register after the expiration time. Assuming the IP phone is connected for the first time to the network (e.g. turned on), the first thing it will do is registering with its SIP registrar.

2.2.2 P2P Applications as Cause of Induced Energy Use

A P2P network is an overlay network on the Internet. P2P hosts (or peers) are directly connected to a group of their neighbors by TCP connections. Neighbors need not be physically nearby, and the process of identifying neighbors involves a bootstrapping process. P2P networks are typically used to share media files, but are also used to share computational resources. File sharing applications became popular in the early 2000s. A relatively new study [73] showed that the use of file sharing applications is still widespread in the Internet. A study presented in [129] found through traces that although a low percentage of P2P users leave the network after a few hours of being connected, about 30% remain with high probability of being connected for more than 16 straight hours. This shows that P2P nodes remain connected to the network for long periods of time.
P2P networks are used not only for file sharing but for other network applications. A survey of P2P architectures is presented in [85], where the authors also discussed possible future uses of P2P networks, including P2P overlay computing, mobile location-based services and mirrored content delivery. The study focuses mostly on the comparison of structured and unstructured P2P networks. In structured networks, the topology is controlled and content is placed in specific locations that will make query routing more efficient. In unstructured networks topology is not controlled and content can be at random places. In unstructured networks the peers participating need to use more of its resources to make the network work. Constant participation in the routing of queries, responses to queries, and traffic control messages is important. Unstructured networks are more commonly used and the Gnutella network is a good example. The well known software Skype uses a P2P network to share traffic routing of the VoIP calls among the peers that are connected to the network. According to classification of architectures presented in that publication, the Skype peers construct an unstructured network. According to [52], Skype uses an ultrapeer-leaf architecture similar to that used by Gnutella. They use the concept of superpeer (supernode) and ordinary peers (nodes), which construct a 2-layer overlay network. The network allows peers to transfer files to each other, send IM messages and conduct VoIP calls. The P2P network is used to send control traffic, IM messages, and requests to initiate VoIP and file-transfer sessions.

It is important to have an idea of how P2P applications can drive power consumption of the hosts they are running on. A P2P application used to share files is nearly idle 99% of the time. Here idle is considered as not uploading a file. This is calculated as follows: A typical P2P host will be downloading or uploading files only 1% of the time (and thus is idle 99% of the time). We calculate this as follows. There are 1 billion downloads per week from 9 million users online at any time [105]. Thus, the average P2P host transfers about 16 files per day. Two different studies calculate the average file size between 4 Mbytes [130], and 8 Mbytes [43]. If a 1 Mb/s download rate is assumed as typical [120], then about 17 minutes per day are spent transferring files, which is about 1% of 24 hours. However hosts running P2P applications must remain “on the network” so that other P2P users can query the host to learn if a requested file is being shared.

Gnutella is a P2P file sharing protocol that uses smart query flooding to find files in the network. It is the most common protocol used to implement file sharing applications because it is open source and the
standard is not complicated. One of the most used P2P applications implementing Gnutella is Limewire [84]. The simplicity and popularity of Gnutella make it a desired target to implement a solution to save energy for hosts running P2P applications.

2.3 Overview of the Gnutella P2P File Sharing Protocol

The first standard Gnutella protocol version is 0.4 and defines five message types; Ping, Pong, Query, Query Hit, and Push [47]. After that, a version 0.6 was published. The main modification of this version is the introduction of ultradeers [48]. Gnutella is a protocol that has evolved after that, although no other version was published. Limewire has introduced modifications to the protocol and usually publishes them in their webpage. Peers can be leaf peers or ultradeers. A leaf connects to ultradeers and shares with it the list of shared files. By default, all new peers connect to the network as leaves. Different applications implementing Gnutella have different ways to select ultradeers. The most common way is by using a selection criterion, such as the time the peer has been connected to the network, or the upload and download speed available. If the criterion is satisfied, the application advertises itself to the network as an ultradeer.

A handshake occurs between two peers when one wants to enter the network. After that, the exchange of Gnutella messages occurs. Files are downloaded from a host using the HTTP protocol. Thus, each Gnutella host is also an HTTP server. The details of the messages are the following:

• The Ping message is used to discover Gnutella hosts in the neighborhood. It is sent to the closest peers of the host sending it, which forward it to their peers. It is one of the first messages sent by a host when it connects the network, and is mainly used to build information about the neighborhood (reachable hosts, bytes shared, etc.) It is also used by some implementations to test if a connection is still alive or not (the pinged peer might not be reachable anymore).

• The Pong message is sent in response to a Ping. It contains information about the host sending it, such as uptime (time it has been connected to the network), amount of bytes shared and if it is willing to receive connections from the host that sent the Ping.

• The Query message is used to find files; a host puts keywords in it and sends the message to its neighbors. The neighbors forward the message for up to a maximum number of times, specified in the TTL field of the Query.
• The Query Hit message is the response from a Gnutella host that received the Query that contains a file or list of files that contain a combination of the keywords in the Query received.

• The Push message is used by a Gnutella host that wants to download a file from a host that is behind a firewall. The host that receives a Push has to now open a new TCP connection with the host that sent the Push. With this connection, the host can request the file from the firewalled host. Other approaches are used if both hosts are behind a firewall.

A host running a P2P application that implements the current version of Gnutella works in 5 steps, from when it connects to the network for the first time until it closes the connections to all the peers it is connected to:

1. A host entering the network connects to at least one peer (from a list received from a bootstrap). From this peer, it will get a list of other possible peers to which it can connect.

2. The new peer explores the neighborhood to discover new peers willing to accept new connections from it. After the maximum number of connections allowed is achieved, the host will just stay connected participating in the network. If the peer acts as a leaf, it will deny attempts from other peers to connect to it. If it is acting as an ultrapeer, it will accept new connections from other peers (ultrapeers or leaves) as long as the maximum slots of peers has not been filled.

3. The peer responds to messages accordingly, and forwards Queries or responds to Queries received from other peers. In case the user generates a search for a set of keywords, it will also submit a Query with the keywords to the list of peers is connected to. It also responds to Pings messages.

4. The peer sends a download request to one or more peers from the list of peers sharing a desired file. It also has to accept upload requests of files it is sharing.

5. The peer repeats steps 3 - 4 or disconnects from the network if the user triggers an action to do so.

2.3.1 Query-Based Protocol

The most important responsibility of a peer in a Gnutella network is to process Query messages by forwarding them if necessary or responding to them if the peer shares one or more media files whose names contain one or more keywords of the received Query. Figure 2.6 describes the timeline which shows how the process of querying for keywords, responding to the Query, and requesting a file works. P2P host A and B are peers connected to a set of peers that do not necessarily have peers in common. In the figure, A is
connected to ultrapeer X and B is connected to ultrapeer Y. P2P host B shares a file named X. It is assumed that Host B is not located behind a firewall. The process happens in the following way:

1. Host A generates a Query message that is sent over the TCP connections established with other peers of the network. The Query is assumed to have a keyword found in the name of a file shared by host B. The message eventually reaches host B after being forwarded by its peers.

2. Host B spends some time to match the keyword with one or more of the files it shares. It then responds to the Query with a Query Hit containing the file X. The Query Hit can be sent over the same TCP connection used to send the Query to host B, or sent directly through UDP to host A. In case it is sent over the TCP connection, it will be forwarded over the same path that was used to send the message from host A to host B. In the example, the message is sent over UDP directly.

3. Host A requests the file X by sending an HTTP request over a new TCP connection and the download starts after host B accepts the request.

**2.3.2 Tasks of a Leaf Peer and Ultrapeer**

While the peer behaves as a leaf, it has to perform several tasks as participant of the network. These tasks have to be performed no matter whether the P2P application is being used (there is a user running
commands) or not. They are independent of other tasks that are only performed if the user wants to (e.g. searching, manually connecting to peers). The tasks of a leaf are the following:

1. Respond to Query messages received only if it shares files that contain the keywords in the Query.
2. Accept or deny requests to upload files to hosts that request a file from it.
3. Establish new TCP connections to hosts that send Push messages to it. This happens when that peer is behind a firewall and wants to download a file from the leaf.
4. Respond to Ping messages received from other hosts with Pong messages containing information about them.

An ultrapeer on the other side helps to diminish the amount of traffic generated by Gnutella networks. It can accept many connections from leaves and routes their Query messages and Query messages from other ultrapeers, based on the knowledge they have about the files shared by leaves. Once a leaf becomes an ultrapeer, it will have an extra set of tasks to perform. It will continue with the first tasks, but now it will also:

1. Accept or deny new connections from other ultrapeers and possible new leaves. The P2P application has a maximum number of connections so that bandwidth and performance of the host is not affected.
2. Retrieve the list of keywords from the files that each of its new leaves are sharing. It will be used to decide to which peer each received Query can be forwarded.
3. Forward Queries received by its leaves or ultrapeers, if the TTL of the message allows it. Queries will be forwarded to its leaf only if a searched keyword is in the list of keywords for the host to which the message will be forwarded.

2.4 Existing Approaches to Save Energy Wasted Due to Induced Use

A technology was invented in 1996 by IBM to remotely wake up PCs without the need to manually turn them on, and was popularized later through an alliance with IBM. The technology is currently known as Wake-On-LAN (WoL) and enables PCs to be woken up by an administrator. The idea consists of using a packet that should trigger the PC receiving the packet through the network to power up from a powered-off state or a sleeping state. This implies that the NIC of the PC has to remain on all the time in order to detect
the packet. To support this technology, motherboards were modified to keep NICs powered-on when the
PC is turned off or sleeping. Manufacturers produced NICs that could detect the patterns used to wake up.

AMD invented a technology called Magic Packet [8] as a packet with a specific pattern to trigger wake-ups. The format of the Magic Packet is shown in Figure 2.7. It contains in its payload 6 bytes where all bits are set to 1, (six times the hexadecimal FF), followed by the MAC address of the PC to be woken up. The packet is sent by an administrator in a MAC frame to the broadcast domain of the PC that is sleeping. The NIC of the PC to be powered-on will detect and recognize its MAC address in the Magic Packet received and will power up the PC. Because there is no way for a PC to know the MAC address of a PC that it is outside of the same LAN it is on, there is no way for a PC to use WoL to wake up a PC outside of its domain. This is the most important limitation of Magic Packet. Routers are usually set to discard direct broadcast frames, so they are not forwarded outside of the LAN. This is done to prevent excess traffic in the network. APIs for NICs, like Network Driver Interface Specification (NDIS) used for most of NICs, have included a set of patterns that NICs can support to trigger a wake-up of the host. These include, among other patterns, TCP SYN packets in IP version 4 and 6, and Magic Packet. The problem with the use of patterns to trigger wake-ups is that the host might wake up too often for TCP SYN packets directed to ports not used by any of the applications running on the host or by applications not running by the time the SYN packet is received.

There have been several efforts to improve reachability of PCs beyond the possibility of being simply woken up. One approach is to enable NICs to respond to ARP packets even when the PC is sleeping. Ethernet NICs that support DMTF’s Alert Standard Format (ASF) Specification 2.0 [5] can respond to ARP packets. Intel has developed a series of technologies implemented in some series of their motherboards to improve wake-up. One of them is Intel vPro targeted to companies that wanted to improve remote manageability of computers inside of their network [61]. The most important feature was to manage
information directly obtained from the PCs even if they were turned off or the operating systems were not working properly. The problem with this improved wake-up is that it implies the use of new hardware (Intel motherboards) and requires modifications of existing applications to interact with Intel. Usually software vendors wait to see if the technology remains in the market to consider modification of their products. Intel also created Remote Wake Technology (RWT) [60]. This technology allows applications running in a PC with an Intel motherboard that has this feature to allow remote wake-ups to the PC using an interface provided by Intel. The problem with this technology is that applications that want to allow the user to remotely wake up the PC have to be modified to interact with the Intel motherboard to allow packets from this application to wake up the PC. Two examples of applications given by Intel are a VoIP application and a remote management application. This technology is then limited to the group of applications modified to work with the Intel RWT API.

A series of companies have created their own improved version of wake-up technology. One company is VERDIEM, which has several power management products. The most important is a solution called Surveyor [137] that uses the Intel vPro technology. It is an infrastructure focused on power managing PCs in a company. The architecture, shown in Figure 2.8, consists of a server that will receive wake-up requests from IT staff or from remote requests from employees. They invented the concept of Wake-On-WAN, which relies on a Surveyor proxy to directly wake up PCs using Wake-On-LAN packets. This proxy receives requests from the Surveyor server, which routes the requests. Another company is 1E which has focused on power consumption of network hosts and has several commercial solutions. They focus mainly
on enhancing wake up capabilities inside of the company network, by assuring remote wake up is possible and by keeping track of whether the wake up attempt was successful or not. At the same time they have been publishing surveys about power consumption and power management policies in office equipment [109].

There are other existing works based on putting hosts (or components of them) to sleep and waking them up when needed. An example presented in [3] is an approach to manage power consumption of mobile devices that run VoIP applications over Wi-Fi. Wi-Fi interfaces represent a large percentage of the power used by mobile devices. Due to the strongly constrained batteries of mobile devices, especially cell-phones, the authors used the approach of turning off the Wi-Fi and turning it on only when necessary. The authors developed an architecture called “Cell2Notify” that wakes up the interface when an incoming VoIP call is detected. For this purpose, it uses the cellular network to detect incoming calls.

2.5 Network Connectivity Proxying (NCP) for Power Management

An NCP is an entity that maintains full network presence for a sleeping network host [100]. Figure 2.9 shows the fundamental concept behind the NCP – the ability of the NCP to “cover” for a sleeping host. The figure shows how state and control is transferred between the host and NCP when the host enters and exits the sleep state.

The operational steps are

1. The host determines that it is time to go to sleep (e.g., based on inactivity),
2. Notice and state are passed to the NCP, and the host goes to sleep,
3. The NCP maintains full network presence, responding to and generating, protocol and application packets as needed,

4. The NCP determines when a packet requiring the full resources of the host has arrived and signals the host to wake up (the host could also wake up on user activity or from an internal timer), and

5. Once the device has fully woken up, state is passed back from the NCP to the host, and the host returns to normal fully powered-on operation.

This dissertation is focused on open problems and questions related to the NCP concept. For example:

1. What are the requirements for the NCP to maintain full network presence for a sleeping host?
2. What is the type of information that the host needs to pass to the NCP for it to maintain a presence?

The use of “performance enhancing proxies” (PEP) to mitigate link-related degradations is described in RFC 3135 [19]. A means of handing TCP connections during periods of disconnection is described whereby a proxy would “zero window” a server host during the disconnection period. Handling optionally reliable links and hosts is further considered under the scope of Delay-Tolerant Networks (DTN) [39].

Proxying for maintaining networking connectivity for sleeping hosts was first explored by different authors in the mid-1990s for shared Ethernet networks [25]. This work was later refined for IP networks in general [28], [53], [100]. A specific goal of being able to proxy low-level tasks including ARP, DHCP, and ICMP was explored. A simple prototype was developed and is described in [53]. In [7] an initial exploration of the architectural constructs required to support selective connectivity was presented. Selective connectivity is the notion that a host can choose the degree to which it maintains a network presence, rather than today’s binary “connected” or “disconnected”. A key architectural construct to support selective connectivity is an assistant that stands in for a host that is sleeping. Thus, [7] begins a more mature thinking of the long-term implications of network presence and how to support it in future networks.

Recently, University of California San Diego (UCSD) has begun to prototype a scheme called Somniloquy whereby a secondary low-power processor covers for the main processor of the PC [4]. The prototype has been developed on a USB-based gumstix device. Somniloquy appears to be a general purpose architecture that can also support applications via application stubs. Somniloquy builds upon many
of the ideas in [28], [53], and [100]. Somniloquy is not able to preserve TCP connections between host awake and sleep, which might be needed for some applications that relay user sessions. The future direction for Somniloquy, and possibly for the work in this dissertation, is to host the NCP on a NIC. NICs are becoming more capable, and many now include onboard processors. Other works on proxying like that of Nedevschi [97], has already started to look at the classification of packets arriving to hosts, presenting different approaches to classify them. The most important approach was to deconstruct types of packets by grouping them into multicast and unicast packets. The grouping was a basis for determining policies about how to process packets (if responses were needed, or if not, how urgent the packets were, and so on).

TCP Chimney is a Microsoft NDIS 6.0 architecture for full TCP offload [89]. TCP Chimney provides a direct connection between applications and an offload-capable NIC to reduce TCP-related processing load on the host CPU. This enables the NIC to perform TCP processing for offloaded connections, including maintaining the protocol state. TCP Chimney does not explicitly address network connectivity beyond that of TCP connections. Other works have attempted to make application connections robust without the direct intention of proxying the connectivity for power management. One approach is by creating an extra layer between the application and the transport layers. The early work by Zhang and Dao [143] and the work by Zandy and Miller [142] created a special library that adds another layer to the communication stack, which makes the connections reliable and independent of mobility. Others have focused explicitly on making TCP connections reliable. The targets of these solutions have usually been mobile devices, where failures in the transmission might occur during the connection [6], [35].

2.5.1 Early Work in Proxying of ARP

Proxying for applications or network protocols has been done for a few decades. In 1987 the idea of implementing a proxy functionality inside of a router to proxy for ARP was published in RFC-1027 [21]. The original purpose was to be used by subnet routers to allow hosts to communicate without being aware of the existence of subnets. This was called an “ARP proxy.” The mechanism worked clearly with the example of a host A trying to communicate with another host B located in another subnet. Both subnets, however are connected by a subnet router. Host A will send an ARP to query for the physical location of host B. The router will respond on behalf of host B, with the knowledge that B is located in a subnet of which the router is aware. Thus, the router proxies the response of the ARP originally sent to host B, and
both hosts are unaware of the existence of the subnets. The same concept can be used by a proxy with the purpose of saving energy. The proxy can send ARP packets on behalf of a proxied host when it is sleeping. The work in [25] explained the design and prototype of a power management proxy server to mainly keep TCP connections open. One of the proposed extensions is to respond to ARP packets sent to the sleeping host, so that applications can reach the host.

2.5.2 Proxying for UPnP

The Universal Plug and Play (UPnP) set of networking protocols [136] is another example of a network application that demands constant connectivity of the hosts connected to the network. UPnP is intended to enable a simple and robust connectivity among different electronic devices, regardless if their network connectivity is wired or wireless. The most used protocol from this set is the Simple Service Discovery Protocol (SSDP), which is the service in charge of discovering network services. The protocol requires all devices participating in the network to be powered-on so that they can send and receive SSDP messages. Devices participating in UPnP network have the capability of offering a service, or being a control point. A printer can offer services like printing or faxing. A control point can make use of those services. Alive messages are sent by devices to advertise a service, discovery messages are sent to discover services. If a device is in a sleep state, it cannot respond to discovery messages and it will not be able to update the advertisement of a service.

In [81] a power management proxy for UPnP was designed and evaluated. This work describes the design of an invisible proxy and a cooperating proxy. An invisible proxy responds to discovery messages sent to a sleeping device, and periodically sends service advertisement messages on behalf of the sleeping device. It is invisible because it requires no modification of the devices being proxied and it does not advertise its presence. It is able to detect when a device is sleeping by checking the last alive message update sent by the device. It can take over a sleeping device by broadcasting an ARP packet to change the location of the IP of the sleeping device. Packets to that IP will be sent to the proxy as long as the device is sleeping. The cooperating proxy on the other side announces its presence to the devices connected to the network. This implies adding a power management capability to devices implementing UPnP. With that service, devices can tell the proxy it is entering or leaving a sleep state. The proxy will then start or stop processing messages on behalf of the device. A similar approach was used in [66] to create a power
management architecture for home networks, composed of different devices. One of the components of the architecture was an Energy-aware Plug and Play protocol used by clients and servers.

2.5.3 Current and Recent Efforts to Define and Standardize Proxying

Different groups have been involved recently in the attempt to standardize Network Connectivity Proxying. The Ethernet Alliance published a white paper in 2007 with the intention of defining proxying for Ethernet-connected devices [100]. The authors of the paper gave an introduction to the problem of energy waste of networks and then gave two possible solutions. They considered one feasible option to be “Encapsulating the intelligence for maintaining network presence in an entity other than the core of the networked devices” [100]. This white paper defined three approaches to implement proxying: 1) Self proxying, 2) Switch proxying, and 3) Third-party proxying. Self proxying means that proxying functionalities would have to be implemented on the hardware of the same host to enable energy savings. Switch proxying puts the functionality into the router to which the host is connected. Third-party proxying puts the functionality in another host that might be located elsewhere in the network other than the router to which the host is directly connected. The work of this dissertation can be classified as Self-proxying and Third-party proxying.

There is a current effort by Ecma International (an industry association dedicated to ICT standardization) to standardize the idea of proxying to support sleeping hosts, as explained in TC32-TG21 [34]. It is an ongoing work where the final purpose is to provide an overall architecture for proxies and provide details of how to proxy for key protocols. Key protocols can be understood as application protocols that are commonly used and that can be generating traffic from/to the host regardless if the host is being used or not. As part the work of the group working on the standard, the interest is also to develop use cases to understand what cases should be in the standard or not. This use cases should explain how to classify an application as key protocol, and how the proxy should act when the host is sleeping. This will help also to determine possible solutions where proxying might not be the answer. The group expects to create a specification that defines the information that has to be exchanged between the proxy and the host, what should be the capabilities of the proxy that the host should be aware of, and what should be the behavior of the proxy if it works on Ethernet or Wi-Fi. As part of this, requirements for network management protocols are being established, like ARP, SNMP, DHCP, IGMP, among others.
Implementations of the NCP for applications like P2P could be extremely constrained by the device on which they are running. With the objective of saving energy in mind, one possible option is a device that consumes much less energy than the proxied host. In this way, the proxy could remain powered-on at all time proxying for the host. For a P2P, processing of Query messages in the proxy is needed, with the purpose of searching the keywords and discarding Query messages that do not match files being shared. This requires keyword search algorithms and data structures that are optimized as much as possible for small and low-power consuming devices that could be used as proxies.

2.6 String Search Algorithms and Data Structures

Different string search algorithms and data structures are explained in this section. The problem consists of finding a string (or pattern) in a text (or set of strings). An algorithm to solve this finds a string by either responding with a Boolean variable if the string exists or not in the text, or by returning a set of all the occurrences of the string in the text. The algorithms can be divided into two types: algorithms that construct data structures over the text to be searched and algorithms that scan the text sequentially to find the pattern (Chapter 8 in [11]). Bloom filters and hash tables are signature data structures, because they store signatures of an original text, constructed by using hashing [11]. The resulting data structure can save space but adds limitations regarding the depth of the information they can represent. Hash tables and Bloom filters are handled with the balls and bins model, where $n$ balls are thrown into $m$ bins [92].

2.6.1 Hash Tables

A hash table is a data structure represented with a table that maps certain keys to their associated elements. Elements that are mapped (or inserted) can be of many different types, for example, strings. In this dissertation, implementations of hash tables will use strings as elements. Elements are mapped in buckets of the table with the use of an element key (also known as element signature). The element key has to be unique for each element mapped, and is later used to locate the element. The mapping is done by using a hash function that has the element key as input, and returns a bucket index of the array. Different element keys might produce the same bucket index, which is called a collision. At this time, different collision resolutions can be applied. Hash tables are used in applications where search time is an issue, because search time on hash tables takes $O(1)$ on average, regardless of the size of the table [31].
There exist different collision resolution approaches used by hash table algorithms. Insertion, deletion and search algorithms depend on what collision resolution approach is used. The most common approaches are the following:

1. **Chaining**: Elements are not stored in the buckets selected by the hash function but instead inserted into a list associated with that bucket. Buckets now are used to point to the list created with all elements mapped to that bucket index. When another element key collides in the same bucket, the insertion algorithm will just add that second element to the same list.

2. **Open Addressing**: Elements are mapped into the bucket selected by the hash function if the bucket is empty. If an element is to be inserted into a non-empty bucket, an approach is used to select the next bucket to be probed, until the next empty bucket is found.

The insertion algorithm then depends on the collision resolution approach used. Figure 2.10 shows the algorithms for mapping a string and finding a string in the hash table `hashTable` of size `b` buckets. The input of the algorithm `mapString()` is the key and the string to be inserted. The input of the hash() function is the key and it returns an index that is used to select the bucket to attempt an insertion. A global variable `collisionRes` is used to store the collision resolution approach selected. If Chaining is used (line 2), the function `insertAtList()` is used to insert the string at the end of the list associated with the selected bucket. If Open Addressing is used, (line 4) first, `newProbe()` in line 5 is called to select the index of the next available bucket. This function will use one of the approaches used to probe for empty buckets. Then the string is inserted into the bucket selected, calling `insertAtBucket()` in line 6.

<table>
<thead>
<tr>
<th><code>mapString(mapKey, mapString)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>index ← hash(mapKey) mod b</code></td>
</tr>
<tr>
<td>2. if (<code>collisionRes = CHAIN</code>)</td>
</tr>
<tr>
<td>3. insertAtList(hashTable, mapKey, mapString)</td>
</tr>
<tr>
<td>4. else if (<code>collisionRes = OPENADDR</code>)</td>
</tr>
<tr>
<td>5. <code>index ← newProbe()</code></td>
</tr>
<tr>
<td>6. insertAtBucket(hashTable, mapKey, mapString)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>findString(inKey)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>index ← hash(inKey) mod b</code></td>
</tr>
<tr>
<td>2. <code>outString ← find(hashTable, index, inKey)</code></td>
</tr>
<tr>
<td>3. return(<code>outString</code>)</td>
</tr>
</tbody>
</table>

Figure 2.10. Hash Table Algorithms to Add a String and to Find a String
A key feature of hash tables is that the number of buckets in the table can be increased as needed. To know when resizing should occur, the concept of load factor is used. Load factor of a hash table is the ratio of non-empty buckets of a hash table. Usually, if the load factor goes beyond a threshold, the table size will be increased. If it falls below another threshold, the size is decreased. The load factor determines how often collisions will occur. It can range from 0 to 1. If the load factor is too high, collisions will occur often, then decreasing the mapping and searching time. Different hash tables implementation check if the load factor is too high given a desired threshold. It is typical to check it after certain number of insertions or deletions of elements have occurred in the table. The implementation decreases the size of the table or increases it if the load factor is below or above the desired threshold, respectively. An algorithm to delete an element of the table is not shown here.

There have been many different improvements to hash tables, most focusing on improving access time. Perfect hashing was proposed in 1984 as an approach to achieve $O(1)$ even in the worst case [45]. Since then, hashing techniques have been considered as perfect hashing if the access time is $O(1)$ in the worst case [31]. The basic idea is to use two schemes of hash tables, the main hash table, and small hash tables associated with each of the buckets of the main hash table. For this purpose two hash functions are used. Different techniques can be used to achieve this, as shown in [32]. Later Silverstein proposed an improvement of previous techniques in [123], work that derived in the Google Sparse Hash Table [49]. In [45] the authors used a sparse array to save space by allocating space only for locations in use. Space is allocated as signatures are inserted. Signatures are pointed to by a second array used as a reference. However, it is not clear which data structure the authors used to implement it.

### 2.6.2 Bloom Filters

A Bloom filter is a probabilistic data structure that represents a set of elements (strings in most of the applications used in this dissertation) [15]. A Bloom filter consists of an array of $m$ bits, initially all set to 0, used to represent $n$ strings. A group of $k$ hashes are used to map (or “store”) strings and to test for membership. Compared with hash tables, Bloom filters are more limited because elements cannot be searched and retrieved. Instead, this data structure can only answer search attempts with a probabilistic response, in form of a true or false, meaning the element probably is or definitively is not in the array. Figure 2.11 shows an example of the mapping and testing in a Bloom filter. The Bloom filter starts with all
bits set to zero when there is no element mapped in. Elements $X_1$ and $X_2$ (and the rest of the set of elements) are mapped, and two elements might set the same position to 1. When testing for elements, $Y_1$ is definitely not in the array (because one of the positions is not set to one), and element $Y_2$ probably is in the array (it might be false positive, or a true positive). Figure 2.12 shows the two key algorithms for a Bloom filter: mapStrings() and testString(). In mapStrings() $m$ strings are hashed $k$ times each, and the hash values are used to identify bit locations in $bloomArray$ to be set to 1. In testString() an $inString$ is hashed $k$ times and the $k$ bit locations are tested. If all $k$ locations are a 1, then probabilistically, the Bloom filter “contains” $inString$. In both algorithms, the hash() function is used to generate a hash and its input are a string and an index. The index is used by hash() to return a different hash each time the function is called for the same string (thus generating $k$ different hashes). The hash() function returns a hash value that then is mapped to one of the $m$ positions in the array (with “mod $m$” in line 5 of mapStrings() and line 2 in testString()).

In the analysis of a Bloom filter false positive rate it is typically assumed that the hash functions are perfect, whereby they produce an independent and random index value for each object and thus the false positive rate is only a function of $m$, $n$, and $k$. The “classic” analysis of Bloom filter false positive rate is as follows (this analysis is often attributed to Bloom [15], but his original analysis was different, this classic analysis probably first appeared in a paper by Mullin [94]). It is assumed that a hash function selects each array position with equal probability. When $n$ objects are mapped in, the probability that an arbitrary bit is not set is $(1 - 1/m)^{kn}$. Then, the probability, $p$, that a given bit is set in a Bloom filter is

![Figure 2.11. An Example of a Bloom Filter [20]](image-url)
A false positive occurs when a tested string that is not a member of the Bloom filter maps to \( k \) bit positions that are set (i.e., have been set by strings mapped into the Bloom filter). This event occurs as

\[
\Pr[\text{false positive}] = p^k.
\]  

The false positive probability can be approximated as

\[
\Pr[\text{false positive}] \approx (1 - e^{-kn/m})^k.
\]  

For a Bloom filter with \( s \) bits set, \( p = s/m \) and thus

\[
\Pr[\text{false positive}] = \left( \frac{s}{m} \right)^k.
\]

The value of \( k \) that minimizes the exact expression in eq. (2.3) can be solved for directly and is

\[
k_{\text{opt}} = \frac{-\ln(2)}{\ln(m - 1) n}.
\]

The value of \( k \) that minimizes the approximate expression in eq. (2.3) can be solved for directly and is
\[ k_{opt} = \ln(2) \frac{m}{n}. \]  

(2.6)

As \( m \) becomes large, the values of \( k_{opt} \) in eq. (2.5) and eq. (2.6) converge to the same. A table of false positive values for varying \( m, n, \) and \( k \) based on eq. (2.6) is presented in [41]. In practice, \( k \) has to be an integer and the chosen \( k \) might be above or below the optimum value. The values exposed there however use an approximation of eq. (2.6). With the optimum \( k \), half of the bits in \( \text{bloomArray} \) are set to 1. For \( m/n = 16 \) (i.e., 16 bits of \( \text{bloomArray} \) allocated for each mapped string), \( k = 11 \). For each string test that results in a “hit,” \( k \) hashes and \( k \) memory accesses (or memory lookups) are needed.

On average, 2 hashes and memory accesses are needed for a string test that results in a “miss”. This can be calculated as follows. The expected fraction of bits of a Bloom filter that are set to 1 after mapping \( n \) elements is 1/2, for large Bloom filters [20]. So, the probability, \( p' \), that when testing a bit position it contains a zero, is 1/2. Let \( U \) be a random variable for the number of lookups.

\[ E_{\text{miss}}[U] = \frac{1}{1 - p'} = \frac{1}{0.5} = 2. \]  

(2.7)

An element can easily be added to a Bloom filter, and the algorithm to do so is similar to \( \text{mapStrings()} \), except that instead of adding a set of elements, only one element is added. It is, however, not possible to remove elements from a Bloom filter without creating the possibility of a false negative. This is a major shortcoming of Bloom filters.

### 2.6.2.1 Applications of Bloom Filters

Bloom filters have found use in spell checkers, distributed databases, distributed caching, and in many other areas. A survey of network applications of Bloom filters is in [20]. Improvements to Bloom filters have been studied by many researchers. Counting Bloom filters was proposed by Fan et al. [40] (and improved by Bonomi et al. [16]) as a means of allowing insertion and deletion of elements (standard Bloom filters do not allow for element deletion). A counting Bloom filter uses multiple bits (typically 4) for each location. Thus, a 4-bit counting Bloom filter requires 4x the amount of memory. Another well known improvement is the compressed Bloom filter [91]. The author presented a novel idea to reduce the number of bits transmitted through the Internet when a Bloom filter is to be sent between two hosts. The number of
bits set in the filter is reduced (compressed) before sending it as message and then the filter is uncompressed back again to the original size.

Bloom filters have been used extensively in caching of web servers. Fan et al. [40] proposed the use of modified Bloom filters to construct summary caches that are to be transmitted among servers. The modification consisted of allowing element deletions. This was achieved by using 4 bits per original bit position, used as a counter for the number of elements mapping to that position. These modified Bloom filters were later called Counting Bloom filters. Bloom filters have also been used for packet inspection. The logic behind packet inspection is a form of string matching. The work in [121] presented a tiered architecture of Bloom filters of different sizes (called hierarchical Bloom filters) to classify payloads of packets. The advantage of their solution is that with the use of this architecture, pieces of the payload could be tested, and it was not necessary to find a match for the whole payload, but excerpts of it. The work in [33] focused on taking advantage of hardware implementations to test several Bloom filters at the same time. The purpose of the hardware implementation was for fast packet inspection. Another approach for packet inspection from the same authors mixes Bloom filters and hash tables to provide exact matches instead of probabilistic matches as the original Bloom filters do. Their results claim that their solution decreases the number of memory accesses per lookup compared to hash tables.

Lumetta and Mitzenmacher [86] applied the idea of power of two choices to Bloom filters to reduce the probability of false positives (called here Power-of-2). Lumetta and Mitzenmacher use two groups of hash functions for mapping elements and testing for element membership. They considered two settings: online and offline. When mapping an element with the online setting, the two groups of hash functions are used to test which one adds the least amount of bits to the Bloom filter. This is done for each of the elements. The algorithm for the online setting (not shown in the original paper) is shown in Figure 2.13. For mapStrings(), each string is checked twice (one per each choice of hash functions) and the choice generating the least amount of bits is used to actually set the bits (in lines 8 and 10 of the algorithm).

When mapping with the offline setting, the whole group of \( n \) elements are mapped with each choice of group of hash functions to see which group yields the least number of bits set in the Bloom filter. Mapping of each element is repeated through rounds to check if choosing another group of hash functions might yield a better rate of bits set. This increase in processing results in a decrease in probability of false
positive. The improvements reported factor of 2 to 3 reduction in probability of false positives. However, the added expense in computation when testing for an element depends on the number of choices. If for example, two choices are taken, two times the number of generated hashes is needed, and thus two times the number of accesses compared to the original Bloom filter. An optimal Bloom filter replacement was studied by Pagh et al. [107]. Their approach is to use dynamic multisets to reduce membership testing time and space usage (and thus also probability of false positives for a given space allocation). This work is theoretical with no reported experimental implementations or results.

```plaintext
mapStrings(stringList)
1. for i ← 1 to m do
2.  bloomArray[i] ← 0
3. for i ← 1 to n do
4.  count1 ← count2 ← 0
6. for j ← 1 to k do
7.  hashVal1[j] ← hash1(stringList[i], j) mod m
8.  if (bloomArray[hashVal1[j]] = 0)
9.    increment count1
6. for j ← 1 to k do
7.  hashVal2[j] ← hash2(stringList[i], j) mod m
8.  if (bloomArray[hashVal2[j]] = 0)
9.    increment count2
6. for j ← 1 to k do
7.  if (count1 < count2)
8.    bloomArray[hashVal1[j]] ← 1
9.  else
10.   bloomArray[hashVal2[j]] ← 1

testString(inString)
1. tempFlag1 ← tempFlag2 ← TRUE
2. for j ← 1 to k do
3.  index = hash1(inString, j) mod m
4.  if (bloomArray[index] = 0)
5.    tempFlag1 ← FALSE
6.    break
7. for j ← 1 to k do
8.  index = hash2(inString, j) mod m
9.  if (bloomArray[index] = 0)
10.   tempFlag2 ← FALSE
11.  break
12. if (tempFlag1 = FALSE) and (tempFlag2 = FALSE)
13.  return (FALSE)
14. return (TRUE)
```

Figure 2.13. Power-of-2 Algorithms From Lumetta and Mitzenmacher [86]
For faster hashing for Bloom filters, Kirsch and Mitzenmacher [77] explored the use of pseudo hashing. Given two hash functions, $h_1(x)$ and $h_2(x)$, additional pseudo hash values can be generated as

$$g_i(x) = h_1(x) + i \cdot h_2(x)$$

(2.8)

where $i$ is the hash value index (where $i = 1, \ldots, k$) and $x$ is the string value being hashed. Kirsch and Mitzenmacher describe the application of this method to Bloom filters. To generate $k$ indexes into a Bloom filter requires only two actual hashes and $k - 2$ iterations of (2.8) which is a considerable reduction in processing compared to needing $k$ actual hashes. It is shown in [77] that using this method does not increase the asymptotic probability of false positives. This approach will result in higher false positive rates when the initial hash values are the same for different strings.

Most application use a large Bloom filter of many MB in length [20], [40]. However, there are applications that use small Bloom filters of a few bytes in length, with some examples as follows. Mullin has investigated the use of small Bloom filters to speed up string search [95]. Mullin employed a first algorithm to build an index of Bloom filters of one per document and a second algorithm to use the resulting Bloom filter index for searching. A specific example was developed and demonstrated where titles and author names from articles in the Communications of the ACM for 1983 and 1984 were used for the search text. The expected (predicted) performance of the new algorithm was not studied by Mullin. Whitaker and Wetherall [139] used small Bloom filters embedded in packets to detect and halt forwarding loops in networks (and to do so faster than a time-to-live field would). In their scheme, named Icarus, an extra field is added to a packet header that consists of a small Bloom filter which registers the network interfaces through which the packet has been by setting pre-determined random bits in the Bloom filter header. If at a given interface the bits do not change (that is, no bits are changed from 0 to 1), the packet is determined to be probably looping (that is, has probably passed through this interface already). Looping packets are removed after more than one such determination. An analysis of expected hop count before a false positive was given as a function of Bloom filter size.

2.6.3 Algorithms for String Searching

String search is an area of information retrieval that has been subject of study for many years [10], [44], [128]. This kind of search can be performed to find exact matches or approximate matches. The
search can end with probabilistic or deterministic responses. String search algorithms are used to find all the occurrences or the first occurrence of a string in a given text [10]. The string to be searched (also known as “pattern” in the literature) and the text are strings over some alphabet. They might however contain some patterns described by a language (e.g., consonants and vowels arranged according to the English language).

Many string search algorithms have been created since the 1970s, and new ones based on complex approaches have been presented relatively recently, like a multiple string pattern matching algorithm based on a compact encoding scheme, presented in [80]. The performance of these algorithms is evaluated by measuring the execution time to search for a string in the worst, average and best case. In the case they need a preprocessing phase, the time needed for preprocessing and the space required are also used to compare performance. The algorithms can be classified using many different criteria. For example, they can be classified using the direction they take to traverse the text to be searched (some can go right-to-left, others can go left-to-right). Another classification uses the preprocessing required as classification criteria.

The preprocessing might be done on the text or on the pattern. Preprocessing on the text can be divided into several types of operations. Typical operations are: 1) lexical analysis of the text to treat punctuation and cases, 2) elimination of stop-words and 3) stemming of the remaining words to remove affixes, among others operations (see Chapter 7 of [11]). Based on the preprocessing required, the classification can be divided into algorithms that do not perform preprocessing at all, and algorithms that perform some kind of preprocessing.

The basic string algorithm uses the naïve approach. It consists on traversing the text one character at a time and each time comparing all the characters of the pattern with the subset of text characters that starts at the current character being checked. The basic algorithm does not perform any preprocessing on the pattern or the text. The Knuth-Morris-Pratt algorithm published in 1977 in [82] takes advantage of unsuccessful characters comparisons and avoids comparing those again, thus speeding the search. To achieve this, the pattern is preprocessed to construct a table that stores for each character the index of the next character to be processed if that character caused a mismatch. This algorithm does not preprocess the text. Another well known algorithm is Boyer-Moore, created also in 1977 [17]. The main characteristic is that it traverses the pattern from right-to-left to find a match in the text. If there is a mismatch, it computes a shift to move the pattern to the right and restart the comparisons. This algorithm is considered the most efficient in the
average case. This algorithm also falls into the category of algorithms that perform some partial preprocessing. Karp and Rabin [74] presented an algorithm that uses a hashing technique. It calculates a signature for each substring of the text of size $m$ and compares it to the signature of the pattern, using of course the same hash function for the text and the pattern. Depending on how well the hash function is chosen, the occurrences of collisions might be low. A collision occurs when the pattern and a substring of the text have the same signature but there is no true match. To avoid false responses, the pattern and the substring are checked and compared character by character to see if they have equal signatures.

2.7 Chapter Summary

In this chapter, research on power management for an ICT host has been described to understand the background of the main work of this dissertation. As part of this, the concept of induced energy use was explained and a detailed description of what are the typical applications that drive the induced energy use of hosts was given. Then the concept of Wake-On-LAN and the improvements that have been proposed were explained. It is clear now what the disadvantages are of relying only on waking up the host when it is needed in order to achieve energy savings. Thus, the existing work on Network Connectivity Proxying (NCP) was explained as a better solution for saving energy, given the advantages of maintaining the network presence of the host while it sleeps. Background on string search algorithms was explained then as a base for potential NCP implementations that run on resource constrained devices.

In the next chapter, the first detailed architecture of an NCP with a design to proxy for IP phones running SIP, and also for some types of client-server applications, is described.
Chapter 3: Proxying as a Means of Reducing Induced Energy Use

Hosts connected to a network maintain their presence to other hosts by generating and responding to messages for network and applications protocols. A network host then, as defined here, is a host that is connected to the Internet using IP version 4. A Network Connectivity Proxy (NCP), as proposed in [100], is an entity that implements the key network presence capabilities of a network host in order to allow the host to sleep, yet it allows the host to appear to other devices as fully operational and connected to the network. Thus, an NCP can enable a network host to sleep and save energy whereas without the NCP it could not sleep even if inactive or idle. For IP version 6, no fundamental change should be needed. There are network management protocols that are new in version 6, but are beyond the scope of this work.

Figure 3.1 shows a network with two network hosts labeled as a host A and remove host and an NCP located within a switch to support host A. The NCP could also be located in another host on the network, within a wireless access point, or within the NIC in the sleeping host. As a first step towards designing an NCP, formal assumptions, goals and requirements are defined.

The following assumptions must hold for the network host covered by an NCP:

1. Have a sleep mode that can be entered/exited on application or operating system command.
2. Be able to fully exit sleep mode in the time span of a few seconds or less.
3. Have a sleep mode that preserves all local protocol and application states.
4. Support a remote packet-based wake-up method such as Magic Packet and/or pattern matching (if the NCP is in a remote location, not in the NIC of the host).
5. Support the ability of applications in hosts to be able to block the host from entering a sleep state if and when the application is actively using CPU, network, or other resources.
The following are the system-wide goals that a system with a network host covered by an NCP should achieve:

1. A host must be allowed to sleep and not lose its network presence.
   a. A host must maintain its IP address and be reachable by edge routers and switches.
   b. Valid incoming requests for TCP connections to a host must be honored.
   c. Existing TCP connections to a host must not be dropped and data must not be lost.
   d. UDP packets sent to a host must not be lost.

2. During the time the host is sleeping, remote state (such as application state in a remote host) must be maintained in all cases.
   a. Application keep-alive messages and/or other application messages must be responded to as required by the application.

3. Changes to network applications and protocols must not be required in the host or the remote host.

3.1 General Requirements for a Network Connectivity Proxy (NCP)

The NCP has assumptions and general requirements. This work is the first to address application level proxying for TCP connections. The work done by Ecma for the proxy specification is focused on working on network management protocols and IP connectivity [34]. The assumptions that hold for the NCP are:

1. It is always fully powered-on and connected to the network.
2. It is within the same MAC-level domain (and thus same IP subnet as well) as the covered host.
3. It has equivalent security measures in place as the host for which it is covering.

The general requirements for the NCP are organized into four categories: IP connectivity, TCP connections, UDP data flows, and network applications and higher-layer protocols. The minimum requirements for supporting IP connectivity are:

1. Have a much smaller incremental energy use than a network host (e.g., 10x less).
2. Know the power state – minimally, off, sleep or on (awake) – of the network host.
3. Use of the IP address(es) of the host for which it is covering.
4. Be able to support ARP, DHCP, and ICMP protocols for a sleeping host to maintain its host reachability, address, and manageability.
5. Be able to operate behind a typical NAT service.

Additional general requirements for supporting TCP connections are:

6. Be able to listen for valid TCP connection requests and other requests coming from other network hosts to a sleeping host.
   a. Be able to wake-up a host for a valid incoming TCP connection request and enable the connection to be established.
7. Be able to maintain permanent TCP connections for a sleeping host and buffer incoming data.
   a. Be able to immediately re-start TCP connection data flow to a host when it wakes up.
   b. Be able to immediately deliver buffered TCP data to a host when it wakes up.
   c. Be able to close TCP connections when it can be determined that a host has been removed and is no longer present.
8. Be able to wake up a sleeping network host when NCP buffers are nearly full to prevent buffers from filling up and potentially losing packets or blocking the server from sending data.

Additional general requirements for supporting UDP data flows are:

9. Be able to buffer incoming UDP packets for sleeping hosts.
   a. Be able to immediately deliver buffered UDP packets to a host when it wakes up.

Additional general requirements for supporting network applications and higher-layer protocols are:

10. Be able to keep network applications executing as if the host was not sleeping.
    a. Be able to respond to routine application messages as required by the application.
b. Be able to generate routine messages as required by the application.

Requirements (1), (2), (6), (7), and (10) (that is, power state signaling and support for TCP connections and keeping network applications executing) are addressed in this chapter and Chapter 5. Requirements (5), (8), and (9) are beyond the scope of this dissertation. Those 3 requirements are simple to design and implement. Requirements (3), (4), and (5) have been largely addressed in previous work (see Section 2.5 of Chapter 2 of this dissertation).

The NCP follows a sequence of steps when it starts proxying for the host and when it stops proxying for it. These steps include getting notification of the power state, states of applications, and maintaining full network presence. When the NCP is covering for a sleeping host it receives packets intended for the host. Each received packet results in one of the following actions: directly respond to the packet, wake up the host, discard the packet, and/or queue the packet for later processing by the host when it is awake. Figure

**Figure 3.2. Packet Processing by the NCP**

Note: Queued packets are delivered to the host when it wakes up.
3.2 shows the flowchart of actions for each received packet. Each packet is evaluated to see if it should trigger a wake-up, and/or if the NCP should generate a response, and/or if the NCP could simply discard it or buffer it. For example, low-level packets such as ARP and ICMP packets can be responded to directly. Other packets, such as an SNMP GET, may require the host to be woken-up if the host is running the service corresponding to the received packet. An SNMP GET has a finite lifespan, it cannot be queued for later response (that is, it does not make sense to respond to an SNMP packet many minutes later). Some application packets can be queued for later processing by the host when it wakes up. Two network applications, where queuing of packets makes sense for later processing and where proxying can enable a host to sleep, were specifically considered and are SSH and IM.

Beyond SSH and IM are a broad range of future applications under the rubric of Rich Internet Applications (RIA). RIAs are web-based applications where the web client executes the user interface and the back-end application server executes the application itself. RIAs support both pull and push of data via TCP connections and potentially split a state between a server and client. RIAs have the potential to have major impact on power management in network hosts. An NCP can support energy efficient operation of RIAs by maintaining connections and buffering data to enable the client host to sleep.

3.2 Covering TCP Connections

For applications like IM and SSH, it is necessary to keep the TCP connections open when the host goes to sleep to be sure the user session is not closed (login with the IM and SSH servers) and for the host to receive messages sent even if it is sleeping. In both cases, if the host goes to sleep, it will not acknowledge bytes received by its TCP stack anymore. The TCP stack of the remote server will start to retransmit the last unacknowledged bytes and buffer the rest of the messages the application attempts to send. If a group of retransmitted bytes is not acknowledged, the stack will back off and calculate the next time at which it will attempt a new retransmission of the same group of bytes. The TCP stack of the remote host does so until it reaches a maximum interval time between retransmissions, (variable called TCP_MAX_RTO in the TCP implementation), after which it will continue retransmissions at that maximum until it reaches the maximum number of retransmissions (variable called TCP_MAX_RETRIES2 in the TCP implementation). The TCP stack then drops the connection if no packet is acknowledged. This will result, in the case of SSH, in the shutdown of the session and the loss of user
session information. In the case of IM, the session with the IM server will be shutdown and messages will not be received. For that reason, the NCP will proxy those TCP connections by keeping them open for as long as the host sleeps. To describe this process, first the NCP components will be explained, and then the prototype and its evaluation are described.

3.2.1 Design of gSOCKS

NCP requirements (6) and (7) are to allow new inbound TCP connections to be established and to preserve TCP connections. To meet these requirements, the SOCKS service is used. The SOCKS standard [83] describes how TCP connections and UDP packet flows can be relayed through an intermediate host that executes a SOCKS server. SOCKS is an Internet service that comprises a client library and a server program. The SOCKS server is typically executed in a firewall, to allow for secure (that is, relayed from a secure host) TCP and UDP client access to a network. The server supports both outbound and inbound TCP connection requests and UDP packet flows. To use SOCKS, a client application must be “socksified” to support encapsulation of key sockets functions using the SOCKS client library. The result of this encapsulation is that a client application can transparently connect via the SOCKS server to an application server, or an application server can transparently connect to a listening client application. Most web browsers and FTP, SSH, and telnet client applications already support SOCKS, since it is a popular service used in firewalls. The SOCKS standard supports IPv6 addresses allowing for future migration to IPv6. The SOCKS service is used here as a key component in the NCP (called here “green SOCKS” or “gSOCKS”) to meet the requirements for supporting TCP connections and UDP data flows.

Figure 3.3 shows the architecture of the NCP with the gSOCKS component. The packet processing component handles packet discard, packet response generation, and wake-up (that is, the first three decision blocks of the flowchart in Figure 3.2). The gSOCKS component consists of an unaltered SOCKS server and a new NCP control point program. The SOCKS service relays TCP connections (similar to what is shown in Figure 3.1) and contains buffering. The use of gSOCKS requires that the NCP be fully powered on at all times so that all TCP connections to be preserved during host sleep are relayed at all times. Short-lived TCP connections (e.g., HTTP connections as part of web browsing that occur only when the client host is awake and in active use) that do not need to be preserved when a host sleeps are not relayed through the NCP gSOCKS component. In Figure 3.3, Control (1) communicates the host sleep state to NCP packet
For the SOCKS service to be used to preserve TCP connections for sleeping hosts one change must be made to TCP in the NCP to prevent a time-out and disconnect of a connection. When a host goes to sleep, any TCP connections from the NCP to the host trying to deliver a packet to the host will go into an exponential back-off period (due to time-out caused by delivery failure). If the host wakes up in the middle of a back-off period, it may still take many seconds before the current back-off period completes and the retry packet is successfully resent and received. Only at this time can any queued packets flow on the connection (as a result of the ACK from the host for the successfully delivered retry packet). In addition, after a fixed number of back-offs (which varies with TCP implementation), a backed-off connection will “give up” and close. To meet NCP requirements (7a) and (7b), the back-off of all NCP-to-host connections should be “frozen” when the NCP determines that the host is asleep (done via control (2) in Figure 3.3). Then, when the NCP determines that the host is again awake, the back-off timer for each NCP-to-host connection.
connection is reset to zero. This will cause an immediate resend of any unacknowledged packets and then all data flows will immediately resume.

It is necessary to design the signaling of the host sleep state to the NCP. The NCP requirement (2) is to know the power state of the network host that the NCP is covering for. How this is done depends on the location of the NCP. If the NCP is located outside of the sleeping host (e.g., in a switch in the network), then host sleep state signaling can be accomplished via a process running in the host that captures operating system sleep and wake-up interrupts (e.g., via ACPI in Windows and Linux), and communicates these interrupts to the NCP via a packet-based protocol (note that if the NCP is located within a NIC, this signaling is not needed). A notification packet is defined that contains a field that can contain two values signifying that 1) the host is now entering a sleep state, or 2) the host is now awake after having been in a sleep state. Figure 3.4 shows the architecture for power state signaling where a TCP connection is established between the host process and the NCP control point. The NCP listens for these connections on a pre-defined port. Before the host goes to sleep, the host opens the TCP connection, sends the state, and closes the connection. When the host wakes up, it opens a new TCP connection, sends the state to the NCP, and closes it.

If a host being proxied physically disconnects from the network, the NCP must stop covering for the host and clean-up NCP resources including closing any TCP connections that it may be relaying via the gSOCKS component. There are several possible ways for an NCP to determine that a host has physically disconnected. They are:

Figure 3.4. Architecture of Power State Signaling
The host periodically wakes up (e.g., based on an internal timer) and “checks in” with the NCP by using the sleep and wake-up notification packets described above.

The NCP periodically polls the host to determine its presence. The poll could be to the host NIC (e.g., via an ARP if the NIC can directly respond to an ARP) or it could be a full wake-up of the host and its operating system and applications (e.g., by sending a Magic Packet) followed by the sending of notification packets.

The NCP determines that the host is disconnected the first time that the NCP needs to wake up, the host and the host fails to wake-up thus signaling its absence.

In the design of an NCP, a periodic timer-based wake-up of the host is required to exchange wake-up and sleep notification packets with the NCP control point. A process in the host can implement this periodic wake-up using a hardware timer that is always running, even when the host is asleep. If a host fails to wake up and communicate with the NCP at its designated time-out, then the NCP will assume that the host is disconnected. A ten minute periodic wake-up is used in the prototype.

### 3.2.2 Prototype Implementation

The gSOCKS component of the NCP was prototyped in a low-end router. A Linksys WRT54G version 2.2 SOHO router was used (see Figure 3.5) and the original firmware was replaced with the open source WhiteRussian distribution of OpenWrt [106]. The WRT54G has a processor running at 216 MHz and 16 MBytes of RAM. The router has a NAT service, so applications running on remote servers connected to a

![Figure 3.5. The Target System: Linksys WRT54G](image-url)
host behind the router only see the router’s IP. The router consumes about 8 W when all four Ethernet ports are connected to a link. Network equipment such as this router would typically remain fully powered-on at all times. The addition of gSOCKS to the router did not change the power use (of the router), so NCP requirement (1) was met.

The WRT54G router does not have a SOCKS server included, however the Srelay package [134] has already been developed for OpenWrt. It is a software program that implements the SOCK protocol, it is written in C and was ported to the OpenWrt firmware. The Srelay was used for the prototype gSOCKS implementation. An NCP control point program for the router and a power state signaling application to run in a Windows-based client host are implemented. The NCP control point is a sockets application that executes in the router. The control point application listens on a pre-determined port for a connection and message from the client host.

The only two messages supported are:

- Host is going to sleep.
- Host is now awake.

The client host application generates these two messages in response to an operating system interrupt (an ACPI interrupt) signaling power state changes in the host. A TCP connection is established and closed just to send the one control message. The message sent by the client contains the IP of the client host. The control program in the client host resumes execution when it wakes up and detects another message from the operating system signaling that the system just woke up. This triggers a “host is now awake” message that is sent to the NCP control point.

A freezing and reset of TCP back-off in the NCP as described in the previous section was not possible to implement. To implement NCP requirement (7b), two TCP variables needed to be modified. The TCP_MAX_RTO was set to 1 second and TCP_MAX_RETRIES2 to very large (it was made 10,000). This means that retransmissions will be made every 1 second as maximum for a period no longer than 10,000 seconds (which is about 3.6 hours). The value set in TCP_MAX_RETRIES2 can be tuned to be as much as needed. This should be guided by the time the host is expected to be sleeping. It can be also set for a reasonable time, after which the host can be woken up to send or receive software updates as it might require. Thus, when a client host goes to sleep, the gSOCKS will retry any undelivered packets every
second and otherwise not “give up” and close the connection. To implement this method, a modification of the TCP implementation in the Linux kernel was necessary to expose the key variables. A customizable TCP Back-off patch presented in [140] was used to achieve this. This method of achieving requirement (7b) generates much more overhead traffic than the stated design (of freezing the time-out), but achieved the goal. This approach works well for a single client supported by gSOCKS. For a gSOCKS supporting many clients (some sleeping and some awake at any given time), a different approach might be needed. To prevent the TCP stack from keeping connections open when the host is disconnected from the network, a timer is used to give a sleeping host a maximum sleep time. If the client does not wake up before the timer expires, all TCP connections are reset and the TCP_MAX_RTO and TCP_MAX_RETRIES2 variables are reset to default values. This satisfies requirement (7c).

3.2.3 System Evaluation

In this section the evaluation of the prototype implementation is described. Specifically, it was addressed that the gSOCKS meets requirement (7). Also it is discussed how requirement (9) could be met, but constraints in the Linksys router prevented full implementation.

The test-bed configuration is shown on Figure 3.6. It consisted of a Linksys router WRT54-G version 3 with the gSOCKS (Srelay SOCKS plus NCP control point) implemented. The client host to be proxied (called “host A”) was a Dell Optiplex desktop PC with a 3.2 GHz Pentium 4 and 1 GByte RAM running Windows XP. The PC had a Broadcom NetXtreme 5755 NIC supporting ASF 2.0 (and thus the ability for the NIC to respond to ARP packets when the PC was asleep). When necessary, two other PCs with the same configuration labeled as host B and C were also used. All hosts were connected to the router, but only host A was covered by the NCP.

Two types of experiments were conducted to show that the NCP successfully proxies for network applications. The experiments types were:

1. Application proxying: the purpose was to show that TCP connections could be preserved for SSH and IM when the client host was sleeping.

2. Throughput experiments: the purpose was to check if the use of relayed TCP connection does have an effect on the throughput of network applications.
The SSH experiment was run as follows: The session-oriented application used for this experiment was the SSH client in the Putty program [113]. Putty has SOCKS support built in. For the purpose of the experiment, a simple test application was developed with the purpose of outputting messages to the console every few seconds (the messages were text strings such as, “Message #1,” “Message #2,” and so on). This application was run on a remote host running a SSH server and messages were sent to host A through the SSH connection to appear on the SSH console window on host A. For the experiment, host A was put to sleep for 30 minutes during which the application was running in host B (and outputting messages on the console window in host A). The experiment was run with and without SOCKS support enabled in Putty. When host A was returned to a fully powered-on state, the contents of the console window were examined for lost messages.

The IM experiment was run as follows: The Yahoo IM client was used for this experiment. The Yahoo IM client had SOCKS support built in. For this experiment three hosts were used (hosts A, B, and C with users Miguel Angel Jimeno, Miguel Jimeno, and Miguel Jimeno Paba, respectively) to participate in a three-way chat session. Only host A had its connection to the Yahoo IM server relayed through the gSOCKS. Host A was put to sleep in the middle of a three-way chat for 30 minutes. On wake-up of host A, the IM client window (in host A) was examined for lost messages.
The results from both experiments showed that no messages were lost when the host A connection was relayed through gSOCKS. The results also showed that the application – the SSH console window and IM client window – updated immediately upon host wake-up. Figure 3.7 shows the results from the IM experiment. The messages that would not be received when host A was not covered by the NCP are shown.

The throughput experiment was run as follows: The throughput achievable on a relayed TCP connection (for FTP) was measured through the Linksys router. A relayed connection was handled by the router control processor. The host was connected to an FTP server located on the USF campus and downloaded a 200 MB file. The throughput of the relayed connection was about 1.5 Mb/s, which is less than the throughput of a non-relayed connection measured on the same host (2.5 Mb/s). This suggests that performance needs to be a future consideration for applications that have long-term TCP connections and have high throughput needs. IM typically does not require high throughput. Most SSH connections are for low data rate console messages, but large file transfers can also be done via SSH. It was also noticed that some client applications (such as Windows Live Messenger) are able to detect when the client host goes to sleep and signal the server of this state change via an application message. This would cause a logoff of the client from the server – an undesirable condition, since the whole purpose of proxying was to prevent logoff to make sleeping invisible to the server- and needs to be further investigated and addressed.
Finally, the evaluation showed that the NAT capability of the Linksys router caused a conflict with SOCKS support for UDP data flows. Specifically, UDP flows bypassed SOCKS and went directly to host A. This should be resolved if the NAT can be “turned off,” or disabled, in the Linksys router.

3.3 Covering Application Protocols – SIP

In this section the design and implementation of the SIP Catcher is described. The SIP Catcher is a proxy that covers for calls directed to IP phones that implement the SIP protocol. The design is based on the design of the NCP. The SIP Catcher enables IP phones to sleep when idle (not making or receiving calls). The proxy will wake up the phones if they receive a call. It is shown that the SIP Catcher can save most of the power currently consumed by IP phones.

3.3.1 Design of SIP Catcher

The goal of the SIP Catcher is to allow IP phones to sleep when not in active use. The following are assumptions that must hold for an IP phone that is covered by a SIP Catcher:

1. The IP phone has a sleep state where it consumes much less power than that consumed when idle or in use. This implies that the phone would need a mechanism to quickly move to a higher power state so that it can be used to make calls in a reasonable time (< 1 second) and can be fully woken up to receive a call.

2. The IP phone has a mechanism that the SIP Catcher can use to detect its power state.

3. The IP phone can be woken up with a Magic Packet or another mechanism and can fully resume operation after this.

4. The transport protocol used by the IP phone is UDP.

The following are the specific requirements for SIP Catcher needed to fulfill general requirement (10) of the NCP (which is be able to keep network applications executing as if the host was not sleeping):

1. To proxy for an IP phone that is allowed to sleep when idle and be woken up when needed.

2. No modification of the hardware or software of the IP phone itself must be necessary.

3. No modification of the SIP protocol must be necessary.

The requirement (1) is a consequence of the general objective of the SIP Catcher. It proxies SIP calls directed to a sleeping IP phone and can enable it to sleep when idle. Wake-ups will be only necessary if the
registration of the phone with a SIP registrar server is necessary or if an incoming call to the phone arrives. Details of how this is achieved are explained during this section. To achieve requirement (2), a system that catches SIP protocol traffic coming/going to/from the IP phone is designed. The system is based on a packet tracer. The system has to be located where the SIP traffic can be captured. The IP phone does not have to be modified because of the assumptions explained before. The SIP Catcher is a program that runs on a host located in such a way that it can capture the network traffic of the IP phone with a packet filter. The filter is based on the tcpdump packet filter [63] and it is used by the program to capture only SIP packets. Figure 3.8 shows a design of the program. The link-level driver makes a copy of each SIP packet that arrives to the network interface of the host and sends it to the SIP Catcher program. All other packets follow the regular path to the protocol stack of the host and other programs. The prototype implementation of SIP Catcher works for only one phone. But only small changes would be necessary for it to proxy for more than one phone, and even then, the SIP traffic will not be high and the load of the host would not be

Note: Control (1) communicates the host sleep state to the SIP Catcher to start or stop catching SIP calls. Control (2) is used to send wake-up packets to the IP phone and for SIP messages forwarding.
affected. This complies with the idea of the system being transparent; this host will be in the middle of the path of SIP packets between the host making the call and host receiving the call. The program uses the tracing capabilities to gather information about the IP phone, such as for example, the SIP user used to register to the SIP registrar server, and IP of that server.

To explain how requirement (3) is achieved, a focus on assumption (4) is needed. Because the IP phone is assumed to use UDP to transport the SIP packets, the SIP Catcher can transparently emulate both sides of a SIP call when necessary. When the phone is sleeping and a caller initiates a call, the SIP Catcher proxies the response of the phone by sending a TRYING message on behalf of it. It then acts on behalf of the caller when it replicates the original INVITE and sends it to the phone when it is finally up. Replicating the message and successfully delivering it to the phone is not possible if TCP is used as transport protocol. In this case, the SIP Catcher would probably have to register to the SIP provider on behalf of the IP phone and modifications to the phone or the SIP protocol might be necessary.

To achieve energy savings, the SIP Catcher has to run on a host that consumes less power than the IP phone or where it does not represent increase of power consumption. Two options for this are possible:

1. Using a network host or equipment that is always powered-on for other reasons (so that it does not represent extra power consumption). This can be one of the servers being used to connect the phones to the VoIP network.

2. Locating the SIP Catcher in the last hop router to which the IP phone is connected. This option does not incur in extra power consumption, because the router is already in use. The disadvantage might be that depending on the capabilities of the router, the performance of the router might be affected by the packet tracer.

For simplicity of the prototype, the second option was selected.

It is necessary to explain how the SIP registration in the server is preserved. The process is as follows:

1. When the phone registers to the SIP registrar of its SIP service provider, the SIP Catcher program running on the router detects where the IP phone is located. The SIP registrar (also the SIP proxy server of the SIP service provider) then knows where that User Agent is located, and responds, accepting the binding and approving the expiration time of that binding.

2. The SIP Catcher detects from the registration the accepted expiration time of the binding.
3. When the phone goes to sleep, the SIP Catcher waits for the expiration time and just before the registration expires, it wakes up the phone for it to update the registration. After a few minutes, the phone will return to a sleep mode if it is not used.

This process repeats so that the phone will be woken up again just before the second expiration time expires. The SIP registrar is the server that approves the expiration time requested by the phone. This expiration time could be set to a maximum in the SIP registrar (according to the maximum time the phone is expected to be idle).

It is important to explain how the design of the SIP Catcher introduces delay time to the initiation of a SIP call. Figure 3.9 shows the timing diagram of the start of a SIP call when the IP phone is sleeping and the SIP Catcher is catching SIP calls directed to it. Assuming the time for the IP phone to wake up is close to zero, the delay appears when the caller receives two TRYING messages. The first message is sent by the SIP Catcher and the second one is sent to the IP phone after it has woken up. The call initiation delay time added (which is the same as the time between the two TRYING messages) is the sum of the following parts: 1) the time that the SIP Catcher takes to send a Magic Packet to the IP phone, plus 2) the time for the IP phone to wake up, process the INVITE, and generate a TRYING response, plus 3) the time that the INVITE message generated by the IP phone takes to arrive to the caller.
3.3.2 Prototype Implementation

Existing commercial IP hard-phones do not allow user-created applications to be installed on them. To test the implementation, a PC running an IP soft phone was used. An application to advertise the power state of the PC was installed on it; this emulates the assumption (2). The PC can be woken up by a Magic Packet and can go to a sleep state that consumes almost zero Watts.

The SIP Catcher was implemented as a program running on the last hop router to which the IP phone is connected. The last hop router selected was a Linksys router WRT54-G version 3. The original firmware was removed and replaced with OpenWrt [106]. Then, the SIP Catcher was developed using the C language and installed in the router as an application package. The Pcap libraries [63] were used in the code to trace all packets directed to the IP phone. A packet filter determined the types of packets that will reach the SIP Catcher. The packet filter was set to pass packets sent to the ports used by the SIP protocols. All Ethernet frames arrive to the SIP Catcher, where the frames have to be parsed to detect the type of message (e.g., if it is actually a SIP message, or if it is an INVITATION message). This has to be done for each packet arriving to the SIP Catcher. A modified version of the power state signaling program used by the NCP and described in Section 3.2.1 (also shown in Figure 3.4) was used by the SIP Catcher to determine the power state of the IP phone. Once the caller starts the call and the SIP proxy server forwards the SIP call to the network where the IP phone is located, the program detects the INVITE message. Immediately it responds with a TRYING message, which makes the caller's device wait. The SIP Catcher then wakes up the IP phone using a Magic Packet. For the prototype of the SIP Catcher, it waits for a few seconds before forwarding exactly the same INVITE message as was received. The time waited was to give time to soft-phone to wake up and be ready to receive messages. However, this time could be set to almost zero if the IP phone can wake up in less than one second as assumption (1) states. After the phone wakes up, the phone is ready to receive the INVITE, reply to it and starts ringing. The ringing in the soft-phone should be a sound generated by the software. It is important to note that there is no need to modify the SIP proxy server to which the proxied IP phone is connected. For the SIP proxy server, the presence of the SIP Catcher is not noticed.
3.3.3 System Evaluation

The prototype of the SIP Catcher used for the evaluation consisted of four parts:

1. IP phone to be proxied. The phone has to use the SIP protocol for session initiation, and has to use UDP as transport protocol. For prototyping purposes, it can be represented by a host running a soft-phone.

2. The SIP Catcher and the host on which it runs.

3. The SIP proxy server: The VoIP service provider that implements the SIP protocol and to where the IP phone connects.

4. The caller: A phone to make calls to the proxied IP phone. Depending on the VoIP service provider used, the caller can be another IP phone or a conventional phone.

The key questions to answer are if the SIP Catcher can successfully proxy for VoIP phones and what is the maximum time the proxy can enable phones to sleep. First, the configuration of the test-bed is shown, an experiment is then described and the results are shown. Two types of experiments were performed:

1. Proxying experiment: This type of experiment was designed to test if the proxy could successfully proxy calls directed to sleeping IP phones.

2. SIP session preservation experiment: This experiment was designed to test if the SIP Catcher can enable the phone to sleep for as long as it remains idle.

The test-bed (shown in Figure 3.10) consisted of an IP phone, a SIP proxy server, the last hop router to which the phone was connected, and a conventional telephone to make calls to the IP phone. The IP phone was represented by a PC running the SJphone soft-phone from SJ Labs [124]. This soft-phone used UDP as the default transport protocol, but has the option of using TCP. The PC used was a Dell Optiplex desktop PC with a 3.2 GHz Pentium 4 and 1 GByte RAM running Windows XP. The SIP service provider consisted of an SIP proxy server and registrar from a VoIP service provider called Tpad [135]. The last hop router was a Linksys router WRT54-G version 3, which runs a NAT server. Traces of the communication were used to check the results of the experiments. A PC (called “the tracer”) was used to trace the packets sent and received by the SIP Catcher and to trace the packets received by the IP phone. Two different types of traces in the test-bed were taken. One trace was for packets from the Internet to the router. Another trace was packets from the router to the IP phone. For the first trace, the router and the tracer were connected to a
hub so that the tracer could detect the packets going in and out of the WAN interface of the router. To trace the packets received by the IP phone, the tracer and the PC with the IP phone were connected to the hub, and the hub to the router. A picture of the IP phone and the Linksys router used for the test-bed is shown in Figure 3.11.

The Wake-On-call experiment was run as follows: The purpose of the experiment was to show that the proxy could successfully proxy a phone call for a sleeping IP phone. At the same time, the intention was to measure the call delay time that the SIP Catcher introduced in the initialization of a SIP call. The design of this experiment is described in the following steps:

1. The PC where the soft-phone is running is put to sleep.
2. While the PC was sleeping, packet traces of the traffic directed to the interface of the PC were taken to verify that the SIP Catcher was successfully proxying the SIP calls.
3. After 5 minutes of the PC being sleeping, a conventional phone was used to make a phone call to the phone number used by the soft-phone.

The SIP session preservation experiment was run as follows: The purpose of this experiment was to determine if the IP phone can be put to sleep for periods of time longer than the expiration time of the binding with the SIP proxy server. This would determine that the phone can sleep even beyond the
expiration time of the session. For this, an option in the configuration menu of the soft-phone was used. This option set the expiration time of the binding with the SIP server. The approach was the following:

1. Set the expiration time to a large value (several hours) to allow the SIP registrar server to negotiate for is the maximum binding expiration time allowed.

2. Set up the control program in the SIP Catcher to wake up the host running the IP phone before the negotiated expiration time expires. Verify the maximum sleep time of the IP phone.

The results of the repetitions of the Wake-On-call experiment were as follows. While the PC was sleeping, the traces (in Figure 3.12) showed the SIP registrar server sending messages to the router to keep the tunnel through the NAT server established in order to communicate with the IP phone. The purpose of this was to prevent the NAT server running on the router from closing the tunnel. No other types of packets were detected from the IP of the SIP proxy server. After some minutes the conventional phone was used to make a phone call to the IP phone. Several seconds after the number to contact the IP phone through its VoIP provider is dialed in the phone, the PC woke up and the phone started to ring. The display of the PC did not turn on given the configuration of the operating system (it does not turn on unless there is user input like keyboard or mouse movement), but it was possible to move the mouse to turn it on and see that the PC was actually up. The phone call could be answered without problem.
The experiment was run again, now using another phone from SJ Labs running in another PC. The purpose of this was to trace the SIP traffic in the caller. To be sure the IP phone was up and ready to receive the INVITE from the router, a delay of 5 seconds was manually set in the SIP Catcher. Other delays below 6 seconds were tried but were not sufficient because the INVITE arrived too early to the IP phone. No call was dropped on the caller side during all the repetitions of this experiment with a delay of 6 seconds (the call initiation delay time response variable). For information purposes, a modification of this experiment consisted on not waking up the phone but sending the TRYING message to the caller. After 1 minute, the caller dropped the call.

The results of the SIP session preservation experiment were as follows. First, the desired expiration time of the binding was changed before the soft-phone connected to the SIP proxy server. A set of expiration times was used in the IP phone (24 hours, 12 hours, and 6 hours). However the SIP proxy server (from Tpad) did not accept those values and negotiated the expiration time to be its maximum of 1 hour or 3600 seconds (the units used by the SIP protocol). The PC with the soft-phone was put to sleep and the proxy program was started in the router. The control program running on the router was set to wake up the PC with the soft-phone when the binding with the SIP proxy server was about to expire. After about a minute of waking up, the IP phone re-registered with the SIP provider. Because the PC did not detect any user input, after a few minutes the PC went back to sleep. For the experiment, this was repeated during several hours. No call was made during that period. The phone remained sleeping most of the time and just woke up a few minutes every hour to re-register. Thus the energy savings could be substantial.

Figure 3.12. Messages Sent by the SIP Registrar Server When Phone Sleeps

The experiment was run again, now using another phone from SJ Labs running in another PC. The purpose of this was to trace the SIP traffic in the caller. To be sure the IP phone was up and ready to receive the INVITE from the router, a delay of 5 seconds was manually set in the SIP Catcher. Other delays below 6 seconds were tried but were not sufficient because the INVITE arrived too early to the IP phone. No call was dropped on the caller side during all the repetitions of this experiment with a delay of 6 seconds (the call initiation delay time response variable). For information purposes, a modification of this experiment consisted on not waking up the phone but sending the TRYING message to the caller. After 1 minute, the caller dropped the call.

The results of the SIP session preservation experiment were as follows. First, the desired expiration time of the binding was changed before the soft-phone connected to the SIP proxy server. A set of expiration times was used in the IP phone (24 hours, 12 hours, and 6 hours). However the SIP proxy server (from Tpad) did not accept those values and negotiated the expiration time to be its maximum of 1 hour or 3600 seconds (the units used by the SIP protocol). The PC with the soft-phone was put to sleep and the proxy program was started in the router. The control program running on the router was set to wake up the PC with the soft-phone when the binding with the SIP proxy server was about to expire. After about a minute of waking up, the IP phone re-registered with the SIP provider. Because the PC did not detect any user input, after a few minutes the PC went back to sleep. For the experiment, this was repeated during several hours. No call was made during that period. The phone remained sleeping most of the time and just woke up a few minutes every hour to re-register. Thus the energy savings could be substantial.
3.4 Chapter Summary

In this chapter formal requirements to support application level proxying in the NCP were presented. The first proxy to keep TCP connections open for hosts that are in sleep states was described. The idea was implemented through the use of a relay server to be used by the host so that the TCP connections of certain network applications were maintained when the host slept. This was achieved by setting variables in the TCP stack implementation of the device where the proxy is running, to prevent it from closing the TCP connections with the sleeping host. At the same time, other variables were set to allow a fast connection recovery when the host came back from the sleep state and was able to restore the connectivity with other network hosts. The proxy was proven to be successful for applications like IM and SSH that rely on user sessions to receive update messages. The update messages were buffered by the proxy and delivered when the hosts woke up.

The SIP Catcher was presented as a proxy for SIP calls directed to IP phones. The proxy now allows IP phones to sleep when idle, and be woken only to receive calls. This was achieved by tracing packets directed to the phone, parsing them to detect SIP calls invitations, generating a response to prevent the call from being dropped and waking up the phone. This proxy allows IP phones to sleep as long as not being used, thus saving most of the wasted energy used by IP phones.
Chapter 4: Improved Probabilistic String Search With an Application to P2P Proxying

In a P2P Gnutella host, string search algorithms have to be used when answering Gnutella Query messages. P2P hosts receiving Query messages have to quickly search the keywords in their list of shared files to check for matches. To make a P2P proxy space and time efficient, string search algorithms used by the proxy have to consume the least possible amount of time and space.

In order to keep the costs of an NCP implementation low, the device where it runs should be limited in both computational and memory capabilities. To store a large list of file names being shared by a host would require considerable memory. Bloom filters have been used to “compress” lists of information that need to be shared [40] to run queries on them. Thus, the application of Bloom filters to proxying for a P2P application and how to make a Bloom filter as time and space efficient as possible are studied here. Bloom filters are considered mostly for applications where set element membership testing is sufficient, and where memory is limited. In more complex cases – where the use of a keyword is needed to retrieve information – more sophisticated data structures and search algorithms are needed.

Bloom filters are useful for devices that are extremely memory space constrained because they are the most space efficient data structures for membership testing. Depending on the device used, a Bloom filter could be used to proxy for P2P applications, but more complex data structures are needed to fully implement Query message responses in a proxy. For this, it is important to study possible improvements to existing string search algorithms. In this chapter, methods to improve string search in P2P applications are presented. The methods presented are of two types: 1) new methods that give only Boolean responses (true or false) and allow false positives as a possible result of a search. (Best-of-N Bloom filter and two-tier Bloom filter), and 2) new methods that are used to retrieve information and do not introduce false positives as a possible result of a search (Signature Array Hash Table (SAHT) and Bloom-filter-based keyword search). Best-of-N is inspired in the work from Lumetta and Mitzenmacher [86], explained in Chapter 2, where the authors used the theory of power of two choices to improve false positive probability of Bloom
filters. First, the goals, assumptions and requirements for better probabilistic string search methods are presented, and then the methods are described and evaluated.

4.1 Requirements for Better Probabilistic String Search Methods

The goal of an improved probabilistic string search method for use on a P2P proxy is as follows: To design, implement and evaluate a new search method with the purpose of using it as search method in a P2P proxy. This proxy is intended to run on a host where memory space and computational capabilities are reduced. The assumptions expected to hold for better string search methods for a P2P proxy are:

1. Objects to be searched can be represented by character strings.
2. The probability of collision of the hash function used by the search method is not statistically significant.

The specific requirement for better string search methods for a P2P proxy is:

1. No false negatives can be accepted due to the nature of P2P applications. Preventing some files of a host from being shared could lead to the isolation of the least requested files.

4.2 The Best-of-N Method for Bloom Filters

The Best-of-N a method that reduces the false positive rate of a Bloom filter but requires preprocessing computations.

4.2.1 Design and Implementation

A Bloom filter with the Best-of-N uses a method that needs additional computation to generate a Bloom filter with a reduced probability of false positives. For a given list of elements (e.g., file name strings) to be mapped into a Bloom filter, the Best-of-N method finds a Bloom filter instance with the least number of bits set to 1. Using \( N \) different groups of hash functions, \( N \) instances of a Bloom filter are generated sequentially. The instance with the least number of bits set to 1 defines 1) the Bloom filter instance and 2) the hash group that is then used to check for membership in the filter. The Best-of-N method is shown in Figure 4.2 (Figure 4.1 defines the variables used). Two functions are given, one to generate the Bloom filter and the other to test for membership. A Bloom filter has \( m \) bits. The constant \( K \) (same as \( k \)) is the number of hash functions in a hash group.
In Figure 4.2, the function hash(string, i) generates a hash value for string using hash function i. The value then has to be moduloed (using mod operand). Lines 8 to 12 map an element and increment setCount.

After all the elements have been mapped, this variable is used to check if the instance just created is the one with the least amount of bits created so far (with “if” statement in line 13). An optimization can be done to
check every time a new bit is set if the new number of values set is already greater than the \( \text{minCount} \). That means the current instance already has more bits than the best instance, and there is no point to continue mapping all the elements. One way to implement multiple hash functions from a single hashing method is to seed the method with the value \( i \). For example, CRC32 can be used as a hashing method and the CRC accumulator is seeded with different values to obtain different hash functions. In the case of SHA1, the digest can be updated every time a different hash is needed. In case the hash function does not accept an update, the string can be modified by appending a different character each time a new hash was needed, thus producing a totally different hash.

Intuitively, the larger the \( N \) value, the lower the probability of false positives and the longer the time required to create the Bloom filter instance. In Section 4.2.2 the analysis of the reduction of probability of false positives as a function of \( N \) is shown.

4.2.1.1 Generating Hash Values With an RNG Method to Reduce Execution Time

The use of pseudo hashing in [77] is extended here to reduce testing and mapping time in Bloom filters. The hash values generation is continued by using a linear congruential generator (LCG) random

\[
\begin{align*}
\text{mapString}(\text{nextString}) & \quad 1. \ seedValue \leftarrow \text{hash}(\text{nextString}) \mod m \\
& \quad 2. \ \text{bloom}[\text{seedValue}] \leftarrow 1 \\
& \quad 3. \ \text{seedRNG}(\text{seedValue}) \\
& \quad 4. \ \text{for } i \leftarrow 1 \text{ to } K \ \text{do} \\
& \quad \quad 5. \ \text{hashValue} \leftarrow \text{randInt()} \mod m \\
& \quad \quad 6. \ \text{bloom}[	ext{hashValue}] \leftarrow 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{testString}(\text{inString}) & \quad 1. \ \text{memberFlag} \leftarrow \text{TRUE} \\
& \quad 2. \ seedValue \leftarrow \text{hash}(\text{inString}) \mod m \\
& \quad 3. \ \text{if } (\text{bloom}[\text{seedValue}] = 0) \\
& \quad \quad 4. \ \text{memberFlag} \leftarrow \text{FALSE} \\
& \quad 5. \ \text{if } (\text{memberFlag} = \text{true}) \\
& \quad \quad 6. \ \text{seedRNG}(\text{seedValue}) \\
& \quad 7. \ \text{for } i \leftarrow 1 \text{ to } K \ \text{do} \\
& \quad \quad 8. \ \text{hashValue} \leftarrow \text{randInt()} \mod m \\
& \quad \quad 9. \ \text{if } (\text{bloom}[\text{hashValue}] = 0) \\
& \quad \quad \quad 10. \ \text{memberFlag} \leftarrow \text{FALSE} \\
& \quad \quad \quad 11. \ \text{break} \\
& \quad 12. \ \text{return} (\text{memberFlag})
\end{align*}
\]
number generator (RNG) where a single hash value is used as a seed to generate additional pseudo hash values. Figure 4.3 shows the method for mapping strings into, and testing for membership in, a Bloom filter. The value $seedValue$ is the initial hash value from an actual hashing of the string. The function seedRNG() seeds the RNG. The function randInt() returns a random integer between 1 and $m$ from the RNG. The returned value is the pseudo hash value. In the function testString(), testing of bits is explicitly halted at the detection of the first 0 bit. Compared to the method of Kirsch and Mitzenmacher to generate hashes (explained in Chapter 2), the RNG method proposed here requires only one actual hash and it can use a “good” LCG RNG algorithm (i.e., one with well-known properties) for generating the pseudo hash values. In the implementation of the RNG hash method the following LCG (from Jain [64]) is used,

$$x_n = 7^5 x_{n-1} \mod (2^{31} - 1)$$

(4.1)

where $x_n$ is the $n$th random integer value. The computation time and probability of false positives were evaluated for both the method from Kirsch and Mitzenmacher, and RNG methods later in this section.

### 4.2.2 Analysis

In this section an expression for probability of false positives for the Best-of-N method is derived. Defining $S$ as the random variable for the number of bits set in a Bloom filter, expressions for mean and variance of $S$ are derived. Assuming a normal distribution of $S$ and using order statistics, a computable expression for probability of false positives as a function of $N$ is reached.

The derivation of the probability of false positives of a regular Bloom filter (a Bloom filter with Best-of-N where $N = 1$) is described in Section 2.6.2. The mean of $S$ is the probability that a bit is 1 multiplied by the total number of bits,

$$E[S] = mp .$$

(4.2)

The variance of $S$ requires the derivation of the second moment of $S$. Let $U_i$ ($i = 1, 2, \ldots, m$) be the random variable that is set to 1 if bit $i$ is set to 1 and 0 otherwise. When $n$ objects are mapped in, the probability that an arbitrary bit is not set is

$$\Pr[U_i = 0] = \left(1 - \frac{1}{m}\right)^{kn} .$$

(4.3)
Then \( S = U_1 + U_2 + \cdots + U_m \) where

\[
\Pr[U_i = 1] = 1 - \left(1 - \frac{1}{m}\right)^{kn}.
\]  

(4.4)

The probability \( \Pr[U_i = 1] \) means the probability that bit \( i=1 \) is set to 1. Hence,

\[
E[S] = E[U_1] + E[U_2] + \cdots + E[U_m] = m \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right).
\]  

(4.5)

For the second moment,

\[
E[S^2] = E[U_1 + \cdots + U_m]^2 = \sum_i E[U_i^2] + \sum_{i \neq j} E[U_i U_j],
\]  

(4.6)

after similar terms are grouped. Since \( U_i^2 = U_i \), it is already known that

\[
E[U_i^2] + \cdots + E[U_m^2] = m \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right).
\]  

(4.7)

It can be noticed that \( E[U_i U_j] \) is equal to the probability that both bits \( i \) and \( j \) are set (this occurs when \( U_i U_j = 1 \)). This is,

\[
\Pr[U_i U_j = 1] = 1 - \Pr[U_i = 0] - \Pr[U_j = 0] + \Pr[U_i = 0 \text{ and } U_j = 0]
\]  

(4.8)

but \( \Pr[U_i = 0] \) is already known from eq. (4.3). And \( \Pr[U_i = 0 \text{ and } U_j = 0] \) is the probability that two bits are not set. So then eq. (4.8) is

\[
\Pr[U_i U_j = 1] = 1 - 2 \left(1 - \frac{1}{m}\right)^{kn} + \left(1 - \frac{2}{m}\right)^{kn}.
\]  

(4.9)

There are \( m(m-1) \) terms in the second summation in eq. (4.6). Thus the second moment can be obtained directly using results from eq. (4.9).

\[
E[S^2] = m \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right) + m(m-1) \left(1 - 2 \left(1 - \frac{1}{m}\right)^{kn} + \left(1 - \frac{2}{m}\right)^{kn}\right)
\]  

(4.10)

From eq. (4.5) and eq. (4.10) the variance \( \sigma^2[S] \) can be obtained.
\[ \sigma^2[S] = E[S^2] - E^2[S], \quad (4.11) \]

Replacing in eq. (4.11) the expressions derived in eq. (4.5) and eq. (4.10), eq. (4.11) is reduced to

\[ \sigma^2[S] = m\left(\frac{m-1}{m}\right)^{kn} + m^2\left(\frac{m-2}{m}\right)^{kn} - m\left(\frac{m-2}{m}\right)^{kn} - m^2\left(\frac{m-1}{m}\right)^{2kn}. \quad (4.12) \]

Order statistics [56] is used to determine the probability function of the first order statistic (the minimum value of samples \( S_1, S_2, \ldots, S_N \) which is selected as Best-of-N). The mean value of \( S \) given \( N \) samples (i.e., \( N \) instances of a Bloom filter) is then calculated. For the random variable \( S \) with probability distribution function \( f(s) \), cumulative distribution function \( F(s) \), and \( N \) independent samples \( S_1, S_2, \ldots, S_N \), the minimum value of the samples is the first order statistic,

\[ S_{\text{min}} = S_{(1)} = \min(S_1, S_2, \ldots, S_N). \quad (4.13) \]

The probability distribution function of the \( r \)th statistic is

\[ f_{r,\text{min}}(s) = \frac{N!}{(r-1)!(N-r)!}(F(s))^{r-1}(1-F(s))^{N-r} f(s), \quad (4.14) \]

from which the probability distribution function of the first order statistic \( f_{\text{min}} \) (here) is then

\[ f_{\text{min}}(s) = N(1-F(s))^{N-1} f(s). \quad (4.15) \]

The mean value of \( S \) can be computed as

\[ E[S_{\text{min}}] = \int_{-\infty}^{\infty} s f_{\text{min}}(s) ds. \quad (4.16) \]

The normality assumption for the distribution of \( S \) can be used now. Thus,

\[ f(s) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(s-\mu)^2}{2\sigma^2}} \quad (4.17) \]

and

\[ F(s) = \frac{1}{2} \left(1 + \text{erf} \left( \frac{s-\mu}{\sigma\sqrt{2}} \right) \right) \quad (4.18) \]

where \( \mu = E[S] \) and \( \sigma = \sigma[S] \). Substituting eq. (4.17) and eq. (4.18) into eq. (4.15) it is derived
It can be calculated now an expression for the probability of false positives of the Best-of-N method,

\[
\Pr[\text{false positive}] = \left( \frac{E[S_{min}]}{m} \right)^k
\]

For a given \(m\) and \(n\), the value of \(k\) to minimize the probability of false positives of a Bloom filter can be determined. The probability of false positives from eq. (2.1) and eq. (2.2) is

\[
\Pr[\text{false positive}] = \left( 1 - \left( 1 - \frac{1}{m} \right)^{kn} \right)^k = \left( 1 - e^{-kn/m} \right)^k.
\]

For a given \(m\) and \(n\) where \(k\) is chosen optimally, the probability of false positives as a function of \(N\) is studied. A software tool is used to calculate eq. (4.21). Figure 4.4 shows a plot of the probability of false positives (i.e., from eq. (4.21)) as a function of \(N\) for \(m/n = 16\) and \(k_{opt} = 11\). Figure 4.4 shows that the
false positive probability can be reduced up to 12% for these Bloom filter parameters (0.0004062 against 0.0004588). The reduction is important when calculated up to 10 instances \((N=10)\) and the improvement stabilizes close to \(N=70\). Figure 4.5 shows the same plot for \(m/n=32\) and \(k_{opt}=22\). The maximum reduction achieved is 18% for these parameters. For Figure 4.4 \(n=1000\) and \(m=16,000\) corresponding to an almost 16 KB Bloom filter with 1000 strings (or 1000 files shared in this P2P application) mapped into it. For Figure 4.5 the same value of \(n\) is used and \(m\) is increased to 32,000. It can be seen that Best-of-N results in a reduction in probability of false positives.

Figure 4.6 shows the improvement factor for \(m/n=8\), \(m/n=16\) (from Figure 4.4), \(m/n=32\) (from Figure 4.5), and \(m/n=64\). Table 4.1 summarizes the improvement for \(N=2, 5, 10, 50,\) and \(100\) from Figure 4.6. It can be seen that as \(m/n\) increases, the improvement factor also increases. For \(m/n=32\) an almost 19% reduction in probability of false positives is achieved for \(N=100\).

Best-of-N is similar to the online setting proposed in [86], where two or three choices of groups of hash functions are used to check which one adds fewer ones to the Bloom filter. The difference is that in Best-of-N the whole set of strings is mapped in and then checked if it increases the number of ones, instead of comparing each time a new string is mapped in. Table 4.2 shows a comparison between the Best-of-N and [86], called here Power-of-2. For Power-of-2, only two choices were taken into account. It can be seen that the probability of false positives for Best-of-N is better than that of Power-of-2 for \(m/n=4\) and

<table>
<thead>
<tr>
<th>(N)</th>
<th>(m/n=8)</th>
<th>(m/n=16)</th>
<th>(m/n=32)</th>
<th>(m/n=64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.021</td>
<td>1.028</td>
<td>1.058</td>
<td>1.057</td>
</tr>
<tr>
<td>5</td>
<td>1.043</td>
<td>1.058</td>
<td>1.083</td>
<td>1.119</td>
</tr>
<tr>
<td>10</td>
<td>1.058</td>
<td>1.078</td>
<td>1.111</td>
<td>1.161</td>
</tr>
<tr>
<td>50</td>
<td>1.086</td>
<td>1.115</td>
<td>1.167</td>
<td>1.243</td>
</tr>
<tr>
<td>100</td>
<td>1.096</td>
<td>1.129</td>
<td>1.188</td>
<td>1.275</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(m/n)</th>
<th>(m/n=4)</th>
<th>(m/n=8)</th>
<th>(m/n=16)</th>
<th>(m/n=32)</th>
<th>(m/n=64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-of-N</td>
<td>1.072</td>
<td>1.096</td>
<td>1.129</td>
<td>1.188</td>
<td>1.275</td>
</tr>
<tr>
<td>Power-of-2</td>
<td>0.710</td>
<td>0.919</td>
<td>1.146</td>
<td>1.388</td>
<td>1.612</td>
</tr>
</tbody>
</table>
m/n = 8, where the improvement is close to 10% for Best-of-N. For these parameters Power-of-2 actually produces even more false positives than using only one choice of groups of hash functions (that is the original Bloom filter). The authors of [86] did not address this issue. For larger values of m/n it can be seen that Power-Of-2 outperforms Best-of-N. The Best-of-N is faster for testing. Only one group of hash functions is used to map the whole set of elements. This allows the application to store the index of the group that was used. In Power-of-2, however, there is no way to know which group was used, so there is the need to perform string tests with all the choices of groups of hash functions, to see which one triggers a
positive. This means that membership testing when the Bloom filter returns a hit is 2 times faster in software implementation of Best-of-N, assuming two choices are used in Power-Of-2.

4.2.3 Experimental Evaluation

Two key measures of performance for a Bloom filter are 1) the probability of a false positive and 2) the computational effort required to test for membership. In this section the analytical model is compared to a real implementation of the Best-of-N method for probability of false positives. The computation time when run on a typical desktop PC is also evaluated. An implementation of a Bloom filter with the Best-of-N method was written in C. The following four hashing methods were implemented (input was a list of strings):

- SHA1, using the implementation from the standard [42]. SHA1 was chosen as a widely used and known hash function, well known for its randomness.
- CRC32, using an 8-bit table look-up implementation from [110]. CRC32 was chosen as a widely used and relatively efficient hash function which can be implemented in software.
- Kirsch and Mitzenmacher’s method from [77], using CRC32 for the seed hashes.
- The new RNG hashing method from Section 4.2.1.1, using CRC32 for the seed hash.

In addition to the above four hashing methods used to hash and map real strings into a Bloom filter, a “perfect hashing” was implemented using an RNG to generate a random sequence of values. This perfect hashing served as a control to eliminate any effects of possible collisions from the real hashing methods.

One critical aspect of the experiments was to create N groups of hash functions based on one hash function implementation. For CRC32, the CRC accumulator is initialized with a different value each time a new hash function was needed. For the SHA1 implementation, a different value at the end of the string is added to be hashed each time a new hash function was needed.

The experiments were designed as follows. All experiments were executed on a Dell OptiPlex GX620 PC (Pentium4, 3.4 Ghz, 2 MBytes cache) with 1 GByte RAM with WindowsXP as the operating system. The gcc compiler (version 3.4.2 mingw-special from Dev C++ [90]) with no optimizations was used in all cases. Execution time was measured using C time functions with an accuracy of 10 ms on Windows XP.

A list of 25,000 strings of unique music file names was obtained using Bearshare [96]. The list of music file names was generated manually by searching names of artists and compiling a list of the songs
retrieved by the queries. Each string consists of artist name and song title. This list of strings was used for the input to all experiments (except for the “perfect hashing” experiments, where no strings were hashed).

Figure 4.7 shows a histogram of string lengths from the test set. The mean string length was 47 bytes. Thus, with \( m/n = 16 \) there are 16 bits (or 2 bytes) per each string that was mapped into the Bloom filter resulting in a storage savings of over 20 times (i.e., \( 47/2 = 23.5 \)).

The two response variables of interest were:

- Probability of false positives for the Bloom filter.
- Execution time to generate a Bloom filter.

Probability of false positives was measured in two ways:

1. Analytically by counting the number of bits set to 1 and using eq. (2.4), and
2. Empirically by testing for membership with a list of 25,000 test strings where none of the strings in the list were already represented in the Bloom filter.

The factors of interest were:

- Hashing method used.
- Bloom filter parameters \( m, n, \) and \( k \).
- Best-of-N parameter \( N \).
- Number of strings used in the string test set.
Experiments were designed to evaluate the probability of false positives, (including comparison of the analytical model to actual implementation) and computation run time (CPU time). All experiments, unless otherwise stated, were executed using the four hashing methods and “perfect hashing” using an RNG. For all experiments $n$ was set to 1000 (corresponding to a reasonable number of files shared by a P2P node) and $m$ set to 16,000 corresponding to $m/n = 16$ ($k$ was chosen optimally as $k_{opt} = 11$ from [41]). The experiment was designed to measure probability of false positive using eq. (2.4) and empirically, and also to measure runtime:

For the setup of the false positive probability calculated with eq. (2.4), $N$ was varied from 1 to 100. Then the number of bits set to 1 was measured and the probability of false positive was calculated using eq. (2.4). The mean from 10,000 iterations for each value of $N$ was collected.

To empirically measure the false positive probability, it was necessary to repeat the previous experiment, except the false positive was measured empirically.

To measure the run-time, it was necessary to repeat the false positive experiment and collect the run-time (CPU time) for each value of $N$. 

Figure 4.8. False Positive Calculated With Eq. (2.4) Using Perfect Hashing
The results of calculating the false-positive probability using eq. (2.4) are shown in Figure 4.8 for perfect hashing and in Figure 4.9 using the four hashing methods studied. In Figure 4.8 it can be seen that the probability of false positive from the implementation and analysis agree perfectly (within 1% for all values of $N$). This validates the analytical model, including the assumption of normality for the number of bits set to 1 ($S$). Figure 4.9 shows that using a real, non-ideal hash function results in a probability of false positives with greater variability compared to using a “perfect hashing” function. However, variability is also below 1% for all values of $N$. 

Figure 4.9. False Positive Calculated With Eq. (2.4) Using Four Hash Functions

Figure 4.10. False Positive Measured Empirically

The results of calculating the false-positive probability using eq. (2.4) are shown in Figure 4.8 for perfect hashing and in Figure 4.9 using the four hashing methods studied. In Figure 4.8 it can be seen that the probability of false positive from the implementation and analysis agree perfectly (within 1% for all values of $N$). This validates the analytical model, including the assumption of normality for the number of bits set to 1 ($S$). Figure 4.9 shows that using a real, non-ideal hash function results in a probability of false positives with greater variability compared to using a “perfect hashing” function. However, variability is also below 1% for all values of $N$. 

80
The results of measuring the false positive probability empirically are shown in Figure 4.10. This figure specifically shows a lower probability of false positives for the RNG method compared to the method of Kirsch and Mitzenmacher. The variability for the Kirsch method is close 8% for \( N=1 \) (0.0004945 for Kirsch against 0.0004588 for the analysis). For \( N=100 \) the variability is close to 9%. The results from Kirsch can still be considered useful. The variability for the RNG method is much better, close to 1% for \( N=1 \) and close to 4% for \( N=100 \). All the other methods are always below 1% of variability.

The results for the run-time experiment are show in Figure 4.11. The graph shows the CPU time to generate one Bloom filter using the Best-of-N method with the four different hashing methods, given a set of 1,000 strings to be mapped into the Bloom filter.

For the test PC the results for \( N = 1 \):

- SHA1 requires 40 milliseconds
- CRC32 requires 4.9 milliseconds
- RNG method requires 1.5 milliseconds
- Kirsch and Mitzenmacher method requires 1.3 milliseconds

These times increase linearly with \( N \). There is some variability with CRC32. For example, between \( N=34 \) and \( N=35 \) the measured run-time values were 176 milliseconds and 172 milliseconds. However the difference can be considered insignificant. The RNG and Kirsch and Mitzenmacher methods are about the same, and CRC32 is 3 times greater in CPU time required than these two methods. For example, for \( N=50 \),
the time for CRC32 is 200 milliseconds against 62 milliseconds in RNG. SHA1 is 8 times greater than CRC32 for \( N=50 \), and the time was 1.7 seconds to calculate 50 instances of Bloom filters with the parameters explained before.

The results in this section clearly show that the effect of using methods like Kirsch or the RNG method instead of the conventional hash functions like SHA1 or CRC32 for creating Bloom filters is not significant, especially in the case of the RNG method. Figure 4.10 shows a variability of up to 4% for RNG when measuring false positives empirically. Figure 4.11 shows that the trade-off of variability is paid-off with the run-time improvement achieved when using Kirsch or RNG.

4.3 Two-Tier Bloom Filter

Testing for element membership in a Bloom filter requires hashing of a test element (e.g., a string) and multiple look-ups in memory. A design of a new two-tier Bloom filter with on-chip hash functions and cache is described. For elements with a heavy-tailed distribution for popularity, like Gnutella Query messages with search for popular keywords, membership testing time can be significantly reduced by using a small Bloom filter to support queries for popular keywords. Less testing time means less computational requirements for resource limited systems. In the literature, Bloom filters have been adapted before to popularity of elements as in [144] where different numbers of hashes were used according to the popularity of the element. Different approaches exist already for improving search in P2P networks, based on the popularity of the elements. The work in [88] stores in the fastest memory the most requested files, and removes them depending on different content management policies (similar to caching). The approach followed in [108] was to cache Query messages in some points of the network. In the case of this dissertation, the proxy used to perform searches are space constrained, so storing the files is not an option.

Membership testing time of Bloom filters is a function of the time to 1) compute up to \( k \) hashes and to 2) perform up to \( k \) lookups in memory where the Bloom filter array is stored. Memory look-up times depend on the type of memory in which the Bloom filter is stored (e.g., high-speed localized SRAM or slower main memory DRAM). In this section, the membership testing time for a Bloom filter is reduced by implementing hashing directly in specialized hardware and by introducing a second tier cache Bloom filter to reduce the number of accesses required into slower main memory.
4.3.1 Design and Implementation

A two-tier Bloom filter design is proposed to reduce membership test time. Figure 4.12 shows the basic design of a single component (such as a specialized chip) containing hashing circuitry and a Bloom filter. The two tiers might exist in a hardware implementation of a Bloom filter, or between processor cache and main memory in a computer. An on-chip Bloom filter can be implemented using fast SRAM (Static Random Access Memory), but it is limited in size to about $m = 4\, \text{Mb}$. Using $m/n = 32$ and $k = 22$ (as recommended in [40]) to achieve a low probability of false positives, the number of elements that can be mapped into the on-chip Bloom filter is $n = 131072$. For applications with more than 131072 elements, the on-chip Bloom filter can be used as a cache for a larger off-chip Bloom filter stored in the main memory (typically implemented in DRAM (Dynamic Random Access Memory), with slower look-up time than SRAM). The two-tier Bloom filter takes as input the element (e.g., a string) to be mapped into, or tested for membership, and outputs the $k$ hash values and a single test output to indicate if the element being tested for was found (or “hit”) in the on-chip cache Bloom filter. If the cache hit output is false, then the computed hash values are used to test the external Bloom filter in main memory. If the element is found in the external Bloom filter the main hit input causes the cache Bloom filter to learn the element (if the cache is not yet full). Thus, the cache Bloom filter contains a subset of the elements mapped into the main memory Bloom filter (the main memory Bloom filter contains mappings for all elements). How well the cache learns the most popular elements determines the performance (measured in membership testing time) of the two-tier Bloom filter.
The parameters of the two-tier Bloom filter are then:

- Size of the external Bloom filter ($m_{main}$).
- Size of the cache Bloom filter ($m_{cache}$).
- Number of elements stored in the external Bloom filter ($N$ or $n_{main}$).
- Number of elements stored in the cache Bloom filter ($n_{cache}$).
- Number of hash functions ($k$).

### 4.3.2 Analysis

Membership testing time ($T_{test}$) is a function of hashing time ($t_{hash}$), cache Bloom filter testing time ($t_{cache}$), main memory Bloom filter testing time ($t_{main}$), and probability of a successful membership test in the cache Bloom filter ($p_{cache}$).

The membership testing time is

\[ T_{test} = t_{hash} + t_{cache} + (1 - p_{cache}) \cdot t_{main} \]  

for the two-tier Bloom filter and $T_{test} = t_{hash} + t_{main}$ for a single Bloom filter in main memory. When testing for an element, first the hashes have to be calculated (which takes $t_{hash}$ time), then the cache has to be tested (which takes $t_{cache}$ time) and then there is a probability $p_{cache}$ that the element is found in the cache. There is a probability $(1 - p_{cache})$ that the element is not found in the cache, in which case the main memory Bloom filter is tested (taking $t_{main}$ time). The summation of these factors are $T_{test}$. In order for the two-tier Bloom filter to have a smaller membership testing time than that of a single Bloom filter in main memory, $t_{cache} - p_{cache} \cdot t_{main} < 0$ must hold. The speed-up ($S$) of $T_{test}$ is the ratio of the time required by the two-tier Bloom filter divided by the time required by a single Bloom filter stored in main memory. The speed-up expresses the relative (percentage) reduction in membership testing time by using the two-tier Bloom filter. Speed-up is,

\[ S = \frac{t_{hash} + t_{main}}{t_{hash} + t_{cache} + (1 - p_{cache}) \cdot t_{main}}. \]  

(4.24)
4.3.3 Experimental Evaluation

The specific target application for the two-tier Bloom filter is membership testing for a file system containing millions of files, each file with a unique identifier (e.g., path plus file name). It is well known that the distribution of the requests for files in some applications such as P2P file sharing and web caching follow a Zipf-like distribution [29], [18], where the probability of requesting the element ranked \( j \)th in popularity among a population of \( N \) elements is

\[
\Pr[j] = \frac{\Omega}{j^\alpha}
\]  

(4.25)

where \( \Omega \) is the normalization constant and \( \alpha \) is the shape parameter. The normalization constant is the inverse sum of \( 1/j^\alpha \) for \( j = 1, 2, \ldots, N \). For \( \alpha = 0 \) the distribution is uniform and as \( \alpha \) increases the distribution becomes skewed and heavy tailed. For P2P file sharing, \( \alpha \) values between 0.60 and 0.83 have been measured [29]. Other studies (e.g., [127] and [55]), have shown similar results.

In this experimental evaluation, simulation is used to study the speed-up, \( S \), of the two-tier Bloom filter compared to a single Bloom filter stored in main memory. The speed-up of the two-tier Bloom filter for a stream of Zipf distributed membership tests is a function of several variables. The factors of this evaluation were then:

- The number of elements, \( N \).
- The popularity of the elements (modeled with a Zipf distribution, parameters \( n_{\text{main}} = N \) and \( \alpha \)).
- The hashing and memory access times \( t_{\text{hash}}, t_{\text{cache}}, t_{\text{main}} \), and \( p_{\text{cache}} \) (the subscripts \( \text{cache} \) and \( \text{main} \) describe the parameter as applying to the cache and main memory Bloom filters, respectively). The probability \( p_{\text{cache}} \) is a function of the size of the cache \( n_{\text{cache}} \).
- How well the cache learns the \( n_{\text{cache}} \) most popular elements.

If the cache Bloom filter learns (and thus represents) the \( n_{\text{cache}} \) most popular elements – called a “perfect cache” here – for the population of \( N \) elements, then the probability of a cache hit is the cumulative probability mass of elements \( 1, 2, \ldots, n_{\text{cache}} \).

\[
P_{\text{cache}} = \sum_{j=1}^{n_{\text{cache}}} \frac{\Omega}{j^\alpha}.
\]  

(4.26)
In reality, the cache will learn a set of elements that are less than perfect. This is called the “realistic cache” in this section and its cumulative probability mass can be computed as follows. Given \( N \) distinct elements that are sampled with replacement, let \( x_j \) be the probability of drawing an element of type \( j \). Sampling continues until \( M \) elements of different types are sampled (this corresponds to the cache being fully loaded). Here \( M \) is \( n_{\text{cache}} \) and \( N \) is \( n_{\text{main}} \). The cumulative probability, \( P \), corresponding to \( p_{\text{cache}} \) for a realistic cache is

\[
P = \sum_{i=1}^{i=M} (x_i) \cdot \Pr[\text{subset } G \text{ is sampled first of all } M \text{ subsets of } N \text{ types}]
\]  

(4.27)

where \( G \) is the subset \( \{u_1, u_2, \ldots, u_M\} \) and where the summation is calculated over all \( M \) subsets of \( N \) types of elements. This summation is likely intractable to compute, given the very large number of subsets possible for a large \( N \). Given this intractability, a simulation model of the two-tier Bloom filter was created. From this simulation model \( p_{\text{cache}} \) for a realistic cache could be experimentally estimated.

The factors explained above were varied in the following way. For the elements, a value of \( N = 8 \cdot 10^6 \) was selected. An example of an application querying such a large set is the caches for web servers found in [40], where queries are made in sets of millions of URLs using Bloom filters. For the Zipf distribution, a parameter \( \alpha = 0, 0.1, 0.2, \ldots, 1.0 \) was selected. This models 8 million elements in the population with popularity ranging from uniform (\( \alpha = 0 \)) to highly skewed (\( \alpha = 1 \)). For hashing and memory access time, \( t_{\text{hash}} = 0 \) (to focus on speed-up from caching effects only), \( t_{\text{cache}} = 1 \), and \( t_{\text{main}} = 5 \) (modeling SRAM as 5 times faster than DRAM). For the cache Bloom filter the parameter values were \( m_{\text{cache}} = 4 \text{ Mb} \) and \( m_{\text{cache}} = 16 \text{ Mb} \), and \( k_{\text{cache}} = 22 \). For the main memory Bloom filter the parameter values were \( m_{\text{main}} = 256 \cdot 10^6 \text{ b} \) (or about 30.5 MB to be able to represent \( N = 8 \cdot 10^6 \) elements with 32 bits for each element), and \( k_{\text{main}} = 22 \).

Figure 4.13 shows the results for \( p_{\text{cache}} \) from eq. (4.21) for a perfect cache and from simulation for a realistic cache. The simulation results for \( p_{\text{cache}} \) for a realistic cache are the average of 30 trials for each value of \( \alpha \). It can be seen that \( p_{\text{cache}} \) increases when \( \alpha \) increases. It can also be seen that \( p_{\text{cache}} \) for the perfect and realistic caches could be up to 100% different for \( \alpha \) between 0.4 and 0.7. Figure 4.14 shows
the results for speed-up. It can be seen that the speed-up is achieved for values of $\alpha > 0.7$ and the larger $m_{cache}$ is, the greater the speed-up. The results show that even with a small cache memory, the two-tier Bloom filter can achieve faster membership testing for a heavy-tailed distribution of elements. The relationship between $t_{cache}$ and $t_{main}$ affects the possible speed-up.

4.4 Signature Array Hash Table (SAHT)

Bloom filters are chosen mainly for their space efficiency. The types of answers they give to queries, (Boolean response: it might be in the set, it is not in the set) limits their usability. For applications like P2P applications, Bloom filters could be used to prevent the application from accessing large and slow data
structures (like the two-tier Bloom filter). But P2P applications need to access information on the element about which it is being queried, i.e., the full filename and properties of the file (size, bitrate, etc.). For that reason, a new data structure is proposed to be able to give additional information when a Query is processed. The particular motivation is to improve Query processing time and search time of proxies in resource constrained devices.

Common to file search applications are the need for 1) fast search for a string match, 2) low memory use (for storing the list of filename strings), and 3) the ability to add and remove filename strings. In this section an alternative probabilistic data structure to Bloom filters is explored. This new data structure is called the Signature Array Hash Table (SAHT) and it is shown that it has many desirable properties and advantages. This work is similar to [45], explained in Chapter 2. This work uses a dense array instead, which allocates space for the expected size of the set. For SAHT, there is no wasted space since the set size is known from the beginning.

4.4.1 Design and Implementation

Hash tables with linked-list chains can be used to store string signature values (where signatures are the hashes of the strings). Hash tables support easy addition and removal of values. However, hash tables require a significant amount of memory due to pointers that are stored with each value in the hash table chains. To eliminate the memory needed for pointers, the idea is to store hash chains in blocks in a signature array and use another array to contain offsets to the blocks. The actual values stored are signatures of the strings. Thus, a false positive will occur if an input string hashes to the same offset location (i.e., chain) and same signature value as a non-matching stored string. Figure 4.15 gives an example of the SAHT. A signature array (sigArray) is created to store signatures. They are grouped in blocks and a table called ptrArray is used to access the signatures in a fast way. The block start positions (sigOffset) and lengths of the blocks in number of signatures (blockLen) are stored in the ptrArray.

Figure 4.17 shows the two key algorithms for the SAHT, the variables defined in Figure 4.16. The algorithm mapStrings() maps m strings into the SAHT and uses a hash function to generate hash values, hash1 and hash2, for each string. Lines 1 to 4 create and sort a temporary array called tempArray. The function sortByHash1() sorts tempArray by hash1 values. Lines 5 and 6 create sigArray by copying the hash2 values from tempArray. The rest of the algorithm creates the ptrArray. On exit, two arrays are
defined and comprise the SAHT: \textit{ptrArray} consisting of \(m\ <\text{sigOffset}, \text{blockLen}\) pairs, and \textit{sigArray} consisting of \(m\ \text{stringSig}\) values, where \text{sigOffset} is the offset into \textit{sigArray} for a chain with index hash\(_1\) modulo \(m\), and \text{blockLen} is the length of the chain stored in this block. A temporary array \textit{tempArray} is defined within \text{mapStrings()}. For a 32-bit machine, hash\(_1\) and hash\(_2\) can be defined to be 32 bits, \text{sigOffset} to be 24 bits, chain\text{Len} to be 8 bits, and string\text{Sig} to be 16 or 32 bits. The values of \text{sigOffset} and chain\text{Len} are derived from hash\(_1\) and hash\(_2\) in the function. A 24-bit \text{sigOffset} value limits \(m\) to be less than \(2^{24} = 16,777,216\) (i.e., no more than 16,777,216 strings can be mapped into the SAHT with the variables defined in size as they are here). The arrays \textit{ptrArray} and \textit{sigArray} are used in the \text{testString()} algorithm to test for membership of an input string. This algorithm first generates the block index (block\text{Index}) and the signature of the string (string\text{Sig}) in lines 1 and 2. Since the line 3, the algorithms test for string\text{Sig} since location block\text{Index}.

4.4.2 Analysis

The probability of false positives of the SAHT can be computed as a balls-and-bins problem where the number of balls and bins are equal. For \(m\) strings and \(b\) bits in string\text{Sig},
mapStrings(stringList)
1. for i ← 1 to m do
2.   tempArray[i].hash1 ← hash(stringList[i], 1) mod m
3.   tempArray[i].hash2 ← hash(stringList[i], 2)
4.   sortByHash1(tempArray)
5. for i ← 1 to m do
6.   sigArray[i] ← tempArray[i].hash2
7.   i ← j ← count ← sumCount ← 0
8. while (i < m) do
9.   if (tempArray[i].hash1 = j)
10.      increment i
11.      increment count
12. else if (count = 0)
13.      ptrArray[j].sigOffset ← 0
14.      ptrArray[j].blockLen ← 0
15.      increment j
16. else
17.      ptrArray[j].sigOffset ← sumCount
18.      ptrArray[j].blockLen ← count
19.      sumCount ← sumCount + count
20.      increment j
21.      count ← 0
22. if (count > 0)
23.      ptrArray[j].sigOffset ← sumCount
24.      ptrArray[j].blockLen ← count

testString(inString)
1.  blockIndex ← hash(inString, 1) mod m
2.  stringSig ← hash(inString, 2)
3.  offset ← ptrArray[blockIndex].sigOffset
4.  len ← ptrArray[blockIndex].blockLen
5. if (len > 0)
6.   for i ← offset to (offset + len) do
7.     if (sigArray[i] = stringSig) return(TRUE)
8. return(FALSE)

Figure 4.17. SAHT Algorithms to Add List of Strings and Test String

Figure 4.16. Variables for SAHT Algorithms
\[ \Pr[\text{false positive}] = \sum_{i=1}^{m} \binom{m}{i} \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{m-i} \left( 1 - \left( 1 - \frac{1}{2^b} \right)^i \right). \] (4.28)

In eq. (4.28) the first binomial expression is the probability of a \( i = 1, 2, \ldots, m \) strings mapping to a given block in \textit{sigArray} and the second expression is the probability of the string having a unique \textit{stringSig} in the chain at the given block and assumes that there may already be duplicate \textit{stringSig} values in a chain.

Eq. (4.28) cannot be simplified into a closed form. However, if it is assumed that each chain contains only non-duplicated values, then the second expression is \( i/2^b \). Depending on how many bits are used for the signature, this assumption is reasonable (e.g., 64 bits per signatures gives \( 2^{64} \) possible different signatures). And eq. (4.28) simplifies to,

\[ \Pr[\text{false positive}] = \frac{1}{2^b}. \] (4.29)

In practice, it was found that equations (4.28) and (4.29) compute to numerical values that are close to each other (less than 1\% of difference).

It is solved now for the mean number of look-ups for the SAHT for the case of a miss and hit. Let \( U \) be a random variable for the number of lookups. Then,

\[ E_{\text{miss}}[U] = 1 + \sum_{i=1}^{m} \binom{m}{i} \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{m-i} = 1 + 1 = 2 \] (4.30)

where the initial “1” is for the look-up in the \textit{ptrTable} and the summation is the mean chain length. The mean chain length of SAHT is calculated the same way the length of collision lists are calculated in hash tables that solve the collisions with chaining, and it has been shown to be 1 [31]. For \( E_{\text{hit}}[U] \) it is conditioned on the probability of a block (or bin) being non-empty. Thus,

\[ E_{\text{hit}}[U] = 1 + \frac{1}{1 - \Pr[\text{chain length = 0}]} \] (4.31)

where

\[ \Pr[\text{chain length = 0}] = \left( 1 - \frac{1}{m} \right)^m = \frac{1}{e}. \] (4.32)
Eq. (4.32) follows same analysis than the probability of false positives of a Bloom filter, shown in the Chapter 2. Then, $E_{hit}[U] = 2.582$ for large $m$.

In the SAHT, a mapped string can be removed by setting its mapped $sigArray$ value to zero (this assumes that the hash function always returns a non-zero value). A new string can be added at a somewhat greater cost: $sigArray$ needs to be shifted-down to create a space for a newly mapped-in $stringSig$.

Using the equations in this section, the memory trade-off between the SAHT and Bloom filter is determined. The SAHT will require more memory for a given false positive rate. Table 4.3 shows the numerical results for an SAHT with 1 million strings mapped-in and $b=16$ and $b=32$, and can be compared with the memory requirements for Bloom filters, shown in Table 4.4. For both cases, a Bloom filter with the closest possible false positive rate was selected. It can be seen that the Bloom filter with approximately matching false positive rate to an SAHT with $b=16$ requires about 2.1 times less memory (i.e., 2.875 million bytes compared to 6 million bytes) and for $b=32$ requires about 1.4x less memory.

However, comparing the number of number lookups to determine a hit between the Bloom filter and the SAHT, the SAHT needs fewer lookups ($E_{hit}[U] = 2.582$), than the Bloom filter if $m/n$ is set to a large value (e.g., $m/n=23$ in Table 4.4). For that $m/n$, the value of $k$ is 16, which means 16 lookups are needed when there is a hit in the Bloom filter.

### 4.4.3 Experimental Evaluation

It is key to compare SAHT and Bloom filters in terms of the performance metrics of the Bloom filters. Thus, the response variables are:

<table>
<thead>
<tr>
<th>$Pr[\text{false positive}]$</th>
<th>Memory required</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b = 16$</td>
<td>$1.53 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>$b = 32$</td>
<td>$2.33 \cdot 10^{-10}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Pr[\text{false positive}]$</th>
<th>Memory required</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m/n = 23$ ($k = 16$)</td>
<td>$1.59 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>$m/n = 46$ ($k = 32$)</td>
<td>$2.52 \cdot 10^{-10}$</td>
</tr>
</tbody>
</table>
The factors of interest for the experiments were:

- Ratio of test strings producing a hit.
- Bloom filter parameters \((m/n, k)\).

The ratio of test strings producing a hit was controlled to measure the number of accesses as a function of ratio of hits in the test strings. A Bloom filter and SAHT were implemented in ANSI C to measure the mean number of lookups (and actual processing time) for hit and miss. For the Bloom filter \(m/n = 23\) was used (similar to numerical analysis in the previous section). The optimum \(k\) was selected for this configuration. CRC32 was used for a hash function. One million strings of mean length 50 bytes were mapped and tests with 6 million strings of the same mean length were performed. The gcc 3.4.5 compiler was used under Windows XP, on a Dell PC with a 3.2 GHz Pentium 4 and 1 GB of RAM. Table 4.5 shows the results. The time to test the 6 million strings was calculated and the average per test was calculated and shown in the table. The ratio of strings producing a hit is 100% for the “All hits” row in the table, and 0% for the “All misses” row. It can be seen that the mean number of lookups closely matches \(E_{\text{miss}}[U]\) and \(E_{\text{hit}}[U]\) from Section 4.4.2. The measured CPU time also corresponds to the relative differences in lookup time for miss and hit. This shows that the SAHT requires less processing time than a corresponding Bloom filter.

### 4.5 Bloom Filter Based Keyword Search

Searching for substrings or keywords within a text file comprised of multiple strings is a common application, not only for P2P applications, but for others such as searching text in web pages containing a given keyword. Fast string search algorithms – such as the Boyer-Moore algorithm [17] that is used in the Unix grep utility [9] – were described in Chapter 2. In this section, a keyword search algorithm that uses
small Bloom filters pre-pended to the strings in the string list (e.g., pre-pended to book titles in a list of book titles) to speed-up the search time is described. The algorithm was first presented in [95] but this dissertation presents the first run-time evaluation. The algorithm determines which strings in a list of strings contain a given keyword. Any strings containing the keyword are output. Strings in the text are separated either by new line characters or by punctuation characters, like periods, question and exclamation marks. Words in a string are separated by spaces or some punctuation characters like commas, and dashes, or hyphens.

**4.5.1 Design and Implementation**

The Bloom filter assisted keyword search algorithm has two parts. The first part (or phase) is genList() (shown in Figure 4.19) that pre-processes a list of strings to generate a matching list of Bloom filters, one
for each string. The second part is searchList() (also shown in Figure 4.19) that uses the generated list of Bloom filters to speed-up the search for a keyword in the list of strings. The variables used in both algorithms are shown in Figure 4.18. Each Bloom filter is generated by hashing and mapping each word in a string. Thus, genList is used to pre-process a list of strings in preparation for multiple fast searches by searchList. The input to genList is a list of strings, stringList, and the output is a list of Bloom filters, bfList. Within genList, genBloom returns a Bloom filter generated by hashing and mapping each space-delimited word in inString. The string inString is an input string consisting of one or more space-delimited words. Then, searchList returns all strings in stringList that contain keyWord. The input to searchList is the list of strings stringList, a list of Bloom filters bfList (from genList), and a keyWord. The output is a list of strings, matchList, which contains the strings in stringList that contain keyWord. The variables keyBloom and bloom are Bloom filters. In searchList, scan() searches string inString for keyWord and returns true if a match is found and false otherwise. The purpose of scan() is to test if the keyword is actually found in the string, since not all Bloom filter matches in line 4 will be true (that is, some will be false positives). An index variable (e.g., a loop counter over all Bloom filters and strings) can be used to match the corresponding inString and bloom in line 5. This index variable has to be stored along with the bfList to know where in the text the corresponding string starts.

The Bloom filter assisted keyword search algorithm trades space for search time. Additional space, over that of the input text file strList, is needed for bfList. The size of bfList is a function of the size of Bloom filter used and the number of strings in strList. The search time is a function of the size of the Bloom filter used in bfList, the number of strings in strList, and the mean length of the strings in strList. A larger Bloom filter will minimize false positives resulting in less need for scanning of strings (that is, in line 6 of searchList) and thus a faster search time.

The Bloom filter based keyword search algorithm was implemented in ANSI C. A Bloom filter size of 32-bits (one unsigned integer word) was used. Hashing for both the genList and searchList implementations was realized using the SHA1 hash with a 20-byte hash. For a 32-bit Bloom filter, a single hash would be sufficient for up to \( k = 32 \) index positions. The list of Bloom filters, bfList, generated by the implementation of genList was stored in a separate file. Each entry in the file – consisting of a 32-bit Bloom filter and 32-bit string pointer – corresponded to one string in strList. The pointer has the offset in
bytes where the string starts in \textit{strList}. An example of the implementation is shown in Figure 4.20. The implementation of \texttt{searchList} took as input \textit{bfList}, \textit{strList}, and the keyword to be searched for, and returned a list, \textit{matchList}, with all strings containing the searched-for keyword. The scan() function in the \texttt{searchList} algorithm was called when a Bloom filter returned a hit (either a true or false positive) and scanned \texttt{inString} to determine if \texttt{keyWord} existed in it. The Boyer-Moore-Gosper implementation in [93] was used to implement the scan() function. The implementation of \textit{strList} was instrumented to determine the number of false hits and the overall execution time in CPU seconds. The execution time was measured only for the algorithm implementation and did include the time for overhead such as loading \textit{strList} from storage into memory. The implementations of \texttt{genList} and \texttt{searchList} are openly available.

\textbf{4.5.2 Analysis}

Of interest are the execution time and memory requirements of the Bloom filter based search algorithm compared to existing string search algorithms. Algorithms for fast substring searches have been the subject of much study. The general idea in these algorithms is to traverse, or scan, the text being searched for a particular substring in different ways. The Boyer-Moore algorithm [17] is generally considered one of the fastest substring search algorithms and is used in the Unix implementation of grep [9], [93]. The Bloom filter based keyword search algorithm implemented in C is compared to the Boyer-Moore algorithm as implemented within grep.

The execution time of the \texttt{searchList} algorithm is a function of the number of Bloom filter hits. For a given text file, \textit{strList}, comprised of \textit{n} strings (and thus also \textit{n} Bloom filters in \textit{bfList}), the following

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.20.png}
\caption{A Bloom Filter Based Keyword Search Algorithm Implementation}
\end{figure}
variables are defined: $T_{\text{all}}$ is defined to be the execution time if all $n$ Bloom filters generate a hit and thus all strings are scanned for a match, $T_{\text{none}}$ to be the execution time if no Bloom filter generates a hit, $H_{\text{true}}$ to be the percentage of strings with a true hit for a given test keyword, and $H_{\text{false}}$ to be the percentage of strings with a false hit (where $H_{\text{true}} + H_{\text{false}} \leq 1$). The sum $H_{\text{false}} + H_{\text{true}}$ can be 1, meaning all the keywords searched report a hit in the Bloom filter. Then, the execution time $T$ for a given list of strings is,

$$T = T_{\text{none}} + (T_{\text{all}} - T_{\text{none}}) \cdot (H_{\text{true}} + H_{\text{false}}).$$  \hspace{1cm} (4.33)

The time $T_{\text{none}}$ is dependent only on the number of strings and the size of the Bloom filters. The time $T_{\text{all}}$ is also dependent on the length of the words in the strings and the number of words per string (string length).

The added memory requirement of the keyword search algorithm is a function of the number and length of strings in $\text{strList}$, the size of the Bloom filter used for each string, and (depending on implementation) the size of a pointer associated with each Bloom filter. The pointer locates the start of each string within the string list that is associated with a given Bloom filter. In the implementation, a 32-bit Bloom filter and a 32-bit pointer were used (thus, being able to index into a 4 GByte text file of strings). Thus, for each string in $\text{strList}$ an additional 64 bits of overhead is added in $\text{bfList}$. Bigger Bloom filters could be used but execution time would not be as good as with 32 bits. Processors with 32-bit registers can load Bloom filters of 32 bits faster than those of 64 bits. The chosen size of the Bloom filter is 32 bits as the use of 64 bits generates two much overhead space for small strings.

4.5.3 Experimental Evaluation

The Bloom filter based search was compared with Boyer-Moore string search algorithm, which is the fastest on average [17]. The response variables of interest were:

- Overhead: space required by the algorithm to store the Bloom filters generated when the text is preprocessed.
- Runtime (Time to search for keyword): time taken since the keyword is preprocessed (by generating the Bloom filter for it) until the first match occurs. This time depends on the number of entries of the keyword in the text.

The factors of interest were the following:
Different factor levels needed to be considered. The number of strings in the text was varied. The text was stored in files, which means then that the file size would vary. Runtime was expected to increase as the number of strings increases, but it was necessary to check if this has any effect when comparing Bloom filter based search against Boyer-Moore. The size of the Bloom filter was set to 32 bits. The number of words per string would affect the Bloom filter based search: The more words in the string, the more bits are set to one in the Bloom filter. This affects the probability of false positives of the data structure. If the Bloom filter has few bits set to one, the number of strings that will be checked due to a false positive will be less than if the bits set to one are more. Two different approaches were used to control the number of words per string. The first approach was to control the number of words per string by synthetically generating a file with strings that contain a fixed number of words per string. The words were generated with a RNG method that output characters from the English alphabet; all contained a fixed number of characters (six, which is the average number of characters per English word). Different files were generated, each one varying the number of words per string. The second approach was to use a file with a large number of book titles. The number of words per string was then distributed (for this purpose, the
Library of the University of South Florida provided a list of 2 million book titles. The titles were cleaned; one-letter words and hyphens were removed. Figure 4.21 shows the histogram of the number of words per string for the file provided by the Library. The file had a total of 1,972,847 strings. The mean number of words per string after articles were removed is 10.62 words.

Runtime was measured in all of the following experiments. For this, 10,000 randomly generated testing keywords were searched in the text file. For testing purposes, the keywords were generated in such a way that none of them would be found in the text files. They consisted of strings with characters randomly taken from the English alphabet plus a control character (a “#” or a “!”), which were not included in the text file. The time to test the 10,000 keywords was measured, and the average time to test each keyword was calculated. The experiments were the following:

- **Fixed-length strings experiment**: Synthetically generated strings are used. Different files were used with a fixed size of 80 MBs, but the number of words per string was changed in each of them. The runtime was measured as a function of the number of words per string. The Bloom filter based search against Boyer-Moore search was compared.

- **File size experiment**: Synthetically generated strings were used. The number of words per string was chosen to be 4 (\( n \) in the Bloom filter), and the optimum \( k \) was selected. With these fixed parameters, files of different sizes were generated. File size in this case was calculated as the number of strings per file (varied from 1 million strings per file to 4 million strings per file). Runtime as a function of file size was measured. The Bloom filter based search against Boyer-Moore search was compared.

- **Variable-length strings experiment**: Files with real strings were used (with variable number of words per string). Four files of different sizes were used. The file size was calculated with the number of MB in the file, which varied between 20 MB and 50 MB. The runtime as a function of file size was measured. The Bloom filter based search against Boyer-Moore search was compared.

Results from the experiments are shown in Figures 4.22 through 4.24. The most interesting results are shown in Figure 4.22, the fixed-length strings experiment. The Bloom filter based search starts with a high search time, compared to the rest of the times for the same algorithm. This can be explained by the number of strings per file. Because the file size was fixed, but the number of words per string was varied, the result
of this is that the file for strings with one word per string contains 10 million strings, but the file with strings of 16 words contains 500,000 strings only, which is 20 times less. Testing 10 million Bloom filters takes more time than testing 500,000 Bloom filters. For Boyer-Moore, the increment of the execution time after 4 words needs further exploration. The execution time, however, augments about 20% after 4 words per string are used. It starts in 32 ms and ends in 38 ms for 16 words per string. The file size experiment (Figure 4.23) shows an expected result. The number of words per string was set to \( n=4 \) to be a representative value for a Bloom filter with only a few words. The size of the file affects the execution time, which increases linearly for both algorithms as the file size increases. The time for the Bloom filter
based search was always about a half of the time of the Boyer-Moore implementation. Figure 4.24 shows the variable-length strings experiment results. This time, the Bloom filter based search was about a third of the time spent by Boyer-Moore to search for a keyword.

4.6 Chapter Summary

The Best-of-N method for Bloom filters was presented as a new method to reduce false positive probability with the trade-off of needing to calculate many instances of the Bloom filters to achieve the improvements beforehand. These calculations do not affect the performance of a P2P proxy because they are pre-processing calculations that can be done in the host to be proxied. With the improvement in the false positive rate, fewer Query messages will trigger unnecessary wake-ups of the proxied host due to incorrect answers to Queries generated by the P2P proxy. The Two-tier Bloom filter saves slow memory lookups by using a small Bloom filter that can be located in small but fast memory. This method can be used by the P2P proxy to speed up the search for keywords found in the Query messages received by it. The SAHT allows a limited amount of information to be retrieved when a Query is checked in the data structure. This gives an advantage against a Bloom filter, which only can reply if the keyword (element) is possibly found in the list of shared files (or set of elements). The Bloom filter based string search previously presented [95] was compared for the first time against the Boyer-Moore search algorithm, and it shown to be useful for a P2P proxy. None of the algorithms presented yield false negatives, which fulfills
the requirement (1) (no false negative can be accepted due to the nature of P2P applications) of the list of requirements for string search algorithms for P2P applications defined in section 4.1. All the algorithms presented here improve at least one of the performance metrics of the existing comparable algorithms.

In the next chapter, the design and implementation of P2P proxies is presented. Depending on the device used to implement them, one or more of the data structures and search algorithms described in this chapter will be of use.
Chapter 5: Design and Evaluation of a Gnutella P2P Proxy

The network adapters used by many hosts to connect to the Internet may be a chip on the motherboard, or a separate add-on card. These adapters do not respond to application layer messages, just low layer packets. It is the TCP/IP implementation within the operating system in the host and the applications running in the hosts, which generate and respond to packets and messages. P2P file sharing is one of the applications that require hosts to be on all the time, as described in Chapter 2. In order to enable hosts running a Gnutella P2P application to be proxied, this chapter first describes the assumptions, requirements and goals of a Gnutella P2P proxy, and then describes the design and evaluation two types of Gnutella proxy. The first type is called a Stateless P2P proxy. The main idea is that TCP connections to other peers are reestablished when the host goes to sleep or wakes up, so the state of the TCP connection is not kept. The second type is called Stateful P2P proxy. The main idea here is that TCP connections to other peers will remain open for as long as the host is sleeping and reply to Gnutella messages that require response.

5.1 Specific Requirements for a Gnutella P2P Proxy

The goal of the Gnutella P2P proxy is to enable hosts running a P2P Gnutella application to stay connected to the Gnutella network and continue advertising the shared files while they are sleeping. The assumptions of the Gnutella P2P proxy are the following:

1. If the proxy is required to establish connections to peers, the proxy is able to find at least one peer willing to accept incoming connections from it.

2. The list of files the P2P application is sharing is accessible by the proxy.

3. The computational capabilities of the proxy are sufficient to at least process some of the Gnutella messages directed to the P2P application when the host is sleeping.
Requirements (1), (2), (3), and (10) explained in Chapter 3 for the NCP are needed here. In addition, the following specific requirement is needed for the Gnutella P2P proxy, to fulfill requirement (10) of NCP:

1. The proxy must have the complete list of files shared by the P2P application in the host for which it is covering.

As stated in the requirements of the NCP to support IP connectivity, and the assumptions expected to hold explained in Chapter 3, the device where the P2P proxy is to run needs to have a much smaller incremental energy use than the covered host. For this purpose, two types of devices were targeted. The first design uses the NIC of the host, thus is part of the hardware of the host, either as a peripheral or embedded in the hardware. The second design uses a router inside of the same LAN in which the host to be covered is located. Because the network equipment is powered-on all the time to interconnect devices in the network, the implementation of proxying capabilities on it represents no additional energy use.

The first option is explored locating the proxy directly on the NIC, thus it is a “SmartNIC.” showing Figure 5.1. In Figure 5.1(a) the SmartNIC is operating as a standard NIC passing all packets to and from the fully powered-on PC. When the host goes to sleep, as shown in Figure 5.1(b), the SmartNIC enables proxying and responds to all packets and wakes up the host only when its full resources are needed (e.g., for a file transfer). In particular, the effective use of a Bloom filter is considered to test if a searched keyword is used by any of the files stored (and shared) within a P2P host and make Query message processing in the proxy as fast as possible. The shared files cannot be stored in the proxy due to its limited

Figure 5.1. The SmartNIC With Proxy Capability
storage capabilities. When a request for a file – in the form of an HTTP GET – is received at the proxy, the proxy will wake up the host and the host will then serve the file.

5.2 Stateless P2P Proxy

In this section, the design, implementation and evaluation of a Stateless Gnutella proxy is presented. This proxy will be able to establish and maintain TCP connections to P2P nodes and respond to Query messages. When an HTTP GET request for a file download is received, the sleeping system is woken up and control transfers from the proxy to the host.

5.2.1 Design

Figure 5.2 shows a P2P host with a Stateless proxy. The proxy can be co-located within the host (e.g., on a NIC) or in another device (e.g., a LAN switch/router). This design considers the use of the NIC. The proxy will be resource constrained and cannot store the files shared by the host. Compared to the host, the proxy can be considered a sub-system because it will perform a subset of the tasks performed by the host or an application running on it. If the proxy is a separate sub-system not physically contained within the host, it may have a different IP address from that of the host.

A P2P application in a host includes a main program that supports a user interface for generating Query messages, displaying Query results, and initiating file downloads. The design of the Stateless P2P proxy assumes that the P2P application to be proxied not only has to respond to Query messages, but also forward them to other peers (Gnutella version 0.4 requires it). Query forwarding, as explained in Chapter 2,
is one of the tasks of a peer implementing Gnutella protocol version 0.4, and version 0.6. According to this, the P2P application must also include capabilities for:

1. Initiating and accepting connections to and from neighbors,
2. Receiving and forwarding Query messages,
3. Generating a Query Hit message for a file that matches the searched keyword contained in a Query message, and
4. Serving files that are requested with an HTTP GET.

A P2P proxy need only support a subset of this, which are capabilities for:

1. Initiating and accepting connections to and from neighbors,
2. Receiving and forwarding Query messages,
3. Generating Query Hit messages, and
4. Waking up the host when an HTTP GET is received.

Figures 5.3 to 5.5 show the programs for the host and the programs for the proxy. The first three lines of both programs shown in Figure 5.3 are the same for both: then will listen for new incoming connections, connect to a set of neighbors’ IPs and process Query messages accordingly. The difference between the two programs is that the program running on the host has an interface that gets Query keywords from the user.
of the application, displays results and triggers file downloads if requested by the user (lines 5 through 11).

Figure 5.4 shows the processes and functions common to both the host and proxy. Figure 5.5 shows the processes specific to only the host (the getServer()) or proxy (the redirector()). The getServer is a simple HTTP server listening for incoming HTTP requests for files. In all cases, processes execute in parallel and do not terminate (e.g., as Windows threads), and functions execute and terminate. The redirector() process uses an HTTP 302 redirect message to cause the requesting host to resend its HTTP GET message. The GET can be forced to be resent to the same or another IP address. A delay has to be used before sending the redirection to give to the host time to wake up. Otherwise, the redirected GET might reach the host before the operating system is ready to accept incoming connections.
There is some state information that needs to be shared between the host and proxy, divided into power states and application states. The power states describe the current power state of the host and the application state contains information that describes the current user session of the application. In the case of a P2P application, a user session is established each time the P2P application connects to the network. For that reason, the following is the state information that the P2P Stateless proxy needs from the host:

- Power state of the host – fully powered-on or sleeping.
- List of names of files shared.
- List of IP addresses of neighbor nodes.

The filenames can be shared between the host and proxy in the form of a Bloom filter. Both host and proxy will have to use the same hash functions to test and map filenames in the Bloom filter. The host needs to create a Bloom filter with the list of shared files, and then make it available to the proxy. When control is transferred from the host to the proxy, TCP connections to neighbors from the host are terminated and then re-established from the proxy. When control is transferred from proxy to host, the opposite occurs.

### 5.2.2 Prototype Implementation

To emulate the implementation of a SmartNIC, an Ethernet development kit was selected, based on the requirement that it must have a low cost processor (to lower the costs of the SmartNIC). The P2P proxy was implemented using a NetBurner MOD5270 Ethernet Development Kit [98] (shown in Figure 5.6) with
the following specifications: 32-bit Freescale ColdFire processor running at 147 MHz, 512 KB of Flash memory, 8 KB Instruction/Data cache and 2 MB of SDRAM. The system runs the uC/OS operating system, which is designed for resource constrained devices. The uC/OS can run several tasks at the same time using different priorities, but it can have only one task at each of the priorities it uses. The host was a Dell OptiPlex PC with a Pentium 4 at 3.2 GHz with 1 GB RAM running Windows XP. Both systems support 100 Mb/s Ethernet. The NetBurner was selected for the low-cost processor it uses and for its ability to be programmed in C. A low-cost processor would be necessary to design a low-cost SmartNIC and its use could be popularized. The implementation was first developed in the host using a customized version of a C++ development environment, and then compiled using a customized C++ compiler and transferred to the NetBurner using a serial cable connected to one of the serial ports shown in Figure 5.6. The implementation was transferred as a binary file executable by the NetBurner processor. The serial port was used only for transferring the compiled implementation and for debugging purposes. There were other processes running at the same time in the NetBurner, like FTP server to upload files, and a DHCP client.

The uC/OS operating system runs threads as sequential tasks. Thus, the proxy program was implemented as a single task with a main loop where all the processes were run in a sequential manner. The proxy implementation used non-blocking sockets that were read using time-outs of one processor tick (which is 100 ns). A non-blocking socket was listening on each of the connections open by the proxy with the list of peers to which it has to connect. It looped through all of them looking for Gnutella messages to read in their buffers. For prototype purposes, a Bloom filter with the list of shared files mapped into it was assumed to be already in the proxy. The approach followed was mapping entire filenames without tokenizing them into keywords. This, however, is a constraint because it forces Queries to be for full filenames to produce a match. For the P2P Stateful proxy, other methods to retrieve the list of shared files and to process Query messages were explored. The list of peers to which the proxy has to connect was also assumed to be in the proxy when the proxy program starts.

5.2.3 Experimental Evaluation

The main purpose of the evaluation of the prototype of the P2P Stateless proxy is to show that a subset of the functionalities of a P2P application can be run on a low-power device dedicated to proxy for the host running the P2P application. As described in Chapter 2, one of the most important tasks of a P2P node is to
respond, and forward Query messages accordingly. A key question is then, how much computational
capability is needed in the proxy sub-system in order to maintain a reasonable Query forwarding rate at all
times? The additional delay time to request and receive a file from a sleeping host also needs to be
considered. Three experiments were designed to evaluate the key measures for the implemented prototype
P2P proxy. The two response variables of the experiments were the following:

- File download time from an awake and sleeping host.
- Query messages forwarding rate.

The file download time measures the delay time introduced by the use of a proxy. It is important also to
measure the Query messages forwarding rate to see how much this important variable will be affected. The
factors of interest are the following:

- Number of Gnutella peers (neighbors) to which the proxy is connected to.
- Ratio of Query Hits: percentage of Queries resulting in a match and then generating a Query Hit.

For evaluation purposes, the application for the P2P proxy is also run in the host. In the host, blocking
sockets could be used in a threaded implementation running parallel processes, instead of non-blocking
sockets and non-threaded functions. The idea is to compare the same implementation in both systems to
evaluate how different the performance of the proxy is in each one.

The Query messages forwarding experiment is as follows: For this, a PC running a “Query blaster”
was used. The Query blaster is a C program that establishes a connection to a P2P host and sends Query
messages as fast as possible. To evaluate the Query messages forwarding rate as a function of the number
of neighbors, the first setup for this experiment was as follows. Other PCs were used to connect to as P2P
peers and the number of peers from 1 to 10 was varied. For this experiment, Query messages for files not in
the host were generated, so the proxy would not respond with a Query Hit message. For the second setup of
this experiment, the Query forwarding rate (where a fixed percentage of the Query messages would result
in a Query Hit message being returned) was evaluated. The percentage of Query messages that resulted in
Query Hit messages was varied from 0% to 10%.

The results from the Query forwarding experiments are shown in Figure 5.7. The figure shows the
Query forwarding rate per connection. The Query forwarding rate for the proxy varied from 360 to 130
messages per second. The rate for the host varied from 12,547 to 324 messages per second. These results
show that as neighbors are added, the query forwarding rate per link decreases. The drop in the rate of the host can be explained by the way the QueryHandler process in Figure 5.4 was designed. Because the process loops through all the open connections, as neighbors are added, the process had to reach the last added connection until returning to the first one and processing packets from that connection again. Although there is a noticeable difference between the rates, it has to be noted that the rates measured on the NetBurner are still useful. The study shown in [58] is one of the most complete studies of the Gnutella network including models and traffic characterization. The study logged 604,000 different Gnutella
sessions and 267 million Gnutella messages. They then proposed a model of the traffic of Gnutella
networks, and created a table that calculated the inter-arrival times of all the Gnutella messages according
to the traces they made. Query and Push message were the messages that arrived more often, with mean
inter-arrival times of 0.03 seconds and 0.06 seconds respectively; which means 33 Query messages every
second, and 16 Push messages arriving every second. So, the rate achieved by the NetBurner means that it
can easily process messages arriving at a typical inter-arrival time. The results for the Query Hit experiment
are shown in Figure 5.8. Similar to Figure 5.7, this figure shows the Query forwarding rate per connection.
As the percentage of Query messages resulting in a Query Hit was increased from 0% to 10%, the Query
forwarding rate remained roughly constant for both the proxy and host. This demonstrates that the overhead
to send a Query Hit message is very low.

The File download time experiment is as follows: It took less than 1 second to download from an
awake host. To measure the time in the NetBurner, the proxy program was configured to wake up the host,
wait a delay time of a few seconds before sending the redirection, and then send it. Several attempts with
different delay times were tried. Delay times of less than 9 seconds prevented the host from being ready to
receive HTTP file requests. It then took 9 seconds to download from a sleeping host that had to be woken
up. The wake-up time of Windows XP was the dominant factor in the 9 seconds. The HTTP request did
not time out, in any case.

5.2.4 P2P Application State Conservation With the P2P Stateless Proxy

It has been shown with the P2P Stateless proxy that a subset of the tasks of a P2P node can be run on a
small system like an Ethernet Development kit and still be able to perform the tasks in a way the typical
participation of a P2P node requires. It is necessary now to explore how to design a P2P proxy that acts on
behalf of the host in a way that preserves the state of the application. The P2P Stateless proxy relies on the
proxied host to send the list of peers the host was connected to before going to sleep. The proxy might
encounter the problem of not being able to reconnect to the list of peer IPs given by the host. A reason for
that might be that the remote peer to which the proxy tries to connect might not be able to accept new
connections and might deny the connection attempt from the proxy. In this case, the proxy would have to
attempt connections to other peers, after which the proxy would then be connected to a different list of
peers than the one to which the P2P application in the host was connected. If this happens, it can be
considered that the state of the P2P application is changing while the host is sleeping. When the host wakes up, it will have to reconnect again and possibly, it will end up not connected to the same original set of peers.

Conservation of the state of the application might not always rely on the preservations of the connections with the same group of remote network hosts. This might not be a problem for client applications that, for example, always connect to the same server (or set or servers). IM applications run 3-way chat conversations through the IM servers, and all the IM servers should have the same login information about the user. This applies for all the applications that need to login to an application server.

5.3 Stateful P2P Proxy

In this section, the design, implementation and evaluation of a P2P proxy called Stateful P2P proxy is presented.

5.3.1 Design

The Stateful P2P proxy is a system that consists of two software components: 1) An implementation of the NCP explained in Chapter 3 with the addition of some modifications to proxy for a P2P application, and 2) the P2P Catcher, in charge of processing Gnutella messages when the proxied host is sleeping. The P2P Catcher is divided into two components: the Gnutella tracer and the Files list manager. Figure 5.9 shows a P2P host being proxied by the P2P Stateful proxy located in the last hop router/switch to which the host is connected. The host has to establish connections to other peers through the proxy. The proxy establishes
the connections to the peers and then forwards all the packets in both directions. The Gnutella messages flow over TCP connections in both parts of the connection: between the host and the gSOCKS, and between the gSOCKS and the peers in the network. When the host is sleeping, the P2P Catcher traces Gnutella messages directed to the host. It cares only for Query messages sent towards the sleeping host, keep-alive messages in the form of Gnutella Pings, and requests for files. Query messages are answered with Query Hit messages if there is a match in the list of shared files with the keyword the Query contains, and Ping messages replied to with Pong messages. File requests in the form of HTTP GET messages trigger host wake-ups. Other messages are ignored.

There is some application state information that needs to be shared between the host and the proxy:

1. The sleep state of the host.
2. Information pertinent to the peer itself. This includes: the Gnutella identifier (the unique 16-bytes identifier used by Gnutella protocol to identify some of the messages sent by the host and connection speed.
3. The list of shared files: information used by the host to respond to Query messages, including file names, file sizes, and checksum of the files being shared, among other file information fields.

The software components are shown in Figure 5.10, as long as the control paths are used by the components. The Stateful proxy is a modified implementation of the NCP, which adds the P2P Catcher to the system. The software components are the following:

- Control point: software in charge of setting configuration variables depending on the sleep state of the host.
- gSOCKS: the SOCKS relay server used to relay TCP connections with Gnutella peers.
- Gnutella tracer (part of P2P Catcher): software that traces messages sent to the same port to which the power managed host is listening, and filters and processes the important messages.
- Files list manager (part of P2P Catcher): software to retrieve the list of shared files from the host and to perform searches on the list when a Query is detected by the Gnutella tracer.

The following is the design of the control program of the NCP. The main task of the control program is to detect the sleep state of the proxied host. For this, the data path (4) is used. A TCP connection is open when the host wants to transmit its state, and closed once it finishes transmitting it. It is also in charge of
communicating the state to other components so that tasks on those components can be performed accordingly. It first uses control (1) in Figure 5.10 to force the gSOCKS to stop forwarding messages to the sleeping host. It also uses control (2) to activate the Gnutella messages tracing functionality of the P2P Catcher. It uses control (3) to change the TCP variables TCP_MAX_RTO and TCP_MAX_RETRIES2 to enable the TCP connections between the host and the relay server to remain open for long periods of time. The values set into these variables can be tuned for an expected host sleep period. When on the contrary, the control program detects the host is awake, the message forwarding will restart, Gnutella messages will not be traced, and the TCP variables explained above needs be set to the original value.

The design of gSOCKS is as follows. It implements a server program using the SOCKS protocol that works as relay server for the TCP connections. The program has a pair of open TCP sockets for each of the

Note: Control (1) communicates host sleep state to gSOCKS. Control (2) communicates host sleep state to P2P Catcher. Control (3) is used to change the TCP retransmission time-out. Data path (4) is used to send response and wake-up packets to the host.

Figure 5.10. Architecture of the P2P Stateful Proxy
connections the host establishes with its peers. Each pair consists of a socket for a connection relay to host, and a socket for the connection relay to peer. It constantly checks on both sockets for messages that will be forwarded to the other socket in the pair once detected in the reading buffer of the socket. The packet forwarding process in the SOCKS server occurs in two different ways, depending on the host sleep state.

They are called rules and work as follows:

- **Awake rule**: action taken when host is awake. There is no specific action which means that messages are forwarded as if the relay server did not implement any proxy capabilities. Messages are passed down to the TCP stack. However, the gSOCKS is checking regularly (every 1 second) for a signal triggered by the control program to determine that the host is sleeping, and that changing the forwarding rule is necessary.

- **Sleep rule**: action taken when the host is sleeping. When the gSOCKS detects that the host is sleeping, it will stop forwarding messages to the sockets listening from the connection established with the sleeping host. The TCP stack of the router will not see any of the packets sent by the peers, because the messages are not being forwarded by the gSOCKS.

The Sleep rule is used to prevent messages from being buffered in the TCP stack when the host is sleeping. In the case of a proxy for P2P, this is important because peers that send Query and Ping messages time out to receive responses. Responses received by the other peers after the time-out are discarded.

The Gnutella tracer follows the same approach followed in Chapter 3 for the SIP Catcher. It consists of a program that runs as one of the components of NCP, and is in charge of detecting specific Gnutella messages sent to the proxied host when it is sleeping, and HTTP requests for a shared file. Figure 5.11 shows the decisions the program takes when a message to the listening port is received. From all the Gnutella messages, the P2P Catcher only cares for the Query, Ping, and Push messages. As the figure shows, Ping and Query messages require the generation of a response. Those messages are then discarded. There should not be Query Hit messages directed to the host because the host should not have triggered a Query, and Pong messages should not be received because they are sent in response of a Ping (which was not sent by the sleeping host). A detected Ping, however, might mean that a peer to which the host is connected wants to test the connection to see if the host is gone or not. Thus, the response of a Ping is important in order to keep the connections with other peers open while the host is sleeping. Query
messages should be responded to so that potential file downloads could be triggered by other peers wanting files being shared by the host. In this way the host continues sharing files even if it sleeping. A request for a file might be received by the P2P Catcher in two different ways, depending if the sleeping host is behind a firewall or not. If it is behind a firewall, Push messages will be received. They mean that there is an attempt to request a file. The approach to be followed is to queue that message and wake up the host. Once the host is awake, the P2P Catcher should deliver the message to the host. A Push should make the power managed host attempt to open a new TCP connection with the peer that sent the Push. This action must only be done by the host once it is awake, because it has the file that will be requested through that new TCP connection. The peer should then request the file (through an HTTP GET). Direct HTTP GET requests should be

Note: Queued messages are delivered to the host when it wakes up.

Figure 5.11. Gnutella Message Processing by the P2P Catcher
detected on the same port used to exchange Gnutella messages. An HTTP GET detected by the Gnutella
tracer generates an HTTP Redirect (302) message as response, which carries an IP to which the file request
should be sent. This Redirect should make the requester peer send the request again to that IP, which is the
same as the power managed host. Before the P2P Catcher sends the HTTP response, it should wake up the
sleeping host.

The design of the file list manager is as follows. It is a software component that is in charge of
retrieving the list of shared files and performing searches for the keywords received in Query Hits. For this,
no modification of the P2P application on the proxied host should be needed. The application selected for
the prototype implementation should use a mechanism to advertise the list of shared files if another peer
requests it. The list is then stored in a data structure. One of the string search algorithms presented in
Chapter 4 will be used as the search method when the P2P Catcher detects a Query directed to the host. The
one that best fits is the Bloom-filter-based search algorithm. This algorithm is able to retrieve information
from files (e.g. the list of shared files with properties for each of them) and it was shown to be faster than
other string search algorithms. Because the device where the NCP is located is a router, some memory can
be traded in order to be able to respond to Query messages with exact matches. The preprocessing part of
this algorithm is applied to this data structure, which generates two files stored in the file system of the
router. This part first creates two files. It first creates a file containing only the filenames, stored in the
same order as in the list of file names. That file is used to create a file with a list of Bloom filters that have
the keywords of the filenames mapped in, and the offset in the file with filenames, so it is easy to locate of
all the information for a file when its Bloom filter returns a match for a keyword.

5.3.2 Prototype Implementation

Figure 5.12 shows the test-bed where the P2P Stateful proxy was implemented and evaluated. The
power managed host uses a Gnutella P2P application that connects to the P2P network through a SOHO
router. A Gnutella ultrapeer running in another host is used as one of the ultrapeers to which the power
managed host is connected as a leaf peer. For testing purposes, another host running as a leaf peer is
connected to the same ultrapeer. With this setup, the leaf will be used as the starting point of Query
messages to be sent to the power managed host.
The implementation of gSOCKS is as follows. The last hop router is a Linksys WRT54G version 3. As for the NCP implementation in Chapter 3, the original Linksys firmware was replaced with an OpenWrt firmware to allow installation of custom made applications called packages. The control program of the NCP is a package running on the router. The most important task is to listen for sleep state changes on the power managed host. It then writes the sleep state to a file that will be read by the other components of the NCP. One of the applications installed in the firmware is the Srelay package, which is a relay server, and is the main part of the gSOCKS. For the P2P Stateful proxy, the relay server had to be modified to implement the message forwarding rules explained in the design. On the main function of the Srelay program, a new process is created that will check the sleep state file that the NCP control program updates when there is a sleep state change. This process was set to sleep 100 ms and updates a global variable with the most current sleep state. This time was selected so that the forwarding rules explained above could be changed as fast as possible, and without generating too many processing requirements on the router processor. A function on the relay server is in charge of forwarding the messages in both directions between a pair of open sockets. This function was modified to check this global variable each time a message was going to be forwarded from one socket to another one.

The P2P application used in the proxied host and the ultrapeer is Limewire. Limewire is an open source software project that is implemented using Java [84]. The Limewire project was modified for prototype purposes only. The list of modifications is the following:

1. The leaf was modified to stop using compression algorithms to send the Gnutella messages. Once this was done, Limewire sent the messages in plain ASCII text, and the P2P Catcher was able to
trace Query and Ping messages. This modification is not mandatory; however, it was simpler for prototype purposes to make the modification in Limewire. A disadvantage of using uncompressed Gnutella messages is that it increases the traffic in the network. This issue was not explored in this dissertation.

2. In the ultrapeer some modifications were also needed. Limewire will connect to the Gnutella network initially as a leaf. To be able to have a manageable environment, Limewire was modified to force it to be an ultrapeer every time it connects to the network.

3. The ultrapeer was modified to prevent it from closing idle connections. An open connection with another peer is considered an idle connection after a timeout is reached. This timeout is reached if no message is received from that connection. Once considered idle, Limewire will send a Ping to it, from which it expects a Pong. If no Pong is received it will close the connection after a few minutes. The timeout used was set to 12 hours, meaning that no connection is considered to be idle before 12 hours. The proxied host was expected to sleep for less than 12 hours.

The P2P Catcher implementation is a program installed as a package in the Linksys router. The main functionality is for the Gnutella tracer. The file list manager implementation is a function inside of the P2P Catcher that is called when the P2P Catcher package is started (it is assumed that the power managed host is up and the Limewire is running when the P2P Catcher is started). Limewire provides a mechanism that is not Gnutella compliant to advertise the list of shared files. The file list manager sends an HTTP GET request to the same port the Limewire uses to listen for incoming connections. If the Limewire is running, it will reply with the list of shared files, containing a file index, the filename, size in bytes, and a 16 bytes checksum. Besides the information of the files, Limewire sends the 16-bytes Gnutella unique identifier that it uses to generate some of the Gnutella messages. This is key for the P2P Catcher to have, because it will be used to generate the Query Hit messages, as recommended by the Gnutella specification.

5.3.3 Experimental Evaluation

For the evaluation, the test-bed shown in Figure 5.13 is used and connected to the Internet through the USF network. It consisted of three Dell PCs, which act as one ultrapeer, and two leaves, the two connected to the ultrapeer. The three PCs used had Intel Pentium 4 processors at 3.2 GHz, and run Windows XP. The ultrapeer has 2 GB of RAM memory and the two leaves 1 GB of RAM. The P2P application running is
Limewire, version 4.18. The proxied host is connected to the LAN of the Linksys WRT54G router shown in the figure. The WAN interface of the router is connected to the network and uses IP 131.247.2.49. The other leaf and the ultrapeer are directly connected to the network of the university and use IPs 131.247.3.150 and 131.247.3.170 respectively. To test that the proxy was successfully working, Wireshark was run on the leaf (IP 131.247.3.150). The purpose of this evaluation is to show that the proxy is fully acting on behalf of the host by responding to Query messages when the host is sleeping, and also, that no buffering of messages sent to the host occurs when the host is sleeping. To show this, the following steps were followed:

1. With the proxy off, and the host being awake, a Query was generated for a keyword containing the word “depeche” on the leaf and the reply of the host was traced.

2. With the proxy on, the gSOCKS on the proxy using the sleep rule, and the host sleeping, a Query for the same keyword was generated and the reply of the proxy was traced.

3. With the proxy on, but the gSOCKS using the awake rule, the host was put to sleep. Three different Query messages from the leaf were generated. The host was woken up, and the messages were traced.
The trace for step 1, taken in the leaf, is shown in Figure 5.14. The Query message is not shown in the trace. It can be seen message number 27 is the Query Hit, and part of the content of the message is highlighted on the right (the name of the file is Depeche Mode Tribute-Ponytail Girl.mp3). Messages 25 and 26 are an exchange of messages between the leaf and the proxied host that does not follow the Gnutella specification (Query Hits are accepted without any need of extra handshaking, as long as a Query has been generated). Figure 5.15 shows a trace obtained in the leaf, which shows the Query response of the proxy.
while the host was sleeping. It can be noted that there is no extra exchange of messages between the leaf and the proxy because this part was not implemented. After the host was woken up, no messages are sent by the host in replay to the Query sent when it was sleeping. This means that the sleep rule worked accordingly by not forwarding the messages to the host when it was sleeping. For step 3, the trace is shown in Figure 5.16. It can be seen that the host tries to start the same message exchange with the leaf as it is shown in Figure 5.14. However, the leaf might have timed out to receive responses for those three Query messages. So, for that reason, no response from the leaf is seen in the trace.

5.3.4 P2P Application State Conservation With the P2P Stateful Proxy

The state of the P2P application being proxied by the P2P Stateful proxy is preserved during the sleep period of the host. Because the TCP connections are being relayed with gSOCKS, they will remain open for as long as the host remains in the sleep state. This assures that the P2P application on the host remains connected to the same set of peers all the time (as long as the remote peer does not disconnect from the network). As explained in Chapter 2, the time the host remains connected to the network, among other factors, is used to promote the P2P application to become an ultrapeer, according to Gnutella version 0.6. One of the consequences is that the peer will reach more peers when generating Query messages. This is the desirable advantage of becoming an ultrapeer, and is reachable by keeping the TCP connections open even if the host goes to sleep.

Figure 5.16. Messages Sent by the Host if Proxy Does Not Discard Messages
Applications that need to preserve the TCP connections with remote servers might benefit from this design. It has been shown already in Chapter 3 that applications like SSH, which establish a session when connected to the SSH server, benefit from the idea of relaying TCP connections to allow the host to sleep.

5.4 Chapter Summary

In this chapter, two types of P2P proxies for P2P Gnutella applications were presented. The first proxy, P2P Stateless proxy, allows the host to close the TCP connections established with the peers before going to sleep. The proxy then reestablishes the TCP connections to the list of peers to which the covered host was connected. When proxying, it is able to respond and forward Query messages directed to the host. It was also shown that this proxy is able to recognize HTTP requests for files directed to the host, and wake up the host to prepare it to serve the file request. In the evaluation of this proxy, it is shown that although the Query message forwarding and response rates are below those rates achievable by the host, this still can be acceptable given existing studies about typical inter-arrival times of Gnutella messages of this type.

The second proxy, P2P Stateful proxy, is located in the last hop router and covers for a host by keeping the connections open for as long as the host is sleeping. It does so by using a software component that consists of a relay server that proxies the TCP connections established by the host. It prevents modifications to be made in the application running host. The use of the Bloom filter based string search algorithm to process keyword searches was explained. The proxy was shown to successfully cover for Query messages sent to the host.
Chapter 6: Summary and Directions for Future Research

This dissertation presents the first investigation of application-level proxying to reduce induced energy use in network hosts. The work presented in this dissertation has been published in several conference papers which are [69], [70], [71], [67] and [68]. Work at UCSD (the Sominoloquy project [4]) and at other universities has built on, (and cites) the work presented here. Most significantly, the work undertaken for this dissertation has directly influenced the Ecma standard (TC32-TG21 – Proxying Support for Sleep Modes) [34], evidence of which is the direct reference to [69] in the draft Ecma standard.

To enable proxies to be lightweight in their processing and memory requirements (and thus be both low cost and low power) improved Bloom filters were investigated. It was shown how Bloom filters are an ideal tool for keyword searching as needed for proxying P2P protocols such as Gnutella. In Chapter 4 a Bloom filter based keyword search algorithm was developed and measured. This algorithm had been previously proposed by Mullin [95]. The work in this dissertation was the first to fully measure the speed-up made possible from using Bloom filters to assist keyword search in a list of strings. Also in Chapter 4 a new method (the Best-of-N method) of mapping elements into a Bloom filter so as to reduce the false positive rate was shown. Best-of-N was shown to be better than existing improvements – such as the Power-of-2 Bloom filter [86] – for small \( m/n \) ratios. It is exactly small Bloom filters with small \( m/n \) ratios that are of interest for the Bloom filter assisted keyword search algorithm. The better Bloom filters – and their application to a fast keyword search algorithm – will directly improve the performance of proxies, such as the P2P proxy described in Chapter 5. This will especially be the case for P2P proxies that cover for multiple leaf nodes sharing millions of files.

The work in better Bloom filters has significant value beyond proxies. Theoretical work in understanding the true false positive rate has now been done and submitted to a journal as [27]. This new work is beyond the scope of this dissertation and was not presented herein.
6.1 Predicted Energy Savings From Application Level Proxying

Depending on the application being proxied, use of the methods described in this dissertation could achieve the following:

- **Gnutella P2P applications:** Assume 10 million PCs running Gnutella P2P applications at all time [105], of which 60% are left on all the time [115]. Using conservative calculations, the idle time of those PCs spent during night (around 10 hours) yields 2,190 GWh/year at 100 W per PC. Given the fact that, only about 1% of the idle time is used to upload files, 99% of the idle time is wasted. Assuming the network connectivity problem due to the use of P2P applications is solved for at least 25% of those PCs with the proxy presented here, the savings could be 542 GWh/year, which is approximately $54 million dollars a year.

- **IP hard-phones running SIP protocol:** There is potential for saving all the energy wasted due to idle time of IP hard-phones, which amounts to 443 GWh/year. With the SIP Catcher proposed in Chapter 3, IP hard-phones could be put to sleep and only be woken up to receive a call. Assuming 25% of those IP hard-phones could be proxied, the savings could be 111 GWh/year, which is approximately $10 million dollars a year.

The savings presented here are considered conservative because in both cases only 25% of the hosts are assumed to be using one or more of the ideas proposed. Also, calculations are only for the applications proxied with the prototypes implemented during this research. Even if only a fraction of the above savings can be achieved, this research will still have made a significant impact in reducing the CO₂ footprint of modern society.

6.2 Directions for Future Research

This research has solved several problems in the area of reducing induced energy use of network hosts, but it has also raised new questions and directions. Figure 6.1 shows a taxonomy of future directions, or next steps, for proxying. Not all of these directions, however, are likely to be fruitful. Fundamentally, one can explore how to improve proxies or finding new uses of proxies. For improving proxies, open questions include:
1. Can a proxy be made completely transparent to the host(s) and applications for which it covers? Proxying for network applications has been shown here to be highly dependent on the application to be proxied. It is necessary to know the application that is to be proxied and certain application session data like the ports, types of messages that need to proxied and any personal account information if any is used during the execution of the application. The proxy should be able to learn what needs to be proxied, and do so without modification of the operating system or the network applications. Although there has been research on what needs to be proxied (e.g. [97]), the results presented on that work reflects just a snapshot of the popular applications today. Proxying should be independent of the type of the network applications that need to be covered.

2. Can a proxy support mobility for wireless hosts? The design of the NCP presented in Chapter 3 relies on connections that are not prone to disruption. In wireless networks, the typical mobility of hosts might lead to disruption in the connectivity. To solve the network connectivity problem for applications running on wireless hosts, the design of the proxy for application connections has to take into account the wireless environment, and overcome problems such as connections being closed due to loss of connectivity to the network blamed on the mobility.

3. What are the potential energy savings of creating a proxy for an application server instead for hosts? The proxy was explored as physically located close to the proxied host (e.g., on the same LAN). This research did not require modifications of the proxied applications. However, to proxy for certain types of network applications, personal user account information to login to the application servers might be needed. One such example is Skype, which uses a proprietary technology [52]. This might prevent some users from sharing their personal information with the
proxy to act on behalf of their applications. For these types of applications, it would be interesting to explore new types of proxies for network application servers. This means that the hosts can use a network connectivity proxy that is in fact part of the services provided by the same company running the application server.

4. What are the computational requirements for a proxy that can serve files on behalf of a sleeping host? Proxying for P2P applications have been explored in this dissertation to enable hosts to sleep when idle. As described on Chapter 4, requests for files in a P2P application follow a Zipf-like distribution, where a very small group of files are requested most of the time. An approach similar to the two-tier Bloom filter could be used in the proxy to cache frequently-requested files in hosts with a large amount of popular shared files. There are some hosts that receive file requests more often than others, because they share a large amount of popular files. With this proxy, hosts could sleep even when constant requests for frequent files arrive because they are served by its proxy. It is important to explore what would be the trade-off between the amount of cached files and the length of sleep periods.

These future directions can be fruitful paths for research, resulting in even greater impact from proxying as a means of reducing the CO₂ footprint of ICT.
References


[38] Energy Information Administration (EIA), 2009. URL: http://www.eia.doe.gov/.


About the Author

Miguel Jimeno received his B.Sc. in Systems Engineering degree from Universidad del Norte, Barranquilla, Colombia, in 2002. He is presently a Ph.D. candidate in the Department of Computer Science and Engineering, at the University of South Florida, Tampa, Florida. His research interests are in performance evaluation of computer networks, and network connectivity proxying of computers. He is a student member of the IEEE.