MODELING AND CACHING OF PEER-TO-PEER TRAFFIC

by

Osama Saleh
B.Sc., La Roche College, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Osama Saleh 2006
SIMON FRASER UNIVERSITY
Fall 2006

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
## APPROVAL

<table>
<thead>
<tr>
<th>Name:</th>
<th>Osama Saleh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree:</td>
<td>Master of Science</td>
</tr>
<tr>
<td>Title of thesis:</td>
<td>Modeling and Caching of Peer-to-Peer Traffic</td>
</tr>
</tbody>
</table>

**Exposing Committee:**
- Dr. Robert Cameron, Chair
- Dr. Mohamed Hefeeda, Senior Supervisor
- Dr. Ramesh Krishnamurti, Supervisor
- Dr. Joseph Peters, Examiner

**Date Approved:**

November 24, 2006
DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Revised: Fall 2006
Abstract

Peer-to-peer (P2P) file sharing systems generate a major portion of the Internet traffic, and this portion is expected to increase in the future. We explore the potential of deploying proxy caches in different Autonomous Systems (ASs) with the goal of reducing the cost incurred by Internet service providers and alleviating the load on the Internet backbone. We conducted an eight-month measurement study to analyze the P2P characteristics that are relevant to caching, such as object popularity, popularity dynamics, and object size. Our study shows that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution, and that several workloads exist in P2P traffic. Guided by our findings, we develop a novel caching algorithm for P2P traffic that is based on object segmentation, and proportional partial admission and eviction of objects. Our trace-based simulations show that with a relatively small cache size, less than 10% of the total traffic, a byte hit rate of up to 35% can be achieved by our algorithm, which is close to the byte hit rate achieved by an off-line optimal algorithm with complete knowledge of future requests. Our results also show that our algorithm achieves a byte hit rate that is at least 40% more, and at most 200%, the byte hit rate of the common web caching algorithms. Furthermore, our algorithm is robust in the face of aborted downloads, which is a common case in P2P systems.
To my family with love.
“The important thing is not to stop questioning. Curiosity has its own reason for existing.

One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity!”

— Albert Einstein
Acknowledgments

I am deeply indebted to my senior supervisor, Dr. Mohamed Hefeeda, for his continuous support, encouragement and guidance through my research. By working with him, I not only know more about distributed and P2P systems, but I understand how to do research. Mohamed provided valuable insights, a lot of help during my graduate career, came on weekends before paper deadlines and made numerous editing and proof-reading of the material presented in this thesis. This thesis would not have been possible without him.

I would like to thank my supervisor Dr. Ramesh Krishnamurti and my thesis examiner Dr. Joseph Peters for being on my committee and reviewing this thesis. I would like to thank Dr. Rob Cameron for taking the time to chair my thesis defense.

I would also like to extend my gratitude to the faculty and staff in the school of computing science at SFU, particularly Dr. Jiangchuan Liu, Dr. Petra Berenbrink and Dr. Qian-ping Gu for what I have learned in their courses.

The following students, friends, and staff made my experience at SFU rewarding and fun: Shameem Sattar was my first lab-mate, and I had numerous discussions with him about caching and content distribution in the Internet; Cheng-Hsin Hsu helped me with Matlab curve fitting; Patrick Lougheed provided a lot of technical support for my frequently failing lab machine; and Murray Patterson helped me laugh at some of the ugliest moments.

I would like to thank Dr. John Todhunter, Dr. Leland McCauley, Dr. Brian Smith and all of La Roche College for their efforts in educating me and getting me interested in computer science. I learned a lot about Oracle database in tutorial sessions run by Dr. John Todhunter in the Summer of 2003. This knowledge helped me tremendously in analyzing the trace data I used in this research.

Last but certainly not least, I would like to thank my family for their unquestioning love, care and support. I would not have made it without them. They inspired me to strive
beyond the ordinary, encouraged me to keep going when I thought I have given it all I can, and gave me the confidence I needed to make it through life. Their expectations have always propelled me to realize my full potential. This thesis is dedicated to them.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Quotation</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Contents</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction and Background

1.1 Introduction .................................. 1

1.2 Previous Work ................................ 3

1.3 Thesis Contributions .......................... 4

1.4 Thesis organization ............................ 6

## 2 Modeling Peer-to-Peer Traffic

2.1 Gnutella P2P System ............................ 7

2.2 Measurement Methodology ........................ 8

   2.2.1 Trace Collection and Processing ........... 9

   2.2.2 Measurement Statistics ....................... 9
2.3 Measuring and Modeling Object Popularity ........................................ 10
2.4 The Effect of Mandelbrot-Zipf Popularity on Caching ...................... 16
  2.4.1 Popularity Dynamics .................................................................. 19
  2.4.2 Measuring and Modeling Object Size ........................................ 21
  2.4.3 Summary of the Measurement Study ......................................... 23

3 P2P Caching Algorithm and Evaluation .................................................. 24
  3.1 P2P Algorithm ............................................................................. 24
  3.2 Evaluation .................................................................................. 27
    3.2.1 Experimental Setup ............................................................. 27
    3.2.2 Caching without Aborted Downloads .................................. 29
    3.2.3 Caching with Aborted Downloads ..................................... 31
    3.2.4 Sensitivity of the P2P Caching Algorithm to Temporal Locality .. 32
    3.2.5 Partial Caching versus Full Caching ................................... 34
    3.2.6 Effect of $\alpha$ and $q$ on P2P Caching ............................... 36
    3.2.7 Effect of Segmentation on P2P Caching ............................... 39

4 Conclusions and Future Work ................................................................. 41
  4.1 Conclusions ................................................................................ 41
  4.2 Future Work .............................................................................. 42

A Data Processing .................................................................................. 44
  A.1 Data format ............................................................................... 44
  A.2 Data processing ........................................................................ 46
  A.3 Data Analysis ............................................................................ 46
    A.3.1 Storing the data ................................................................. 47
    A.3.2 Querying the data ............................................................. 48

Bibliography .......................................................................................... 50
List of Tables

2.1 Summary statistics of the measurement study. .................. 10
2.2 Object popularity in several AS domains. ...................... 16
List of Figures

1.1 The P2P proxy cache sits at the edge of an Autonomous System (AS) 5

2.1 Object popularity in P2P traffic in different ASes. The popularity can be modeled by Mandelbrot-Zipf Distribution. Plots are sorted from left to right based on the $q$ values. 11

2.2 Continued from Fig 2.1. Object popularity in P2P traffic: in different ASes (a)-(b), and across all ASes (c). The popularity can be modeled by Mandelbrot-Zipf Distribution. 12

2.3 Zipf versus Mandelbrot-Zipf for different $q$ (plateau factor) values. 14

2.4 The impact of number of hosts and the average number of downloads in an AS on the plateau factor $q$. Data shown for the top 20 ASes. 17

2.5 Effect of Mandelbrot-Zipf popularity distribution on the cache performance using: (a)LFU (analytic), (b)LRU (simulation) 18

2.6 Effect of Mandelbrot-Zipf popularity distribution on the byte hit rate using optimal algorithm (trace-based). 18

2.7 Popularity dynamics in P2P systems. The figure shows popularity of the 100 most requested objects in (a) the top 1st AS, (b) top 2nd AS. 20

2.8 Popularity dynamics in P2P systems. The figure shows popularity of the 100 most requested objects in all ASes. 20

2.9 Distribution of object sizes in P2P systems in (a) Gnutella, and (b) BitTorrent. The figures show that the size distribution has multiple peaks which could be attributed to the multi-workload nature of P2P traffic. 21

3.1 Proportional Partial Caching Algorithm for P2P traffic. 27

3.2 Byte hit rate for different caching algorithms. No aborted downloads. 29
3.3 Relative byte hit rate improvement of P2P algorithm compared to traditional caching algorithms in the top ten Ases. No aborted downloads. 

3.4 Byte hit rate for different caching algorithms using traces with aborted downloads.

3.5 Relative byte hit rate improvement of P2P algorithm compared to traditional caching algorithms in the top ten Ases with aborted downloads.

3.6 Byte hit rate for different caching algorithms using traces with preserved popularity and temporal correlations.

3.7 Sensitivity of P2P caching algorithm to temporal correlations using the LRUSM from citeBW02.

3.8 The importance of partial caching in P2P traffic. The figure shows the byte hit rate achieved using our P2P algorithm with and without partial caching in AS95 (a), and in top ten ASes (b).

3.9 Byte hit rates achieved by our online heuristic partial caching algorithm (P2P) and by the MKP offline optimal in top ten ASes.

3.10 The impact of $\alpha$ on caching.

3.11 The impact of $q$ on caching.


A.1 A raw query message

A.2 A query hit message
Chapter 1

Introduction and Background

In this chapter, we describe the problem of caching Peer-to-Peer (P2P) traffic and its importance. We differentiate between P2P systems and web and video streaming systems justifying the need for new caching algorithms. Then, we describe the related work and summarize the contributions of this thesis.

1.1 Introduction

Peer-to-peer (P2P) systems are distributed systems in which participating computers (peers) cooperate by sharing resources. The resource can, for example, be CPU cycles, files, storage space, communication services and bandwidth, or real-time media streaming. P2P systems have been mostly used for P2P file-sharing where each peer acts as a server (by serving files to others) and a client (by requesting files from others) at the same time.

P2P file-sharing systems have gained tremendous popularity in the past few years. More users are continually joining such systems and more objects are being made available, enticing more users to join. Currently, traffic generated by P2P systems accounts for a major fraction of the Internet traffic [8], and it is expected to increase [13]. Some studies show that traffic generated by P2P file-sharing systems has surpassed Web traffic. The sheer volume and expected high growth of P2P traffic have negative consequences, including: (i) significantly increased load on the Internet backbone, hence, higher chances of congestion; and (ii) increased cost on Internet Service Providers (ISPs) [15], hence, higher service charges for their Internet users. A potential solution for alleviating those negative impacts is to cache a fraction of the P2P traffic such that future requests for the same objects could be
served from a cache in the requester's autonomous system (AS). An autonomous system is a collection of hosts and routers that belong to one administrative domain.

Caching in the Internet has mainly been considered for web and video streaming traffic, with little attention to the P2P traffic. Many caching algorithms for web traffic [22] and for video streaming systems [19] have been proposed and analyzed. Directly applying such algorithms to cache P2P traffic may not yield the best cache performance, because of the different traffic characteristics and caching objectives. For instance, reducing user-perceived access latency is a key objective for web caches. Consequently, web caching algorithms often incorporate information about the cost (latency) of a cache miss when deciding which object to cache/evict. Although latency is important to P2P users, the goal of a P2P cache is often focused on the ISP's primary concern; namely, the amount of bandwidth consumed by large P2P transfers. Consequently, byte hit rate, i.e., minimizing the number of bytes transferred, is more important than latency. Moreover, P2P objects tend to be larger than web objects [8, 25] reducing the number of complete objects that can be held in a cache.

Furthermore, although objects in P2P and video streaming systems share some characteristics, e.g., immutability and large size, streaming systems impose stringent timing requirements. These requirements limit the flexibility of caching algorithms in choosing which segments to store in the cache. Therefore, new caching algorithms that consider the new traffic characteristics and system objectives need to be designed and evaluated.

In this thesis, we first develop a deeper understanding of the P2P traffic characteristics that are relevant to caching, such as object popularity, object size and popularity dynamics. We do that via an eight-month measurement study on a popular file-sharing system. Then, we design and evaluate a novel P2P caching algorithm for object admission, segmentation and replacement.

We perform trace-based simulations to evaluate the performance of our algorithm and compare it against common web caching algorithms, such as Least Recently Used (LRU), Least Frequently Used (LFU) and Greedy-Dual Size (GDS) [4], and a recent caching algorithm proposed for P2P systems [35]. Our results show that with a relatively small cache size, less than 10% of the total traffic, a byte hit rate of up to 35% can be achieved by our algorithm, which is close to the byte hit rate achieved by an off-line optimal algorithm with complete knowledge of future requests. Our results also show that our algorithm achieves a byte hit rate that is at least 40% more, and at most triple, the byte hit rate of the common web caching algorithms.
1.2 Previous Work

We first summarize previous P2P measurement studies, justifying the need for a new study. Then, we contrast our caching algorithm with other P2P, web and multimedia caching algorithms.

Several measurement studies have analyzed various aspects of P2P systems. Gummadi et al. [8] study the object characteristics of P2P traffic in Kazaa and show that P2P objects are mainly immutable, multimedia, large objects that are downloaded at most once. The study demonstrates that the popularity of P2P objects does not follow Zipf distribution, which is usually used to model the popularity of web objects [2]. The study provides a simulation method for generating P2P traffic that mimics the observed popularity curve, but it does not provide any closed-form models for it. Sen and Wang [31] study the aggregate properties of P2P traffic in a large-scale ISP, and confirm that P2P traffic does not obey Zipf distribution. Their observations also show that a few clients are responsible for most of the traffic. Klemm et al. [16] use two Zipf-like distributions to model query popularity in Gnutella. Because the authors are mainly interested in query popularity, they do not measure object popularity as defined by actual object transfers.

While these measurement studies provide useful insights on P2P systems, they were not explicitly designed to study caching P2P traffic. Therefore, they did not focus on analyzing the impact of P2P traffic characteristics on caching. The study in [8] highlighted the potential of caching and briefly studied the impact of traffic characteristics on caching. But the study was performed in only one network domain.

The importance and feasibility of caching P2P traffic have been shown in [17] and [15]. The study in [17] indicates that P2P traffic is highly repetitive and responds well to caching. The authors of [15] show that current P2P protocols are not ISP-friendly because they impose unnecessary traffic on ISPs. The authors suggest deploying caches or making P2P protocols locality-aware. Both [17] and [15] do not provide any algorithm for caching.

The closest work to ours is [35], where two cache replacement policies for P2P traffic are proposed. These two policies are: MINRS (Minimum Relative Size), which evicts the object with the least cached fraction, and LSB (Least Sent Byte), which evicts the object which has served the least number of bytes from the cache. Our simulation results show that our algorithm outperforms LSB, which is better than MINRS according to the results in [35].

Partial and popularity-based caching schemes for web caching, e.g., [11], and video
streaming, e.g., [12,21] have been proposed before. [11] proposes a popularity-aware greedy-dual size algorithm for caching web traffic. Because the algorithm focuses on web objects, it does not consider partial caching, which is critical for P2P caching due to the large sizes of objects. Jin et al. [12] consider partial caching based on object popularity, encoding bit rate, and available bandwidth between clients and servers. Their objectives are to minimize average start-up delays and to enhance stream quality. In contrast, our partial caching approach is based on the number of bytes served from each object normalized by its cached size. This achieves our objective of maximizing the byte hit rate without paying much attention to latency. A partial caching algorithm for video-on-demand systems is proposed in [21], where the cached fraction of a stream is proportional to the number of bytes played back by all clients from that stream in a time slot. Unlike our algorithm, the algorithm in [21] periodically updates the fractions of all cached streams, which adds significant overhead on the cache.

Finally, our caching algorithm is designed for P2P systems, which contain multiple workloads corresponding to various types of objects. This is in contrast to the previous web and streaming caching algorithms which are typically optimized for only one workload.

1.3 Thesis Contributions

The problem we are dealing with is building an effective caching system to store some objects in a proxy cache deployed at the edge of an AS, such that the number of bytes served from the cache (i.e., byte hit rate) is maximized. The cache sits at the gateway of an autonomous system or a local network, intercepts incoming P2P traffic and caches a subset of the observed traffic as shown in Fig. 1.1. Our objective is to reduce the overall WAN traffic, hence reducing cost incurred by ISPs and congestion in the Internet backbone. To be able to achieve this goal, we conduct a measurement study to understand P2P traffic and then develop and evaluate an effective caching algorithm.

Specifically, our contributions can be summarized as follows [9,27].

- We develop a large-scale passive measurement infrastructure to capture P2P traffic from thousands of ASes across the world over eight months. Our measurement infrastructure infers what objects exist on which peers in which ASes without actively participating in the network or requiring access to gateways in those ASes. Our traces
are one of very few, if any, traces that are being made available to the research community.

- We develop new models for P2P traffic based on our measurement study. We show that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution, which is a generalized form of Zipf-like distributions with an extra parameter. This extra parameter captures the flattened head nature of the popularity distribution observed near the lowest ranked objects in our traces.

- We show the existence of multiple workloads in P2P traffic corresponding to different types of content. We also study the popularity dynamics of P2P objects and show that objects popularity is short-lived and it reaches a peak and dies out in 10-15 weeks.

- We use trace-based simulation and simple analysis to study the impact of the Mandelbrot-Zipf popularity model on caching. We show that relying on object popularity alone may not yield high hit rates/byte hit rates.

- We design and evaluate a new caching algorithm for P2P traffic that is based on segmentation, partial admission and eviction of objects. Our results also show that
our algorithm achieves a byte hit rate that is at least 40% more, and at most 200%, the byte hit rate of the common web caching algorithms. Furthermore, our algorithm is robust against various traffic patterns with different degrees of temporal correlations and against aborted downloads which are common in P2P systems.

1.4 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 describes the Gnutella P2P system, the measurement setup, and the data collection and processing techniques. It presents the new model for object popularity, and analyzes the effect of this model on cache performance. It also studies popularity dynamics and object size. In Chapter 3, we describe our P2P caching algorithm and evaluate its performance using trace-based simulation. We also study the sensitivity of our P2P algorithm to temporal correlations in this chapter. Chapter 4 summarizes the conclusions of this thesis and outlines future directions for this research. Appendix A provides more details about trace data collection, processing and analysis.
Chapter 2

Modeling Peer-to-Peer Traffic

We are interested in deploying caches in different autonomous systems (ASes) to reduce the WAN traffic imposed by P2P systems. Thus, our measurement study focuses on measuring the characteristics of P2P traffic that would be observed by these individual caches, and would impact their performance. Such characteristics include object popularity, object sizes, popularity dynamics and number of P2P clients per AS. We measure such characteristics in several ASes of various sizes. In this chapter, we first describe the Gnutella file sharing system on which we performed our measurement. Then, we describe our measurement methodology for data collection and processing. We end by presenting our traffic models and studying their impact on cache performance.

2.1 Gnutella P2P System

Gnutella [7] is one of many P2P file-sharing systems in existence today. It is regarded as the first fully decentralized P2P file-sharing system. Since it was first released in 2000, Gnutella has gained a lot of popularity and is one of the most widely deployed file-sharing systems [33,36]. During the past year, the population of users on the Gnutella network has tripled, and is approximated at 2 million users today [34].

Gnutella connects participating peers in an overlay network where each peer acts as a client and as a server at the same time. A peer joins the network by finding an active set of peers in the system and establishing TCP connections with them. According to the Gnutella protocol specifications, peers exchange several types of messages including PING, PONG, QUERY and QUERYHIT. A PING message is sent by a joining peer to discover
more neighbors. If a peer receives a PING message, and is willing to establish a connection to the sending peer, it responds with a PONG message. QUERY and QUERYHIT messages are the most pertinent to our study. A QUERY message contains search keywords, a TTL (time to live) field and the address of the immediate neighbor which forwarded the message to the current peer. Query messages are propagated to all neighbors in the overlay for a hop distance specified by the TTL field. A typical value for TTL is seven hops. If a peer has one or more of the requested files, it replies with a QUERYHIT message. A QUERYHIT message is routed on the reverse path of the QUERY message it is responding to, and it contains, among other things, a list of names, URNs (uniform resource name) and sizes of matching files, and the IP address of the responding peer. The list of matched files could contain one or more entries. Upon receiving replies from several peers, the querying peer chooses a set of peers and establishes direct connections with them to retrieve the requested file.

To improve scalability and limit the overhead of query flooding, most implementations of the Gnutella protocol adopt a two-tier overlay network architecture. In this architecture, two kinds of peers exist: ultra-peers, characterized by high bandwidth and long connection periods, and leaf-peers which are ordinary peers that only connect to ultra-peers. When a leaf-peer connects to an ultra-peer, it uploads a list of the URNs of all of its shared files to the ultra-peer. Ultra-peers use such lists to forward queries only to peers that might have matching files. Leaf-peers do not forward query messages.

2.2 Measurement Methodology

We conduct a passive measurement study of the Gnutella file-sharing network [7]. For the purposes of our measurement, we modify a popular Gnutella client called Limewire [18]. We choose to conduct our measurement on Gnutella because (i) it is one of the most widely-deployed P2P systems [36], (ii) it supports the super-peer architecture which facilitates non-intrusive passive measurements by observing traffic passing through super peers, and (iii) it is easier to modify since it is an open source protocol.

Previous studies show that Gnutella is similar to other P2P systems. For example, early studies on the fully-distributed Gnutella and the index-based Napster systems found that clients and objects in both systems exhibit very similar characteristics such as the number of files shared, session duration, availability and host uptime [29]. Another study on
BitTorrent [23] made similar observations regarding object characteristics and host uptime. Also the non-Zipf behavior of object popularity in Gnutella (as we show later) has been observed before in Kazaa [8]. Studies have also shown that Gnutella is one of the top-three most popular P2P systems [36]. Therefore, we believe that the Gnutella traffic collected and analyzed in our study is representative of P2P traffic in general.

2.2.1 Trace Collection and Processing

We run our measurement node in an ultra-peer mode and allow it to maintain up to 500 simultaneous connections to other Gnutella peers. On average, our measurement node was connected to 265 nodes, 72% of which were ultra peers.

We implemented a Listener class and registered callback routines to limewire's MessageRouter class. The MessageRouter class is responsible for routing all Gnutella messages that pass through the node. This enabled us to receive all QUERY and QUERYHIT messages that pass through our measurement node. We strip the headers of messages, append a timestamp of when the message was recorded and write them into trace files. Later on, we read from those files and use the java regular expressions package to process the contents of these raw messages and write them into processed trace files. At this stage we map all found IP addresses to their corresponding AS, ISP, city, and country. We were able to resolve 76.3% of all IP addresses. The remaining IP addresses were either private IP addresses or addresses of peers behind NAT (Network Address Translation) boxes. We set up an Oracle database, and use SQL* Loader to do a bulk load of all those trace files into database tables. We use SQL queries and PL/SQL scripts later on to get data in and out of the database. Appendix (A) describes data processing and management providing examples of SQL scripts we used.

2.2.2 Measurement Statistics

The measurement study was conducted between 16 January 2006 and 16 September 2006. Our measurement peer was located at Simon Fraser University, Canada. But since the Gnutella protocol does not favor nodes based on their geographic locations [16], we were able to observe peers from thousands of ASes across the globe. During the eight months of the measurement, we recorded more than 288 million QUERY messages and 134 million QUERYHIT messages issued from approximately 38 million peers distributed over more
Table 2.1: Summary statistics of the measurement study.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of QUERY messages</td>
<td>287,875,754</td>
</tr>
<tr>
<td>Number of QUERYHIT messages</td>
<td>133,952,125</td>
</tr>
<tr>
<td>Number of unique objects</td>
<td>16,687,320</td>
</tr>
<tr>
<td>Number of IP addresses</td>
<td>38,952,881</td>
</tr>
<tr>
<td>Number of unique ASes</td>
<td>17,369</td>
</tr>
<tr>
<td>Number of ASes with more than 100,000 downloads</td>
<td>127</td>
</tr>
<tr>
<td>Total traffic volume exchanged in all ASes</td>
<td>6,262 terabytes</td>
</tr>
</tbody>
</table>

than 17 thousand different ASes. Table 2.1 summarizes the measurement statistics. The large scale of our measurement study enables us to draw solid conclusions about P2P traffic. The measurement data is stored in several trace files with a total size of approximately 20 giga bytes. The trace files are available to the research community at [20].

2.3 Measuring and Modeling Object Popularity

In this section, we explain how we measure object popularity in different ASes. Then, we present and validate a simple, and fairly accurate, popularity model for objects in P2P systems.

The popularity of an object is defined as the probability of requesting that object relative to other objects. Object popularity is critical for the performance of the cache. Intuitively, storing the most popular objects in the cache is expected to yield higher hit rates than storing any other set of objects.

Since we are primarily interested in the performance of individual caches, we measure the popularity of objects in each AS. To measure the popularity of an object in a specific AS, we count the number of replicas of that object in the AS considered. The number of replicas indicates the number of downloads that were completed in the past. This means that if a cache were deployed, it would have seen a similar number of requests. This assumes that most of the downloads were supplied by peers from outside the AS, which is actually the case because peers in most current P2P networks have no sense of network proximity and thus do not favor local peers over non-local peers. In fact, previous studies [8] have shown that up to 86% of the requested P2P objects were downloaded from peers outside the local network even though they were locally available.
Figure 2.1: Object popularity in P2P traffic in different ASes. The popularity can be modeled by Mandelbrot-Zipf Distribution. Plots are sorted from left to right based on the $q$ values.
Figure 2.2: Continued from Fig 2.1. Object popularity in P2P traffic: in different ASes (a)–(b), and across all ASes (c). The popularity can be modeled by Mandelbrot-Zipf Distribution.
CHAPTER 2. MODELING PEER-TO-PEER TRAFFIC

To count the number of replicas of a given object, we extract from our trace all QUERY-HIT messages which contain the unique ID (URN) of that object. QUERYHIT messages contain the IP addresses of the responding nodes that have copies of the requested object. We can determine the number of replicas by counting the number of unique IP addresses. Then, we map these unique IP addresses to their corresponding ASes by using the GeoIP database.

We compute the popularity of each object in each of the top 18 ASes (in terms of sending and receiving messages). These top ASes contribute around 43% of the total traffic seen by our measurement node. We also compute the popularity across all ASes combined. We rank objects based on their popularity, and we plot popularity versus rank. Fig. 2.1 and Fig. 2.2 show a sample of our results. Similar results were obtained for other ASes. As shown in the figure, there is a flattened head in the popularity curve of P2P objects. This flattened head indicates that objects at the lowest ranks are not as popular as Zipf-like distributions would predict. This flattened head phenomenon could be attributed to two main characteristics of objects in P2P systems: Immutability and large sizes. The immutability of P2P objects eliminates the need for a user to download an object more than once. This download at most once behavior has also been observed in previous studies [8]. The large size of objects, and therefore the long time to download them, may make users download only objects that they are really interested in. This is in contrast to web objects, which take much shorter times to download, and therefore, users may download web objects even if they are of marginal interest to them. These two characteristics reduce the total number of requests for popular objects.

Figures 2.1 and 2.2 also show that, unlike the case for web objects [2], using a Zipf-like distribution to model the popularity of P2P objects would result in a significant error. In log-log scale, the Zipf-like distribution appears as a straight line, which can reasonably fit most of the popularity distribution except the left-most part, i.e., the flattened head. A Zipf-like distribution would greatly overestimate the popularity of objects at the lowest ranks. These objects are the most important to caching mechanisms, because they are the best candidates to be stored in the cache.

We propose a new model that captures the flattened head of the popularity distribution of objects in P2P systems. Our model uses the Mandelbrot-Zipf distribution [32], which is the general form of Zipf-like distributions. The Mandelbrot-Zipf distribution defines the
probability of accessing an object at rank $i$ out of $N$ available objects as:

$$p(i) = \frac{K}{(i + q)^\alpha},$$  

(2.1)

where $K = \sum_{i=1}^{N} 1/(i + q)^\alpha$, $\alpha$ is the skewness factor, and $q \geq 0$ is a parameter which we call the plateau factor. $\alpha$ controls the skewness (slope) of the curve. $q$ is called the plateau factor because it is the reason behind the plateau shape near the left-most part of the distribution. Notice that the higher the value of $q$, the more flattened the head of the distribution will be. When $q = 0$, Mandelbrot-Zipf distribution degenerates to a Zipf-like distribution with a skewness factor $\alpha$. Fig. 2.3 compares Zipf distribution versus Mandelbrot-Zipf distribution for different $q$ values. Notice that, there is about an order of magnitude difference in frequency between the two distributions at the lowest ranks.

To validate this popularity model, we fit the popularity distributions of objects in each of the top 18 ASes to Mandelbrot-Zipf distribution using the Matlab distribution fitting tool. Our results, some of them are shown in Figures 2.1 and 2.2, indicate that Mandelbrot-Zipf distribution models the popularity of P2P objects reasonably well. Table 2.2 summarizes $\alpha$ and $q$ values for eighteen representative ASes.
The significance of the plateau factor $q$ is that it controls the left-most part of the distribution; i.e., the fraction of total requests received by objects at the lowest ranks. As such, $q$ serves as an important indication of the feasibility of caching and the achievable byte hit rate; the larger the value of $q$ the lesser the benefit of caching. To understand what parameters control the value of $q$ in an autonomous system, we perform two experiments using values from the top 20 ASes. In the first experiment, we study the impact of the number of hosts (peers) versus the plateau factor $q$. The number of hosts in an AS is computed by counting the number of unique IPs in our traces that map to this AS. We use the GeoIP database [6] to perform the mapping. The plateau factor $q$ is computed from fitting the object popularity curve to a Mandel-brot Zipf distribution. Fig. 2.4(a) shows that the larger the number of hosts in an AS, the smaller the value of $q$. This is because when the number of hosts is large, objects at low ranks have higher frequency since there are a lot of potential requesters. This high frequency results in a less flattened popularity curve near low ranks, hence a smaller value for $q$. On the other hand, ASes with a small number of hosts tend to have larger values of $q$. This is primarily due to the download-at-most-once behavior. The frequency of objects at the lowest ranks is bounded above by the number of hosts that could potentially download them.

The small number of hosts per AS alone is not a definite indication of a large value of $q$. In ASes with small number of hosts, the value of $q$ is also affected by the average number of downloads per host. This can be seen in Table 2.2 where AS18538 has 592 hosts, and a $q$ value of 121, whereas AS1161 has 2,046 hosts with a $q$ value of 500. This is because the total number of downloads in AS 1161 is six times the number of downloads in AS18538. As the average number of downloads increases, the frequency of downloading the most popular objects slowly approaches the upper bound, while the frequency of less popular objects increases. This is because peers are forced to choose from the pool of less popular objects once they consume the most popular objects. This results in smaller differences between frequency of popular objects and less popular objects, resulting in a more flattened popularity curve, and a larger value for $q$. The main reason behind this behavior, again, is the download-at-most-once behavior. To validate this argument we plot in Fig. 2.4(b) the average number of downloads per host versus the plateau factor $q$. The figure shows that the value of $q$ increases as the average number of downloads per host increases.
### Table 2.2: Object popularity in several AS domains.

<table>
<thead>
<tr>
<th>AS No.</th>
<th>unique IPs</th>
<th>unique objects</th>
<th>MZipf($\alpha$, $q$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14832</td>
<td>652</td>
<td>64,183</td>
<td>(0.52, 8)</td>
</tr>
<tr>
<td>18173</td>
<td>71</td>
<td>766,841</td>
<td>(1.62, 511)</td>
</tr>
<tr>
<td>18538</td>
<td>592</td>
<td>40,756</td>
<td>(0.60, 121)</td>
</tr>
<tr>
<td>2161</td>
<td>1,448,323</td>
<td>1,407,310</td>
<td>(0.66, 5)</td>
</tr>
<tr>
<td>223</td>
<td>565,056</td>
<td>735,329</td>
<td>(0.68, 3)</td>
</tr>
<tr>
<td>105</td>
<td>660,260</td>
<td>754,398</td>
<td>(0.78, 5)</td>
</tr>
<tr>
<td>95</td>
<td>104,840</td>
<td>126,568</td>
<td>(1.2, 5)</td>
</tr>
<tr>
<td>1859</td>
<td>723,974</td>
<td>1,235,716</td>
<td>(0.72, 5)</td>
</tr>
<tr>
<td>11715</td>
<td>849,069</td>
<td>1,079,858</td>
<td>(0.68, 0)</td>
</tr>
<tr>
<td>1403</td>
<td>1,125,465</td>
<td>1,052,114</td>
<td>(0.70, 0)</td>
</tr>
<tr>
<td>2120</td>
<td>746,429</td>
<td>830,866</td>
<td>(0.78, 6)</td>
</tr>
<tr>
<td>1161</td>
<td>2,046</td>
<td>251,800</td>
<td>(0.75, 500)</td>
</tr>
<tr>
<td>397</td>
<td>636,836</td>
<td>768,445</td>
<td>(0.74, 2)</td>
</tr>
<tr>
<td>11803</td>
<td>539,818</td>
<td>726,778</td>
<td>(0.70, 0)</td>
</tr>
<tr>
<td>1711</td>
<td>486,552</td>
<td>680,428</td>
<td>(0.56, 4)</td>
</tr>
<tr>
<td>2609</td>
<td>16,084</td>
<td>233,187</td>
<td>(0.74, 3)</td>
</tr>
<tr>
<td>1782</td>
<td>773,921</td>
<td>885,218</td>
<td>(0.75, 4)</td>
</tr>
<tr>
<td>9406</td>
<td>527,536</td>
<td>711,192</td>
<td>(0.70, 20)</td>
</tr>
<tr>
<td>9548</td>
<td>814,555</td>
<td>991,747</td>
<td>(0.68, 3)</td>
</tr>
</tbody>
</table>

### 2.4 The Effect of Mandelbrot-Zipf Popularity on Caching

In this section, we analyze the impact of the Mandelbrot-Zipf popularity model on the cache hit rate and byte hit rate using simple analysis and simulation. Hit rate is defined as the fraction of requests that could be served from the cache. Byte hit rate is the fraction of requested bytes that were served from the cache.

We start with a simple analysis of an LFU (Least Frequently Used) policy. Under LFU, the $C$ most popular objects are stored in the cache. For simplicity, we assume that all objects have the same size and the skewness parameter $\alpha$ is 1. We are mainly interested in exploring the impact of the plateau factor $q$ on the hit rate. The hit rate $H$ of an LFU cache is the cumulative probability of requesting one of the $C$ objects. The probability of
accessing an object at rank \( i \) is given by Eq. 2.1. Hence, \( H \) is given by:

\[
H = \sum_{i=1}^{C} p(i) = \sum_{i=1}^{C} \frac{K}{(i+q)} \\
\approx \int_{i=1}^{C} \frac{K}{(i+q)} \, d_i = K \ln \left( \frac{1 + C/q}{1 + 1/q} \right). \tag{2.2}
\]

Eq. (2.2) implies that increasing \( q \) results in a decrease in hit rate. When \( q \rightarrow \infty \), i.e., the head is very flat, the hit rate approaches zero. In contrast, for a Zipf-like popularity distribution \( (q = 0) \), the hit rate is \( H = K \ln C \). To further illustrate the impact of Mandelbrot-Zipf on the cache performance, we plot in Fig. 2.5(a) the relative loss in hit rate between Zipf and Mandelbrot-Zipf distributions. The relative loss in hit rate is computed as \( \left( H^{Zipf} - H^{MZipf} \right) / H^{Zipf} \), where \( H^{Zipf} \) and \( H^{MZipf} \) are the hit rates achieved by an LFU cache if the popularity follows Zipf and Mandelbrot Zipf distributions, respectively. As the figure shows, significant loss in hit rate could be incurred because of the flattened-head nature of the Mandelbrot-Zipf popularity distribution. The loss in hit rate is higher for smaller relative cache sizes and larger values of \( q \). A relative cache size is the percentage of the total cache size that can store all objects requested.

Next, we consider an LRU (Least Recently Used) cache. We use simulation to study the impact of the popularity model on the hit rate. We generate synthetic traces as follows.
CHAPTER 2. MODELING PEER-TO-PEER TRAFFIC

Figure 2.5: Effect of Mandelbrot-Zipf popularity distribution on the cache performance using: (a) LFU (analytic), (b) LRU (simulation).

Figure 2.6: Effect of Mandelbrot-Zipf popularity distribution on the byte hit rate using optimal algorithm (trace-based).
We consider 4,000 equal-sized objects and randomly generate requests for these objects according to the Zipf and Mandelbrot-Zipf distributions. We run the traces through an LRU cache with a relative cache size that varies between 0 and 100%. We compute the hit rate in each case. The results are shown in Fig. 2.5(b). As the figure indicates, the situation is even worse under LRU: higher drops in hit rates are observed using Mandelbrot-Zipf distribution, especially under small cache sizes.

Finally we use traces from the first three months of our measurement study and compute the maximum achievable byte hit rate in two different ASes. We pick two ASes from our traces with similar $\alpha$ values but different $q$ values: AS397 with $q = 8$ and AS14832 with $q = 55$ (as observed in the first three months). We use an optimal off-line algorithm which looks at the trace of each AS and stores in the cache the objects which will serve the most number of bytes, hence, achieves the highest byte hit rate. We perform trace-based simulation and compute the byte hit rate under various cache sizes for both ASes. As can be seen from Fig. 2.6, with a cache size of 400 GB, a byte hit rate of 24% is achieved under AS397, while only 9% byte hit rate is achieved under AS14832 using the same cache size. This means that the top popular objects which can fit in a cache size of 400 GB receive 24% of the total outgoing requests in AS397, in comparison to 9% of the total outgoing requests in AS14832.

These observations and experiments imply that caching schemes that capitalize on object popularity alone may not yield high hit rates/byte hit rates and may not be very effective in reducing the ever-growing P2P traffic.

2.4.1 Popularity Dynamics

In designing a dynamic caching algorithm, it is important to understand the time-scale at which the popularity of objects changes, so that we can maintain in the cache the most popular objects. This is particularly important in P2P systems since the total amount of traffic far exceeds the cache size.

To measure the turnover of popularity in P2P traffic, we perform the following experiment. We choose the top 100 most popular objects during the third month of our measurement as seen in the top 1st AS, top 2nd AS and all ASes. We trace the popularity of those objects by counting the number of requests they received per week for the entire eight months of our measurement study. Fig. 2.7 shows that popular objects gain popularity in a relatively short time-scale reaching their peak in about 5-10 weeks in the top 1st and
CHAPTER 2. MODELING PEER-TO-PEER TRAFFIC

Figure 2.7: Popularity dynamics in P2P systems. The figure shows popularity of the 100 most requested objects in (a) the top 1\textsuperscript{st} AS, (b) top 2\textsuperscript{nd} AS.

Figure 2.8: Popularity dynamics in P2P systems. The figure shows popularity of the 100 most requested objects in all ASes.
2nd AS. The popularity of those objects drops dramatically after that. The same behavior is observed for all ASes (Fig. 2.8). As the figures show, we observe as much as a sixfold decrease in popularity in a matter of 5-10 weeks. This means that if we store objects in the cache based on their frequency of requests, they will be retained in the cache long after they lose their popularity. Thus, frequency-based caching policies, e.g., LFU, may not be very effective in caching P2P traffic. Also notice that the popularity of objects is not very short-lived. As shown in the figure, a popular object enjoys about 3 months of popularity before its request frequency dies out. This indicates that incremental partial caching of objects is beneficial since there is enough time for the cache to build popularity profiles, identifying which objects are truly popular and incrementally cache them until they are completely available in the cache. If the time scale were small, i.e., in the order of days, full caching of objects would be a better approach. We discuss incremental partial caching versus full caching further in Sections 3.1 and 3.2.5

2.4.2 Measuring and Modeling Object Size

Many web caching algorithms use object size in their decision to evict objects. Other algorithms, such as Greedy-Dual Size [4], incorporate size with other characteristics to maximize cache performance. With the growing amount of P2P traffic and the large size
of exchanged objects, cache size is a very important resource, and any effective policy must strive to achieve the objective of placing in the limited-size cache the best objects that will maximize byte hit rate.

To understand object size distributions, we collect information about objects in two popular P2P file-sharing systems: BitTorrent and Gnutella. In BitTorrent, there are a number of web servers that maintain meta data (torrent) files about objects in the network. These servers are known as torrent sites. We developed a script to contact four popular torrent sites and download random torrent files. We downloaded 100 thousand .torrent files, 49.3% of which are unique. Each .torrent file contains information about the shared object including object size, number of segments in the object and the size of each segment. For the Gnutella system, we extract from our traces the sizes of all unique objects seen during our measurement.

As shown in Fig. 2.9, the histogram of the size distribution in both systems exhibits several peaks, where each peak corresponds to a different workload. The term workload is used here to describe objects with similar content and size properties. For example, a peak around 700 MB corresponds to most shared CD movies; another peak around 300 MB corresponds to high quality TV video series and some software objects while a peak around few mega bytes corresponds to video and audio clips. The location of those peaks are almost identical in both Gnutella and BitTorrent as the figure shows. However, BitTorrent has two more peaks: one peak around 1.5 GB corresponding to high quality avi movies, and another smaller peak at 4.5 GB (not shown in the figure) corresponding to DVD movies. We have checked the validity of the peak-to-content-type mapping by randomly sampling many files from the content and checking the type of the sampled files. The similarity of workloads in Gnutella and BitTorrent leads us to believe that similar distributions exist for shared content on other file-sharing P2P systems.

The existence of multiple workloads has several consequences on the cache design. For example, audio files tend to be requested more often than large video files. Thus using an LFU policy would be biased against large objects. On the other hand, using object size as a replacement criterion, i.e., evicting objects with the smallest size, would be biased against smaller objects. Worse yet, this might result in a scenario where we evict tens of popular mp3 objects to make space for a not-so-popular large video object that would be requested only once. Therefore, any P2P caching algorithm will have to consider the intrinsic properties of each workload and the fact that P2P object size extends from few kilobytes to
Understanding object size and the multiple workloads that exist guided the design of our P2P caching algorithm. We show in Chapter 3 how our algorithm leverages the existence of multiple workloads to divide objects of each workload into equal-sized segments to improve cache performance.

2.4.3 Summary of the Measurement Study

We designed our measurement study to analyze the P2P traffic that would be observed by individual P2P caches deployed in different ASes. We found that popular P2P objects are not as highly popular as their web counterparts. The popularity distribution of P2P objects has a flattened head at the lowest ranks, and therefore, modeling this popularity as a Zipf-like distribution yields a significant error. We also found that a generalized form of the Zipf distribution, called Mandelbrot-Zipf, captures this flattened head nature and therefore is a better model for popularity of P2P objects. Furthermore, we found that the Mandelbrot-Zipf popularity has a negative impact on hit rates and byte hit rates of caches that use the common LRU and LFU policies. In addition, our study revealed the dynamic nature of object popularity in P2P systems, which should be considered by caching algorithms. Finally, we found that objects in P2P systems have much larger sizes than web objects, and they can be classified into several categories based on content types.
Chapter 3

P2P Caching Algorithm and Evaluation

In the previous chapter, we analyzed P2P traffic characteristics that are relevant to caching, which guided the design of our algorithm. In this chapter, we first describe our P2P caching algorithm. Then, we use trace-based simulation to thoroughly evaluate our algorithm under several traffic patterns.

3.1 P2P Algorithm

With the understanding of the P2P traffic we developed, we design and evaluate a novel P2P caching scheme based on segmentation and partial caching. Partial caching is important because objects in P2P systems are large. Furthermore, P2P traffic is composed of multiple workloads, each workload corresponds to a different content type (e.g., audio, video, documents), and occupies a subrange of the total object size range. Moreover, because of the Mandelbrot-Zipf popularity model, the most popular objects may not receive too many requests. That is, the portion of the overall traffic that is directed towards those popular objects is small compared to the amount of total traffic. Thus storing an entire object upon a request may waste cache space. P2P caching should take a conservative approach toward admitting new objects into the cache to reduce the cost of storing unpopular objects. To achieve this objective, our algorithm divides objects into small segments and incrementally admits more segments of an object to the cache as the object receives more requests.
We use different segment sizes for different workloads, because using the same segment size for all workloads may favor some objects over others and introduce additional cache management overhead. For example, if we use a small segment size for all workloads, large objects will have large number of segments. This introduces a high overhead of managing the large number of segments within each object. Such overhead includes locating which segments of an object are cached, and deciding which segments to evict. On the other hand, using a large segment size for all workloads will favor smaller objects since they will have smaller number of segments and get cached quicker. Making segment size relative to object size has the same two problems. This is because the range of P2P object sizes extends to several gigabytes. Using relative segmentation, as a percentage of an object size, may result in large objects having segments that are orders of magnitude larger than an unsegmented smaller object.

Consistent with our observations of multiple workloads, we use four different segment sizes. For objects with sizes less than 10 MB, we use a segment size of 512 KB, for objects between 10 MB and 100 MB, a segment size of 1 MB, and for objects between 100 MB and 800 MB a segment size of 2 MB. Objects whose size exceeds 800 MB are divided into segments of 10 MB in size. We call this segmentation scheme variable segmentation scheme. Note that our segmentation procedure is in line with most segmentation schemes used by real P2P systems. Our analysis of more than 50,000 unique torrent files shows that the vast majority of BitTorrent objects are segmented into 256 KB, 512 KB, 1 MB and 2 MB segments. E-Donekey uses segments of size 9.28 MB [5]. Depending on the client implementation, segment size in Gnutella can either be a percentage of object size, as in BearShare [1], or fixed at few hundred KBs [7]. We opt not to follow a protocol-specific segmentation scheme so as to make our algorithm independent of the underlying P2P protocol. In addition, a protocol-specific segmentation scheme might not always be in line with the objective of maximizing byte hit rate. This is confirmed by our evaluation in Section 3.2.

The basic idea of our algorithm is to cache a portion of each object that is proportional to its popularity. That way popular objects will be incrementally given more cache space than unpopular objects. To achieve this goal, our algorithm caches a constant portion of an object when it first sees that object. As the object receives more hits, its cached size increases. The rate at which the cached size increases grows with the number of hits an object receives. The unit of admission into the cache is one segment; that is the smallest
granularity of our algorithm for caching/evicting is one segment.

For each workload, the cache keeps the average object size in that workload. The average is computed from our traces, but could also be computed online in a real cache. Denote the average object size in workload $w$ as $\mu_w$. Further let $\gamma_i$ be the number of bytes served from object $i$ normalized by its cached size. That is, $\gamma_i$ can be considered as the number of times each byte in this object has been served from the cache. The cache ranks objects according to their $\gamma$ value, such that for objects ranked from 1 to $n$, $\gamma_1 \geq \gamma_2 \geq \gamma_3 \geq \cdots \geq \gamma_n$. We refer to an object at rank $i$ simply as object $i$. When an object is seen for the first time, only one segment of it is stored in the cache. If a request arrives for an object of which at least one segment is cached, the cache computes the number of segments to be added to this object’s segments as $(\gamma_i/\gamma_1)\mu_w$, where $\mu_w$ is the mean of the object size in workload $w$ to which object $i$ belongs. Notice that this is only the number of segments the cache could store of object $i$. But since downloads can be aborted at any point during the session and that we store at least one segment of an object, the number of segments actually cached upon a request, denoted by $k$, is given by

$$k = \min(\text{missed}, \max(1, \frac{\gamma_i}{\gamma_1}\mu_w)),$$

where $\text{missed}$ is the number of requested segments not in the cache. This means that the cache will stop storing uncached segments if the client fails or aborts the download, and that the cache stores at least one segment of an object.

The pseudo-code of our P2P caching algorithm appears in Fig. 3.1. At a high level, the algorithm works as follows. The cache sits at the gateway of an autonomous system or a network. It intercepts client requests and extracts the object ID and the requested range. If no segments of the requested range are in the cache, the cache stores at least one segment of the requested range. If the entire requested range is cached, the cache will serve it to the client. If the requested range is partially available in the cache, the cache serves the cached segments to the client, lets the client download the missing segments from the target server, and decides how many of those missing segments to cache using Eq. (3.1). In all cases, the cache updates the average object size of the workload to which the object belongs and the $\gamma$ values of the requested object.

The algorithm uses a priority queue data structure to store objects according to their $\gamma$ values. When performing eviction, segments are deleted from the least valued objects. Currently, our algorithm evicts contiguous segments without favoring any segments over
CHAPTER 3. P2P CACHING ALGORITHM AND EVALUATION

Proportional Partial Caching Algorithm

/* upon a request for object i */
1. if object i is not in the cache
   2. add one segment of i to cache, evicting if necessary
   3. else
   4. hit = cached range \cap requested range
   5. $\gamma_i = \gamma_i + \text{hit} / \text{cached size of } i$
   6. missed = (requested range - hit)/segment size
   7. $k = \min\{\text{missed}, \max(1, \frac{\gamma_i}{\gamma_l} \mu_k)\}$
   8. if cache does not have space for k segments
      9. evict k segments from the least valued object(s)
   10. add k segments of object i to the cache
   11. return

Figure 3.1: Proportional Partial Caching Algorithm for P2P traffic.

others. This is because P2P downloads are likely to start from anywhere in the file [35].
Our algorithm needs to perform $O(\log N)$ comparisons with every hit or miss, where $N$ is
the number of objects in the cache. But since objects are large, this is a reasonable cost
considering the small number of objects the cache will contain.

3.2 Evaluation

In this section, we use trace-driven simulation to study the performance of our P2P caching
algorithm under several scenarios, and compare it against three common web caching algo-
rithms (LRU, LFU and GDS) and a recent caching algorithm proposed for P2P systems.

3.2.1 Experimental Setup

Traces and performance metrics. We use traces obtained from the first three months of our
measurement study to conduct our experiments. Our objective is to study the effectiveness
of deploying caches in several ASes. Thus, we collect information about objects found in
a certain AS and measure the byte hit rate that could be achieved if a cache were to be
deployed in that AS. We use the byte hit rate as the performance metric because we are
mainly interested in reducing the WAN traffic. In all experiments, we use the ASes which
have the most amount of traffic seen by our measurement node. This ensures that we
have enough traffic from an AS to evaluate the effectiveness of deploying a cache for it. In addition, we study the impact of the plateau factor $q$ and the skewness parameter $\alpha$ of the Mandelbrot Zipf distribution, as well as the segment size on the performance of our P2P caching algorithm.

Algorithms implemented. We run several AS traces through the cache and compare the byte hit rate achieved using several caching algorithms. In addition to our algorithm, we implement the Least Recently Used (LRU), Least Frequently Used (LFU), Greedy-Dual Size (GDS) \cite{4} and Least Sent Bytes (LSB) \cite{35} algorithms. We also implement the off-line optimal (OPT) algorithm and use it as a benchmark for comparison. LRU capitalizes on the temporal correlation in requests, and thus replaces the oldest object in the cache. LFU sorts objects based on their access frequency and evicts the object with the least frequency. GDS sorts objects based on a cost function and recency of requests, and evicts the one with the least value. We use object size as the cost function in GDS to maximize byte hit rate as indicated by \cite{4}. LSB is designed for P2P traffic caching and it uses the number of transmitted bytes of an object as a sorting key. The object which has transmitted the least amount of bytes will be evicted next. OPT looks at the entire stream of requests off-line and caches the objects that will serve the most number of bytes from the cache.

Evaluation scenarios. We evaluate the algorithms under several scenarios. First, we consider the case when objects requested by peers are downloaded entirely, that is, there are no aborted transactions. Then, we consider the case when the downloading peers prematurely terminate the downloads during the session, which is not uncommon in P2P systems.

We also analyze the effect of temporal locality on caching algorithms. Recall that temporal locality is defined as the likelihood of requesting an object in the near future \cite{11,24}. Temporal locality has two components: popularity and temporal correlations \cite{24}. Popularity of an object is the number of requests it receives relative to other objects. Temporal correlation means that requests to the same object tend to be clustered or correlated together in time. Temporal correlation is usually measured in the number of intervening requests between two consecutive requests to the same object. A shorter sequence of intervening requests imply higher temporal correlations. We study the effect of each component of temporal locality in isolation from the other. We also study the combined effect of the two components.

To isolate the popularity component, we randomly shuffle our traces to eliminate the
CHAPTER 3. P2P CACHING ALGORITHM AND EVALUATION

3.2 Caching Algorithm and Evaluation

3.2.1 Cache Size

Figure 3.2 shows the byte hit rate for two representative ASes with different characteristics. These two ASes have different maximum achievable byte hit rates, which is defined as the fraction of traffic downloaded more than once, i.e., cacheable traffic, over the total amount of traffic. As shown in the figure, our policy outperforms other policies by as much as 200%. For instance, in AS397 (Fig. 3.2(a)) with a cache of 600 GB, our policy achieves a byte hit rate of 24%, which is almost double the rate achieved by LRU, LFU, GDS, and LSB policies. Moreover, the byte hit rate achieved by our algorithm is about 3% less than...
CHAPTER 3. P2P CACHING ALGORITHM AND EVALUATION

30

Figure 3.3: Relative byte hit rate improvement of P2P algorithm compared to traditional caching algorithms in the top ten ASes. No aborted downloads.

that of the offline optimal algorithm. Our trace indicate that the amount of traffic seen in AS397 is around 24.9 tera bytes. This means that a reasonable cache of size 600 GB would have served about 6 tera bytes locally using our algorithm; a significant saving in the WAN bandwidth.

We believe that traditional policies perform poorly for P2P traffic due to the effect of unpopular objects. For example, one-timer objects are stored entirely under traditional policies on a miss. Under our policy, however, only one segment of each one-timer will find its way into the cache, thus minimizing their effect. The same could be said about 2nd-timers, 3rd timers and so on. Thus, our algorithm strives to discover the best objects to store in the cache by incrementally admitting them. Similar results were obtained for the other top ten ASes. Our policy consistently preforms better than traditional policies.

Fig. 3.3 summarizes the relative improvement in byte hit rate that our policy achieves over LRU, LFU, GDS, and LSB for the top ten ASes, with a cache of size 500 GB. The relative improvement is computed as the difference between the byte hit rate achieved by our policy and the byte hit rate achieved by another policy normalized by the byte hit rate of the other policy. For instance the improvement over LRU would be \( \frac{P2P - LRU}{LRU} \). Notice that the lines connecting dots are used for presentation purposes, and do not imply continuity.
of the curve. The figure shows a relative improvement of at least 40% and up to 180% can be achieved by using our algorithm. That is a significant gain given the large volume of the P2P traffic. We notice that the relative gain our policy achieves is larger in ASes with a substantial fraction of one-timers.

We also observe that the achievable byte hit rate is between 15% and 40% with reasonable cache sizes. This is similar to the achievable byte hit rate for web caching, which is practically in the range 20%—35% (CISCO technical paper [10]), or as other sources indicate 30%—60% [25]. But due to the large size of P2P objects, a small byte hit rate amounts to savings of tera bytes of data.

As a final comment on Fig. 3.2(a) and Fig. 3.2(b), consider the byte hit rates achieved under our algorithm and the optimal algorithm. Notice that although the percentage of cacheable traffic in AS397 is less than that of AS95, the byte hit rate is higher in AS397. This is because popular objects in AS95 do not get as many requests as their counterparts in AS397. That is, the popularity distribution of AS95 has a more flattened head than that of AS397. We computed the skewness factor $\alpha$ and the plateau factor $q$ of the Mandelbrot-Zipf distribution that best fits the popularity distributions of these two ASes. We found that AS95 has $\alpha = 0.6$ and $q = 50$, while AS397 has $\alpha = 0.55$ and $q = 25$. Smaller $q$ values mean less flattened heads, and yield higher byte hit rates. Section 3.2.6 elaborates more on the impact of $\alpha$ and $q$ on the byte hit rate.

### 3.2.3 Caching with Aborted Downloads

Due to the nature of P2P systems, peers could fail during a download, or abort a download. We run experiments to study the effect of aborted downloads on caching, and how robust our caching algorithm is in face of download failures. Following observations from [8], we allow 66% of downloads to abort anywhere in the session. To achieve this, we use the same traces used for the previous experiments with the same number of objects plus two aborted download sessions for each object in the trace. While our policy is designed to deal with aborted downloads, web replacement policies usually download the entire object upon a miss, and at times perform pre-fetching of objects. This is reasonable in the web since web objects are usually small, which means they take less cache space. But in P2P systems, objects are larger, and partial downloads constitute a large portion of the total amount of traffic. Fig. 3.4 compares the byte hit rate with aborted downloads using several algorithms. Compared to the scenario of caching under full downloads (Section 3.2.2), the performance
of our algorithm improves slightly while the performance of other algorithms declines. The improvement in our policy could be explained by the fact that fewer bytes are missed in case of a failure.

The performance of LRU, LFU, GDS, and LSB declines because they store an object upon a miss regardless of how much of it the client actually downloads. Hence, under aborted download scenarios, the byte hit rates for traditional policies suffers even more than it does under full download scenarios. Similar results were obtained for the top ten ASes with a cache of size 500 GB. Our policy consistently outperforms LRU, LFU, LSB and GDS with a wide margin in all ten ASes. Fig. 3.5 shows that the relative improvement in byte hit rate is at least 50% and up to 200%.

3.2.4 Sensitivity of the P2P Caching Algorithm to Temporal Locality

As we discuss in Section 3.2.1, temporal locality is caused by popularity of objects and temporal correlations of requests. In the previous experiments we isolated the effect of popularity from the effect of temporal correlations by shuffling the traces. In this section, we first study the combined effect of the two components of temporal locality. Then we isolate temporal correlation and study its effect on caching while fixing popularity.

For the combined effect of popularity and temporal correlations, we use the original unshuffled traces, with preserved popularity and temporal correlations. Figs. 3.6(a) and
CHAPTER 3. P2P CACHING ALGORITHM AND EVALUATION

(b) show the byte hit rate in two ASes: AS95 and AS397, respectively. LRU and GDS perform slightly better than LSB and LFU because they make use of temporal correlation between requests. However, the achieved byte hit rate under the four algorithms is still low compared to the byte hit rate achieved by our algorithm. Note that the byte hit rate under our algorithm is slightly smaller than in the previous experiments, where we only used popularity. This is because our algorithm does not capitalize on temporal correlation. However, this reduction is small, less than 3%. The fact that the performance of our algorithm does not suffer much under temporal correlation and still outperforms other algorithms (e.g., LRU and GDS) could be explained as follows. We believe that object size is the dominant factor in caching for P2P systems, because the maximum cache size we used (1000 GB) is still small compared to the total size of objects, less than 5%—10% in most cases. As a consequence, object admission strategy is a key element in determining the byte hit rate, which our algorithm capitalizes on. We obtained similar results for other ASes.

Now, we fix popularity and study the effect of temporal correlations. Since we cannot control the degree of correlation in our traces, we generate synthetic traces with various degrees of temporal correlations. This is done by using the LRU Stack Model [3], which
generates correlated requests by using a stack of depth $n$ to keep an ordered list of the last requested $n$ objects such that the subset of objects in the stack have a higher probability of being accessed again than they would if they were not in the stack. The stack depth reflects the degree of temporal correlations in the generated traces: higher depths indicate stronger temporal correlations.

The authors of [3] provide an open-source trace generation tool called ProWGen. We modify ProWGen to use Mandelbrot-Zipf distribution and provide object sizes from our traces. We fix the popularity by using $q = 20$ and $\alpha = 0.6$. To study the sensitivity of our algorithm to temporal correlations, we vary the depth of the LRU stack between 0 and 500. As Fig. 3.7 shows our P2P caching algorithm is not sensitive to the degree of temporal correlations, and the achieved byte hit rate stays fairly constant across a wide range of temporal correlations. This shows the robustness of our algorithm in the face of different traffic patterns with various degrees of temporal correlations.

### 3.2.5 Partial Caching versus Full Caching

As we show in previous experiments, the performance of traditional policies suffers when they are used to cache P2P traffic. This is mainly because they cache entire objects, thus running the risk of storing in the cache large unpopular objects. Our algorithm takes a conservative approach by incrementally caching objects in proportion to their request frequency.
To study how much partial caching contributes to the performance of our P2P caching algorithm, we perform the following experiment. We run the traces of the top 10 ASes through a 500 GB cache that runs a modified version of our algorithm without the partial caching capability. That is, we use a utility function based on the number of bytes served from an object normalized by its cached size, and evict the object with the least utility value. Fig. 3.8(a) shows the results for a sample AS where the performance of our algorithm clearly suffers when partial caching is not used. In Fig. 3.8(b), we show the byte hit rate achieved using our algorithm with and without partial caching capability for the top ten ASes using a cache of size 500 GB. As the figure shows, in some ASes, partial caching contributes as much as 60% of the byte hit rate achieved by our algorithm.

To further investigate the importance of partial caching for P2P traffic, we compare our algorithm versus an offline algorithm that looks at the entire trace and fills the cache with the most popular objects. We call this algorithm the Most K-Popular (MKP) algorithm. MKP is optimal in terms of object popularity, i.e., it achieves the optimal hit rate, not the optimal byte hit rate. We mainly use it because it stores the most popular objects entirely.
CHAPTER 3. P2P CACHING ALGORITHM AND EVALUATION

3.2.6 Effect of $\alpha$ and $q$ on P2P Caching

As we mention in Chapter 2, P2P traffic can be modeled by a Mandelbrot-Zipf distribution with two parameters: a skewness parameter $\alpha$ and a plateau factor $q$. In this section, we study the effect of $\alpha$ and $q$ on the byte hit rate of our algorithm via simulation. We did not use our traces because they may not capture the performance of our algorithm for all
Figure 3.9: Byte hit rates achieved by our online heuristic partial caching algorithm (P2P) and by the MKP offline optimal in top ten ASes.

Figure 3.10: The impact of $\alpha$ on caching.
possible values of $\alpha$ and $q$. We randomly pick 100,000 objects from our traces and generate their frequencies using Mandelbrot-Zipf with various values for $\alpha$ and $q$. We fix the cache size at 1,000 GB and we assume a no-failure model where peers download objects entirely.

To study the effect of different $\alpha$ values, we fix $q$ and change $\alpha$ between 0.4 and 1. As shown in Fig. 3.10, the byte hit rate increases as the skewness factor $\alpha$ increases. This is intuitive since higher values of $\alpha$ mean that objects at the lowest ranks are more popular and caching them yields higher byte hit rate. Thus ASes whose popularity distribution is characterized with high $\alpha$ values would benefit more from caching than those with low $\alpha$ values.

Another parameter that determines the achievable byte hit rate is the plateau factor $q$ of the popularity distribution. $q$ reflects the flattened head we observed in Chapter 2. Fig. 3.11 shows that the byte hit rate decreases as $q$ increases. The reason for this is that a higher value of $q$ means popular objects are requested less often. This has a negative impact on a cache that stores those popular objects, because they receive less downloads resulting in a decreased byte hit rate. Notice, though, that we are only exploring how changing the Mandelbrot-Zipf parameters impacts the caching performance and not suggesting using
specific values for $q$ and $\alpha$. This means that there exists some ASes for which Mandelbrot-Zipf accurately captures the popularity model by a small value of $\alpha$ and a large value of $q$ resulting in popularity being spread out among objects more evenly, i.e., approaching a uniform distribution. But in such model all caching algorithms will suffer since the portion of requests received by popular objects decreases drastically.

### 3.2.7 Effect of Segmentation on P2P Caching

Our algorithm uses a variable segmentation scheme whereby objects belonging to the same workload have the same segment size. As we discuss in Section 3.1, variable segmentation has several advantages over fixed segmentation. Using different segment size for different workloads helps reduce the overhead of managing too many segments if large objects were to have the same segment size as small objects. It also helps increase byte hit rate because larger segments for larger objects means that they will be available in the cache after relatively few requests. For example, if we use 1 MB segmentation, then an object of 4 GB will have 4000 segments to manage. On the other hand, using 10 MB segment size would reduce the number of segments to 400 and still give a higher byte hit rate.

To measure the impact of segmentation on byte hit rate, we repeat the experiment of
aborted downloads under AS397. Using different segment sizes, we first evaluate the byte hit rate when all objects have the same segment size (1 MB, 50 MB, and 100 MB) regardless of their size. Then we evaluate our variable segmentation scheme which we described in Section 3.1. As can be seen from Fig. 3.12, as the segment size increases, the byte hit rate degrades. This is because the cache evicts and admits segments only, and when a download aborts in the middle of downloading a segment, the rest of it will be unnecessarily stored. Similarly, under large segment sizes, we may admit more unpopular objects since admitting one segment of a one-timer amounts to admitting a substantial part of the object. The figure also shows that our variable segmentation scheme achieves similar byte hit rate as a uniform 1 MB segmentation, which imposes higher cache management overhead.
Chapter 4

Conclusions and Future Work

This chapter summarizes the conclusions of this thesis and outlines some future research directions for this research.

4.1 Conclusions

In this thesis, we conducted an eight-month measurement study on the Gnutella P2P system. Using real-world traces, we studied and modeled characteristics of P2P traffic that are relevant to caching, such as popularity, object size and popularity dynamics. We found that the popularity distribution of P2P objects cannot be captured accurately using Zipf-like distributions. We proposed a new Mandelbrot-Zipf model for P2P popularity and showed that it closely models object popularity in several AS domains. We also studied object sizes in P2P systems and showed the existence of multiple workloads. Our measurement study indicates that: (i) The Mandelbrot-Zipf popularity has a negative effect on byte hit rates of caches that use LRU or LFU policies, and (ii) Object admission strategies in P2P caching are critical to the performance of P2P caches.

We designed and evaluated a new P2P proportional partial caching algorithm that is based on segmentation and incremental admission of objects according to their popularity. Using trace-driven simulations, we showed that our algorithm outperforms traditional algorithms by a significant margin and achieves a byte hit rate that is up to triple the byte hit rate achieved by other algorithms. Our results indicate that partial caching is crucial for the performance of caching algorithms for P2P systems. We showed that our algorithm is robust against various workloads with different degrees of temporal correlations and against
aborted downloads which are common in P2P systems.

4.2 Future Work

The research presented in this thesis can be extended in several directions. We summarize some of these directions below.

- Extending the measurement infrastructure to monitor other popular P2P systems, such as Kazaa, BitTorrent and Emule. It would be interesting to model the traffic from several systems and study object and user characteristics that are relevant to caching across all of them.

- Analyzing and fine-tuning our algorithm. Along this line, we may analytically analyze aspects of our P2P caching algorithm such as: (i) How close it is to the off-line optimal algorithm; (ii) the effect of popularity dynamics on caching performance; and (iii) the impact of partial admission of objects on the achieved byte hit rate. We could also study preferential segment eviction whereby different segments of the same object carry different weights.

- Designing and implementing a software prototype for a P2P proxy cache. The cache should be able to manage concurrent downloads from many clients using different P2P protocols. Several research aspects need to be addressed for this prototype to work, including: (i) P2P traffic identification [14,30], (ii) transparent caching and serving of objects from different P2P protocols, and (iii) designing object segmentation schemes such that segments could be stored efficiently in the file system and served quickly to requesting clients.

- Designing a cooperative P2P caching protocol, where proxy caches in different ASes cooperate to serve requests for each other in the hope of minimizing WAN traffic. This project is being carried out in our research group as a continuation of this work.

- Designing efficient cache zoning techniques where different workloads occupy different zones in the cache with different cache spaces. This will allow the cache to optimize on different metrics at different times. For instance, a cache that is interested in enhancing the user experience would more likely give large portion of the cache space to audio objects. The same cache could also be tuned to minimize the outgoing traffic.
by allocating a larger zone for large video objects. In all cases, different algorithms might work better for different zones and could be run in the same cache concurrently. Cache zones might also cooperate by evicting the least valuable object from any of them to optimize a common desired metric.
Appendix A

Data Processing

During the eight months of our measurement, we collected over 50 GB of raw data. This huge amount was almost impossible to process using computer memory. Thus, we deployed an enterprise Oracle database on a server in our NSL [20] lab to store and manage the traces. In this appendix, we describe the process of collecting, processing and analyzing the trace data.

A.1 Data format

Our measurement nodes write logs into two different sets of files: Query files and reply files. Query files contain all query messages, while reply files contain all query hit messages observed by our node.

A typical format of a query message is as follows.

```
```

Figure A.1: A raw query message

Where each message has the following fields:

- timestamp in Unix format of when the message was received.
- search keywords entered by the user
APPENDIX A. DATA PROCESSING

- **ttl**: time to live of the query message.
- **hops**: number of hops the message has encountered so far.
- **meta**: query meta-data.
- **address**: IP address of the immediate neighbor in the overlay that forwarded the query to the current node.
- **urns**: Uniform Resource Names the user could provide for the query.
- **all**, **video**, **audio**, **document**, **images**, **winProg**, **lnxProg** indicate the type of content the query is for.

A reply message contains one or more hits from a certain node, and looks as follows:

```
1139427652604,QueryReply::
2 hits
guid=6212A89DD55C0345AE19FC731B3A408F, ttl=3, hops=2, priority=0
ip: 192.168.1.97
[index: 43
size: 5615494
name: Freeway - Turn Out The Lights (Produced By Kanye West) (Instrumental).mp3
xml document: null
urns: [urn:sha1:XDUGOB627F32LOADBGR16OU5X7QBUKT],
index: 146
size: 6211081
name: Instrumentals-Kanye-West,-Mos Def, Freeway - 2 Words (Instrumental) 1.mp3
xml document: null
urns: [urn:sha1:ZMXGECSUXWTZQESP74B7FCHQRO3G7YW]]
```

Figure A.2: A query hit message

Where each query hit message has the following fields:

- **timestamp** in Unix format of when the message was received.
- **n** hits indicates the number of hits returned in this message.
- **GUID**: unique identification for the message.
- **ttl**: time to live of the message.
- **hops**: the number of hops the message has encountered.
APPENDIX A. DATA PROCESSING

- priority: message priority.
- ip: IP address of the responding node with the files.
- index: index of the file on the local disk of the responding node.
- size: file size in bytes.
- name: file name on the responding node.
- URN: unique identified of the file (hash of the content).

In this research, we relied mostly on reply messages, hence we will mainly concentrate on the processing and analysis of reply messages in the remaining of this appendix.

A.2 Data processing

To compress the size of our trace files and eliminate bogus data, we process raw data into new processed files organized by month. We use the java regular expressions for fast processing. The following regular expression patterns can be compiled using the java regular expression package to parse the query-hit message above. The following line parses the responding peer's information,

```
.*?(\d+), QueryReply.+(\d+(?=(?<= hits))).*ttl=(\d+), hops=(\d+).*ip: ([\d.]+).*?\[.*?
```

and the following line parses a single hit/entry in the query reply message as illustrated in Fig. A.2.

```
```

The previous expressions efficiently parse the traces without any need for complex code. The brackets contain groups we want to obtain from the traces. Check the java regular expressions library [26] for more details on how they are used.

The processed logs are written into separate files with '&&' as a separator between the fields we described earlier. At this stage, we map all IP addresses to their corresponding city, country, ISP and AS number and store those fields in the processed log files.

A.3 Data Analysis

We set up an Oracle 10g database in our lab and use it to store and manage our data and retrieve information. In this section, we describe some of the important SQL scripts/queries we used in storing and querying the data.
A.3.1 Storing the data

We create a table called replies using the following SQL command:

```
CREATE TABLE REPLIES
(
    TIMESTAMP NUMBER(38),
    TTL NUMBER(38),
    HOPS NUMBER(38),
    ADDRESS VARCHAR2(18),
    CTRYCODE VARCHAR2(5),
    CITY VARCHAR2(64),
    ISP VARCHAR2(256),
    ASNUM NUMBER(38),
    FILESIZE NUMBER(38),
    FILEINDEX NUMBER(38),
    FILENAME VARCHAR2(512),
    FILEURN VARCHAR2(36)
)
/
```

We load the data from the processed trace files using SQL * loader using the following SQL script:

```
LOAD DATA
INFILE '/Data/january.dat'
APPEND INTO TABLE REPLIES
FIELDS TERMINATED BY '$&'
(
    TIMESTAMP INTEGER EXTERNAL,
    TTL INTEGER EXTERNAL,
    HOPS INTEGER EXTERNAL,
    ADDRESS CHAR(18),
    ASNUM INTEGER EXTERNAL,
    ISP CHAR(256),
    CITY CHAR(64),
    CTRYCODE CHAR(5),
    FILEINDEX INTEGER EXTERNAL,
    FILESIZE INTEGER EXTERNAL,
    FILENAME CHAR(512),
    FILEURN CHAR(36)
)
A.3.2 Querying the data

In the following, we describe some of the complex queries we used to retrieve information from the database. To get the top ranking tuples from a table, e.g., top ranking AS numbers, we use a query of the form:

\[
\begin{align*}
&\text{select } * \text{ from (select ASNUM, count(*)}, \text{rank() over (order by count(*))} \\
&\text{DESC}) \text{ r from REPLIES group by ASNUM order by 2 desc where r } \leq 20; \\
\end{align*}
\]

To get the frequency of objects in top 20 ASes into a file, we use the following script, repeating lines 3–7 and changing AS number (line 5) and the output file (line 4) only for the rest of the ASes:

```
set heading off;
set termout off;
spool on;
spool /Data/TopAs/onel4832.out
select count(*) from REPLIES where ASNUM like '14832' group by fileurn, address order by count(*) desc;
spool off;
set heading off;
set termout off;
```

This script took over 8 hours to complete. Upon completion, the frequency of objects in all top 20 ASes were written to 20 separate files.

To retrieve file sizes for all ASes sorted in a descending order, we use the following script

```
set heading off;
set termout off;
spool on;
spool /Data/AllSizes.out
select avg(F1LESIZE) a from replies group by fileurn order by a desc;
spool off;
set termout on;
set heading on;
```
APPENDIX A. DATA PROCESSING

Notice that, since fileurn uniquely identify a file, \( \text{avg}(	ext{FILESIZE}) \) simply gives the size of the file. We use the function \( \text{avg} \) since \( \text{FILESIZE} \) is not used in the group by clause. The query took about 40 minutes to complete.

To get the top 10 most requested objects in the top 2 ASes during the month of March 2006, we use the following command:

\[
\begin{align*}
\text{select} & \quad * \quad \text{from} \quad (\text{select} \quad \text{FILEURN}, \quad \text{count}(*), \quad \text{rank}() \quad \text{over} \quad (\text{order} \quad \text{by} \quad \text{count}(*)) \quad \text{DESC}) \\
& \quad \text{r} \quad \text{from} \quad \text{replies} \quad \text{where} \quad \text{ASNUM} \quad \text{in} \quad (14832,18173) \quad \text{and} \quad \text{TIMESTAMP} \quad \text{BETWEEN} \quad 1141171200123 \\
& \quad \text{and} \quad 1143676800123 \quad \text{group} \quad \text{by} \quad \text{FILEURN} \quad \text{order} \quad \text{by} \quad 2 \quad \text{desc}) \quad \text{where} \quad r \quad \leq \quad 10;
\end{align*}
\]

Once we obtain the top most requested objects, we can issue another query to obtain the timestamps at which those objects were observed by our measurement node.

Due to the huge size of the data, some queries took a long time to execute (in the order of hours). To monitor the progress of a query, we execute the following SQL command:

\[
\begin{align*}
\text{SELECT} & \quad \text{opname}, \quad (\text{elapsed} \quad \text{seconds} \quad + \quad \text{time} \quad \text{remaining}) \quad \text{total} \quad \text{time}, \quad \text{time} \quad \text{remaining}, \\
& \quad 100 - ((\text{time} \quad \text{remaining}/(\text{elapsed} \quad \text{seconds} \quad + \quad \text{time} \quad \text{remaining})) \quad * \quad 100) \quad \text{percentage} \quad \text{complete} \\
\text{FROM} & \quad \text{v$session_longops} \quad \text{WHERE} \quad \text{time} \quad \text{remaining} > 0;
\end{align*}
\]
Bibliography


