TOWARDS POWER-EFFICIENT INTERNET STREAMING
TO MOBILE DEVICES

by

Yao Liu
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

Dr. Songqing Chen, Dissertation Director
Dr. Robert Simon, Committee Member
Dr. Fei Li, Committee Member
Dr. Brian L. Mark, Committee Member
Dr. Sanjeev Setia, Department Chair
Dr. Kenneth S. Ball, Dean, The Volgenau School of Engineering

Date: Spring Semester 2013
George Mason University
Fairfax, VA
Towards Power-Efficient Internet Streaming to Mobile Devices

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Yao Liu
Master of Science
George Mason University, 2012
Bachelor of Science
Nanjing University, 2007

Director: Dr. Songqing Chen, Professor
Department of Computer Science

Spring Semester 2013
George Mason University
Fairfax, VA
Dedication

I dedicate this dissertation to my parents.
Acknowledgments

I am deeply grateful to my advisor, Dr. Songqing Chen, for his help during my PhD studies. Dr. Chen introduced me to research and trained me to become a researcher. Throughout my graduate studies, he consistently provided me with guidance, support, and encouragement. Dr. Chen is a role model whose enthusiasm, dedication to research, and commitment to high standards I can only hope to emulate.

I want to thank Dr. Robert Simon, Dr. Fei Li, and Dr. Brian L. Mark for their time and effort serving on my dissertation committee. I also want to thank them for their valuable advice that helped to improve this dissertation.

I am also grateful to our collaborators, Dr. Lei Guo and Dr. Yang Guo, for their valuable advice and constructive comments regarding my research, and to Dr. Bo Shen, for graciously providing us with server-side logs and for helpful discussions.

Additionally, I would like to give my thanks to my lab mates and friends at George Mason University: Dongyu Liu, Lei Liu, Vinay Devadas, Baoxian Zhao, Bo Zhang, Zhi Zhang, Mohammed Anowarul Hassan, Mohammad Atiq Haque, Mengbai Xiao, Ming Zhang, An Wang, just to name a few.

I also have to give my special thanks to Yingyu Bu at University of California, Irvine and Zhenhua Li at Peking University, who spent a lot of time discussing research with me over years.

Finally, I would like to thank my parents for always giving their full support to my studies.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>xi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Internet Streaming to Mobile Devices</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Power Efficiency of Internet Streaming Protocols</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 RTSP Streaming</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Pseudo Streaming</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3 Chunk-Based Streaming</td>
<td>4</td>
</tr>
<tr>
<td>1.2.4 Peer-to-Peer (P2P) Streaming</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Dissertation Organization</td>
<td>7</td>
</tr>
<tr>
<td>2 An Empirical Investigation of Internet Streaming Delivery to Mobile Devices</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Background and Workload Overview</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Characterization of Mobile Device Heterogeneity</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 Mobile System Heterogeneity</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2 Audio/Video Codec Heterogeneity</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Characterization of Mobile Streaming Videos</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 Video Playback Duration and File Size</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Popularity of Mobile Videos</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3 Popularity of Different Video Versions</td>
<td>25</td>
</tr>
<tr>
<td>2.4.4 Popularity Evolution</td>
<td>26</td>
</tr>
<tr>
<td>2.4.5 Correlation Between Popularity and Video Length</td>
<td>29</td>
</tr>
<tr>
<td>2.5 Trade-off between CPU and Storage</td>
<td>30</td>
</tr>
<tr>
<td>2.6 Related Work</td>
<td>33</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>34</td>
</tr>
<tr>
<td>3 Towards Power-Efficient Internet P2P Streaming to Mobile Devices</td>
<td>36</td>
</tr>
</tbody>
</table>
# Table of Contents

3.1 Introduction .................................................. 36  
3.2 Background and Related Work ................................. 39  
3.3 Measurement and Analysis ..................................... 40  
3.3.1 Methodology and Result Overview ......................... 40  
3.3.2 Impact of Control Traffic ................................. 43  
3.3.3 Impact of Uploading Traffic ............................... 47  
3.3.4 Impact of the Number of Neighbors ....................... 48  
3.3.5 Summary of Measurement Findings ....................... 49  
3.4 Design of BlueStreaming ..................................... 49  
3.4.1 Motivation ................................................. 49  
3.4.2 BlueStreaming Overview .................................. 51  
3.4.3 Traffic Classifier: Diverting Control Traffic to Bluetooth ... 52  
3.4.4 Traffic Shaper: Shaping Ingress Streaming Traffic ........ 53  
3.4.5 Uploading Scheduler: Scheduling Upload Wisely .......... 55  
3.5 Implementation ................................................ 57  
3.5.1 Prototype Implementation ................................. 57  
3.5.2 Infrastructure vs. Hybrid Mode ............................ 58  
3.5.3 Channel Selection and Competition Avoidance .......... 59  
3.6 Performance Evaluation ...................................... 60  
3.6.1 Experimental Setup ....................................... 60  
3.6.2 Energy Saving in the Infrastructure Mode ................. 60  
3.6.3 Hybrid Mode .............................................. 64  
3.6.4 Retransmission Rate ...................................... 66  
3.7 Summary ....................................................... 68  
4 Eliminating Redundant Internet Streaming Traffic to iOS Devices in C/S Streaming 69  
4.1 Introduction .................................................... 69  
4.2 HTTP Range Request and Streaming to iOS .................. 72  
4.3 Internet and Local Measurements ............................. 73  
4.3.1 A Server-Side View ...................................... 73  
4.3.2 A Client-Side View ...................................... 75  
4.4 Analysis of Redundant Streaming Traffic .................... 77  
4.4.1 MediaPlayer RE-REQUESTs downloaded data after the entire file is downloaded ......................... 78  
4.4.2 The client frequently ABORTs the connection, causing data in flow useless .............................. 81
4.4.3 Additional OVERLAPPING requests are sent to compensate slow connections ................................. 87
4.4.4 Summary of Analysis Findings ................................. 89
4.5 Design and Implementation of CStreamer ................................. 89
  4.5.1 CStreamer Design ................................. 90
  4.5.2 CStreamer Implementation ................................. 92
  4.5.3 CStreamer iOS App ................................. 94
4.6 Performance Evaluation ................................. 94
4.7 Related Work ................................. 97
4.8 Summary ................................. 98
5 Conclusion and Future Work ................................. 100
  5.1 Conclusion ................................. 100
  5.2 Future Work ................................. 101
Bibliography ................................. 103
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of Workload</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>System Heterogeneity of Mobile Devices</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Video Codecs</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Video Resolution and Encoding Rate</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Video Quality</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary of Video Accesses</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of Statistics on Laptop</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Comparison of Sleep Time (%)</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Simple Strategies Do Not Work Well</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>PPTV at 400 Kbps</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>QQLive at 500 Kbps</td>
<td>64</td>
</tr>
<tr>
<td>3.6</td>
<td>RTT based on customized Ping</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Devices Used in Experiments</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>Average Transferred Size vs. File Size (Bytes)</td>
<td>77</td>
</tr>
<tr>
<td>4.3</td>
<td>Extra Traffic Delivered after Downloading is Finished (Bytes)</td>
<td>80</td>
</tr>
<tr>
<td>4.4</td>
<td>Estimated Start-up Delay (seconds)</td>
<td>96</td>
</tr>
<tr>
<td>4.5</td>
<td>Average WNIC Sleep Time (%)</td>
<td>97</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Daily Accesses in November 2010</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Hourly Accesses On November 1st 2010</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Weekly Accesses from November 8 to November 15 2010</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Most Popular Resolutions</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Audio Codecs</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Video Codecs</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td># of Versions Per Video (CDF)</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Video Playback Duration Distribution</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Video File Size Distribution</td>
<td>21</td>
</tr>
<tr>
<td>2.10</td>
<td>Daily Video Popularity Distribution</td>
<td>22</td>
</tr>
<tr>
<td>2.11</td>
<td>Monthly Video Popularity Distribution</td>
<td>24</td>
</tr>
<tr>
<td>2.12</td>
<td>Daily Version Popularity Distribution</td>
<td>25</td>
</tr>
<tr>
<td>2.13</td>
<td>Monthly Version Popularity Distribution</td>
<td>26</td>
</tr>
<tr>
<td>2.14</td>
<td>Percentage of Change in Top Videos</td>
<td>28</td>
</tr>
<tr>
<td>2.15</td>
<td>% of Videos Remaining in Top</td>
<td>29</td>
</tr>
<tr>
<td>2.16</td>
<td>% of Sessions for Videos of Different Length (minute)</td>
<td>29</td>
</tr>
<tr>
<td>2.17</td>
<td>Cache Performance over Different Time</td>
<td>32</td>
</tr>
<tr>
<td>3.1</td>
<td>PPTV (400kbps) 30-min Test</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>PPS (400kbps) 30-min Test</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>SopCast (530kbps) 30-min Test</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>QQLive (500kbps) 30-min Test</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>Variation of Upload Throughput (averaged over 10 seconds)</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Overview of BlueStreaming Architecture</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>PPTV 30-min Tests: Inter Packet Delay of WiFi (CDF)</td>
<td>61</td>
</tr>
<tr>
<td>3.8</td>
<td>Traffic Pattern of PPTV</td>
<td>62</td>
</tr>
<tr>
<td>3.9</td>
<td>Energy Consumption Comparisons</td>
<td>66</td>
</tr>
<tr>
<td>3.10</td>
<td>Retransmission Rates</td>
<td>67</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.1 iOS Traffic from Server-side Log</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>4.2 Statistics of iOS Devices Watching YouTube</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>4.3 HTTP Range Requests in one YouTube Experiment</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>4.4 Received Range per Request in one YouTube Experiment</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>4.5 TCP Stream Connections</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>4.6 Time and Traffic Delivered During Half-closed TCP</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>4.7 Throughput (KBytes/s) of Aborted Connections (CDF)</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>4.8 Byte-Range of Requests in Slow Connections</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>4.9 Overview of CStreamer</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>4.10 Tests with YouTube (a video of 36.7 MB)</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>4.11 Tests with Dailymotion (a video of 26.2 MB)</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>
Abstract

TOWARDS POWER-EFFICIENT INTERNET STREAMING TO MOBILE DEVICES
Yao Liu, PhD
George Mason University, 2013
Dissertation Director: Dr. Songqing Chen

Internet streaming services are very popular today. According to Cisco, video traffic accounts for more than 51% total Internet traffic. With the pervasive adoption of various mobile devices in practice in the past several years, today Internet streaming services are receiving a rapidly growing number of requests from various mobile devices. As a result, more than 50% of the data consumed by mobile devices is video streaming traffic. However, streaming delivery to mobile devices is more challenging than to its desktop counterpart.

In this dissertation, we first empirically investigate Internet mobile streaming practices. For this purpose, we conduct detailed analysis on a one-month server log (with 212 TB delivered video traffic) from a top Internet mobile streaming service provider serving worldwide mobile users. We investigate mobile streaming from various perspectives, including hardware and software heterogeneity, different characteristics of mobile videos, and different user access patterns. The results provide us in-depth understanding on the current Internet mobile streaming services.

A critical constraint on mobile devices for receiving Internet streaming services is that they have limited battery capacities. While watching streaming videos, the battery power is depleted at a fast rate by the wireless network interface card (WNIC) for streaming data
transmission, by the CPU or GPU for computationally intensive video decoding, and by the display for rendering the video. Among these power-consuming sources, transmission power consumption is very significant: for a mobile device receiving streaming services, about 30% to 40% of the power is consumed by the WNIC for streaming data transmission. So in order to prolong the battery lifetime, it is important to save the battery power consumed by the WNIC. For this purpose, we design and implement new schemes that can effectively save battery power consumption while maintaining good streaming quality to mobile devices. In particular, we focus on peer-to-peer (P2P) streaming and client-server streaming as they are widely used. We aim to save battery power consumption from two aspects: (1) how the data is received, and (2) how much data is received.

Through extensive Internet experiments, we find that the widely used 802.11 power saving mode (PSM) cannot effectively save power for popular P2P streaming services: several unique characteristics of P2P traffic, such as highly-frequent and delay-sensitive control packets, often prevent the wireless interface from switching into power saving mode. To overcome these challenges, we design BlueStreaming, a system that can intelligently leverage Bluetooth to transmit P2P control traffic. Experiments using a BlueStreaming prototype with a number of popular P2P streaming applications show that BlueStreaming can achieve a reduction of power consumption by up to 46% compared to the commodity PSM scheme.

Besides how data is delivered, the battery power consumption on mobile devices is also affected by how much data is received. However, in the current practice, we find from experiments that iOS users always receive a significant amount of redundant traffic when accessing Internet streaming services such as YouTube in a client-server architecture. Such redundant traffic not only leads to more battery power consumption but also can incur monetary cost. After analyzing the underlying reasons, we design and implement a system called CStreamer that can minimize such redundant traffic. In experiments, a CStreamer prototype deployed on Amazon EC2 completely eliminates redundant traffic and reduces battery power consumption by up to 40%.
Chapter 1: Introduction

Internet streaming to mobile devices is increasing in popularity. However, unlike streaming to desktop computers, the limited battery power supply poses a critical constraint for streaming to mobile devices. In receiving streaming services on a mobile device, there are three major power drainage sources: the wireless network interface card (WNIC) for streaming data transmission, the CPU or GPU for computationally intensive video decoding, and the display for video rendering.

This dissertation aims to investigate effective ways to reduce the power consumption during Internet streaming to mobile devices.

1.1 Internet Streaming to Mobile Devices

With the reduced price and the improved processing speed, mobile devices, such as iPhone and iPad, are becoming more and more popular. By September 2010, over 58.7 million people in the US owned smartphones [1]. With the most recent technology advancements, people today are more inclined to use their mobile devices to access the Internet. For example, it has been reported that iPad users generated 2.5 times more Internet traffic than iPhone users for accessing popular services, such as USA Today, Google Maps, Bloomberg, and eBay, after its release [2].

To support mobile Internet accesses, 802.11 and the third generation (3G) wireless networks are also quickly developing. For example, WiFi operates in more than 595,000 public hotspots [3]. Jupiter Research also estimated that 65% households in the United States have home-deployed WiFi access points (APs) [4].

Besides the general web surfing on the Internet, these days more and more accesses from mobile devices are directed to all kinds of Internet streaming services. For example,
YouTube [5] is among the earliest to provide video streaming services to mobile devices. Today both iOS and Android have native support for YouTube. Other popular streaming service providers, including Netflix [6] and Hulu [7], also provide streaming services to subscribed mobile users via APPs built in various mobile operating systems. As a result, mobile video traffic now dominates the Internet mobile traffic. According to Cisco’s report [8], mobile video traffic accounted for more than 50% of total mobile data traffic in 2012, and is predicted to exceed two thirds by 2017.

However, using a mobile device to receive Internet streaming services can quickly exhaust its limited battery supply because receiving Internet streaming services often involve continuous and bulk data transmissions, decoding, rendering, and displaying. Among them, wireless data transmissions consume a significant amount of energy. For example, it was reported that the WiFi radio on HTC Tilt 8900 series consumes up to 5x more power than the base energy consumption of the phone [9]. As a result, power saving on the wireless network interface card (WNIC) has been a major focus of many research efforts.

The battery consumption of the WNIC is dominated by the following two aspects: (1) how much data is transmitted during the streaming session, and (2) how such data is transmitted. The first aspect is very intuitive. Under the same circumstances of all other factors, transmitting more data would cause the WNIC to work longer and thus consume more battery power. The second aspect, however, is more complicated. Since most commodity WiFi interface cards now support the IEEE 802.11 power saving mode (PSM) by default, it is straightforward that the PSM should be leveraged to minimize the battery power consumption during streaming services. However, the standard PSM has limited effectiveness for streaming applications since streaming applications often involve bulk data transmission in a continuous fashion, which does not provide many opportunities for the WNIC to switch into the PSM. Thus, much research has been conducted to discover methods to cause the WNIC to sleep as much as possible during streaming to mobile devices. For example, PSM-throttling [10] proposes a client-centric scheme to send streaming data in bursts, maximizing the sleep time of the WNIC. In Proxy buffering [11], a local intermediate
proxy buffers packets then sends them at client-specific intervals. The key idea of these schemes is to shape traffic so that packets can be sent in bursts without degrading streaming quality. By utilizing traffic shaping, the WNIC of mobile devices can increase the time they spend in low power sleep mode, which decreases power consumption.

1.2 Power Efficiency of Internet Streaming Protocols

We next briefly discuss the streaming protocols that are currently being used by existing Internet mobile streaming applications, paying particular attention to the the power efficiency of these protocols.

1.2.1 RTSP Streaming

Real-time streaming protocol (RTSP) [12] is a standard Internet streaming protocol for communication between a media player and a media server. In practice, today YouTube [5] and Vuclip [13] use RTSP to deliver streaming content to mobile devices such as BlackBerry, Nokia, and SonyEricsson mobile phones. RTSP is a stateful protocol that sets up a media session between the client and the server and allows the client to execute VCR-like commands such as play, pause, fast forward and rewind. The client maintains a small buffer to absorb network jitter and ensure smooth playback. With RTSP, the streaming server delivers streaming data at the rate equivalent to the video encoding rate. Because of the constant rate delivery, the WNIC can barely sleep, and it is among the least power efficient protocols [14].

1.2.2 Pseudo Streaming

While RTSP is a standard streaming protocol, Internet streaming services today also widely use HTTP for streaming content delivery, which is referred to as pseudo streaming or progressive downloading. With pseudo streaming, a client downloads the media file from an HTTP server. The playback can start before the entire file is downloaded. Pseudo streaming is used in many popular Internet streaming services today, including YouTube,
Dailymotion [15], Veoh [16], and Vuclip. Pseudo streaming is employed in a mobile version of YouTube, which allows iOS and Android users to use to watch videos in their native browsers [5].

Fast Start [17] is often used in pseudo streaming. Fast start allows the server to deliver the data at a higher speed than the encoding rate at the beginning of the session and then uses a low speed to deliver the rest of the file. Fast Start thus tends to reduce the startup delay perceived by a user and smooths network jitter during playback. With Fast Start, a large amount of data is transmitted to the mobile device at the beginning, which provides more opportunities for the WNIC to sleep when receiving the rest of data, resulting in battery power savings [14]. On the other hand, Fast Start can also introduce a significant amount of redundant traffic, which causes more data than needed be transmitted and consumes additional battery power.

1.2.3 Chunk-Based Streaming

Chunk-based streaming also uses HTTP to deliver streaming content. However, different from pseudo streaming that progressively downloads a single video file, the media content is segmented into many chunks, and a client periodically queries the streaming server for new file chunks. Many streaming services today are chunk-based, including Apple HTTP Live Streaming [18], Flash HTTP Streaming [19], and Microsoft Smooth Streaming [20].

Typically, in chunk-based streaming, the media file is segmented into smaller chunks, each containing a few seconds (e.g., 2 seconds for Microsoft Smooth Streaming, and 10 seconds for Apple HTTP Live Streaming) of media content. For streaming access, the client requests the segmented chunks. These chunks are reassembled after being downloaded, and fed into the MediaPlayer for playback. With the help of traffic shaping, chunk-based streaming data can be delivered in a burst, which allows the WNIC on a mobile device to sleep often [14].
1.2.4 Peer-to-Peer (P2P) Streaming

Peer-to-Peer (P2P) has demonstrated in practice to be scalable for file sharing and Internet streaming. Many Internet streaming systems have adopted P2P techniques. For example, PPTV [21], PPStream [22], SopCast [23], and QQLive [24] are all Internet-scale P2P streaming systems that provide hundreds of TV channels and serve millions of Internet users every day. Within a P2P system, a peer downloads streaming content for the playback from other users and uploads downloaded content to other peers simultaneously. While different protocols have been extensively studied, in practice, most of deployed Internet P2P streaming systems today are mesh-based and they often use a pull approach for data acquisition. That is, peers in these systems need to request data chunks from a dynamically changing set of neighbors. To pull/request data from neighbors, a peer first needs to exchange data chunk availability information (e.g., buffermap) with neighbors. After knowing which data chunks available at which neighbors, the peer then sends Streaming Data Chunk Requests to request missing chunks from different neighbors. Typically, the unit of a data chunk is small (e.g., 1280 bytes) compared to the video file size. Thus such data chunk requests are fine-grained, and are highly frequent. These buffermap messages and fine-grained data chunk requests are referred to as control packets [25] [26]. Because a user has to frequently exchange control packets and upload data to other peers, the WNIC cannot sleep for long time as in the pseudo streaming or chunk-based streaming [14].

1.3 Contributions

In this dissertation, we investigate how to save battery power consumption on mobile devices while maintaining streaming quality. Our approach generally proceeds in two steps: (1) Via extensive measurements, we first reveal inefficiencies in today’s Internet streaming services. (2) We design and implement systems that can address the problems and effectively save energy. We focus on the following two aspects that affect the WNIC power consumption:
How is the streaming data transmitted?

Because the power consumption of WNIC in sleep mode is significantly less than in active mode, we show how to manipulate the transmission of streaming data so that the WNIC can spend a maximum amount of time in low power consumption sleep mode and save the power consumption.

How much data is transmitted?

During the streaming session, transmitting more data via the WNIC causes the WNIC work longer and consume more power. We thus investigate the total amount of traffic received at mobile devices during Internet streaming and show how power consumption can be saved by effectively eliminating the redundant streaming traffic.

The major contributions of this dissertation are summarized as follows:

• We first empirically study Internet mobile streaming services from the server-side via one month of server log data collected from one of the largest Internet mobile streaming service providers. Through detailed analysis, we show great hardware and software heterogeneity of mobile devices, different characteristics of mobile videos, and different user access patterns from those in traditional Internet streaming services. To deal with heterogeneity among mobile devices and reduce decoding power consumption at mobile devices, streaming servers leverage online-transcoding at the server-side. Online transcoding poses a great challenge for mobile streaming servers as it requires a huge amount of CPU resources. We show that caching at the server side with a proper replacement policy can effectively trade limited storage size for significant savings of CPU cycles.

• For mobile clients receiving P2P streaming, through Internet measurement and analysis, we find that battery power consumption of the WNIC is highly affected by the control, uploading traffic, and the dynamics of neighboring peers. Motivated by the measurement results, we design and implement BlueStreaming, which takes into
account the unique characteristics of Internet P2P streaming and leverages the commonly existing Bluetooth interface to transmit delay-sensitive control traffic while using WiFi for streaming data traffic. BlueStreaming works with all existing P2P streaming applications and effectively saves battery power by trading the power consumption of Bluetooth for greater power savings on the WiFi interface via intelligent traffic shaping. Our experimental results based on our BlueStreaming prototype show that BlueStreaming can reduce energy consumption by up to 46% compared to the commodity PSM scheme.

- For mobile clients receiving streaming services via pseudo streaming, through analysis of server-side workload and experiments in a controlled lab environment, we find that the current practice has introduced a significant amount of redundant traffic particularly when using iOS devices. This extra traffic not only over-utilizes Internet resources but also results in additional battery power consumption at the WNIC as well as more monetary costs for mobile users. To investigate the potential reasons of such redundant traffic, we conduct experiments and analyze the captured traces. We find that the redundant traffic is mainly caused by the limited available memory and too fast or too slow network connections. To reduce redundant traffic without requiring changes at the server side or the client side, we design CStreamer, which transparently works between the client and the server. In experiments, a CStreamer prototype deployed on Amazon EC2 completely eliminates redundant traffic and can reduce battery power consumption by up to 40%.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we first investigate Internet streaming services to mobile devices from the server side. In Chapter 3, we discuss the unique characteristics of P2P streaming and our design and implementation of BlueStreaming. In Chapter 4, we present our measurement results about the redundant
traffic in pseudo streaming and our design and implementation of CStreamer. Chapter 5
summarizes the dissertation and discusses some future work.
Chapter 2: An Empirical Investigation of Internet Streaming Delivery to Mobile Devices

2.1 Introduction

To understand the key challenges of Internet mobile streaming and the difference from traditional Internet streaming, a number of studies have been performed. As today the majority of Internet mobile streaming services are delivered in a client-server architecture, many studies have focused on the resource consumption and streaming quality received on the mobile device. For example, Xiao et al. [27] studied energy consumption when watching YouTube on mobile devices. Huang et al. [28] investigated fetching policies of different mobile video players, and Finamore et al. [29] examined the potential causes for inferior streaming quality of mobile YouTube accesses.

However, these studies mainly focus on the client side by examining specific devices [27, 28] or via local experiments [29]. As the key to the current Internet mobile streaming services, the server side plays a critical role in the entire streaming delivery process. Unfortunately, so far, little is known about the server side, possibly due to the limited availability of data from the server side.

To provide in-depth understanding of the current Internet mobile streaming services, in this chapter, we set to investigate the server side in streaming delivery to mobile devices. For this purpose, we have collected a 30-day server log from a top Internet mobile streaming service provider. In 30 days, about 105 million video sessions were served with about 212 TB video traffic delivered. Through our analysis, we have a number of findings. While the details are presented later in the chapter, some highlights are as follows:

- A unique challenge for Internet streaming delivery to mobile devices is rooted from the
fact that mobile devices are very heterogeneous. In this workload, we find over 2800 different hardware models with 92 different screen resolutions running 14 different mobile operating systems, using 3 audio codecs and 4 video codecs. This greatly challenges the traditional Internet streaming delivery infrastructure where the bottleneck often lies in the limited bandwidth.

- To deal with the device heterogeneity, runtime transcoding is used to customize a video to the appropriate versions on the fly for different devices. A video clip could be transcoded into more than 40 different versions in order to serve requests from different devices.

- Compared to videos in traditional Internet streaming, mobile streaming video clips are typically of much smaller size (with a median of 1.68 MBytes) and the video duration is shorter as well (with a median of 2.7 minutes). Furthermore, the daily mobile user accesses are more skewed following a Zipf-like distribution but users' interests also shift quickly, resulting in a stretched-exponential distribution in the long term.

To reduce the huge CPU cycles demanded for transcoding on the fly, we further explore caching at the server side by trading off storage for CPU cycles. Our study shows that a policy that considers different versions of a video altogether outperforms other intuitive ones (e.g., a file based one) when the cache size is limited. As far as we know, we are the first to provide a server-side analysis on a Vuclip-like Internet mobile streaming service. Our findings provide new insights and lay some foundations to improve the current Internet mobile streaming delivery.

Contributions described in this chapter were published in [30]. The rest of the chapter is organized as follows. We describe some background and the workload overview in section 2.2 and study the device hardware and software heterogeneity in section 2.3. We examine various mobile video properties in section 2.4 and further explore the trade-off between the storage and the CPU at the server side in section 2.5. Some related work is described in section 2.6 and we summarize this chapter in section 2.7.
2.2 Background and Workload Overview

To investigate how current Internet streaming services are delivered to mobile devices, we have collected a 30-day server log from one of the largest Internet mobile streaming service providers, Vuclip [13]. Vuclip provides mobile users with the search-and-delivery services. It allows users to search for and watch any videos on any video-enabled mobile phones and devices.

Different from many existing services that only provide streaming services to specific mobile devices, Vuclip can serve any type of mobile devices that are capable of streaming playback. Vuclip allows any mobile user to search for interested video available on the Internet, and transcodes them on-demand and on-the-fly based on the type of the mobile device. To serve different types of mobile devices, Vuclip employs on-demand transcoding at the server side. Transcoding is a process to convert the requested video clip to the appropriate codecs, format, and size at runtime upon a request so that the video can be properly rendered and played on the requesting mobile device. Vuclip transcodes a video into different versions by choosing the best audio/video codecs, frame size, frame rate, and quality level combination for the mobile device. According to our analysis, each video was accessed in more than 2 versions on average (as shown in Table 2.1), and the most popular video was accessed in 41 different transcoded versions (as shown later in Figure 2.7).

To deliver video content, Vuclip uses the traditional client/server (C/S) architecture. The video file is delivered via pseudo streaming over HTTP. That is, when the requested content is available on the server, the client would issue an HTTP GET request to download the content. A video may be downloaded via several HTTP GET requests with different partial ranges specified (i.e., range requests). To differentiate video requests from HTTP requests, we define a request as a single HTTP transfer between the client and the server, and a session as the set of requests that are involved in downloading an entire video clip.

The data we collected is from Nov. 1st to Nov. 30th, 2010. In this one-month server log, there are about 105 million sessions watching more than 4 million different videos. There are a total of about 192 million HTTP requests. The total traffic delivered from the
Table 2.1: Summary of Workload

<table>
<thead>
<tr>
<th>Workload Length</th>
<th>30 Days</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Sessions</td>
<td>105,389,370</td>
</tr>
<tr>
<td># of Requests</td>
<td>192,255,173</td>
</tr>
<tr>
<td># of Requests from Mobile Devices</td>
<td>181,556,344</td>
</tr>
<tr>
<td># of Unique Videos Accessed</td>
<td>4,052,740</td>
</tr>
<tr>
<td>AVG. # of Formats Per Video</td>
<td>2.31</td>
</tr>
<tr>
<td>Total Traffic Volume</td>
<td>212 TB</td>
</tr>
</tbody>
</table>

server in these 30 days is about 212 TB. Table 2.1 gives a summary of this workload. Note that among all these requests, some are from desktop/laptop computers instead of mobile devices. In order to focus on the requests from mobile users, we differentiate them in the server log through the User-Agent strings specified in each HTTP request. By analyzing the User-Agent, we find there is a total of 150,072 unique User-Agent strings. Among them, 84,281 (56%) represent mobile devices. However, examining the received requests, we find most of them come from User-Agent strings representing mobile users: more than 94% (181 million out of 192 million) requests are from mobile devices.

With the exclusion of desktop/laptop traffic, Figure 2.1 gives an overview of the server side traffic in 30 days. Note that the left $y$-axis represents the total number of requests per day, while the right $y$-axis represents the total traffic volume per day. During this one-month period, despite a small decrease in the middle, the number of the requests and the delivered traffic amount kept increasing, indicating the popularity of Vuclip.

Figure 2.2 shows the hourly mobile streaming access patterns in a day. The figure indicates that hourly accesses peak around 17:00 GMT. Furthermore, the total number of requests and the traffic volume served during peak hours almost double these in non-peak hours. Figure 2.3 further depicts the hourly pattern from Nov. 8th to Nov. 15th, 2010 (a week). The figure shows clear peak and off-peak hourly patterns for each day. The figure also shows some drop after Nov. 12th, 2010. It is likely due to the fact that Nov. 13th was
a Saturday and Nov. 14th was a Sunday. We can observe the increase of accesses again on Monday.
2.3 Characterization of Mobile Device Heterogeneity

2.3.1 Mobile System Heterogeneity

To provide Internet streaming services to all kinds of mobile devices like Vuclip, a unique challenge is the heterogeneity among mobile devices. Different from the pre-encoding approach that was taken by many other service providers to serve specific types of mobile devices, transcoding has to be used to customize the video into a proper format for the requesting mobile device. Although transcoding is very flexible and desirable to serve heterogeneous mobile devices, transcoding demands huge CPU cycles on the fly.

To get a realistic picture of the mobile device heterogeneity, we use WURFL [31] to retrieve detailed device information based on the User-Agents information we have extracted from the server log. Among the 84,281 User-Agents that represent mobile devices, we are able to get the brand and model information from more than 74,708 (88.64%) distinct User-Agent strings. The rest only have browser information.

As shown in Table 2.2, accesses to Vuclip in these 30 days came from 2864 different
Table 2.2: System Heterogeneity of Mobile Devices

<table>
<thead>
<tr>
<th>Models</th>
<th>2864</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolution</td>
<td>92</td>
</tr>
<tr>
<td>Mobile OSes</td>
<td>14</td>
</tr>
</tbody>
</table>

device models. These devices have different screen sizes that can support video playback with different resolution rates. Delving into this, we find that these devices have 92 different resolutions (width and height combinations), ranging from $84 \times 48$ to $1600 \times 1200$. Figure 2.4 shows the most popular resolutions, including $320 \times 240$, $480 \times 360$, and $480 \times 320$. They also run on 14 different mobile operating systems.

![Figure 2.4: Most Popular Resolutions](image_url)
Figure 2.5: Audio Codecs

Figure 2.6: Video Codecs
2.3.2 Audio/Video Codec Heterogeneity

To play video on a mobile device, both audio and video codecs are required. On different devices, the supported codecs may be different as well. Such heterogeneity would further increase the load for the server if the server conducts transcoding for the mobile device. Note that if such transcoding is done at the client side, it would lead to excessive battery power consumption.

To examine the codec heterogeneity, we further look into the supported audio/video codecs on these 2864 hardware models. We find that typically there are 3 audio codecs being used, namely AAC, AMR, and WMA, and there are 4 video codecs being used, namely H.263, H.264, MPEG-4, WMV. Figures 2.5 and 2.6 show the popularity of these codecs. As shown in these figures, AMR is the most popular audio codec, as more than 59% devices support it, and H.263 and MPEG-4 are the most popular video codecs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Video</th>
<th>Audio</th>
<th>#Videos</th>
<th>Sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASF</td>
<td>WMV</td>
<td>WMA</td>
<td>293,025</td>
<td>2,031,161</td>
</tr>
<tr>
<td>3GP</td>
<td>H.264</td>
<td>AAC</td>
<td>692,004</td>
<td>12,636,639</td>
</tr>
<tr>
<td>3GP</td>
<td>H.263</td>
<td>AMR</td>
<td>2,805,494</td>
<td>46,790,565</td>
</tr>
<tr>
<td>3GP</td>
<td>MPEG-4</td>
<td>AMR</td>
<td>138,022</td>
<td>1,213,319</td>
</tr>
<tr>
<td>3GP</td>
<td>MPEG-4</td>
<td>AAC</td>
<td>1,762,132</td>
<td>36,552,760</td>
</tr>
<tr>
<td>3GP in Total</td>
<td></td>
<td></td>
<td>3,746,548</td>
<td>97,193,283</td>
</tr>
</tbody>
</table>

With 3 audio codecs and 4 video codecs, we expect a total of 12 combinations of different audio/video codecs. In practice, however, not all these combination of the audio and video codec are used. In the workload, we only find 5 combinations. Table 2.3 shows the 5 video+audio encoding schemes used. While more than 4 million (4,052,740) unique videos were accessed, we are able to extract about 3.7 million (3,789,229) that are accessed as videos. The rest were only accessed as audio. Table 2.3 shows that H.263+AMR and
MPEG4+AAC are the most popular encoding schemes, accounting for 84% of total viewing sessions. This is not surprising as H.263 and MPEG4 are the most widely supported video codecs on the 2864 models of mobile devices.

In addition to different codecs, video files are also encoded into two different formats, i.e., two types of containers, 3GP and ASF. 3GP is the 3GPP file format, which is a multimedia container format defined by the Third Generation Partnership Project (3GPP) for 3G UMTS multimedia services. 3GP is often used on 3G mobile phones. On the other hand, ASF (Advanced Systems/Streaming Format) belongs to Microsoft Media framework and it is a proprietary digital audio/digital video container format. Apparently, 3GP is much more widely used in practice than ASF for mobile videos.

Table 2.4: Video Resolution and Encoding Rate

<table>
<thead>
<tr>
<th>Quality</th>
<th>Frame Width</th>
<th>Encoding Rate (Kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>176</td>
<td>51 - 55</td>
</tr>
<tr>
<td>Low</td>
<td>320, 360</td>
<td>71 - 187</td>
</tr>
<tr>
<td>High</td>
<td>176</td>
<td>81 - 147</td>
</tr>
<tr>
<td>High</td>
<td>320, 360</td>
<td>172 - 335</td>
</tr>
<tr>
<td>WiFi</td>
<td>320, 360</td>
<td>358 - 423</td>
</tr>
</tbody>
</table>

Besides the above hardware and software heterogeneity, mobile devices may have different network speed, due to various reasons, such as accessing through cellular network or WiFi. To support different mobile Internet access speed, Vuclip also transcodes video clips into 3 different quality levels: Low Quality, High Quality and WiFi Quality. Table 2.4 shows the corresponding range of object encoding rate for different quality levels. Consider the variety of resolutions, videos are also customized into 3 different frame widths: 176, 320, and 360. As we can observe from the table that a larger resolution (width) video does not necessarily come with a high encoding rate. On the other hand, a video with a high encoding rate typically comes with a larger resolution.
Table 2.5: Video Quality

<table>
<thead>
<tr>
<th>Quality</th>
<th># of Videos</th>
<th># of Sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>1,694,108</td>
<td>26,365,900</td>
</tr>
<tr>
<td>High</td>
<td>3,323,211</td>
<td>72,392,729</td>
</tr>
<tr>
<td>WiFi</td>
<td>91,761</td>
<td>465,815</td>
</tr>
</tbody>
</table>

Table 2.5 further shows the number of videos that are of Low, High, and WiFi Quality as well as the number of their requested sessions. The videos in High Quality are mostly requested: 73% viewing sessions are for videos encoded with High Quality. Consider that Vuclip transcodes the video content on-demand, it is not surprising that 87% of video contents have at least one version encoded with High Quality. WiFi Quality, however, is the least requested quality level. This is likely due to the relatively slow mobile accessing speed and tiered data plan billing model today.

Figure 2.7: # of Versions Per Video (CDF)
Since Vuclip transcodes the original video to accommodate mobile devices with different codecs, frame width, and quality level, for the ease of presentation, we use versions to refer to different transcoded video files for each video in the rest of the chapter. On the other hand, we use videos to refer to a set of video clips that correspond to the same content. Figure 2.7 shows the CDF of number of versions each video has. As shown in the figure, in this workload, about 59% videos have only one version, and about 3% videos are accessed in 10 or more versions. The largest version number is 41.

2.4 Characterization of Mobile Streaming Videos

The previous section has shown that mobile device heterogeneity is a great challenge to the service provider. With such a level of heterogeneity, what kind of video clips are being served is of our great interest. In this section, we further analyze the mobile video clips that we have collected from the server log in order to reveal the commons with and differences from the traditional Internet streaming content.

2.4.1 Video Playback Duration and File Size

Figure 2.8 depicts the distribution of video playback duration in seconds. In this figure, videos are sorted in decreasing order of the playback duration, and the y-axis is in log scale. As shown in the figure, video clips accessed by mobile users are mostly short in terms of playback duration: more than 97% videos are less than 10 minutes long, and the median playback duration is 162 seconds (less than 3 minutes). Compared to the longer duration of traditional Internet streaming video clips, such a shorter duration makes it more feasible for mobile devices because video streaming consumes a lot of limited resources on mobile devices, including the network for data receiving, the CPU for decoding, and the display for rendering. Such resource consumption can drain the limited battery power supply at a very high rate.

Correspondingly, Figure 2.9 shows the file size (bytes) distribution. Again, we sort the video files (versions) based on their sizes in decreasing order. As shown in Figure 2.9, the
video file distribution is similar to that of the duration as shown in Figure 2.8. Note that, here in this figure, each video may have been accessed in several versions in different formats and file sizes. As we can see, most video files accessed by mobile devices are smaller than
8 MBytes, with a mean file size of 2.78 MBytes and the median file size 1.68 MBytes. This shows that videos accessed by mobile devices are mostly small in terms of bytes. This can reduce the total network transmission for downloading the video file. Note the network interface card could consume 30% to 40% of the total battery power consumed during a streaming session to a mobile device [32], [14].

Moreover, compared to the size of the traditional Internet video files [33, 34], the size distribution we find in this server log is much smaller. This provides a great opportunity for reducing the transcoding cost as we discuss later in section 2.5.

2.4.2 Popularity of Mobile Videos

![Figure 2.10: Daily Video Popularity Distribution](image)

Figure 2.10 shows the popularity pattern of videos accessed site-wide on Nov. 1st, 2010. In this figure, the $x$-axis represents videos ranked by the number of requested sessions in decreasing order, plotted in log-scale, while the $y$-axis represents the number of viewing sessions of this video, also plotted in log scale. This figure shows that, in log-log scale, the
The popularity distribution of videos accessed can be well fitted with a Zipf-like distribution

\[ y_i \propto \frac{1}{i^\alpha}, \]

where \( i \) is the popularity rank of the video, \( y_i \) is the number of requested sessions for the video, and \( \alpha \) is the skewness parameter. Moreover, we find \( \alpha = 0.955 \) fits our data very well with the goodness of fit value \( R^2 \) very close to 1, indicating the popularity distribution is not only Zipf-like, but also very close to the Zipf’s law where \( \alpha = 1 \). Similar patterns have been found for other days in the workload.

The Zipf-like distribution is known to be efficient in modeling web traffic, and is the premise for efficient web caching. Specifically, \( \alpha \) is an indicator of request concentration, and proxy caching can be more efficient with a larger \( \alpha \) value. For example, it was reported in [35] that \( \alpha \) varies between 0.64 and 0.83 for web traffic, while it tends to be smaller for media traffic (for example, work [34] reports 0.56 for YouTube traffic). Different from previous measurement studies where data were collected at edge networks, the mobile video accesses are highly concentrated at the server side. Such discrepancy is reasonable as collecting traffic at edge networks can only reflect the local users’ accesses, while the server logs can provide a complete view of the video popularity. Furthermore, this also means caching at the server side is more effective than caching at the edge/client side, if caching at the server side is needed. Note for content delivery, caching at the server side is typically not for reducing network traffic as caching at the client side.

While Figure 2.10 shows short-term (one day) popularity distribution, Figure 2.11 shows the corresponding distribution in long-term, spanning over the entire 30 days of our workload. In this figure, the left \( y \)-axis is in powered scale while the right \( y \)-axis is in log scale. The \( x \)-axis is in log scale as well. As shown in the figure, the monthly video popularity deviates from a straight line in log-log scale, meaning not a Zipf-like distribution. Instead, it can roughly be fit with a stretched exponential (SE) distribution as shown by the left \( y \)-axis in powered (by a constant \( c \)) scale [36]. With SE distribution, the rank distribution
function can be expressed as

\[ y_i^c = -a \log i + b. \]

An SE distribution is fit by several parameters as shown in the Figure. For example, parameter \( c \) is also called the *stretch factor*, which characterizes the median file size of workload [36]. It was reported that for media workloads with a median file size \(< 5\, \text{MBytes}\), the stretch factor is \( \leq 0.2 \). Our analysis confirms this with a \( c \) of 0.065 and a median file size of 1.68 MBytes. Parameter \( a \) in an SE distribution increases with the duration of workload as well as the ratio of media request rate to new content birth rate, and it causes the distribution to deviate from a straight line in log-log scale.

The stretched exponential distribution has been used to characterize many natural and economic phenomena, as well as the access patterns of Internet media traffic [36]. It was shown that under an SE distribution, media caching is much less efficient than under a Zipf distribution. This poses a new challenge if long-term caching is needed on the server side.
2.4.3 Popularity of Different Video Versions

We have shown in Figure 2.10 that the daily video popularity follows a Zipf-like distribution. However, as discussed before, each video may be accessed by very diverse mobile devices, resulting in multiple transcoded versions. We thus further examine the popularity distribution of all versions accessed on Nov. 1st 2010 where each version is counted as a distinct object. Figure 2.12 shows that when different versions are considered as different objects, the popularity cannot be well-fitted with the Zipf distribution. On the one hand, due to the increased number of video versions (2.31 versions per video on average), the skewness factor $\alpha$ decreases from around 0.95 to 0.7. On the other hand, as shown in the figure, accesses to the Top-1000 versions are much larger than what Zipf predicts, indicating significant deviation from Zipf-like distribution.

Figure 2.13 further depicts the monthly version popularity. This figure shows that although the daily version popularity cannot fit well with either Zipf or SE distributions, the monthly version popularity pattern can be well fitted with a SE distribution. The goodness value of the fit is 0.998532, very close to 1.
Figure 2.13: Monthly Version Popularity Distribution

The version-based popularity patterns we have examined above indicate that different from video based popularity, accesses to different video versions are more distributed. This means poorer caching performance if caching is needed for these versions.

2.4.4 Popularity Evolution

We have shown that mobile users’ daily accesses for mobile videos are highly concentrated, but monthly accesses patterns are flatter. In this subsection, we further examine how video popularity changes over time, aiming to shed light on such popularity changes. We first study the commons between accesses in consecutive days, and then, more specifically, we consider the temporal locality characteristics of these accesses.

Table 2.6 summarizes the daily accesses and the corresponding videos that were requested in the system. According to our analysis, 292K out of 502K (58%) unique video clips accessed daily are not accessed in the previous day on average. Among these 292K video clips, about 92K are new video clips. This indicates that about 18% video clips accessed every day are new. The rest 200K (40%) are unpopular ones that were in the system,
but were not frequently accessed. In total, new video clips account for about 68% of total unique video clips accessed during 30 days. Since new video clips are generated at a high rate of 18%, this confirms the implication of SE-distribution that a monthly static caching scheme may not be so efficient as a more frequently updated one.

Unlike traditional video on-demand streaming systems, Vuclip has a larger repository as well as a faster new content generation rate. We next examine if temporal locality is helpful in predicting what will be popular in the future in such a highly dynamic system.

First, we examine the temporal locality at the server side. We have shown in Table 2.6 that about 18% new video clips are added into the video repository daily. We further examine if top accessed videos change at a similar rate. Figure 2.14 shows the percentages of change in Top-100, Top-500, Top-1000, Top-2000, Top-5000 and Top-10000 accessed videos every day in 30 days. The percentages of change in Top-500, Top-1000, Top-2000 and Top-5000 requested video clips are all about 20%, which is similar to the birth rate of new video clips everyday. The Top-100 videos change at a higher rate, fluctuating between 20% and 35%. Previous studies on a video-on-demand (VoD) system report the daily rate of change in Top-100 videos is less than 15% [33]. Compared to the traditional VoD system, the result here indicates that mobile users tend to shift their interest faster. The change rate of Top-10000 videos is relatively stable at about 25%, higher than that of Top-500, Top-1000, Top-2000, and Top-5000 videos. This is likely due to the fact that the Top-10000

Table 2.6: Summary of Video Accesses

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Daily New Videos</td>
<td>92 K</td>
</tr>
<tr>
<td>Average Daily Accesses Videos</td>
<td>502 K</td>
</tr>
<tr>
<td>Percentage of Daily New Videos</td>
<td>18%</td>
</tr>
<tr>
<td>Average Daily Accessed Videos Difference</td>
<td>292 K</td>
</tr>
<tr>
<td>Percentage of Average Daily Difference</td>
<td>58%</td>
</tr>
<tr>
<td>Average Daily Requested Sessions</td>
<td>3512 K</td>
</tr>
<tr>
<td>Total Accessed Videos in 30 Days</td>
<td>4052 K</td>
</tr>
<tr>
<td>Percentage of New Videos in 30 Days</td>
<td>68%</td>
</tr>
</tbody>
</table>
list also includes videos that are less popular, e.g., new videos that were only active for a short period of time (one-day), and the 40% old videos that were not accessed in the previous day.

Figure 2.15 keeps track of the top videos on the first day (Nov. 1st, 2010) of our workload, and examines how many of them would remain on the top list for the next 29 days till Nov. 30th, 2010. We notice that during the first 5 days, a great percentage of top video clips are purged out of the top list, indicating quick user interest shift. However, such a change becomes relatively stable beyond the first 5 days, indicating videos that were popular during the past $N$ days are likely to be popular in the future. Nevertheless, despite such a quick change of user interests in a short period, video popularity is still relatively stable in the long term. For example, although only about 10% video clips remain in Top-100 after 29 days, this percentage increases to 30% for Top-500, 40% for Top-1000, 50% for Top-2000 and about 60% for Top-5000 and Top-10000, which indicates that with a large cache size, the cached video clips can be flushed less frequently.
2.4.5 Correlation Between Popularity and Video Length

Figure 2.15: % of Videos Remaining in Top

Figure 2.16: % of Sessions for Videos of Different Length (minute)
We have shown in section 2.4.1 that the mobile video files are often short and the video popularity in a short-term has a Zipf-like distribution. Thus one may wonder whether shorter videos get more accesses. To examine this, we group video files into 1 minute interval based on their lengths. Figure 2.16 shows the total number of requested sessions decreases as the video length increases, and videos shorter than 5 minutes account for about 70% total requests. We further calculate the correlation between video popularity and video length, and examine if shorter videos tend to be more popular. Our results show that the correlation coefficient is 0.006, which indicates the correlation is weak. We have also conducted similar tests based on versions, and the results are similar. This indicates the large percentage of requests for short videos are due to the large population of short video contents instead of user preferences.

### 2.5 Trade-off between CPU and Storage

As aforementioned, in order to conduct on-demand transcoding to serve all kinds of heterogeneous mobile devices, Vuclip has to employ a lot of computer clusters. This challenges service providers both technically and economically with the growing popularity of Internet mobile streaming services and it is thus very desirable to reduce the huge demand of CPU cycles for such transcoding.

In the previous section, we have shown that the mobile users’ accesses are more concentrated (skewed) than those in the traditional Internet streaming services. Thus, caching on the server side, sometimes called reverse caching, could be explored to temporarily cache some transcoded objects so that on-the-fly transcoding would not be necessary if the same type of mobile devices access the same video. Such full-object caching is possible for mobile videos because mobile video objects are typically smaller with a median of 1.68 MBytes as we showed before. Note that different from the traditional caching objectives such as web proxy caching for reducing the network traffic, caching at the server side here is to reduce the CPU cycles demanded for transcoding. That is, a trade-off between the storage and the CPU.
On the other hand, we have also shown that mobile users’ interests shift quickly. This provides some hints for cache replacement. Regardless whether the cache is implemented via disk and/or memory, the cache replacement policy is the key to the cache performance. Typical cache replacement strategies (such as popularity-based policies) may work, however, the complexity added by Vuclip-like services comes from the fact that a video often has multiple transcoded versions. Intuitively, these different versions could be considered as separate objects in the cache. However, if we consider video popularity, different versions of the same video are internally related. Therefore, we next explore different replacement strategies via simulations for Vuclip-like systems.

A simple strategy is to ignore the internal relationship of different versions of a video, and consider each version as a distinct object. Under this assumption, the existing web proxy cache replacement policies can be adopted. Since we are dealing with video objects, we thus first consider a version-popularity based replacement policy, in which a utility function is defined as the ratio of the version access number to the storage size occupied by that version. The version with the least utility is the victim to be purged from the cache.

On the contrary, if we consider that different versions of a video are related because along the diminishing popularity of a video, all of its versions may get fewer and fewer accesses, we can also consider a policy in which all different versions of a video are bundled together as one object in the cache. Taking the popularity of this object as the sum of popularity of all its versions, we can design a video-popularity based replacement policy, in which the entire object (with all versions) is replaced once it is identified as the victim based on the utility function defined as the ratio of the total access number to this video to the total size of occupied storage.

With the above two strategies, naturally, one may wonder if a hybrid policy could perform better. That is, neither considering the version independently as the version-popularity based policy does, nor considering all versions of a video as one object as the video-popularity based policy does. In a hybrid strategy, a utility function can be defined for each video as in the video-popularity based replacement policy, but the victim is the
least popular version of the least popular video.

To study the effectiveness of these different strategies, we conduct trace-driven simulations using the collected workload to compare their performance. With the accesses of Nov. 1st, Figure 2.17(a) shows that when the cache size is smaller than 27% of the total size of accessed objects of that day, a video-popularity based replacement policy can achieve the highest cache hit rate, and save roughly about 55% CPU cycles, while a version-popularity based strategy performs the worst. This is likely due to the highly concentrated video access
pattern as shown in Figure 2.10 compared to less concentrated version access pattern shown in Figure 2.12. The hybrid strategy performs consistently worse than the video-popularity based one, but still a little better than the version-popularity based policy. These results are consistent with our analysis on the video and version popularity. When the cache size increases, the version-popularity based policy has more flexibility in choosing the best version to cache, and thus achieves the best cache hit ratio among the three.

Figure 2.17(b) further shows the results when one-week trace from Nov. 1st to Nov. 8th 2010 was simulated. The cache size percentage used in this simulation is based on total accessed objects of Nov. 1st. We find that the cache performance over one week can reach as high as that of a day. However, the cache performance over a month as shown in Figure 2.17(c) is worse. This is because with a SE distribution for monthly video accesses, caching is much less efficient than with a Zipf distribution for daily accesses.

2.6 Related Work

The Internet has witnessed the sharp increase of Internet video traffic in the recent years with all kinds of Internet streaming systems, such as VoD and Internet P2P-based streaming systems. Lots of research has been conducted to study these Internet streaming systems. For example, Yu et al. [33] examined server logs of a traditional VoD system with a total of over 6700 unique videos, and analyzed user access patterns, session length, and video popularity. Yin et al. presented the access logs of a live VoD system [37], which shows different user and content properties compared to [33]. Krishnappa et al. collected Hulu traffic at a campus edge network, and examined the potential benefits of performing caching and prefetching at edge networks [38]. For P2P-based streaming systems, Wu et al. investigated P2P streaming topologies in UUSee [39]. Huang et al. conducted a large scale measurement to study the PPLive-based on-demand streaming [40].

Along the increasing popularity of user-generated content (UGC), studies have also been conducted to characterize UGC videos. For example, Cha et al. studied user behaviors and
video popularity of YouTube, and compared them with non-UGC content from Netflix [41]. Work [34] examined the traffic characteristics of YouTube at a campus edge network.

With the rapid increase of Internet-capable mobile devices in recent years, mobile Internet video services and accesses are surging. A few studies have been conducted to investigate the performance of mobile streaming applications, mostly focusing on the resource utilization for receiving streaming data on mobile devices. For example, Xiao et al. studied the power consumption of mobile YouTube [27]. Finamore et al. collected traffic from several edge locations and studied the potential reasons for the inferior streaming experience of mobile YouTube users [29]. Previously, we have also conducted measurements to study the resource utilization of different streaming approaches to mobile devices [14].

However, to the best of our knowledge, no prior work has examined Internet mobile streaming services from the server side, which is the key to provide Internet mobile streaming services. In this chapter, we have investigated the commons and differences of mobile Internet streaming services with/from the traditional Internet streaming services. Our study reveals a critical challenge in Internet mobile streaming services is the hardware and software heterogeneity of mobile devices. Our analysis also shows different access patterns of mobile videos from traditional Internet streaming videos. Furthermore, we have also shown that caching at the server side with a proper replacement policy can significantly reduce the resource consumption for Vuclip-like Internet streaming systems in dealing with heterogeneity.

2.7 Summary

The wide adoption of mobile devices in practice has made pervasive Internet streaming possible. While a number of studies have been conducted to examine the streaming services from the client’s perspective, in this chapter, we have studied the Internet mobile streaming services from the server side via one-month server log collected from one of the largest Internet mobile streaming service providers. Through detailed analysis, we have shown the great hardware and software heterogeneity of mobile devices, different characteristics of
mobile videos, and different user access patterns from those in traditional Internet streaming services. As a great challenge that Vuclip-like system faces is the huge demand of CPU resources for online transcoding to deal with heterogeneity, we show that caching at the server side with a proper replacement policy can effectively trade-off limited storage size for great savings on CPU cycles. These results provide some basic guidelines for building and optimizing future Internet mobile streaming systems.
Chapter 3: Towards Power-Efficient Internet P2P Streaming to Mobile Devices

3.1 Introduction

As aforementioned, P2P is a popular technique for delivering Internet streaming services. Today, PPTV [21], PPStream [22], SopCast [23], and QQLive [24] are all Internet-scale P2P streaming systems that provide hundreds of TV channels and serve millions of Internet users every day. CNN has also started to use P2P technology provided by Octoshape to deliver high quality live video to its users [42]. According to Cisco’s report [43], today, global P2P TV systems have generated over 280 petabytes (and 6% of) Internet video traffic per month, and is growing at a rate of 47% annually.

However, using mobile devices to receive Internet streaming services can quickly exhaust the limited battery supply because receiving Internet streaming services often involve continuous and bulk data transmissions, decoding, rendering, and displaying. Among them, wireless data transmissions consume a significant amount of energy. For example, it was reported that the WiFi radio on HTC Tilt 8900 series consumes up to 5x more power than the base energy consumption of the phone [9]. A lot of efforts have been made to save battery power consumed by the wireless network interface cards (WNICs) by exploiting sleep opportunities via proxying [11], batching [44], PSM-throttling [10], etc. However, existing power saving strategies mainly focus on client/server (C/S) based streaming applications. Compared to C/S based streaming, Internet P2P streaming can further aggravate the battery power consumption on mobile devices due to the following unique characteristics.

First, different from C/S based streaming, a client participating in P2P streaming needs to continuously exchange extra control traffic, such as buffermaps and fine-grained data
chunk requests, with neighbors. Such control traffic, on the one hand, greatly facilitates the streaming content distribution in a scalable manner. On the other hand, it also remarkably increases the total number of IP packets transmitted and reduces the inter-packet delays, which further reduces the sleep opportunities of the WNIC.

Second, critical to the scalability of a P2P system, a client participating in P2P streaming has to upload the downloaded content to other neighboring peers. Such uploading requirement does not exist in C/S based streaming. The uploading traffic increases the total traffic volume for a P2P client, resulting in more battery power consumption.

Third, a client participating in P2P streaming receives streaming content from multiple sources instead of a single server. Furthermore, these streaming sources are changing dynamically along time since a P2P client often needs to frequently connect to new neighbors to download missing file chunks. In addition, the dynamics of the streaming sources result in different round-trip times to the client, making data receiving at the client side totally random, and making it difficult for the WNIC to switch into the power saving mode (PSM).

In this chapter, we first conduct Internet measurements to study in-depth the impact of Internet P2P streaming on battery consumption on mobile devices with several popular Internet P2P streaming applications. Our measurement results show that: (1) a client in P2P streaming needs to transmit an extremely large number of small packets for control traffic (e.g., twice as many as streaming data packets). Such a large number of frequent and delay-sensitive control packets not only increase the traffic volume, but also significantly reduce the inter-packet delay and the potential sleep time of the WNIC on the mobile device; (2) the amount of uploading traffic a mobile client needs to transmit to its neighbors changes from time to time and from application to application, ranging from 10 Kbps to over 1.5 Mbps. In addition to increasing the transmission load on the WNIC, such highly un-predictable uploading further reduces the inter-packet delay and the sleep opportunities of the WNIC for power saving; and (3) a client in P2P streaming receives packets from dynamically changing sources (e.g., 3 to 20 peers), making it difficult to employ server or client-centric traffic shaping techniques to save power consumption on mobile devices.
Motivated by the measurement results, we seek to address these challenges for power-efficient Internet mobile P2P streaming services. Accordingly, we propose and design the BlueStreaming system by leveraging the commonly existing Bluetooth interface on mobile devices. In BlueStreaming, instead of having WiFi and Bluetooth interfaces to work alternately, we always keep Bluetooth active to transmit delay-sensitive control traffic. With intelligent traffic shaping techniques applied on the WiFi interface for streaming data traffic, the extra battery power consumed by the Bluetooth interface can be over-compensated by the greater power saving on the WiFi interface. Furthermore, the uploading traffic from the client is opportunistically scheduled on both interfaces to minimize battery power consumption incurred.

To evaluate the performance of BlueStreaming, we have implemented prototypes on both Windows and Mac OS to participate in several popular Internet P2P streaming services. Our experimental results show that BlueStreaming can effectively reduce battery power consumption by up to 46% without affecting streaming quality. In summary, this chapter makes the following contributions:

- Through Internet measurement and analysis, we show that when a mobile device participates in Internet P2P streaming services, its battery power consumption is highly affected by its control traffic, uploading traffic, and the dynamics of neighboring peers.

- We design and implement BlueStreaming, a power-efficient Internet P2P streaming system that takes into account the unique characteristics of Internet P2P streaming and trades Bluetooth's power consumption for greater power saving on the WiFi interface via intelligent traffic shaping.

- We evaluate our BlueStreaming prototype with popular Internet P2P streaming applications in both infrastructure and hybrid modes, and show that BlueStreaming can improve energy saving by up to 46% compared to the commodity PSM scheme.

Contributions described in this chapter were published in [45]. The remainder of the
chapter is organized as follows. Section 3.2 describes some related work. We present our measurement results in section 3.3 and the design of BlueStreaming in section 3.4. The implementation is described in section 3.5 and the evaluation is discussed in section 3.6. We summarize this chapter in section 3.7.

3.2 Background and Related Work

With pervasive wireless Internet accesses, power saving has been considered on the commodity WNIC. Today most WNIC can operate in two power modes: Constantly Awake Mode (CAM) and PSM. It has been reported in [9] that a mobile smartphone in CAM uses as much as 1120 mW power, while it consumes much less power of 72 mW in PSM. In practice, commodity WiFi devices often use adaptive PSM (PSM-A) instead of static PSM. PSM-A switches between CAM and PSM modes. If there is no network activity (idle) for a pre-determined time period, called *idle timeout interval*, the WiFi interface would switch to PSM by sending a NULL data frame with the *Pwr Mgt* field set to 1. It only wakes up to check the Traffic Indication Map (TIM) per beacon interval (e.g., 100 ms). The WiFi interface can notify the access point (AP) that it is ready to receive buffered data by sending another NULL data frame with the *Pwr Mgt* field set to 0. To further improve the energy efficiency, a lot of prior work has sought to exploit idle opportunities for the WNIC. For example, µPM [46] allows the WiFi interface to exploit short intervals in terms of microseconds between MAC frames to enter the low power mode. Catnap [44], on the other hand, exploits large sleep intervals by scheduling TCP transfers at the granularity of application data units with the help of workload hints.

Meanwhile, since mobile devices are commonly equipped with multiple network interfaces that have different power consumption rates, research has been conducted on alternatively activating different interfaces for battery power saving, focusing on substituting a high-power consumption interface with a low-power consumption interface whenever possible. For example, Agarwal et al. proposed to wake up the WiFi interface for incoming
calls using GSM radio [47]. In CoolSpots [48], policies were proposed to intelligently switch between the Bluetooth and WiFi interfaces. However, CoolSpots is not appropriate for today’s Internet streaming delivery to mobile devices because Bluetooth alone cannot afford the constant bandwidth demanded by modern streaming applications (see section 3.4.1). In contrast to CoolSpots that uses the two interfaces alternatively, BlueStreaming uses both WiFi and Bluetooth interfaces simultaneously, and relies on the greater power saving on the WiFi interface to overcompensate the extra power consumption incurred by the Bluetooth interface.

Focusing on Internet streaming to mobile devices, Bertozzi et al. proposed to switch off the WiFi interface during playback and turn it on when running out of streaming buffer [49]. Linear prediction was proposed to select sleep time intervals for the WNIC [50]. PSM-throttling [10] utilizes client-side reshaping of TCP traffic to improve energy saving. Compared to existing work, this study shows that a client in P2P streaming not only generates extra uploading traffic, but also transmits highly frequent and delay-sensitive control traffic, making direct adoption of existing solutions ineffective.

3.3 Measurement and Analysis

To investigate the battery power consumption on mobile devices in popular Internet P2P streaming applications, in this section, we conduct measurements with several existing Internet P2P streaming applications. We then analyze the traffic in-depth in order to study underlying factors that impact the battery power consumption.

3.3.1 Methodology and Result Overview

To study the battery power consumption on mobile devices in receiving P2P streaming services, we first use a 2nd generation iPod Touch (iTouch) running iOS 3.1.2 to receive streaming services. We conduct experiments with TVUPlayer [51], and Justin.tv [52]. To the best of our knowledge, TVUPlayer is the only P2P based streaming application available
on iTouch at the time of this measurement, while Justin.tv is a C/S based Internet streaming service.

In these experiments, we set up Wireshark [53] to listen on the same channel to capture all incoming/outgoing packets. Since we log packets at the data-link layer, we are able to get frame control information from IEEE 802.11 headers. We use Pwr Mgt flag in the frame control field to determine the Power Management mode that the WiFi interface will switch to after the transmission of the current frame. All experiments last 30 minutes and are conducted with a dedicated AP running 802.11g in order to minimize traffic contention.

We set both TVUPlayer and Justin.tv to watch channels with the same streaming rate (281 Kbps), and examine the potential power consumed by the WiFi interface. Our results show that during 30-minute tests, C/S based Justin.tv allows the WNIC to sleep for 83% time, while in P2P based TVUPlayer, the WNIC can only sleep for 26% time, meaning 3 times more power consumption in receiving P2P streaming services. Given that iOS uses an extremely aggressive PSM-A policy (idle timeout interval is only about 20-25 ms [9]), the energy performance of TVUPlayer on other mobile devices could be even worse.

Although the above results provide an overview of the power in-efficiency P2P streaming services for mobile devices, TVUPlayer is not as widely used as other P2P based Internet streaming services such as PPTV (PPLive is renamed as PPTV), PPStream (PPS), SopCast and QQLive, each of which serves millions of Internet users daily. To study in-depth the battery power consumption of mobile devices in these applications, we further conduct experiments with a laptop computer since these applications do not have a version for mobile devices. We again compare their results against C/S based Justin.tv (J.tv) [54]. In these experiments, the laptop computer connects to a dedicated AP running 802.11n and watches different channels of these popular applications. The laptop runs Windows 7 with Maximum WiFi Power Saving enabled. Again, Wireshark is used to capture all incoming and outgoing packets to/from the laptop at the data-link layer. Because each peer in P2P streaming is required to upload to its neighbors and such uploading throughput can be highly unpredictable (we will show later in section 3.3.3), we thus conduct experiments at various
Table 3.1: Summary of Statistics on Laptop

<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>Encoding Rate (Kbps)</th>
<th>Total # of IP Packets</th>
<th>Total # of Control Packets</th>
<th>Total # of Streaming Packets</th>
<th>Sleep Time (%)</th>
<th>Average # of Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPTV</td>
<td>P2P</td>
<td>400</td>
<td>191716</td>
<td>116819</td>
<td>70873</td>
<td>4.42</td>
<td>12</td>
</tr>
<tr>
<td>PPS</td>
<td>P2P</td>
<td>396</td>
<td>433935</td>
<td>322779</td>
<td>85180</td>
<td>0.09</td>
<td>20</td>
</tr>
<tr>
<td>SopCast</td>
<td>P2P</td>
<td>530</td>
<td>305826</td>
<td>202240</td>
<td>92457</td>
<td>0.99</td>
<td>3</td>
</tr>
<tr>
<td>QQLive</td>
<td>P2P</td>
<td>500</td>
<td>293474</td>
<td>183814</td>
<td>108827</td>
<td>7.12</td>
<td>4</td>
</tr>
<tr>
<td>J.tv</td>
<td>C/S</td>
<td>433</td>
<td>123655</td>
<td>56689</td>
<td>66966</td>
<td>21.44</td>
<td>N/A</td>
</tr>
</tbody>
</table>

times of a day and first present the results of tests that involve the least amount of uploading traffic. This makes the comparison against C/S based J.tv more meaningful because otherwise the additional high uploading from a mobile device undoubtedly increases the battery power consumption. Note that these tests are not necessarily accessing unpopular channels, since uploading of a peer is also affected by a number of other factors.

Table 3.1 summarizes some raw data based on the traces we have collected in the experiments. Note that Total # of Streaming Packets refers to the downloaded streaming data packets only. Although the video encoding rates are different across four P2P applications, it is not necessarily correlated to the amount of time that the WiFi interface has spent in the PSM: both PPTV and PPS stream at similar rates, but they have different power efficiency; SopCast allows the WiFi interface to sleep for less than 1% of the session time with a streaming rate of 530 Kbps, but at a similar rate of 500 Kbps in QQLive, the WiFi interface operates in the PSM for 7% of the session time, and is the most power efficient.

Nevertheless, compared to C/S based J.tv in which the WiFi operates in the PSM for 21% of the session time, none of these four popular P2P streaming applications is power efficient. Note that all above experiments involve a small amount of uploading traffic. If the laptop has to upload more to neighboring peers, the power efficiency of P2P streaming applications would be even worse (we will show soon in section 3.3.3).

While the sleep time varies in different P2P streaming applications, above results on
both the mobile device and the laptop show that compared to C/S based streaming, a mobile client in existing P2P streaming applications is much less power efficient. As aforementioned, this is likely due to the uploading and control traffic. To quantify their impact, next we further analyze the incoming and outgoing traffic to/from our testing client.

3.3.2 Impact of Control Traffic

Figure 3.1: PPTV (400kbps) 30-min Test

Figures 3.1, 3.2, 3.3, and 3.4 show more detailed traffic analysis in the above experiments.
for PPTV, PPS, SopCast, and QQLive, respectively. Figure 3.1(a) shows the packet size distribution of PPTV. As indicated by this figure, the packet size distribution is highly bi-modal. Via reverse-engineering, we find that only about 37% packets are streaming packets with the size around 1408 bytes while about 60% packets have a size less than 300 bytes. Similarly, Figures 3.2(a), 3.3(a), and 3.4(a) show that there are more than 74%, 66%, and 62% small packets in the streaming sessions of PPS, SopCast, and QQLive, respectively.

Recall that existing P2P streaming systems mainly use a pull based mesh structure, in which neighboring peers need to frequently exchange control information. For example,
peers exchange buffermaps to determine data chunk availability among neighbors. Peers also need to send out file chunk requests to retrieve missing streaming data chunks. In addition, PPTV, PPS, Sopcast, and QQLive all use UDP to transmit packets. Thus, these small packets can only be control traffic of the P2P protocol. Via reverse-engineering, it is confirmed that these packets include both buffermaps exchanged with neighbors and file chunk requests sent to neighbors.

Table 3.1 shows that in all these P2P streaming applications, there are more control
packets (the 5th column) than streaming data packets (the 6th column). PPS even generated about 3 times more control packets than data packets. Regardless of the total traffic amount of these control packets, such a large number of control packets can significantly reduce the inter-packet delay and thus idle time of the WiFi interface. This would result in more battery power consumption. Figures 3.1(b), 3.2(b), 3.3(b), and 3.4(b) show the inter packet delay distribution in these experiments. In these figures, Inter Streaming Packet delay considers only ingress streaming data packets. Inter Control Packet delay considers both incoming and outgoing control packets, and Inter Packet delay takes all
packets transmitted into account. As shown in these figures, while 10% and 3% successive streaming packets of PPTV and PPS arrive with a delay larger than 100 ms (meaning opportunities for power saving), about 60% control packets in PPTV and 42% in PPS are sent/received within 1 ms from the preceding one. This causes the overall Inter Packet delay to significantly deviate from Inter Streaming Packet delay patterns.

Although control packets are much more frequent than streaming data packets, the throughput of control traffic is relatively small. As shown in Figures 3.1(c), 3.2(c), 3.3(c), and 3.4(c), the throughput of control traffic of the four P2P streaming applications is less than 100 Kbps for most of the time. In these figures, the throughput is averaged over 10 seconds. The relatively small control traffic throughput is mainly due to the small size of control packets. This is a very important feature that we could explore to save power consumption in BlueStreaming in section 3.4. Although control traffic may increase with the increase of uploading, typically the control traffic throughput is still much smaller than the streaming data rate.

3.3.3 Impact of Uploading Traffic

The measurement results we have presented above involve a minimum amount of uploading traffic. In practice, a P2P client may upload more and the uploading throughput may also change from time to time. Figure 3.5 shows the uploading variations of our client with a minimum uploading average and a maximum uploading average in our over fifty 30-minute tests at various times. As shown in this figure, the throughput of uploading traffic could reach as high as over 1 Mbps in PPTV, PPS, and SopCast. Such a high uploading throughput clearly consumes more battery power. As shown in Table 3.2, with a larger amount of uploading traffic transmitted, the sleep time (%) of the WiFi interface in all four applications decreases significantly, and in PPS it even could not switch into the PSM at all.
3.3.4 Impact of the Number of Neighbors

Both control traffic and uploading traffic are affected by the number of neighbors a peer communicates with at a time. A larger number of neighbors not only increase the control information exchanged with different peers and the uploading traffic amount, but also divide large idle time intervals into smaller ones because the response time of different peers to our client varies significantly. During the experiments, we have kept track of the number of peers that our streaming client downloaded data from. As shown in the last column of Table 3.1, during the 30-minute test, our client in QQLive exchanges streaming data with
Table 3.2: Comparison of Sleep Time (%)

<table>
<thead>
<tr>
<th></th>
<th>PPTV</th>
<th>PPS</th>
<th>SopCast</th>
<th>QQLive</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>4.42</td>
<td>0.09</td>
<td>0.99</td>
<td>7.12</td>
</tr>
<tr>
<td>MIN</td>
<td>0.08</td>
<td>0.00</td>
<td>0.22</td>
<td>4.33</td>
</tr>
</tbody>
</table>

4 neighboring peers on average every 10 seconds, while it involves 12 neighbors on average in PPTV.

3.3.5 Summary of Measurement Findings

Our study shows that in Internet P2P streaming (1) although of small throughput, the control traffic is highly frequent, which remarkably reduces the potential sleep time of the WiFi interface; (2) the uploading required from a peer changes dynamically and could reach a very high throughput, which directly affects the battery power consumption and worsens the client side traffic pattern; (3) the dynamics of peers’ neighbors directly affect both control and uploading traffic, and the variance of response time among neighbors further shortens inter-packet delay.

3.4 Design of BlueStreaming

As discussed in the last section, P2P streaming differs from traditional client-server based streaming on the additional control traffic, uploading traffic, and data receiving from multiple peers. These can lead to excessive battery power consumption on mobile devices.

3.4.1 Motivation

To save power consumed for data transmission, an intuitive solution is to always switch to a low power interface, e.g., Bluetooth connection when available. Another solution is to use WiFi interface only, but aggressively shapes the traffic by brutally buffering all packets for
long enough and transmit them in a very large MAC frame periodically.

<table>
<thead>
<tr>
<th>Name</th>
<th>Total # of Streaming</th>
<th>Sleep Time (%)</th>
<th>Distortion/Freeze Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM-A</td>
<td>109800</td>
<td>5.24</td>
<td>0</td>
</tr>
<tr>
<td>Bluetooth only</td>
<td>37228</td>
<td>N/A</td>
<td>96</td>
</tr>
<tr>
<td>WiFi with traffic shaping</td>
<td>121598</td>
<td>26.89</td>
<td>38</td>
</tr>
</tbody>
</table>

We implement both of these schemes. Table 3.3 shows the results when accessing a QQLive channel of 500 Kbps for 30 minutes. Bluetooth only saves energy consumption by only using Bluetooth for all transmissions. However, Bluetooth alone cannot afford the constant bandwidth demanded by the application. As a result, Table 3.3 shows that the client with Bluetooth only receives about only 34% streaming data packets compared to PSM-A. This causes the playback to be frozen all the time during the entire streaming session.

On the other hand, in WiFi with traffic shaping, the control packets are also buffered and sent in burst periodically via WiFi. Table 3.3 shows that such an approach can make the WiFi interface sleep for 26% of the session time. However, the delayed control packets have caused the client to receive 10% more streaming packets, indicating the delay-sensitivity of control traffic. Moreover, by analyzing the captured frames of the playback and comparing with those captured in PSM-A (since we do not have access to the original video), we find that quality degradation (playback freezing or distortion) happens for a total of 684 seconds, which is 38% of the entire streaming session.

The above results demonstrate that these intuitive strategies using either interface may cause problems in both user perceived quality and the efficiency of transmission. We next present our design of BlueStreaming to intelligently utilize both WiFi and commonly existing Bluetooth interfaces on mobile devices for Internet P2P streaming accesses.
3.4.2 BlueStreaming Overview

To efficiently deal with control traffic in P2P streaming, BlueStreaming leverages the Bluetooth interface on a mobile device to transmit control traffic. For this purpose, BlueStreaming always classifies the incoming and outgoing traffic to/from a client into two classes: ingress/egress control traffic and ingress/egress streaming data traffic. BlueStreaming uses Bluetooth to transmit both ingress and egress control traffic. In addition, since uploading is inevitable and Bluetooth is always active in BlueStreaming, Bluetooth is also used to upload to neighbors whenever there is available bandwidth. The WiFi interface is mainly used for streaming data downloading, as well as uploading opportunistically upon excessive uploading.

Figure 3.6: Overview of BlueStreaming Architecture

Figure 3.6 depicts the architecture of BlueStreaming. Logically, BlueStreaming consists of three main components: the traffic classifier at both the AP and the client sides to decouple control traffic from streaming data traffic, the traffic shaper at the client side for shaping data traffic to save battery power, and the uploading scheduler at the client side to handle uploading traffic. The successful design of these three components, however, needs to address the following challenges:

- **traffic classifier**: given that in most P2P streaming, control traffic and streaming data traffic use the same channel (port number), how can control traffic be decoupled from streaming data traffic transparently (transparent to the application)?
• **traffic shaper**: since most existing P2P streaming traffic is delivered over UDP and a client often dynamically communicates with a set of different peers at a time, how shall BlueStreaming shape the downstream traffic to a client?

• **uploading scheduler**: because uploading from a peer is necessary and critical to the scalability of a P2P system, how does a client perform uploading while minimizing the corresponding battery power consumption?

### 3.4.3 Traffic Classifier: Diverting Control Traffic to Bluetooth

Since Bluetooth operates with about 20% power compared to WiFi in an active state, with BlueStreaming, a mobile device seeks to use Bluetooth to transmit the large number of small delay-sensitive control packets in order to put the WiFi interface into sleep mode for a longer time. This requires Bluetooth to provide sufficient bandwidth to transmit control traffic. Commodity Bluetooth interfaces today typically offer a low data rate of 1 Mbps or so. Although we have shown in section 3.3 that control traffic may fluctuate from time to time depending on a number of factors such as the number of neighbors, the control traffic rate is often very low. Using Bluetooth to transmit control traffic also needs to consider the time sensitivity of control packets. A previous study shows that the delay of the control traffic would cause repetitive requests of the same data chunks [55]. Thus, to ensure timely delivery of control packets, in BlueStreaming, the Bluetooth interface stays active all the time so that control packets are transmitted without any delay.

To transmit control packets via Bluetooth, control traffic must be separated from data traffic at runtime. In BlueStreaming, the traffic classifier is responsible for decoupling control packets from data packets. Since there are both incoming and outgoing control packets, the traffic classifier works at both the AP and the client sides. Because (1) the current design of BlueStreaming aims to be application transparent, (2) both control traffic and data traffic in typical P2P streaming services use a same port, and (3) the classification must be real time, BlueStreaming has to avoid deep packet inspection for any application specific information.
Thus, in BlueStreaming, we leverage a feature that we have found via our measurement for runtime packet decoupling. In section 3.3, Figures 3.1(a), 3.2(a), 3.3(a), and 3.4(a) show the packet size distribution. As shown in these figures, the control packets are of a much smaller size compared to that of streaming data packets. Thus, in BlueStreaming, we leverage the packet size to differentiate different types of packets. Although the classification is based on our observation in the measurements, we have further conducted measurements with these P2P streaming services for prolonged time in order to validate its accuracy.

Once the control traffic is decoupled, it is easy to divert to the Bluetooth interface. Upon packet arrival, merging received packets from different interfaces is straightforward since these P2P streaming applications use UDP.

**3.4.4 Traffic Shaper: Shaping Ingress Streaming Traffic**

After all control packets are diverted to the Bluetooth channel, only streaming data traffic is transmitted through WiFi. Thus, more energy could be saved by exploiting the idle time of the WiFi interface.

Figures 3.1(b), 3.2(b), 3.3(b), and 3.4(b) show the inter-packet delays. Although the Inter Packet delay distribution of WiFi (with control traffic diverted to Bluetooth) would be similar to the distribution of Inter Streaming Packet delay as shown in these figures, the increase of sleep opportunity due to control traffic elimination is still small. This is likely due to the multiple uploading peers instead of only a single server. Multiple uploading peers have different response time to this client, resulting in different ingress streaming packet arrival time as observed in our measurements.

Thus, after diverting control traffic, BlueStreaming still needs to handle incoming traffic from different neighboring peers. A natural scheme is to do traffic shaping at the AP. Since BlueStreaming mainly deals with UDP based streaming, the traditional TCP based traffic shaping in the client-server based architecture, such as PSM-throttling, does not work. In addition, due to the QoS requirements of Internet streaming services, such streaming packets cannot be buffered for too long.
To this end, in the traffic shaper, BlueStreaming utilizes the buffer maintained by the WiFi AP: the streaming packets arrived over time are combined into a single MAC frame burst. As a result, the interval between bursts is significantly enlarged and the WiFi interface can exploit these opportunities to switch into the PSM.

To minimize the change to AP, the traffic shaper in BlueStreaming works by not sending the NULL or PS-Poll frame even when the TIM indicates that there are buffered frames for the client. This forces the AP to continue to buffer frames destined to this client. A subtle issue here is to determine when to wake up and retrieve buffered data. In BlueStreaming, we set it based on: (1) streaming bitrate, (2) effective WiFi bandwidth, and (3) QoS requirements. The first two are easy to determine. The third one, however, is highly application dependent. On the one hand, it impacts the client perceived streaming quality directly, although the playout buffer can absorb most of such delay. On the other hand, P2P applications usually use a timer to determine if any data chunk needs to be re-requested. For certain chunks that are not received in time, they will either be forfeited if their deadline has already passed or they will be requested again [55]. Hence, if streaming packets are buffered at the AP for too long, the application would believe such packets are lost, and request them again.

On the other hand, if the WiFi interface keeps staying in the sleep mode without responding to AP’s beacon messages, the AP would believe that the client is no longer associated. After a pre-determined timeout interval, the AP would start dropping its buffered frames. Thus, how long the data packets should be buffered is critical, which directly determines the sleep duration of the WiFi interface.

Taking all the above considerations into account, we set the packet buffering duration as follows:

$$T_{buf} = \min\left( T_{re-req} \times \left(1 - \frac{Rate}{BW_{WiFi}}\right), T_{AP-timeout}\right)$$

where $T_{re-req}$ is the re-request timer of P2P applications, $Rate$ is the bitrate of the streaming channel, $BW_{WiFi}$ is the estimated bandwidth of the WiFi link, and $T_{AP-timeout}$ is the
timeout duration after that the AP starts to drop buffered data for the client.

### 3.4.5 Uploading Scheduler: Scheduling Upload Wisely

In P2P streaming, peers should upload data to neighbors. This is the key to the scalability of any P2P application. But such uploading brings new complexities. First, uploading downloaded data to other peers directly incurs the consumption of the uploading bandwidth and battery power. Second, uploading incurs a comparable number of control packets. Third, uploading is very dynamic and hard to predict in P2P streaming since it depends on how many peers the client is serving at a time. As a result, uploading may seriously offset our energy saving efforts.

Aiming to solve these issues efficiently so that the minimum battery power is consumed for such uploading, we design an uploading scheduler working at two levels: *Priority-based Bluetooth Uploading* and *Opportunistic WiFi Uploading*. In short, BlueStreaming tends to fully utilize the always-on Bluetooth first, and then opportunistically utilize WiFi for uploading if needed.

**Priority-based Bluetooth Uploading**

In the previous measurement, we have shown that control traffic, although resulting in a large number of packets, occupies a small portion in the traffic volume. Utilizing Bluetooth for control traffic transmission typically consumes less than 20% of the available Bluetooth bandwidth. This leaves a lot of Bluetooth bandwidth un-used because the Bluetooth interface in BlueStreaming is always active. Thus, when there are uploading packets pending, un-used Bluetooth bandwidth is firstly exploited to transmit the uploading traffic. Since Bluetooth is kept active to transmit frequent control traffic, diverting uploading packets to it does not incur much more power consumption.

While the above idea works intuitively, a design pitfall is the delay of control packets due to the streaming data uploading to neighbors. In order to ensure timely delivery of control packets, upstream data packets should not be diverted to Bluetooth when the bandwidth
utilization of Bluetooth reaches a threshold. For this purpose, different priorities should be assigned to control traffic and streaming data traffic. A higher priority should always be given to the control packets when streaming data packets also present.

**Opportunistic WiFi Uploading**

Since WiFi provides a much higher data rate compared to Bluetooth, when high uploading is demanded and cannot be completed by Bluetooth, BlueStreaming considers opportunistically using WiFi to deliver uploading traffic in bursts. For this purpose, similar to downstream/incoming packets, intermittent upstream/outgoing packets are combined and sent at a higher data rate by buffering outgoing streaming packets for a certain amount of time at the network layer at the client side.

The timing in the above process is critical. Although it is appealing to buffer the packets for long enough and then transmit them in a very large MAC frame burst, it would impact the performance of the requesting peer and may cause our BlueStreaming client to receive excessive repetitive requests. This can further increase the number of control packets and negatively impact the throughput of the Bluetooth channel. In order for WiFi to upload with a minimum consumption of extra battery power, the scheduler should work seamlessly with the PSM mechanism on the client.

Thus, to opportunistically utilize WiFi for uploading when needed, the uploading scheduler transmits uploading traffic in a burst before or right after “shaped” downstream traffic based on the PSM configuration.

- If the idle timeout interval used in the PSM is only based on outgoing traffic activities, BlueStreaming starts to upload while the WiFi interface is in the PSM. The WiFi interface will switch from the PSM to the CAM and the transmission is scheduled to finish right before the start of downstream packet transmissions. Note that the end-timing of uploading is not stringent, because the AP will not start to send buffered downstream packets until it receives a NULL wake-up or PS-Poll frame from the client. Under such a scheduling policy, the time spent for downloading streaming
packets would be counted toward the idle timeout interval.

- If the idle timeout interval for the PSM is based on both incoming and outgoing traffic activities, the scheduler can transmit the uploading via WiFi after being notified by the AP that it has no more buffered data packets.

In either case, the client can sleep for:

\[
T_{\text{sleep}} = \min \left( T_{\text{re-req}} \times \left( 1 - \frac{\text{Rate}}{\text{BW}_{\text{WiFi}}} \right), T_{\text{APtimeout}} \right) - \frac{\text{SIZE}_{\text{Up-Buf}}}{\text{BW}_{\text{WiFi}}} \tag{3.2}
\]

where \( \text{SIZE}_{\text{Up-Buf}} \) denotes the volume of uploading packets buffered in the outgoing buffer at the client side.

### 3.5 Implementation

To evaluate the performance of BlueStreaming, we have implemented prototype systems on both Mac OS and Windows. In this section, we discuss some implementation and deployment issues.

#### 3.5.1 Prototype Implementation

Our current prototypes run on laptops since laptops have implemented more complete Bluetooth profiles including Personal Area Network (PAN) as desired by BlueStreaming. BlueStreaming could be totally AP-transparent if control traffic can be separated at the transport layer. However, as we mentioned before, existing P2P streaming applications do not differentiate control traffic from data traffic at the transport layer. Therefore, we need a traffic classifier at both the AP and the client side. We prototyped BlueStreaming AP using a MacBook as the AP for both WiFi and Bluetooth. It shares its wired Ethernet connection via WiFi and Bluetooth interfaces. In a nutshell, the traffic classifier works
at the IP layer. It intercepts packets at the IP layer, modifies destination addresses, and re-injects the packets to the corresponding interfaces.

The traffic shaper utilizes the buffer maintained at the WiFi AP by allowing the client not to wake up even if it has buffered packets. For this purpose, we implement a buffer at the network layer of our AP, which buffers downstream streaming data packets and then forwards to the link layer in bursts periodically. The buffering time is set according to Equation (3.1).

The uploading scheduler is implemented at the client side. In our experiments, the idle timeout interval of PSM set by the device driver is solely based on outgoing network activities. Therefore, we schedule outgoing uploading traffic before the bursty downloading of incoming traffic. To implement this, we set up a client side buffer at the network layer to buffer its outgoing streaming data packets. It also logs the timestamp of the first arrived packet of the latest incoming burst via WiFi, and determines when to start transmitting bursty uploading traffic based on Equation (3.2).

### 3.5.2 Infrastructure vs. Hybrid Mode

BlueStreaming can be deployed in two modes in practice: (1) an Infrastructure mode where a dedicated AP provides both WiFi and Bluetooth connections, and (2) a Hybrid mode where an intermediate device is used.

A BlueStreaming client needs both WiFi and Bluetooth interfaces, so that both interfaces can be used in parallel. In an infrastructure mode, the AP also needs to have both interfaces so that a BlueStreaming client and the AP can direct different types of traffic to different subnets (e.g. wireless LAN and bluetooth PAN).

In practice, however, a Bluetooth-enabled AP is not so common compared to a WiFi AP. If WiFi AP does not support Bluetooth, we allow a BlueStreaming client to operate in a hybrid mode by utilizing other Bluetooth-enabled network devices nearby. The intermediate devices should have a Bluetooth interface as well as a connection to the AP (wired or wireless). In practice, the hybrid mode can be easily deployed by using a plug-and-play
Bluetooth dongle. In this mode, a BlueStreaming client can form an ad-hoc network with the intermediate device, and ask it to relay its traffic.

Another natural concern is the range mismatch problem of WiFi and Bluetooth. The expected communication range of WiFi is about 100 meters, while Bluetooth often covers only about 10 meters. In BlueStreaming, the hybrid mode can help deal with this issue. That is, with a relay in the middle, the communication distance of Bluetooth can double. Multi-hop relay could also be leveraged if necessary. The impact of relay in the hybrid mode is studied in section 3.6.3. If there is no relay possible at all, the user has to use WiFi as before.

On the other hand, the new Bluetooth standard may shed light on this range mismatch issue. For example, with the development of Bluetooth 4 [56], Bluetooth can improve its coverage to 50 meters or more. Better yet, with new low energy techniques, such an improved communication range may not necessarily increase the energy consumption.

3.5.3 Channel Selection and Competition Avoidance

Co-existence of Bluetooth and WiFi has attracted a lot attention during the past years, because Bluetooth and WiFi may operate in the same spectrum. When a Bluetooth transmission and a WiFi transmission happen at the same frequency, interferences could happen and packets may be lost.

Recent advances of wireless technology have made significant progress to address this problem. For example, hardware vendors today commonly use specific co-existence algorithms to handle Bluetooth and WiFi traffic intelligently when both interfaces are in operation. Operating systems such as Windows 7 also implement Adaptive Frequency Hopping (AFH) to minimize interferences. In our experiments, our laptop is equipped with the latest Bluetooth/WiFi combo chipset from Broadcom and runs Windows 7. We will study retransmission rates in section 3.6.4.
3.6 Performance Evaluation

3.6.1 Experimental Setup

To evaluate BlueStreaming, we run our Windows prototype to access PPTV, PPS, SopCast, and QQLive. A laptop (MacBook) is set as the BlueStreaming Access Point for both WiFi and Bluetooth, and WiFi runs 802.11n at 2.4GHz. A second laptop runs Windows 7 and acts as the BlueStreaming client to run the four applications. The third laptop associates with the AP and captures WiFi traffic in a promiscuous mode. In a hybrid mode, a fourth laptop is used to relay the control traffic between our testing client and the AP by associating with the AP via WiFi, while the AP turns off its Bluetooth interface and only serves as AP for WiFi. In all experiments, laptops are placed 3 meters from each other except that the traffic sniffer (the third laptop) and our testing client (the second laptop) are placed close to each other.

3.6.2 Energy Saving in the Infrastructure Mode

We first examine the effectiveness of BlueStreaming in infrastructure mode. For each of the four applications, three tests are conducted: (1) PSM-A (without BlueStreaming), (2) BlueStreaming with only the traffic classifier enabled, and (3) BlueStreaming with all three components enabled. Because there is always uploading traffic from our testing client, the traffic shaper has to work with the uploading scheduler in order to maximize power saving. So we did not test BlueStreaming with only the traffic classifier and the shaper enabled. Each of these tests is conducted for 30 minutes. For each application, three tests are carried out consecutively in the hope that the Internet conditions and dynamics of these P2P services have not remarkably changed.

Table 3.4 gives an overview of the power saving on our testing client when accessing a channel of 400 Kbps in PPTV. During all three tests, our testing client receives smooth playback and captured frames show no quality degradation when compared to that in PSM-A. However, the amount of the battery power consumed by the WiFi interface for receiving
Table 3.4: PPTV at 400 Kbps

<table>
<thead>
<tr>
<th>Name</th>
<th>Total # of Streaming</th>
<th>Total # of Control</th>
<th>Sleep Time (%)</th>
<th>Consumed Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM-A</td>
<td>66465</td>
<td>147554</td>
<td>0.56</td>
<td>2005</td>
</tr>
<tr>
<td>Classifier only</td>
<td>66121</td>
<td>121512</td>
<td>25.82</td>
<td>1745</td>
</tr>
<tr>
<td>BlueStreaming</td>
<td>70622</td>
<td>132466</td>
<td>60.50</td>
<td>1090</td>
</tr>
</tbody>
</table>

Figure 3.7: PPTV 30-min Tests: Inter Packet Delay of WiFi (CDF)
streaming data is vastly different. With PSM-A, the WiFi interface spent only 10 seconds in the PSM and more than 99% of the session time in the active mode. When the traffic classifier is enabled, the amount of power saving is greatly improved: the WiFi interface spent more than 25% of session time in the PSM. When BlueStreaming is fully enabled, the WiFi interface spent over 60% time in the PSM, a 134% improvement over the case with only the traffic classifier enabled.

The additional sleep time of WiFi does not necessarily mean additional energy saving since BlueStreaming keeps Bluetooth active all the time and thus constantly consumes some
battery power. We need to take that into account to evaluate the total power consumption of BlueStreaming. Since we do not use an instrument to measure the absolute battery consumption, we rely on the specification given in [48] and [9] to estimate the total power consumed. That is, the WiFi interface consumes 1120 mW in the CAM and 72 mW in the PSM; the Bluetooth interface consumes 120 mW in the active mode, and 25 mW in the idle mode. Note that we consider Bluetooth to be always active, which is conservative for the total power saving. The last column of Table 3.4 shows that considering the total power consumption of both the WiFi and Bluetooth interfaces, the classifier only and BlueStreaming can save over 13% and 46% energy, respectively, when compared to PSM-A.

Figure 3.7 further shows the Inter Packet Delay distribution of WiFi traffic in these three tests. Compared to Figure 3.7(a), Figures 3.7(b) and 3.7(c) show that the Inter Packet delay distribution and Inter Streaming Packet delay are similar to each other, indicating the effectiveness of Traffic Classifier in diverting control traffic and exploiting sleep opportunities for the WiFi interface. In particular, Figure 3.7(c) shows a very pronounced bimodal pattern when all BlueStreaming components are enabled: about 95% packets arrive within 1 ms of the preceding one, while the rest arrive over 600 ms of the preceding packet, demonstrating the effectiveness of downstream traffic shaping.

Figure 3.8 further plots a snapshot (10 seconds) of the traffic pattern on the WiFi interface of our testing client in these three tests. Both incoming and outgoing packets are included. Figure 3.8(a) shows the traffic pattern when PSM-A is used. The traffic pattern shows a stair-case shape, and the small intervals between successive packets imply few opportunities for the WiFi interface to switch into PSM. Streaming with the classifier only can divert the frequent control packets to Bluetooth, resulting in more sleep opportunities as shown in Figure 3.8(b). However, due to the variation of response time from different neighbors, the traffic pattern is still not bursty enough. With the help of Traffic Shaper in BlueStreaming, Figure 3.8(c) shows that bursty traffic is sent more regularly and periodically, given that the interval between bursts is determined by Equation (3.1).

Table 3.5 shows the results of our testing client accessing a 500 Kbps channel in QQLive.
Table 3.5: QQLive at 500 Kbps

<table>
<thead>
<tr>
<th>Name</th>
<th>Total # of Streaming</th>
<th>Total # of Control</th>
<th>Sleep Time (%)</th>
<th>Consumed Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM-A</td>
<td>109800</td>
<td>206259</td>
<td>5.24</td>
<td>1917</td>
</tr>
<tr>
<td>Classifier only</td>
<td>105702</td>
<td>190428</td>
<td>34.37</td>
<td>1584</td>
</tr>
<tr>
<td>BlueStreaming</td>
<td>103600</td>
<td>187311</td>
<td>52.89</td>
<td>1234</td>
</tr>
</tbody>
</table>

The table shows that although QQLive transmits 50% more packets than PPTV, it has more power saving in PSM-A and Classifier-only than in PPTV. This is likely due to the smaller variation of response time from a different number of neighboring peers. However, the request time-out $T_{re-req}$ of QQLive is smaller than that of PPTV (based on our reverse engineering), and thus traffic shaping is less effective for QQLive. Even so, the extra amount of battery power saved by applying traffic shaping and upload scheduling is still prominent: the WiFi interface spent more than 52% of the session time in the PSM.

The average energy saving for all four P2P streaming applications, including PPS and Sopcast is shown in Figure 3.9, where Blue-I means BlueStreaming under the infrastructure mode and Blue-H means BlueStreaming under the hybrid mode. Because PPS has a very small request time-out value, our client saves about 10% energy in BlueStreaming compared to PSM-A. SopCast, on the other hand, has a larger $T_{re-req}$ similar to that of PPTV, and thus our client can save about 45% energy with BlueStreaming.

### 3.6.3 Hybrid Mode

We have shown that BlueStreaming is effective in saving power in the infrastructure mode where the AP supports both Bluetooth and WiFi. In practice, if Bluetooth is out of range or the AP does not support Bluetooth, we propose to use the hybrid mode. In this section, we evaluate BlueStreaming in hybrid mode with a relay node in the middle. The intermediate node forms a Bluetooth Ad-hoc network with our BlueStreaming client, and joins the subnet of BlueStreaming AP via the WiFi connection.
Since control traffic in P2P streaming is delay-sensitive, a particular concern on the hybrid mode is the delay of relaying, which could increase the response time at the application layer and deteriorate streaming quality. Because of the highly dynamic nature of P2P streaming overlay, we could not directly use application layer response time to compare the delay in different scenarios. Instead, we use \texttt{Ping} to measure the approximate round trip time from our BlueStreaming client to the AP via different paths, and estimate the increased control traffic delay. In short, the control traffic could be delivered under three scenarios: (1) \texttt{PSM-A} via a direct WiFi connection when BlueStreaming is not enabled; (2) \texttt{BlueStreaming-Infrastructure} via a direct Bluetooth connection; and (3) \texttt{BlueStreaming-Hybrid} via a relayed Bluetooth device. For \texttt{PSM-A} and \texttt{BlueStreaming-Infrastructure}, we use standard \texttt{Ping} to measure the RTT. For \texttt{BlueStreaming-Hybrid}, we run a customized \texttt{Ping} at our BlueStreaming client that measures the RTT of the relayed connection to AP.

<table>
<thead>
<tr>
<th>Name</th>
<th>RTT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM-A</td>
<td>3</td>
</tr>
<tr>
<td>BlueStreaming-Infrastructure</td>
<td>22</td>
</tr>
<tr>
<td>BlueStreaming-Hybrid</td>
<td>25</td>
</tr>
</tbody>
</table>

We send out one \texttt{Ping} request every 1 second with 32 bytes payload, which is the payload of a typical control packet. For each scenario, we take the average of 400 RTT results. Table 3.6 shows that with a direct 802.11n WiFi connection, the RTT is about 3 ms; with a direct Bluetooth connection in the infrastructure mode, the RTT increases to about 22 ms on average, which is likely due to the smaller throughput of Bluetooth; with a relayed Bluetooth connection in the hybrid mode, the RTT slightly increases to 25 ms, which is due to the additional WiFi transfer occurred between the relay node and the
WiFi AP. Apart from the potentially increased control traffic response time, the streaming quality during the tests has no degradation in either the infrastructure or hybrid mode.

Figure 3.9 further summarizes the average energy consumption across four applications in these tests. The figure shows that the estimated energy consumption is comparable in both modes.

Due to high dynamics of P2P streaming overlay, it is not appropriate to say if one mode is better/worse than the other in terms of power saving even if they have demonstrated slightly different performance on energy consumption. Nevertheless, it is reasonable to conclude that BlueStreaming can achieve good power saving in both modes.

### 3.6.4 Retransmission Rate

Throughout our experiments, we have been using 802.11n at 2.4 GHz band (instead of 5 GHz), which is the same band that Bluetooth operates on. In addition to that, using
airport scan on the MacBook to scan all available WiFi APs nearby, we find that all non-overlapping channels (channel 1, 6, and 11) of 2.4 GHz are occupied by more than 6 APs each during our experiments because our experiments are done in a lab covered by the AP hotspots. Because we run our experiments during 1pm to 5pm local time, there are a lot of wireless users around, meaning lots of background traffic. Both sharing 2.4 GHz band and significant background traffic may cause collisions and interferences.

Figure 3.10: Retransmission Rates

To examine the situation of such interferences, we study the re-transmission rate during our experiments. Figure 3.10 depicts the average retransmission rate of WiFi traffic throughout our experiments. The result shown in the figure is the mean of five runs. When the Bluetooth interface is not used, the retransmission rate under PSM-A across four applications is about 0.01. When we turn on BlueStreaming and use the infrastructure mode to transmit P2P streaming traffic, the retransmission rate increases to above 0.04. This shows
in our environment, actively using Bluetooth does mildly increase retransmissions. When BlueStreaming uses an intermediate node to relay control traffic in the hybrid mode, the retransmission rate increases to about 0.06, possibly because the relay node uses WiFi to communicate with our BlueStreaming AP and deliver control traffic. Even so, the streaming quality in the experiments is not degraded, demonstrating the practicability of BlueStreaming. In an environment with less background traffic, the retransmission rate could further decrease.

3.7 Summary

Recently the Internet has witnessed the rapid increase of P2P streaming traffic and the quick shift of using mobile devices for Internet accesses. Unfortunately, accessing delay-sensitive streaming services on a mobile device is hindered by the limited battery power supply, because a mobile client in P2P streaming needs to transmit more traffic, including uploading to neighbors as well as exchanging a large number of control packets with neighbors. In order to enable power-efficient P2P streaming to mobile devices, in this chapter, we have designed and implemented BlueStreaming, a system that simultaneously utilizes WiFi and commonly existing Bluetooth interfaces. The energy consumed by the Bluetooth interface is effectively over-compensated by the greater power saving on the WiFi interface. Extensive experiments with our implemented prototypes demonstrate the effectiveness and practicability of BlueStreaming.
Chapter 4: Eliminating Redundant Internet Streaming
Traffic to iOS Devices in C/S Streaming

4.1 Introduction

The recent years have witnessed the rapid and pervasive adoption of various mobile devices. For many applications, mobile devices are quickly replacing their desktop counterpart, including for receiving Internet streaming services. Mobile devices typically use wireless connections for receiving Internet streaming services. Be it WiFi or 3G/4G, typically, the capacity of a wireless connection still could not keep up with its wired counterpart, while streaming applications often involve bulk data transmission in a continuous fashion. This constrains the video quality (e.g., video encoding rate bits/second) that could be effectively delivered to mobile users. In addition to the quality, this could cause additional monetary cost to mobile users if cellular network connections have been used, because today the cellular data plan usually uses a tiered billing model [57]. Watching videos online can not only quickly generate a significant amount of traffic, but also result in the usage tier to be reached sooner, and extra monetary cost for subsequent data usage.

Moreover, the limited battery supply is still the Achilles’ heel of any mobile device. Receiving, decoding, and displaying a bulk amount of streaming data inevitably depletes the limited battery power supply at a fast pace.

Therefore, for mobile devices and mobile users, it is very important that the streaming data should be delivered in a precise fashion without unnecessary traffic or extra monetary cost. However, in this chapter, through server-side workload analysis and client-side measurements and experiments in a controlled lab environment, we find that current Internet mobile streaming practices introduce a significant amount of redundant traffic. In particular, for the popular iOS based mobile devices, accessing streaming services typically
involves about 10% to 70% unnecessary redundant traffic if a user watches the requested video from the beginning to the end. That is, such redundant traffic is not due to the early termination of the client access. Through experiments and analysis, we further investigate why such a significant amount of redundant traffic is transmitted. Our results show that: (1) to improve user’s experience of potentially re-watching the video, the MediaPlayer on iOS devices constantly re-downloads the beginning part of the video again after finishing downloading the entire file; (2) when the downloading speed is fast, the MediaPlayer frequently aborts the HTTP connection and then sends the request again, causing data in flow to be wasted; and (3) when the downloading speed is slow, the MediaPlayer continuously sends additional and overlapping requests to smooth the playback.

Such a significant amount of redundant traffic not only wastes network bandwidth, but also over-utilizes server-side resources. A streaming server is often short of bandwidth and processing power today due to the rapid increase of video files and requests. Moreover, even if such redundant traffic is for the sake of user’s perceived streaming performance, it is detrimental to the mobile device’s interest (in terms of battery power consumption) and the mobile user’s interest (in terms of potentially extra monetary cost).

Motivated by our measurement results, we examine the potential causes for such unnecessary traffic in normal mobile streaming accesses. We find these problems are mainly due to the limited available memory and the too fast/slow network connections. These findings motivate us to seek effective solutions to alleviate and minimize such redundant traffic without modifying the server side or the client side. For this purpose, we design and implement CStreamer that can transparently work between the client and the server. CStreamer partitions the video content into small segments. To eliminate the re-downloaded traffic, CStreamer synchronizes the downloading with the MediaPlayer’s playback progress. To refrain from sending too fast, it serves the segments periodically, instead of all at once. To deal with slow connections, CStreamer allows the MediaPlayer to seamlessly switch to a lower quality version of the same video provided by the server during playback.

To evaluate the effectiveness of CStreamer in minimizing the redundant traffic, we have
implemented a prototype of CStreamer and have deployed it on Amazon EC2. Different iOS devices are instructed to access various streaming services via this prototype. Our experimental results show that CStreamer can completely eliminate the redundant traffic without affecting user’s perceived streaming experience and save up to 40% power consumption. In summary, this chapter makes the following contributions:

- Through server-side workload analysis and client-side measurements, we find that the current Internet streaming services to iOS mobile devices often generate 10% to 70% redundant traffic that is detrimental to the server (for delivery), the network (for transmission), the mobile device (for battery consumption), and the mobile user (for money).

- Conducting experiments in controlled environments, we investigate the potential causes of such redundant traffic. We find it is mainly attributed to the limited available memory and too fast or too slow network connections.

- Motivated by our findings, we design and implement CStreamer that transparently works between the client and the server. We evaluate our CStreamer prototype with various popular Internet streaming services, and show that CStreamer can completely eliminate redundant traffic without degrading the user’s QoS and save up to 40% power consumption.

A portion of the work this chapter was published in [58]. The rest of the chapter is organized as follows. We present some background about HTTP range requests and streaming to iOS devices in section 4.2. Both the server-side and client-side measurements are presented in section 4.3. We present our analysis in section 4.4. Our design and implementation of CStreamer is presented in section 4.5 and we evaluate its performance in section 4.6. Some related work is presented in section 4.7 and we summarize this chapter in section 4.8.
4.2 HTTP Range Request and Streaming to iOS

Among the popular mobile devices, iOS based devices are leading the market [59]. According to Freewheel, 80% of wireless video views take place on iOS devices [60]. On iOS, pseudo streaming is often used when a video streaming service, such as YouTube [5] and DailyMotion [15], is accessed. For example, a mobile version of YouTube allows iOS and Android users to use pseudo streaming to watch videos in either their native web browsers or native YouTube applications. That is, the client can download the media content from an HTTP server. The playback can start before the entire file is downloaded. In order to support VCR-like control, such as fast forward and rewind, the client also uses HTTP range requests to request part of the video file. An HTTP range request, or range request in short, is an HTTP request with ranges specified in the header of the request, indicating the desired data range of the requested file. The server only needs to respond with that part of file instead of the entire file. However, the entire file can be requested with the range specified from 0 to $\text{filesize} - 1$.

The MediaPlayer identifies itself with the user agents (e.g., AppleCoreMedia/1.0.0 ...). It would first check the cache on its storage. If it could not find the requested video in the cache, it would ask the server for meta-data information about the video file, including file size, last modified time, etc. This is achieved by sending out an HTTP GET request specifying a range of 0–1. Then, the MediaPlayer would send multiple HTTP requests for the file, and specifies a range to download in each request.

If the requested video is cached, the MediaPlayer would send a \texttt{If-Modified-Since} request to the server. If the server replies with HTTP 304 \texttt{Not Modified}, the MediaPlayer would start to play the cached content. However, due to the limited storage and the potentially large video file size, MediaPlayer usually does not cache the entire file. Instead, it only caches the beginning part (e.g., several hundred KBytes) of the video file. So it would issue multiple consecutive HTTP range requests to request the non-cached part of the file. Correspondingly, the server would reply with HTTP 206 \texttt{Partial Content} for
each request.

4.3 Internet and Local Measurements

In order to investigate the Internet mobile streaming delivery to iOS, we conduct measurement and analysis from both a server side and a client side.

4.3.1 A Server-Side View

To gain a collective view of the current Internet mobile streaming practice, we are able to access server-side logs from a top Internet mobile streaming service provider, Vuclip [61]. This service allows users to search and play videos on mobile devices. Pseudo streaming is available from this service. We collected one-month server log from Feb 1st to Feb 28th, 2011. While accesses from various mobile devices are logged, we extract accesses from iOS devices based on User-Agent String in HTTP requests, and compare the logged traffic amount with the actual file size.

In the one-month server-side workload, we extract 397,940 unique video sessions from iOS devices accessing the Vuclip during February 2011. Our following analysis focuses on these iOS sessions only. Note that in each session the video is not necessarily watched from the beginning to the end, as a user may find the video not interesting and terminate the session early.

Each viewing session consists of multiple HTTP (range) requests. Figure 4.1(a) shows the distribution of # of HTTP requests that are served by the server per viewing session. For about 50% of the sessions, more than 12 requests are used. More than 20% of the sessions have issued more than 40 requests.

Figure 4.1(b) shows the distribution of HTTP response size. Note here when we count the size of a response, we have excluded the response header information and only count the size of the response body. In general, the response size is determined by two factors: (1) if the request is successfully served, the response size is the same as the requested range; (2) if the request was aborted by the client, then the response size is smaller than the specified
As shown in the figure, more than 35% responses are about 64 KBytes (reasons to be discussed in section 4.4.3).

For each unique session, we sum up the body size of HTTP responses, termed as transferred size, and compare it with the actual size of the requested file. Figure 4.1(c) shows the distribution of the ratio between the transferred size against the actual file size. Overall, more than 28% sessions received more data than the actual file size. Besides the extra data received, Figure 4.1(a) shows these sessions also initiated more HTTP requests to download the video. Given that we are considering all sessions here, such a number would
be much larger if we only consider sessions that had watched the entire video. Figure 4.1(d) further shows that for sessions that last longer than three quarters of the video duration, 78% of them received more data than the video file size. 39% of them even received over 50% more traffic than the actual file size.

4.3.2 A Client-Side View

In addition to server-side workload analysis, we also conduct experiments in our lab in order to investigate the redundant traffic. We have conducted experiments with four devices running different versions of iOS as shown in Table 4.1:

<table>
<thead>
<tr>
<th>Name</th>
<th>iOS version</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPod Touch</td>
<td>3.1.2</td>
<td>128 MB</td>
</tr>
<tr>
<td>iPhone 3G</td>
<td>4.2.1</td>
<td>128 MB</td>
</tr>
<tr>
<td>iPhone 3GS</td>
<td>5.0.1</td>
<td>256 MB</td>
</tr>
<tr>
<td>iPhone 4S</td>
<td>5.1</td>
<td>512 MB</td>
</tr>
</tbody>
</table>

These devices are instructed to access two types of services. On the one hand, they are used to access the streaming service of YouTube [5], Dailymotion [15], and Veoh [16] via MobileSafari. On the other hand, we also setup our own HTTP server in our lab running Nginx 0.7.65 [62] to do controlled experiments in order to further investigate why and how the redundant traffic is delivered.

During the experiments, to record all the incoming and outgoing packets, we setup Wireshark [53] to listen on the same channel as the testing device in promiscuous mode and capture all packets received/delivered from our testing device. We analyze the traffic that is received by these testing devices, with a focus on the amount of redundant traffic that is received.
Our first set of experiments is to investigate whether such a redundant traffic phenomenon is unique to Vuclip or it exists for other services as well. For this purpose, we first use our iOS devices to watch a same YouTube video repeatedly at different times. We capture all the incoming and outgoing packets in the streaming sessions, and compare the actual size of the video file (36.7 MBytes) with the total number of bytes in the HTTP responses that are received by our testing devices. While the server-side workload may have included early terminated viewing sessions, in the client-side experiments, all viewing sessions are normal sessions without early termination.

Figure 4.2: Statistics of iOS Devices Watching YouTube

Figure 4.2(a) shows the results of our iOS devices watching the same YouTube video at different times of a day. Most of the streaming sessions received more than 40 MBytes of responses, which is 10% more than the file size. It is noticeable that iPhone 3G even received more than 74 MBytes of traffic in some sessions, which is more than doubled the size of the video file. Note that we only count the payload (i.e., video data) of HTTP responses here, without taking into account the protocol headers.

We further conducted experiments with all three different video streaming services,
accessing 3 different video files from our testing devices. Table 4.2 shows the average received traffic of our 4 testing iOS devices compared to the actual file size from 10 tests each. We find that, on average, mobile devices’ received traffic is consistently more than 110% of the actual file size across all three video files and four different devices.

The impact of such an extra amount of redundant traffic is multi-fold. First, this increases the traffic on the Internet. More importantly, this adds additional load on the server while a server is constantly busy with serving multiple clients. Besides unnecessarily over-utilizing the server and Internet resources, such traffic is also detrimental to the user and mobile devices’ interests. On one hand, receiving more data would make the wireless network interface card (WNIC) on the mobile device work longer and thus consume more battery power. On the other hand, if the video is downloaded using a cellular network connection, it would lead to the data plan tier be reached sooner than expected and generate more monetary cost because cellular data plans today often use a tiered billing model.

### 4.4 Analysis of Redundant Streaming Traffic

To find out why such redundant traffic is transmitted, we closely study the captured work-loads and further conduct experiments to validate our findings.
4.4.1 MediaPlayer RE-REQUESTs downloaded data after the entire file is downloaded

We have discussed in section 4.2 that to request the video file for playback, the MediaPlayer would send out multiple HTTP range requests for the file. We thus examine how the requested range changes as playback proceeds. Figure 4.3 shows the Byte-Range of consequent HTTP requests during one typical YouTube experiment for our 4 iOS devices, respectively. In all experiments, we instruct our testing devices to watch a same 480-second long video on YouTube. The file size of the video is 38,517,389 bytes (i.e., the YouTube video in Table 4.2).

Figure 4.3(a) shows two phases, one before 262 seconds and one after 262 seconds. While the entire video file is fully downloaded in the first phase, we notice that there are quite some range requests afterwards (the second phase). Note that the video plays for 480 seconds. For iPhone 3G, Figure 4.3(b) also shows similar two phases. The first one before 220 seconds, and the second one after 220 seconds. For iPhone 3GS shown in Figure 4.3(c), the two phases are before 190 seconds and after 190 seconds. For iPhone 4S shown in Figure 4.3(d), the two phases are before 109 seconds and after 109 seconds. However, this is not due to re-watching or rewinding, because we watched the entire video without seeking backwards during the playback or clicking on the replay button. That is, after the video file has been completely downloaded, the MediaPlayer automatically starts to request again from the cached point. Such a behavior is consistently observed in not only YouTube but also other Internet streaming services and our HTTP server.

To study these re-downloading, we further examine the repeated YouTube tests. In these viewing sessions, we study the amount of extra traffic as being transmitted after the entire file has been downloaded. The results are shown in Table 4.3. For 14 out of 18 tests on iPod Touch, the MediaPlayer downloaded the beginning part of the video again after finishing downloading the entire file, and the average amount of extra traffic is 22.9 MBytes. For iPhone 3G, such traffic is seen in a higher percentage of tests (14 out of 16), with an average of 47.7 MBytes, which is even larger than the file size. Usually, the first
Figure 4.3: HTTP Range Requests in one YouTube Experiment
such re-request would request from the cached point (\texttt{Range.Start}) to a certain point in the middle of the file (\texttt{Range.End}). In succeeding requests, the \texttt{Range.Start} would increase with an interval of 64 KBytes as would \texttt{Range.End}.

After multiple experiments, our conjecture on this behavior is that the MediaPlayer downloads the data from beginning again in case the user wants to re-watch the video. So it requests the missing part of the video that is not or no longer cached in the memory or storage. At this point, the memory is mostly occupied by downloaded but not yet played data, and thus the available free memory space is small. However, the downloading speed is so fast that it is \textit{about} to fill up the available memory. Then the MediaPlayer decides to abort that connection, and not to replace the un-played data.

In general, mobile devices have a smaller memory size compared to desktop/laptops. For example, the iPod Touch and iPhone 3G we used for experiments have only 128 MBytes memory, and the system daemons use up more than 2/3 of the memory, leaving about only 30 MBytes memory for applications running on the mobile device. Because of the small memory, sometimes it is not feasible for a mobile device to fit the entire video in the available memory. One may wonder if by increasing the memory size would solve the problem. However, as we have shown in Figures 4.3(c) (d) and Table 4.3, with memory size increased to 256 MBytes in iPhone 3GS and 512 MBytes in iPhone 4S, such re-downloading behavior still persists.

While the iOS devices are mobile devices, we further use Safari 5.1.1 to watch the same video on mobile YouTube [5] (instead of the www.youtube.com site) from a MacBook Pro.
running Mac OS 10.7 Lion with 4 GBytes of memory. While multiple repeated experiments have been conducted, with no exceptions, no re-requests are issued for the downloaded data after the entire file has been downloaded. Clearly, MacBook Pro has a much larger memory size, allowing it to cache the entire video in the memory and serve directly from the memory if the user wants to re-watch the video.

On iOS mobile devices, however, with the increased video quality, higher screen resolutions, and multi-tasking capability, the available memory would always be limited. And as a result, to better accommodate the user’s potential replay request, the MediaPlayer’s decision to download the video again has potentially contributed a significant amount of redundant traffic. Even worse, the user may decide not to re-watch the video, and the re-downloaded data is completely wasted.

4.4.2 The client frequently ABORTs the connection, causing data in flow useless

While Figure 4.3 shows the range of the requests sent from the mobile devices, Figure 4.4 shows the corresponding responses to these requests. As we can observe by comparing both figures, while the MediaPlayer often sends out an HTTP range request from the start or a later point to the end of the file, the MediaPlayer also often aborts the connection before it receives all the requested data, and starts a new connection after that. A new HTTP range request would be sent to the server through a new connection, with Byte-Range from the last successfully received byte in the previous connection all the way to the end of the file. Figure 4.4(a) shows that 113 connections were set up and terminated during the first 262 seconds when the MediaPlayer was actively downloading the file. For iPhone 3G/iPhone 3GS/iPhone 4S as shown in Figures 4.4(b) (c) and (d), the received ranges are also consistently smaller than the requested ranges as shown in Figures 4.3(b) (c) and (d).

Figure 4.2(b) shows the total number of TCP connections used to download the entire video file from the YouTube server. On average, iPod Touch uses 114 TCP connections to watch the video, while iPhone 3G uses 208, iPhone 3GS uses 61, and iPhone 4S uses
Figure 4.4: Received Range per Request in one YouTube Experiment
These numbers are also consistent with what we have observed from the server-side log shown in Figure 4.1(a): among the sessions that received more traffic than the file size, 47% of them sent out more than 100 HTTP requests.

Figure 4.5: TCP Stream Connections

Figure 4.5(a) shows the distribution of traffic amount that has been successfully received and acknowledged at the TCP layer of the client for each connection. More than 95% of the TCP connections received less than 1 MB data, much smaller than the \texttt{Byte-Range} specified.
in the HTTP requests as shown in Figure 4.3. Given the small amount of traffic transferred, it is not hard to imagine that the TCP connections do not last long either. Figure 4.5(b) shows the distribution of time elapsed from the client sends out TCP-SYN to start a TCP stream till it decides to terminate the connection and sends out TCP-FIN. It is shown that about 78%-88% TCP streaming sessions of our testing iOS devices lasted less than 1 second before they were terminated. A subsequent TCP connection was immediately started after that. As shown in Figure 4.5(c), the mean time difference between the transmission of TCP-FIN of the previous connection and the TCP-SYN of a new connection is about 1 second for iPod Touch/iPhone 3GS/iPhone 4S, and 200 ms for iPhone 3G.

Such abnormal aborts apparently can cause the data in flow to be wasted as typically it takes at least a round-trip time for the server to respond to the termination. One may wonder such frequent aborts and new requests may be due to the slow connection speed. However, these experiments were conducted in our lab with a dedicated AP and at different times. As shown later, we have validated that this is not the reason. Before we try to look for answers, we first show how much traffic has been wasted due to such abnormal aborts.

First, in our experiments, we have observed that some of the packets received and acknowledged at TCP layer are requested again in the subsequent connection. For example, suppose the MediaPlayer requested the range from 100 KB to the end of file in a request, and 500 KB are received and acknowledged at TCP layer before TCP-FIN is sent out. Ideally, the subsequent request should be from 600 KB to the end of file if the ACK-ed packets are delivered to the application. However, we consistently observe the subsequent request to request from anywhere between 100 KB and 600 KB, to the end of the file, causing duplicated traffic transmission. Figure 4.5(d) shows the distribution of the wasted traffic amount. In about 36% and 27% of all aborted connections for iPod Touch and iPhone 3G, more than 120 KB data are ACK-ed at TCP layer, but requested again in the subsequent connection. For iPhone 3GS and iPhone 4S, 20% aborted connections wasted more than 93 KB and 46 KB data, respectively.

Second, besides ACK-ed packets that are not successfully delivered to the application
layer, more data is wasted in half-closed connections. When the MediaPlayer decides to terminate a connection, it sends out a TCP-FIN, and expects the server to reply with TCP-FIN. The HTTP server, however, interprets this as a half-closed TCP connection in which there is nothing to transmit from the client side. The server continues to send out the response that has already been in the TCP buffer before replying with a TCP-FIN, given that the TCP window size at the client side is often set to 131,072 bytes (128 KB) in the last segment. However, as we observed in the traces, for each subsequent none-TCP-FIN packets received at the client side, the client would send out a TCP-RST, asking the server to stop sending to this connection. Moreover, most video servers today use asynchronous I/O, which might cause more packets to be sent out before the TCP-RST is processed.

![Graphs showing time and traffic delivered during half-closed TCP](image)

**Figure 4.6:** Time and Traffic Delivered During Half-closed TCP

Because of the mismatch between the client and the server, the server continues to send packets after receiving TCP-FIN. We analyze the RST-ed traffic of traces from our experiments. Figure 4.6 depicts the distribution of time between when TCP-FIN is sent out and when the last packet of the TCP connection is received at the client side. For most TCP stream sessions, it takes the YouTube server about 10 ms to 100 ms to stop sending
packets to the closed TCP connection. As a result, as shown in Figure 4.6(b), a median of 28 KB to 54 KB data is wasted per closed connection for 4 testing devices. Recall that we have shown in Figure 4.2(b) that more than 60 TCP connections on average are used for a single viewing session in our experiments, wasting 54 KB data per connection would lead to more than 3 MB redundant traffic in total for these RST-ed packets.

**Reasons for frequent connection aborts**

As we have observed a high number of aborted connections, we set to investigate why such TCP behaviors happen. We first examine whether the serving speed would impact the MediaPlayer’s decision to abort a connection. We categorize all TCP connections observed in our experiments into two categories: TCP connections that are aborted before the full HTTP response is received (termed as **Aborted** connections), and normal TCP connections that received the full HTTP response (termed as **Normal** connections).

![Figure 4.7: Throughput (KBytes/s) of Aborted Connections (CDF)](image)

Figure 4.7(a) shows the average throughput per connection in our experiments with our local Nginx HTTP server. In these experiments, we set different bandwidth limits to
the server, and examine the connections that are aborted or closed normally. We calculate average throughput as the traffic amount that has been received and acknowledged divided by the TCP connection time between TCP-SYN and TCP-FIN. As shown in the figure, about 80% of Normal TCP connections have a throughput smaller than 600 KBytes/s, while more than 85% of Aborted connections’ throughput is more than 600 KBytes/s. Overall, although there is no clear distinction, Aborted connections generally have higher throughput than Normal connections. The results from Internet (YouTube) experiments as shown in Figure 4.7(b) are also similar. This confirms that such frequent connection aborts are actually not caused by slow connections, but rather fast connections.

Fast connections can impact the downloading behavior from two aspects. On one hand, when the MediaPlayer finds the downloading speed is so fast that the downloaded but unplayed part of the video has nearly filled up all the available memory, it may decide to abort the connection, and let the playback buffer be consumed before resuming the downloading. On the other hand, with fast connections, the MediaPlayer finds the downloading speed is too fast, but it does not know if the user would continue to watch the video. If not, continuing to download at such a high speed would waste both traffic and battery power on the mobile device. So it decides to abort the current connection, and start a new one afterwards if the user is still watching.

4.4.3 Additional OVERLAPPING requests are sent to compensate slow connections

While the fast serving/downloading speed from the server can cause redundant traffic, we find that a slow serving/downloading speed causes problems as well. This happens when the MediaPlayer finds the downloading speed of the current connection is not fast enough to keep up with the playback progress. Such a slow downloading speed may be caused by either server-side bandwidth constraint or client-side connection limit. Server-side bandwidth constraint is potentially caused by: (1) the server is serving too many connections; and (2) the server is throttling the per connection serving speed. On the client side, the mobile
device’s slow connection speed can be caused by competition for wireless channel, be it either cellular or WiFi. When the connection speed is slow, the MediaPlayer would start a new connection to request data in the unit of 64 KB if the file is small. This helps smoothly play the video by fetching desired data directly. However, the original “slow” connection is not terminated, continues to download the file.

![Graph](image)

**Figure 4.8: Byte-Range of Requests in Slow Connections**

This pattern is very common in the server log we collected. As shown in Figure 4.1(b), more than 35% HTTP responses are around 64 KB. We also observe this phenomenon in our HTTP server when we set the serving speed limit. The requested range and the served size as logged are usually 65,536 bytes (64 KB). For example, Figure 4.8 shows the Byte-Range as specified in each HTTP request when we limit the serving speed to 82 KBytes/s and instruct iPhone 4S to view a video of 480 seconds and 38,517,389 bytes (80 KBytes/s). The initial request was from the beginning all the way to the end of the file, and was not aborted throughout the session. It took the server 469 seconds to finish delivering the data. During the 469 seconds when the response was being sent, the MediaPlayer further sent out 79 requests, each with a Byte-Range of 65,536 bytes. As a result, 5,117,344 bytes were received to compensate the slow connection. After the entire file was downloaded, the MediaPlayer started to download the file again from the beginning, as shown at 487 seconds in the figure. Because these requests were aborted before the MediaPlayer received
the full response, they only led to a total of 86,271 bytes of traffic. Overall, the MediaPlayer received 13.7% more traffic than the actual file size.

### 4.4.4 Summary of Analysis Findings

Through experiments and analysis, we find the redundant streaming traffic can be mainly attributed to the following: (1) the limited available memory causes the MediaPlayer to re-download the previously downloaded data in order to accommodate potential replay requests from the user; (2) the limited available memory and the fast connection speed cause HTTP connections to be frequently aborted, wasting a lot of data in flow; (3) a slow connection can also cause the MediaPlayer to issue overlapping requests to provide better experience to end users.

### 4.5 Design and Implementation of CStreamer

The redundant traffic is mainly caused by the limited available memory on mobile devices and the mismatch between the client and the server for connection aborts. Such redundant streaming traffic not only over-utilizes the Internet and server resources, but also ultimately incurs extra battery power consumption and potentially monetary cost to users.

Unlike desktop operating systems, mobile operating systems today do not use swap/virtual memory to extend memory size. Moreover, as we have shown, even if the physical memory size is increased from 128 MBytes in iPhone 3G to 512 MBytes in iPhone 4S, the problem persists. This is likely due to the increased screen resolution of iPhone 4S that uses more memory for display, and the increased degree of multitasking on iPhone 4S. As the quality level of mobile videos also keeps increasing, the limited memory size is likely to continue as a bottleneck for Internet mobile streaming.

Furthermore, iOS is a closed system, which makes it difficult for users to modify the iOS MediaPlayer. One may argue that such a problem is due to design pitfall or a software bug, and can be fixed by software updates. However, such a problem is seen in different iOS versions from 3.1.2 to 5.1 with millions of devices installed. Updating existing software
may not be easy and quick.

With these considerations in mind, we have built a middleware system, which we call CStreamer. With CStreamer, redundant traffic can be eliminated without changing either the iOS MediaPlayer or the many media sites which serve videos via pseudo streaming.

4.5.1 CStreamer Design

While pseudo streaming to iOS generates redundant traffic due to three reasons as discussed in section 4.4, we find that such phenomenon does not happen when videos are delivered with HTTP Live Streaming (HLS). This suggests a straightforward solution for mitigating the redundant traffic in pseudo streaming: convert pseudo streaming into HLS. The challenge here, however, is how such conversions can be done in a transparent approach.

![Diagram of CStreamer](image)

Figure 4.9: Overview of CStreamer

Figure 4.9 shows the architecture of the CStreamer. CStreamer combines an iOS App with a proxy-like CStreamer server. The iOS App works with the CStreamer server to rewrite pseudo streaming video links so that the MediaPlayer requests streaming data using HLS from the CStreamer server. When the CStreamer server receives such a video request with the re-written URL, it downloads the desired video from the video server using a
single HTTP GET request. Then it segments the video according to HLS, and transmits the segments to the iOS devices for playback. Converting pseudo streaming to HLS with CStreamer brings the following benefits:

**When downloading speed is fast**

With pseudo streaming, a client would request the media file aggressively. In CStreamer, however, the MediaPlayer would request file segments sequentially and periodically. That is, a subsequent request is not sent out immediately following the current one. Rather, it waits for its turn until the playback progress has reached its scheduled time.

This allows the MediaPlayer to take into consideration the playback progress when issuing requests. Depending on the memory available at the client side and the connection speed, the request is at least 1, at most 5 segments ahead of the current playback. On one hand, this reduces the unwatched data if the user stops watching in the middle. On the other hand, when the available memory size is small, the request rate is not as aggressive as in pseudo streaming, and therefore HTTP requests would not be aborted. Moreover, the MediaPlayer does not re-download the beginning portion of the video after finishing downloading the entire video.

**When downloading speed is slow**

While downloading a video using pseudo streaming under a slow connection, the iOS MediaPlayer issues parallel, overlapping requests for video ranges, leading to redundant traffic and even slower effective downloading speeds. When the MediaPlayer goes through CStreamer transparently, however, it would always wait to receive the full response of the current request, without sending out any additional overlapping requests.

To deal with various connection speeds, many video service providers today, including YouTube, offer different versions of the same video encoded in different rates. This allows a user to switch to a lower quality version when the downloading speed is slow compared to the streaming rate. To adaptively deliver the video when the connection speed is slow,
CStreamer requests a high quality version and a low quality version of the same video, segments both versions, and puts the meta-information of both versions in the same playlist, allowing the user to seamlessly switch between different versions.

4.5.2 CStreamer Implementation

Our CStreamer prototype consists of four major components:

**Request Handler**

The Request Handler processes two types of requests sent by the mobile device: meta-info requests and video requests. For *meta-info request* (e.g., requesting a file containing video name, duration, and video link), the Request Handler would request the desired content from the video server. However, before it delivers the response, it rewrites the pseudo streaming link in the response to a new URL: the CStreamer URL. This URL is an HLS URL that points to a new playlist file on the CStreamer Media Server. After the mobile device receives the response containing the CStreamer URL, if the user decides to watch the video, the MediaPlayer would send out a *video request* directing for the CStreamer URL. When the Request Handler receives such a video request, it would call the Media Downloader.

**Media Downloader**

The Media Downloader receives the request from the Request Handler. It extracts the original pseudo streaming link from the CStreamer URL, and starts immediately to download the requested video at the highest speed. As the video is being downloaded, the Media Downloader pipelines content to the Media Segmenter, which segments video without waiting for the download to complete. This pipelining procedure results in a minimal user perceived start-up delay.
**Media Segmenter**

The Media Segmenter consists of two parts: Container Changer, and Segmenter. Videos deliverable to iOS devices via pseudo streaming today, are often put into either MP4 or 3GP format other than MPEG2-TS used by HLS. The video file must be put into MPEG2-TS container format to be segmented. However, unlike video transcoding which is CPU intensive and slow, changing only the container format does not require changing the audio/video encoding and is fast enough to be conducted at real-time.

The Media Segmenter receives pipelined output from the Media Downloader, feeds the data into the Container Changer to change the container format. The Container Changer further pipelines its output to the Segmenter, which segments the video into segments. The pipelined execution of the Media Downloader and the Media Segmenter makes CStreamer very fast to prepare the video content.

After the requested video has been processed, the Media Downloader and the Media Segmenter can move on to process another version of the same video, either in higher quality or lower quality.

**Media Server**

While the Media Downloader and Media Segmenter are still processing, the Media Server allows the user to download and watch the first segment. Without an `EXT-X-ENDLIST` tag in the playlist file, the MediaPlayer would wait and retrieve the playlist again later from the Media Server, which contains updated playback meta-information.

To efficiently utilize the storage at the Media Server, and save the downloading bandwidth cost, the Media Server also maintains a database with information about the original video file (e.g., web service, video id, video link, etc.) and its corresponding segmented files (e.g., location, playlist file, etc.). This allows more requests for the same video to be served directly from the Media Server, without repeating the downloading and segmenting processes.
4.5.3 CStreamer iOS App

For an iOS device to use CStreamer, the end user can set the CStreamer server as an HTTP proxy to handle the requests. However, manually configuring the iOS device is inconvenient for end users, and proxying all traffic through CStreamer puts a lot of burden on the CStreamer server. To mitigate such drawbacks, we have also implemented a CStreamer App. To end users, the CStreamer App is a web browser. However, it monitors all requests, and identifies meta-info requests. For example, the request URL for a YouTube video meta-info starts with http://m.youtube.com/watch?ajax=1. The response to this request contains a json file with video’s pseudo streaming link in it. The CStreamer App redirects such video meta-info requests to the CStreamer server, where the response is rewritten by the Request Handler.

4.6 Performance Evaluation

To evaluate the effectiveness of CStreamer, we deploy our prototype on Amazon EC2 [63]. We run CStreamer on an EC2 Micro Instance, and instruct our iOS devices to access the video services of YouTube and Dailymotion via CStreamer. For each access, we have repeated the experiments 10 times consecutively. In our experiments, we focus on whether CStreamer can serve user’s requests in a timely manner, so we do not consider the case when the video can be directly served from CStreamer cache. After each experiment with CStreamer, we would empty the media server’s storage.

Figure 4.10 shows the traffic patterns of two consecutive experiments we conducted to watch a 480-second YouTube video on iPhone 3GS using pseudo streaming and CStreamer respectively. Results with other iOS devices are similar. With pseudo streaming, over 59 MBytes of traffic were delivered. With CStreamer, the 480-second video is segmented into 48 segments. In the current implementation, the segments are delivered periodically. Thus each segment is delivered every 10 seconds, except for the first 5 segments, which were requested aggressively by the MediaPlayer. Figure 4.10(b) shows that: (1) each segment is
downloaded only once and no RE-REQUEST is observed, even if the last segment finishes downloading 40 seconds earlier before the end of playback; (2) The MediaPlayer on the iPhone 3GS did not abort any connections, and each segment is downloaded in only one connection. As a result, no redundant traffic is transmitted during the entire streaming session. As a result, about 31% of traffic is saved compared to using pseudo streaming.
Similarly, Figure 4.11 shows the traffic patterns of watching a 478-second video on Dailymotion. More than 50 MBytes of traffic was transmitted using pseudo streaming, while CStreamer did not cause any redundant traffic.

While CStreamer can eliminate redundant traffic as shown in Figures 4.10(b) and 4.11(b), one may wonder if the user perceived streaming quality could be affected due to additional processing between the client and the server. A vital metric here is the start-up delay. We thus examine how long does it take from the user choosing to watch a video to the MediaPlayer starts playback.

Table 4.4: Estimated Start-up Delay (seconds)

<table>
<thead>
<tr>
<th>Name</th>
<th>Pseudo Streaming</th>
<th>CStreamer</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouTube</td>
<td>1.78</td>
<td>1.75</td>
</tr>
<tr>
<td>DailyMotion</td>
<td>2.42</td>
<td>2.87</td>
</tr>
</tbody>
</table>

We estimate the start-up delay of pseudo streaming by examining the period between when the HTTP request for the video is sent and when 10 seconds of streaming data is received. For CStreamer, we examine the period between when the video request is sent and when the first segment was downloaded. To make the comparison more meaningful, we compare a pair of experiments that are conducted sequentially. Table 4.4 shows the results. For YouTube, we find that the video server is close to our testing location. So with pseudo streaming, it took only 1.78 seconds to download the initial 10 seconds of playback data. With CStreamer, despite the communication between our client and CStreamer server as well as the processing delay, it took only 1.75 seconds to download the first 10-second segment. This is potentially because the EC2 instance we used to run CStreamer is also close to the YouTube server on the Internet, and therefore it can download the video file at very fast speed. Similar to YouTube, we find that DailyMotion does not experience much additional delay either. This indicates the start-up delay, which is important to user’s
perceived QoS is not affected by using CStreamer.

Table 4.5: Average WNIC Sleep Time (%)

<table>
<thead>
<tr>
<th>Name</th>
<th>Pseudo Streaming</th>
<th>CStreamer</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouTube</td>
<td>80.9</td>
<td>87.7</td>
</tr>
<tr>
<td>Dailymotion</td>
<td>79.8</td>
<td>90.5</td>
</tr>
</tbody>
</table>

By eliminating redundant traffic, CStreamer allows the wireless network interface card (WNIC) on the mobile device to spend more time in low power sleep mode, and thus saves battery power consumption. The battery saving comes from two aspects: the reduced total traffic amount and the bursty traffic delivery. For example, for the YouTube experiment presented in Figure 4.10(a), the WNIC is able to sleep 86.8% (416 seconds) of time during the 480 second playback; while it only sleeps 85.0% of time in our succeeding test when watching the same video via pseudo streaming. For Dailymotion, using CStreamer allows the WNIC to sleep 91.7% (439 seconds) of the time over 478 seconds, while it can only sleep 83.6% time when using pseudo streaming. The average of the 10 experiments is shown in Table 4.5. Similar to section 3.6, we utilize the specification given in [9] to estimate the power consumption at the WNIC. We find that on average, the power consumption is reduced by 26% for YouTube and 40% for Dailymotion.

4.7 Related Work

In recent years, the Internet streaming traffic has increased dramatically. Plenty of previous work had mainly focused on characterization, measurement, and analysis of all kinds of VoD and P2P-assisted streaming services. For example, different user access patterns, session lengths, video popularity, and other content properties have been studied in traditional and live VoD systems [33], [37]. Krishnappa et al. examined Hulu traffic, focusing on the
potential caching and prefetching at the edge networks [38]. For P2P-assisted streaming systems, there are also lots of studies that aimed to characterize and improve the existing performance [64], [26], [39], [40].

Similar studies have been conducted on popular video sites, such as YouTube. For example, user behaviors and video popularity of YouTube were studied and compared with non-UGC content from Netflix [41]. The video properties and access patterns of YouTube were analyzed in [65]. Gill et al. reported the traffic characteristics of YouTube at a campus edge network [34].

With the increase of Internet mobile streaming services, Xiao et al. studied the power consumption of mobile YouTube [27]. Finamore et al. collected traffic from several edge locations and studied the potential reasons for the inferior streaming experience of mobile YouTube users [29]. Rao et al. characterized and compared the traffic pattern of YouTube and Netflix on desktops and mobile devices [66]. Erman et al. examined mobile video traffic over cellular networks from the ISP’s perspective [67]. Li et al. examined an iOS-based mobile TV based on server-side logs [68]. In this chapter, focusing on the dominant iOS devices, we find that streaming to these iOS devices has introduced a significant amount of redundant traffic in the current practice, which is detrimental to both the server and the client, as well as the resource utilization on the Internet. Our proposed solution can effectively address this problem without requiring changes at either the client side or the server side.

4.8 Summary

Internet mobile streaming traffic has started to dominate the Internet mobile data traffic, and it continues to increase with wider adoption of all kinds of mobile devices. Precisely delivering streaming traffic to mobile devices is not only important to the service providers and the Internet, but also important to mobile devices (battery power wise) and mobile users (monetary cost wise). In this chapter, through measurement and analysis, we find
that there is non-trivial redundant traffic delivered when existing mobile streaming services are accessed on iOS devices. Motivated by the analysis results, we design a middleware that can transparently reduce such redundant traffic. Having evaluated with a prototype installed on Amazon EC2, we find that our solution can completely eliminate such redundant traffic without degrading end users’ performance and save up to 40% power consumption.
Chapter 5: Conclusion and Future Work

5.1 Conclusion

Internet streaming to mobile devices is becoming increasingly popular. Based on logs collected from the server-side, we investigate the unique characteristics of Internet mobile streaming in the current practice. We show the great hardware and software heterogeneity of mobile devices, different characteristics of mobile videos, and different user access patterns from those in traditional Internet streaming services.

One significant challenge for Internet mobile streaming today is that mobile devices are equipped with limited battery supply, while watching streaming videos on mobile devices may consume the battery power at a fast pace. Therefore, reducing the power consumption during streaming data transmission is essential.

In this dissertation, we have designed and implemented two systems that can effectively reduce power consumption during Internet streaming to mobile devices. The first, BlueStreaming, improves the power-efficiency of P2P streaming. It intelligently utilizes the additional Bluetooth interface to transmit the highly frequent, delay-sensitive, and low throughput control traffic in P2P streaming, thus trading the power consumption of Bluetooth interface for greater power savings at the WiFi interface. Our experimental results based on a BlueStreaming prototype system show that BlueStreaming can achieve up to 46% power saving. Our second system, CStreamer, addresses the redundant streaming traffic problem in C/S streaming which we find via extensive experiments with iOS devices. Designed as a middleware system, CStreamer does not require any changes from either the client side or the server side. We have implemented a prototype system of CStreamer and deployed on Amazon EC2. Experiments based on this prototype show that CStreamer can
completely eliminate the redundant streaming traffic and save up to 40% power consumption.

5.2 Future Work

In the future, I plan to expand the scope of our research and improve the power-efficiency of other mobile system components. For example, cellular data interfaces, such as 3G, 4G LTE, are inherently different from WiFi. They have much longer tail time compared to WiFi (more than 10 seconds vs. approx. 70 ms) in which the interface must operate with high power consumption. On the other hand, LTE also employs a low power DRX (Discontinuous Reception) mode to save energy. We plan to explore ways to reduce energy consumption when cellular data interfaces are used for both uplink (e.g., video uploading/sharing) and downlink (e.g., video streaming) mobile accesses.

With the increasing popularity of Internet streaming services as well as the increasing demand for high quality video streaming to the heterogeneous computing devices and/or mobile devices, the server side also faces significant challenges in providing good mobile streaming services. Such challenges include demands for lower response times, higher throughput, and a requirement to transcode and store huge collections of videos. We plan to investigate modifications to the basic Internet streaming service architecture and improve Internet streaming services under these challenges. As a first step, we plan to study buffer cache replacement and prefetching policies that consider block access patterns associated with video accesses. By thorough analysis of these patterns, we plan to design a caching policy specifically tailored to improve the video service response time. In addition to buffer cache in the main memory, we also plan to investigate strategies that allow flash memory to be used effectively as a cache between the main memory and the hard disks for video servers. Flash memory is known to have fast random read throughput and low power consumption but poor random write performance and can only be erased a limited number of times.
Therefore, we plan to explore strategies that combine effective flash memory cache replacement policies with appropriate writing schedules to reduce the number of random writes and erasures.
Bibliography
Bibliography


Curriculum Vitae

Yao Liu received a Bachelor of Science degree in Computer Science from Nanjing University in 2007. She joined Volgenau School of Engineering at George Mason University to begin PhD studies in Computer Science in 2007. She was one of two recipients of the Volgenau School of Engineering Outstanding Graduate Award in 2012.