An Energy Efficient HTTP Adaptive Streaming Protocol Design for Mobile Hand-held Devices

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Tanjil Hossain

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Internet traffic generated from mobile devices has experienced a huge growth in the last few years. With the increasing popularity of streaming applications in mobile devices, video traffic generated from mobile devices is also increasing. One of the big challenges of streaming applications on mobile devices is the energy intensive behaviour of such applications. Energy management has always been a critical issue for mobile devices. A wireless network interface consumes a significant portion of the total system energy while active. During video streaming, the network interface is kept awake for a long period of time. This causes a large energy drain. There are several research works focused on reducing energy consumption during video streaming on mobile devices.

HTTP adaptive streaming is gaining popularity as a method of video delivery because of its significant advantages in terms of both user-perceived quality and resource utilization. By using rate adaptation via changes in the requested video version, it adapts to varying network available capacity. There are several research work that aim to increase the performance of rate adaptation. None of the previous works have focused on reducing energy consumption during HTTP adaptive streaming.

In this thesis, an energy efficient HTTP adaptive streaming protocol is designed. The new protocol uses an efficient buffer management approach and a three step bitrate selection mechanism. The proposed protocol is implemented by modifying the Adobe OSMF player version 1.6. Performance evaluation of the new protocol is carried out by running a number of experiments in both a lab environment and three real world environments. The experimental results show that the proposed protocol is able to achieve high amounts of sleep time (by more than an estimated 70% for WiFi and more than 35% for 3G/EDGE) and reduce energy consumption during data transfer. It can also reduce data wastage by 80% in case of playback interruption in the video playback.

Acknowledgements

I would like to take this opportunity to thank and express my gratitude to the people who helped me and made the successful completion of this thesis possible.

First and foremost, I would like to express my genuine gratitude and sincere appreciation to my supervisors Dr. Derek Eager and Dr. Dwight Makaroff who helped me all the way from the beginning of my study at University of Saskatchewan. When I started my program, I had very little idea about doing research. Both of my supervisors guided me in the right directions from the beginning of my program and helped me understanding the art of research by explaining everything in details and even answering my silly questions without being annoyed. They provided me with lots of support, encouragement, motivation and ideas. They extended their helping hands whenever I needed any suggestions. I also appreciate their efforts in correcting my thesis which took a lot of their valuable time and yet they were patient. I could not have imagined having a better guidance for my M.Sc. study.

Besides my supervisors I would like to thank the rest of the members of my thesis committee: Dr. Nadeem Jamali, Dr. Chanchal Roy, and Dr. Francis M. Bui for their suggestions and insightful comments.

I am very thankful to my roommates and friends here in Saskatoon who provided unconditional support and encouragement for the successful completion of my thesis.

Last but not the least, I would like to express my sincere gratitude to my parents and siblings who were always there for me. I would have been lost without them.

Contents

List of Tables

List of Figures

List of Abbreviations

3G	Third Generation (mobile communication system)
AAC	Advanced Audio Coding
AHDVS	Akamai High Definition Video Streaming Service
APV	Average Playback Version
CAM	Constantly Awake Mode
CDN	Content Delivery Network
CPU	Central Processing Unit
CST	Cellular Sleep Time
DASH	Dynamic Adaptive Streaming over HTTP
DR	Download Ratio
EDGE	Enhanced Data for GSM Evolution
EWMA	Exponential Weighted Moving Average
GPRS	General Packet Radio Service
HD	High Definition
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
LCD	Liquid Crystal Display
MAMP	Macintosh, Apache, MySQL and PHP
MBI	Multi Bitrate Information
MHD	Mobile Hand-held Device
MPEG	Motion Picture Experts Group
NAT	Network Address Translation
NIC	Network Interface Card
OS	Operating System
OSMF	Open Source Media Framework
OTT	Over the Top
P2P	Peer to Peer
PDA	Personal Digital Assistant
PS	Playback Smoothness
PSM	Power Save Mode
RAM	Random Access Memory
RTCP	RTP Control Protocol
RTSP	Real Time Streaming Protocol
RTT	Round Trip Time
SMP	Strobe Media Playback
SR	Switch Ratio
SWF	Small Web Format
TCP	Transmission Control Protocol
TIM	Traffic Indication Map
TTFB	Time To First Byte
UDP	User Datagram Protocol
VBR	Variable Bitrate
WCDMA	Wide band Code Division Multiple Access
WLAN	Wireless LAN
WNIC	Wireless Network Interface Card
WST	WiFi Sleep Time

Chapter 1

INTRODUCTION

Rapidly advancing Internet infrastructures have enabled high bandwidth video streaming applications [6]. These applications are being deployed in a variety of end user devices, from desktops to laptops to mobile hand-held devices. Video traffic, excluding video exchanged through peer-to-peer (P2P) file sharing, occupies more than 50% of Internet Protocol (IP) traffic.[1] One of the main reasons behind the rapid growth of Internet traffic is the mobile traffic generated from different mobile hand-held devices.

There are many types of mobile hand-held devices available now. Smartphones and tablets are the most common mobile devices available today. A mobile device is dependent on a battery for energy. Unfortunately, battery capacity is severely restricted by constraints on the size and the weight of the device. Since enhanced battery life helps to ensure a satisfactory user experience [30], energy efficiency is an important attribute of mobile hand-held devices.

In the last few years, mobile devices have experienced a vast expansion in functionality. Communication intensive applications are considered the most energy intensive applications for mobile devices, because such applications keep the wireless radio (WiFi or cellular) active during most of the application's lifetime. Radios consume a significant amount of system energy when active and can cause the battery to rapidly drain. Due to the communication-intensive nature of video streaming, in particular, its energy consumption is very high ([6], [11], [28]).

Internet video applications use different video delivery techniques to transfer data from server to client. Among them, the most popular video delivery technique is streaming, wherein the client can begin playback before all of the video data has been received. Typically, the contents are delivered to a client by a unicast connection from the content provider, either using a proprietary streaming protocol running directly on top of TCP or UDP, or using HTTP over TCP. The HTTP-based streaming approach known as HTTP adaptive streaming is a content delivery approach in which clients are able to adaptively control the quality of the video they receive. In this approach, multiple encodings of the same video are provided, each divided into small "segments". The client selects the next segment to retrieve based on network conditions, buffer state, and the goal of maximizing

[1] `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html`, accessed 10-August-2012

"quality of experience".

In this thesis, a new protocol is proposed and evaluated for reducing mobile hand-held device energy consumption during HTTP adaptive streaming. The experimental results show that the proposed protocol is effective in reducing energy consumption. It also reduces the amount of data wastage in the case of playback interruption by the user early in the video playback. The energy efficiency and video playback performance with the proposed protocol are evaluated under different network conditions by running experiments in a variety of locations. Some variants of the proposed protocol are also investigated, so as to determine the impacts of important protocol design choices.

1.1 Mobile Hand-held Devices

A mobile hand-held device (or simply "mobile device") is a small, hand-held computing device with a variety of functionalities. Mobile hand-held devices typically have a touch or non-touch display screen and sometimes a mini keyboard. There are many types of mobile devices. Mobile phones, smartphones, PDAs, pagers and Personal Navigation Devices (e.g. GPS) are the most common among them.

Current mobile devices are not limited to only basic telephony functionalities, but typically also include high speed wireless network capabilities, and sufficient processing and memory capabilities to run enterprise, multimedia, and gaming applications. Mobile devices can support storing of personal data, surfing the web, sending short messages (SMS) and/or multimedia messages (MMS), checking e-mail, sending instant messages (IM), audio or video calling, watching streaming videos, taking pictures and other functionalities.

A typical mobile device can be viewed as having four architectural levels: hardware, operating system (OS), middleware, and applications [27]. The most energy-intensive components at the hardware level are the CPU, the Wireless Network Interface Cards (WNICs), and the Liquid Crystal Display (LCD). Most mobile devices are equipped with WiFi and cellular radio, Bluetooth, and GPS in order to get Internet connectivity, connect with other Bluetooth supportive devices, and to get geographic position information, respectively. The main power source of these devices is a lithium-ion battery. The next higher level is the OS. The OS can access physical devices through well-defined driver interfaces. The OS supports the execution of various software applications known as apps. Next is the middleware level. This level acts as a communication medium between the OS and the application layer. It provides cost effective and transparent access to proprietary services and resources for mobile device applications. Finally comes the application level, which consists of the applications themselves.

Smartphones and tablets are the most common and widely-used mobile devices. These devices are popular because of their small size and advanced functionalities. According to a report pub-

lished in the year 2012 by IAB Canada,[2] 85% of Canadians use cell phones and 45% of them are smartphone users. According to the US Digital Media Usage report,[3] half of the mobile users in the United States used a smartphone in the year 2011 and usage will increase 18.4% in the year 2012. As well, a 62.8% increase in the number of tablet users compared to the year 2011 is forecast, due to the increasing popularity of the iPAD in the year 2012 throughout the world. From the rapid growth of mobile devices, it is clear that people are becoming more dependent on such devices. Performance improvement of mobile hand-held devices has become an important research issue.

1.1.1 Energy Consumption In Mobile Hand-held Devices

Mobile hand-held devices are dependent on their battery for energy. Battery capacity is severely restricted by constraints on the size and weight of the device. In the last few years, such devices have experienced a vast expansion in their functionality. The rapid evolution of mobile device functionality has led to use of communication and computation intensive applications, increasing energy consumption of the device. Therefore, the demand for more effective means of energy management has correspondingly increased. As enhanced battery life helps to ensure satisfactory user experience [30], energy efficiency is an important quality in mobile devices.

As noted previously, the CPU(s), LCD, and wireless NIC(s) are considered the three major energy consuming components of mobile hand-held devices. LCD energy consumption is dependent on system settings. On the other hand, the energy usage of a NIC depends on the extent of its usages. From an investigation [20] using an HTC Magic phone,[4] it was found that the WiFi radio consumes 38mW during sleep mode. On the other hand, while active it consumes more than 700mW, and up to more than 1000mW depending on transfer rate. Compared to the WiFi radio, the CPU consumes only a small amount of energy during a data transfer operation.

There are several power management modes used in IEEE 802.11 standard WiFi radios, with corresponding variations in power consumption from mode to mode. Typical WiFi devices support at least two power modes: a high power Constantly Awake Mode (CAM), and an energy-saving Power Save Mode (PSM), in which the radio periodically wakes up to receive data [37]. Many WiFi devices today also implement a technique known as adaptive PSM [22], where the device switches between PSM and CAM based on some heuristics.

[2] `http://www.iabcanada.com/wp-content/uploads/2012/04/IABCanada\_MobileInCanada\_041012\_FINAL.pdf`, accessed 13-July-2012

[3] `http://www.csmediagroup.com/what-to-expect-in-mobile-for-2012-2/`, accessed 13-July-2012

[4] `http://www.adobe.com/products/flashmediaserver/pdfs/FlashMediaServer3_WhitePaper_ue_v1.pdf`, accessed 15-August-2012

1.1.2 Mobile Data Traffic

Mobile web access has established itself as one of the most popular and widely-used mobile device services. It grew 280% globally during each of the years 2009 and 2010 and is expected to double annually up to the year 2015.[5] Due to portability and easy Internet access, people are becoming reliant on mobile web access for their everyday life. In some countries, the number of mobile web users is almost as large as the number of PC-based Internet users [12]. A large portion of mobile data traffic is streaming video [16].

Widely-deployed WiFi networks on university campuses, in business enterprises, public utilities, and residential homes has made mobile Internet usage more pervasive. According to a report released by comScore, more than one third of all mobile traffic was generated via WiFi in the United States in August 2011.[6] Tablets, which typically require a WiFi connection to access the Internet, are also generating a significant amount of mobile traffic by using mobile broadband access.

1.2 Video Streaming

Video streaming is a popular, and bandwidth intensive mobile device application. In 2011 65% of tablet users watched short-length streaming videos on their tablet while half of them watch on-demand TV shows, full-length movies, and live broadcasts.[6] Video streaming has become one of the most demanding applications on the Internet over the last three to four years. It already accounts for more than half of the aggregate Internet traffic [2].

With Over the Top (OTT) service for on-line video and audio, content is delivered directly from provider to viewer without the application-level involvement of an Internet Service Provider (ISP).[7] With OTT services, content is delivered to clients through unicast connections from content provider servers or Content Delivery Network (CDN) nodes. OTT video can be classified into distinct categories, including user-generated content (such as YouTube videos), professionally generated content from studios and networks that promotes commercial offerings and programming (such as from ABC.com and Hulu), and movies and TV series (such as provided by Netflix and Apple TV). There is also a significant amount of growth of managed video services such as video-on-demand cable TV, and IPTV. Such services are run by an ISP over its managed network using unicast or multicast delivery, and maintaining quality of service (QoS) features.

Progressive Download has also become very popular for video streaming due to its simplicity. In this mechanism, an entire video is requested using a single HTTP request and delivered over a TCP

[5] `http://www.ericsson.com/thecompany/press/releases/2010/03/1396928`, accessed 13-July-2012

[6] `http://www.comscore.com/Press_Events/Presentations_Whitepapers/2011/Digital_Omnivores`, accessed 13-July-2012

[7] `http://www.pace.com/global/our-thinking/over-the-top-services-ott/`, accessed 13-July-2012

connection. Playout starts after some threshold amount of data has been buffered. YouTube is the third most popular website on the Internet according to its Alexa[8] traffic rank. It delivers 4 billion hours of video per month by using this approach.[9] One major drawback of this approach is its lack of adaptivity [2]. Some content providers like YouTube provides the opportunity to select a version from multiple available versions with different resolutions. In most cases, it is very difficult to choose the most appropriate version without prior knowledge of the available bandwidth. Because network bandwidth fluctuates, the selected version may become inappropriate and the viewer might experience occurrence of frequent video freezes and rebuffering. In order to resolve these problems, a new HTTP-based streaming approach has been developed, called HTTP adaptive streaming [25], where multiple versions of the same video are provided, with different bitrates, each divided into small segments. The player is able to adaptively choose from which version the next segment should be selected, according to dynamic network and player buffer conditions to maximize the quality of the user experience.

The use of HTTP offers several service benefits for video streaming. HTTP is one of the most widely-used protocols. Most of the currently available devices support HTTP in some form. Due to the pull-based nature of the HTTP protocol and its use of TCP, video streams can easily traverse firewalls and NAT devices. HTTP servers are more scalable as no HTTP state information is kept at the server. Finally, the caching infrastructure developed for web objects can be exploited, possibly with each video segment being individually cacheable [2].

1.3 Thesis Motivation

One of the main challenges in mobile device video streaming is its energy intensive behaviour. During video streaming, typically the network interface is awake almost all the time which results a large energy drain. Energy consumption of mobile devices has received significant attention from researchers. Researchers have proposed different proxy based methods ([4], [15], [33], [37]) as well as client centric methods ([35], [40], [46]) for energy efficient mobile device data transfer. There are also several research projects ([1], [5], [6], [11], [21], [28], [43], [47]) that are concerned specifically with energy efficient multimedia streaming.

HTTP adaptive streaming has gained popularity recently. There are several commercial players in the market that use this technique. Some of the research work on HTTP adaptive streaming ([2], [8], [9], [13]) have tried to measure the bitrate adaptation efficiency of the current HTTP adaptive streaming players during fluctuations in bandwidth availability. Other work ([14], [23] , [25], [44]) have proposed different bitrate adaptation techniques for HTTP adaptive streaming.

[8] `http://www.alexa.com/topsites`, accessed 13-July-2012
[9] `http://www.youtube.com/t/press_statistics`, accessed 15-July-2012

Most of the previous work have focused on the efficiency of bitrate adaptation techniques. None of these work has considered the problem of reducing energy consumption during HTTP adaptive streaming. The main motivation of this thesis is to reduce client-end energy consumption during HTTP adaptive streaming.

1.4 Thesis Objectives

The objectives of this thesis are as follows:

- Design an energy efficient HTTP adaptive streaming protocol.
- Implement the proposed protocol.
- Evaluate the performance of the proposed protocol by running experiments in different locations under varying network conditions in order to show the achieved performance improvements.

1.5 Thesis Findings

Major findings of the thesis are as follows:

- By using an energy efficient HTTP adaptive video streaming protocol, it is possible to achieve substantial WNIC sleep times during data transfer (in the experiments reported in this thesis, up to more than 70% of the data transfer time for WiFi, and 35% of the data transfer time for cellular), at the cost of somewhat reduced video quality.
- By deploying an efficient buffer management policy, it is possible to increase the sleep time during data transfer (by up to 6.4% of the data transfer time in the experiments reported in this thesis, for WiFi, and up to 22% for cellular). Also an adaptive buffer management policy can reduce data wastage (in the scenario considered in the thesis experiments, by up to 82% compared with using a fixed size 30 second buffer).
- A three step bitrate selection mechanism is proposed. Measurement experiments verified that this mechanism is able to predict the likelihood of the WNIC being able to enter sleep mode during a data transfer operation with high accuracy (up to 95% in some of the scenarios considered).

1.6 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes background knowledge about mobile device energy consumption patterns and different types of media streaming techniques,

including HTTP adaptive streaming. Chapter 3 discusses previous work done in the areas of HTTP adaptive streaming, mobile device energy efficiency, and energy efficient multimedia streaming. Chapter 4 defines the proposed energy efficient HTTP adaptive streaming protocol and describes its important characteristics. Chapter 5 describes the experimental methodology, while experimental results are discussed in Chapter 6. Finally, Chapter 7 summarizes the thesis and discusses some possible future research directions.

Chapter 2

BACKGROUND

Energy efficiency is an important attribute of a mobile device. Communication-intensive applications like video streaming consume a large amount of system energy while operational. The system resources used to stream the data vary, depending on the data transfer rates, the signalling protocol used, and the compression technique applied to the raw video data. There are several video streaming protocols available today. This chapter presents an overview of mobile device energy consumption patterns in Section 2.1. Section 2.2 describes the MPEG video compression technique. In Section 2.3, different video streaming protocols are described. Finally, Section 2.4 provides a description of HTTP adaptive streaming protocols and different adaptive streaming players.

2.1 Mobile Device Energy Consumption

In a mobile hand-held device, three components are major sources of power consumption: display (LCD), CPU, and network hardware (WiFi or 3G radio) [28]. Each of these will be described in the remainder of this section.

The backlight is responsible for most of the energy drain of the display. The LCD power consumption increases with the increase of brightness. Since backlight brightness is controllable by predefined user settings, setting a lower brightness level can reduce the LCD energy consumption.

Wireless communication is one of the primary causes of high energy consumption in mobile devices. Wireless technologies such as WiFi and 3G/EDGE radio have been identified as among the most energy intensive components [10], [45].

The CPU energy consumption depends on several factors, such as the input data to be processed, the processing time of the software, and the CPU internal architecture [28]. The CPU energy consumption is highly dependent on the applications running on it and their data inputs. For example, in the case of multimedia streaming applications, multimedia decoding is responsible for most of the CPU energy consumption [28].

Figure 2.1 shows the energy consumption rate of some major components of a gPhone HTC Magic series smartphone during both WiFi and cellular data transfer for two separate experiments where data was transferred to a HTC magic phone from a server by using a TCP socket connection

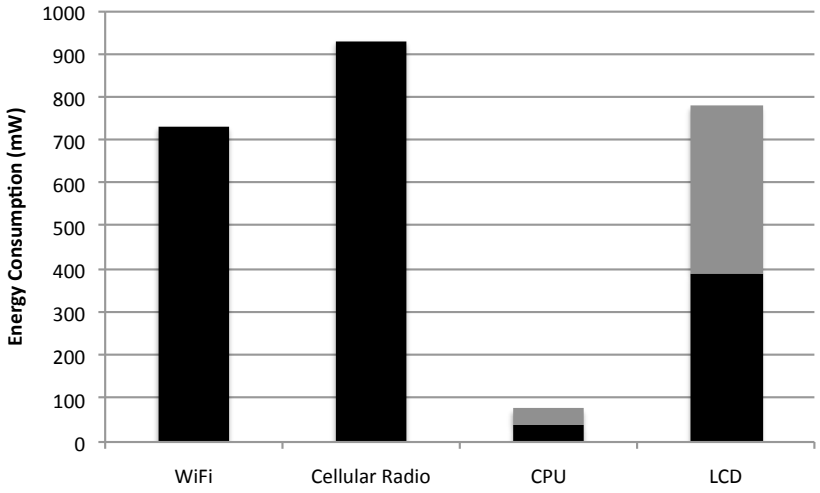


Figure 2.1: Energy consumption rate by the major components of an HTC Magic Smartphone [20]

[20]. In the first experiment, WiFi radio was used for data transfer, whereas cellular radio was used in the second experiment. The gray bar defines the range of energy consumption. Energy consumption was measured by using PowerTutor,[1] an application for Google phones for measuring the power consumed by major system components of a smartphone. The energy consumed by the CPU and the LCD vary based on transfer rate and brightness, respectively. From this graph, it is clear that the WiFi and cellular radios consume a considerable amount of system energy while active.

The LCD power consumption rate for HTC Magic phone varies from 392 mW to 781mW for brightness level from 30 to 255 [20]. Typically while the device is in the idle state, it consumes a very small amount of energy, ranging from 150 mW to 200 mW.

Carroll *et al.* [10] produced a breakdown of power distribution to CPU, memory, touchscreen, graphics hardware, audio, storage, and various networking interfaces (WiFi and GPRS) for Openmoko Neo Freerunner[2] smartphone. According to the authors, during the suspended state (in which only the communication processor is active), the GSM antenna consumes most of the system energy to maintain connection with the network. On the other hand, during idle state (in which the device is active but no application is running), graphics consumes the highest amount of system energy. In the active state, the energy consumption of network components (WiFi and GPRS) is much higher than CPU and RAM energy consumption. For WiFi, CPU and RAM energy consumption is comparatively higher than when using GPRS due to high throughput. However, GPRS consumes more power than WiFi by a factor of 2.5.

[1] `http://ziyang.eecs.umich.edu/projects/powertutor/`, accessed 15-August-2012
[2] `http://www.openmoko.com/freerunner.html`, accessed 15-August-2012

Scenario	CPU	WiFi	LCD	Total system energy
WiFi OFF	>= 24 mW	0	392 mW	>=415 mW
WiFi ON but sleep	>=35 mW	38 mW	392 mW	>=460 mW
WiFi ON and transmitting	>=45 mW	733 mW-738 mW	392 mW	>=1170 mW

Table 2.1: Energy consumption rate by major smartphone components in different scenarios in an HTC Magic smartphone [20]

Xiao *et al.* [45] investigated energy consumption when using a mobile YouTube player on Nokia N95.[3] This work compared the energy consumption using Wide-band Code Division Multiple Access (WCDMA) and Wireless LAN (WLAN) access technologies for different mobile YouTube use cases. Their results showed that when using WLAN, less energy was consumed than when using WCDMA during video download and upload. The total energy consumption of playback during progressive download was equal to that of download followed by playback from cache, using either WCDMA or WLAN access technologies.

Network technologies consume a significant amount of system energy, especially during data transfer. Energy usage can be classified into three categories according to Balasubramanian *et al.* [7]: energy required to switch to the high-power state (ramp energy), energy consumption during data transfer (transmission energy), and energy spent in high-power state after the completion of the transfer (tail energy). In Section 2.1.1. and Section 2.1.2 respectively, WiFi and 3G/EDGE radio energy consumption is examined in more detail.

2.1.1 WiFi Energy Consumption

When in WiFi Power Saving Mode (PSM), there is a very small cost of maintaining the network association. The WiFi radio can start data transmission without any additional ramp energy consumption. The amount of tail energy consumption is also small, since the time spent in the high-power state following a data transfer is relatively short (approximately 1 second in HTC Magic phone). Most of the energy consumed with WiFi is transmission energy, which can rise up to 737 mW depending on the transfer rate [20]. While in sleep mode, WiFi consumes only 38 mW.

Table 2.1 shows the energy consumed rate in an HTC magic smartphone by the CPU, the WiFi, and the LCD, as well as total energy consumption rate, in different scenarios, when the CPU is

[3] `http://www.nokia.com/gb-en/support/product/n95/`, accessed 15-August-2012

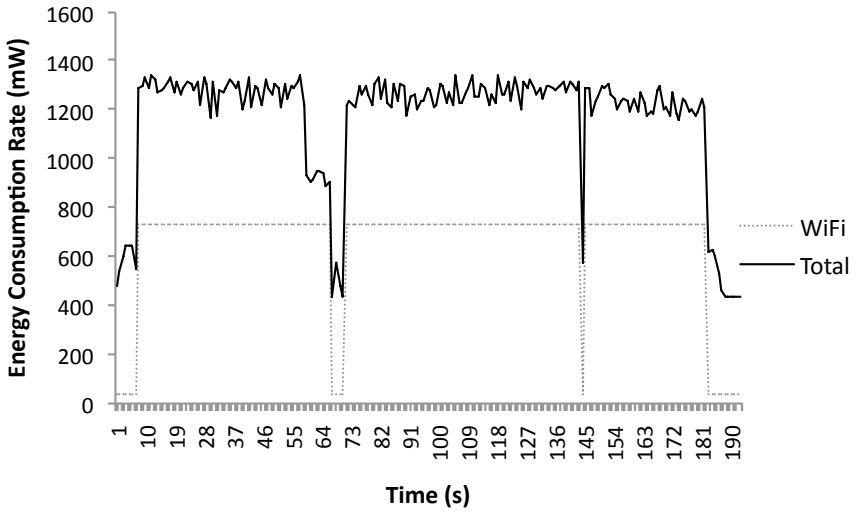


Figure 2.2: WiFi and system total energy consumption rate in an HTC Magic smartphone [19]

minimally active and the LCD is set on the lowest level of brightness [20]. Energy consumption was measured by using PowerTutor. While the WiFi is on but not transmitting, it consumes a small amount of energy compared to that when transmitting, although this small amount of energy is still more than 8% of the total system energy. On the other hand, during transmission, the WiFi consumes over 62% of the total system energy.

Figure 2.2 shows the WiFi and the system total energy consumption rate during a WiFi data transfer, and minimal activation of the CPU and the LCD [19]. Energy consumption was measured by using PowerTutor. With the start of data transfer, the WiFi energy consumption experiences a rapid growth, which makes a large contribution towards total system energy consumption. During data transfer, there are some drops in the WiFi energy consumption due to delays in data packet arrivals. The WiFi energy consumption drops permanently when data transfer ends. In this figure, it is clear that the total system energy consumption follows almost the same pattern as the WiFi energy consumption.

The WiFi energy consumption does not depend to any significant extent on transfer rates. Table 2.2 shows the WiFi energy consumption per second, for different transfer rates [20]. For different data rates, the consumption varies only between 733 mW and 734 mW.

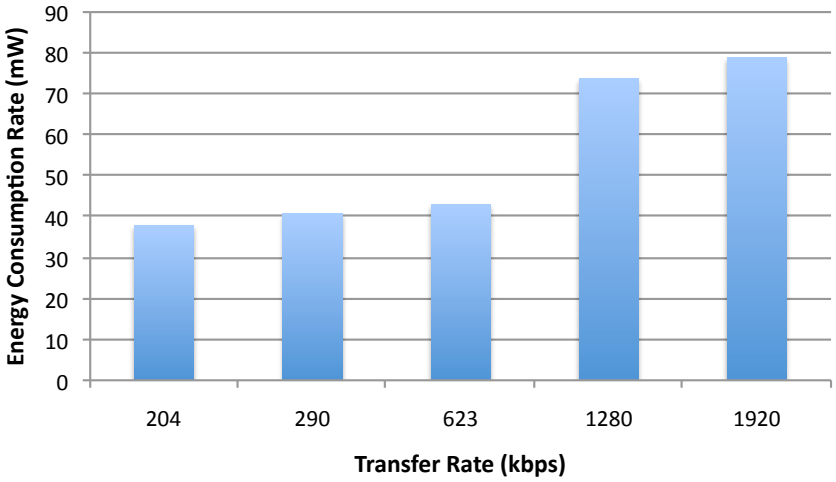


Figure 2.3: CPU energy consumption rate for different transfer rates [20]

Table 2.2: WiFi energy consumption rate for different transfer rates in an HTC Magic smartphone [20]

Transfer Rate	Energy Consumption Rate
204 kbps	733 mW
290 kbps	733.2 mW
623 kbps	733.81 mW
1.25 Mbps	733.91 mW
1.87 Mbps	733.98 mW

Depending on the transfer rate, the CPU energy consumption varies significantly. It increases directly with the transfer rate. Figure 2.3 shows the trend of increasing CPU energy consumption with increased transfer rate [20]. Since the CPU consumes a small amount of energy compared to the WiFi radio, this variation does not have a significant effect on the total system power consumption. The initial transition between the WiFi idle and awake states increases the CPU energy consumption slightly. This initial energy cost also depends on transfer rates. The WiFi state changes during data transfers have a small impact on the CPU energy consumption.

WiFi Sleep Modes

Mobile devices typically offer different types of sleep modes that allow sub-components of the system to sleep in different states. Most WiFi devices exploit Constantly Awake Mode (CAM) and Power Save Mode (PSM) [37]. With CAM, the radio is always fully powered on. With PSM,

the radio can be powered down, and the Access Point(AP) then buffers packets destined for the device. The AP sends a beacon containing a traffic indication map (TIM) once every *BeaconPeriod* (typically 100ms), to indicate whether or not there are any queued data packets for the mobile device. The client device NIC wakes up to listen for beacons at a fixed frequency and sends out a PS-POLL message to receive buffered data. The AP informs the mobile device whether there are more buffered packets in the AP by setting the MORE bit; otherwise the AP informs the client to stop sending the PS-POLL messages by clearing the MORE bit, if data transfer is complete. A mobile device can be configured to skip *ListenInterval* beacons between listen times. Mobile devices can wake up at anytime to send data without waiting for a beacon [22].

PSM reduces energy consumption but with a significant performance cost. Network latency is increased due to the "buffer at AP and check once in a while" behaviour. This approach may cause significant performance losses in scenerios where low network latency is important (for example online gaming, voice and media streaming, etc.).[4] On the other hand, CAM ensures high performance with no WiFi radio down-time.

Many recent WiFi devices support both PSM and CAM simultaneously [22]. Based on some heuristics (e.g., TIM bit set, reception of a threshold number of packets or lack of network activity for a pre-defined duration), the WiFi radio switches between PSM and CAM. The AP is notified of a transition between PSM and CAM by sending a NULL data frame with the power management bit set to 1 (PSM) or 0 (CAM). This approach may incur some energy loss due to the idle timeout period when transitioning from CAM to PSM [37].

2.1.2 Cellular Radio Energy Consumption

The tail energy consumption with 3G/EDGE radios (i.e., the energy consumed after an active period before the radio is powered down) is higher than with WiFi. Figure 2.4 shows radio and device total energy consumption during 3G/EDGE data transmission for an HTC Magic smartphone [19]. Like with WiFi, the device total energy consumption is strongly impacted by the radio energy consumption. Radio energy consumption can be divided into 4 phases [19]. In the first "ramp energy" consumption phase, which persists for 2 seconds, energy consumption rate is 413 mW. The second phase is the "data transmission energy" consumption phase, where the radio energy consumption rate is more than 900 mW. The length of this phase is equal to that of the data transfer time. The third and fourth phases are "tail energy" consumption phases. In the third phase, the device consumes a comparatively higher amount of energy per second (the same as in the second phase). The device waits in this phase for 5 seconds, in case there is further data transmission activity. In the last phase, the radio operates at a lower power level (413 mW) for 5-8 seconds.

[4] `http://www.comscore.com/Press_Events/Presentations_Whitepapers/2011/Digital_Omnivores`, accessed 13-July-2012

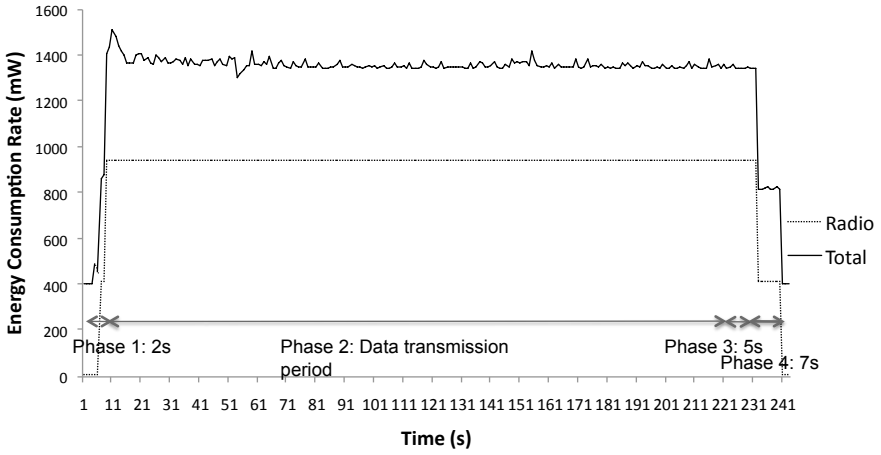


Figure 2.4: Radio and system total energy consumption rate for an HTC Magic smartphone [19]

While in the last two phases, data transmission can be quickly resumed. For the HTC magic phone length of tail phase is 12 seconds [19].

This energy consumption pattern is not unique. Similar behaviours hold across different manufacturers and OSes [7]. For example, for an HTC Touch smartphone with Windows Mobile 6.1, the length of the tail phases is 17 seconds, and that of the ramp energy consumption phase is 1.5 seconds [16].

2.2 MPEG Video Compression

The process of reducing redundancy in video data and converting into a format that consumes fewer resources during storage and transmission is known as video compression. While an encoder converts video into a compressed format, a decoder decompresses it. Nearly all video compression uses lossy compression methods, but the difference between the original and the video after compression and decompression is ideally not noticeable to the viewer. Most of the audio-video content delivered on mobile devices is compressed by using the MPEG compression standard. The MPEG audio-video compression standards are based on complex adaptive encoding and simple fixed actioned decoding techniques [41]. For video, a combination of image and motion compensation is used. The encoder processes a frame of video in blocks of 16 x 16 neighbouring pixels, known as macroblocks. For complex video scenes with more motion, additional data needs to be added to the compressed video, and the encoding is variable bitrate (VBR) [32]. An average bitrate can be

calculated across the entire video, but the actual bitrate will fluctuate.

MPEG defines the notion of a "group of pictures" or GOP [36] within the compressed video. A GOP generally contains three types of frames: Intra coded frames (I-Frames), Predictive coded frame (P -frames), and Bidirectionally predictive coded frame (B-frames). I-frames are larger than P-frames or B-frames, and are intra-coded, meaning that the compression of each pixel blocks is carried out without reference to pixel blocks of other frames [36]. Both P-frames and B-frames are encoded using motion compensation. P-frame encoding uses previous frames, while for B-frames both previous and subsequent frames are used (in all cases, only frames with in the same GOP). During video fragmentation as used in HTTP adaptive streaming, one option is to set the segment length equal to the GOP length.

H.264/AVC is one of the most widely used MPEG-based video compression formats currently available. It is a block-oriented motion-compensation-based codec standard with enhanced motion prediction capabilities and higher video quality with lower bitrate [42].

The MPEG standards group standardized Dynamic Adaptive Streaming over HTTP (DASH), also known as MPEG-DASH[5] in 2011. It is based on 3GPP[6] Adaptive HTTP Streaming (AHS), also adopted by the Open IPTV Forum (OIPF)[7] as a baseline standard.

2.3 Media Streaming

There are different ways to transmit content between different nodes of a network. The communication method used between nodes can depend on the content type, the network conditions, and the delay tolerance of the application. In the case of simple file transfer over a lossy network, the main focus is reliable delivery of the content. In this situation, redundancy is added in order to protect packets against losses, and/or retransmission takes place to recover lost packets. On the other hand, in the case of audio/video media delivery with real time viewing requirements, the main goal is to ensure low latency and jitter, and efficient transmission. In this type of content delivery, occasional losses might be tolerated. There are different media streaming protocols available currently. These can be classified into push-based and pull-based protocols, based on whether the server or client determines what data gets delivered [8].

2.3.1 Push-based Media Streaming Protocols

In push-based streaming protocols, the server initiates packet streaming to the client after a client-server connection is established. The server continues to transmit packets until the client stops or interrupts the session. In this type of streaming, the server maintains session state, which the

[5] `http://mpeg.chiariglione.org/meetings/geneva11-1/geneva_press.htm`, accessed 7-August-2012
[6] `http://www.3gpp.org/About-3GPP`, accessed 7-August-2012
[7] `http://www.oipf.tv/`, accessed 7-August-2012

client can change using a session control protocol. The most widely used session control protocol in push-based streaming is the Real Time Streaming Protocol (RTSP) [39]. In order to perform real-time control of media delivery from the server, the client issues stream control commands, such as play and pause.

Push-based streaming often uses the Real-time Transport Protocol (RTP) [38] for data transmission. In order to monitor transmission statistics and quality of service (QoS), and aid synchronization of multiple streams, RTP Control Protocol (RTCP) [18] is often used with RTP. While RTP carries the media streams, RTCP is responsible for providing QoS feedback and synchronization between media streams. RTP generally runs on top of the User Datagram Protocol (UDP). UDP does not provide any rate control mechanism itself. In this architecture, the client/server intercommunication bitrate does not depend on the underlying transport protocol; rather it depends on the client/server application-level implementation. UDP provides a low latency and best-effort media transmission facility for RTP [8].

Generally, in the case of push-based streaming, in order to match with the client consumption rate, the server transmits contents at the media encoding bitrate. In this situation, the main goal is to keep the client buffer level stable and at a sufficient level for smooth video playback. As the client can only consume data at the encoding bitrate, delivering data at a higher rate may cause unnecessary load on the network. In this regards, push-based delivery ensures optimized network resource usage. On the other hand, due to packet loss or transmission delay the client packet reception rate can drop below the consumption rate. This may cause buffer underflow, resulting in playback interruption.

In the particular situation where playback interrupts occur due to a low transfer rate or high packet loss, resulting in buffer underflows, bitrate adaptation can play an important role. In this type of situation, the server can dynamically switch to a lower bitrate stream. With a lower bitrate stream, the client side media consumption rate reduces, reducing the likelihood of buffer underflow. Similarly, the server can switch to a higher bitrate stream when network condition improve [8].

In push-based adaptive streaming, it is necessary to monitor the achievable transmission rate to the client and compute different network metrics such as the round trip time (RTT), packet loss, and network jitter periodically in order to make stream selection decisions. Clients can make the stream selection decisions on their own, or can transmit network metrics to the server, for the server's use in making the decisions. The main disadvantage of sender-driven rate adaptation is that on the server needs to take the extra burden of selecting an appropriate bitrate for individual clients.

2.3.2 Pull-based Media Streaming Protocols

With pull-based media streaming protocols, the client initiates the transmission by sending a request to the server. After receiving the client's request, the server starts data transmission to that particular client if the request is accepted. The transfer rate between client and server depends on the client and the available bandwidth. Generally HTTP is the most common and widely used protocol for pull-based media streaming.

One of the most popular and widely used pull-based media streaming approaches used today is progressive download. In progressive download, in response to the client's HTTP request to the server, the server starts data transmission. The client tries to pull data from the server as early as possible. Once the client fills up a minimum required buffer level, it starts playback of the media while in the background as it continues content downloading from the server. Playback continues without interruption as long as the download rate is at least as high the playback rate. During playback, if the network condition degrades, the download rate might become lower than the playback rate. In this situation, there may be buffer underflow and the user may experience interruption and jitters in playback.

In order to react dynamically to changes in network conditions, pull-based streaming protocols can use a bitrate adaptation approach known as receiver-driven rate adaptation, in which the client makes rate selection decisions. The main advantage of receiver-driven rate adaption over sender-driven rate adaptation is that the server doesn't need to take on the burden of maintaining receiver connection state information, and collecting and processing network metrics concerning the quality of the client connection.

There are three types of mechanisms for bitrate adaptation to the available bandwidth [14]: 1) transcoding-based [34], 2) scalable encoding (H.264/SVC) based [23] [44] and 3) stream-switching [14]. In the transcoding-based approach, the raw content is transcoded to match with a specific bitrate based on the client request [34]. These algorithms can achieve a fine granularity by throttling the frame rate, video resolution, and compression with the cost of increased processing load and poor scalability as per client basis transcoding is required. In scalable encoding, the raw video is encoded once, and adapted on-the-fly based on the scalability feature of the coder. These two approaches are difficult to use with CDN supported content delivery, since content can not be easily cashed using standard CDN cashing technology. On the other hand, a stream-switching algorithm encodes the raw content into several versions. Based on client side network conditions, the bitrate that best matches the client bandwidth is choosen. The main disadvantages of this approach are the requirements of comparatively higher storage, and coarser stream quality granularity, due to having a discrete set of bitrate versions.

2.4 HTTP Adaptive Streaming

HTTP adaptive streaming uses receiver-driven rate adaptation together with HTTP based delivery.[8] The main goal of HTTP adaptive streaming is to deliver streaming video to the user more efficiently by dynamically switching between different streams of varying quality and size during playback. HTTP adaptive streaming ensures the best possible video quality to the user based on client-side network bandwidth and player buffer conditions. Another important goal of HTTP adaptive streaming is to make video quality smoother by ensuring smooth and unnoticeable transitions between different bitrate qualities without disrupting the continuous playback.

With the rapid growth of technology, high bandwidth network connections equipped with faster-performing client hardware are available today. High-Definition (HD) video transfer over the Internet is very much possible today. Generally, larger video starting at 640 x 480 pixel dimensions and increasing up through 720p towards 1080p is considered as HD web video.[9] With the introduction of HTTP adaptive streaming, users can experience high quality video materials.

One of the biggest challenges for high quality and longer duration video streaming, especially HD video, is the fluctuating Internet connection at the user end. This is a common problem with most network connections and can get worse when several applications generate huge workloads from the same connection. This fluctuating network bandwidth may cause interruption and repeated re-buffering of the high quality streaming video and impacts the user experience if the selected stream bandwidth is unsustainable for the duration of the data transfer. In this situation, HTTP adaptive streaming can play an important role. It detects bandwidth fluctuation and automatically selects the bitrate among streams of available bitrate versions suitable for the current network conditions and switches into that bitrate version. Similarly, some video playback may experience low bandwidth at the beginning of the playback and bandwidth may increase later. In this scenario, HTTP adaptive streaming starts the playback with a comparatively low bitrate version and up-shifts the bitrate version whenever it detects a sustainable increase in network bandwidth.

In some players (e.g. YouTube), there are multiple available versions with different resolutions for the user to choose. Generally most of the users stick with default video format [17]. In this situation, users may experience playback interruption and re-buffering if the default video format is inappropriate for their network connection. HTTP adaptive streaming plays an important role to solve these kind of issues.

User's hardware performance limitations can also interrupt video playback, especially with large-dimension HD video and full-screen playback. If the CPU is not capable of faster video stream decoding, frames may be dropped. In this situation, using lower quality segments will reduce frame

[8] `http://www.adobe.com/products/hds-dynamic-streaming.html/`, accessed 13-July-2012
[9] `http://www.adobe.com/products/hds-dynamic-streaming.html/`, accessed 13-July-2012

drops and ensure better playback quality.

HTTP adaptive streaming maintains several versions of the same video, encoded with different bitrates and quality levels. Video content is partitioned into segments of a few seconds duration of each. A client can request different segments with different encoding bitrates depending on the underlying network bandwidth [2]. As bitrate selection is done on the client's site, this increases server site scalability. The player can also adjust buffer size based on the segment request rate.

Figure 2.5 shows the basic working principle of an HTTP adaptive streaming player. Media content is divided into short duration media segments in the server. Each segment is encoded with various bitrates and can be decoded independently. Initially, once the user initiates the playback in the client side by clicking the play button, the client sends an HTTP GET request for the manifest file of that particular video to the server. In response to the client request, the server sends the manifest file. This file generally contains some basic information about that particular video such as location of the video content, video playback duration, available bitrate versions, video resolution for different bitrate versions, and metadata information. This file may also contain several other types of information which varies from player to player. The HTTP adaptive streaming client then sends the HTTP GET request in order to retrieve a particular segment by using the information of the manifest file. The client initially requests the lowest bitrate version from the server. Based on the network bandwidth, the client further dynamically picks the segment with the right encoding bitrate that matches or is below but closest to the available network bandwidth. In response to the client request, the server sends video content to the client. By playing these segments back to back, the original video is reconstructed at the client side.

It is also possible to incorporate multiple request-response streams in HTTP adaptive streaming [24]. Request-response streams are able to scale with the available bandwidth by increasing the number of concurrent streams. According to Kuschnig *et al.* [24], larger segment sizes more effectively utilize network bandwidth. The disadvantage of using multiple streams is that they quickly saturate the network. Larger segment sizes reduces the frequency of bitrate adaptation.

2.4.1 Available HTTP Adaptive Streaming Players

There are several HTTP adaptive streaming players available in the market, such as Microsoft Silverlight, Akamai Adaptive Streaming video player, Netflix, Apple HTTP live streaming, Move Network, and Hulu. In the following the main features of some of the HTTP adaptive streaming players are summarized.

Microsoft Silverlight Smooth Streaming is an open source player. It is possible to change the basic algorithm of this player and change it as per developers need. Benno *et al.* [9] defined the basic working policy of this player. In the basic deployment, all the chunks are of two seconds duration. In response to a client's request, the server initially sends the manifest file. This file

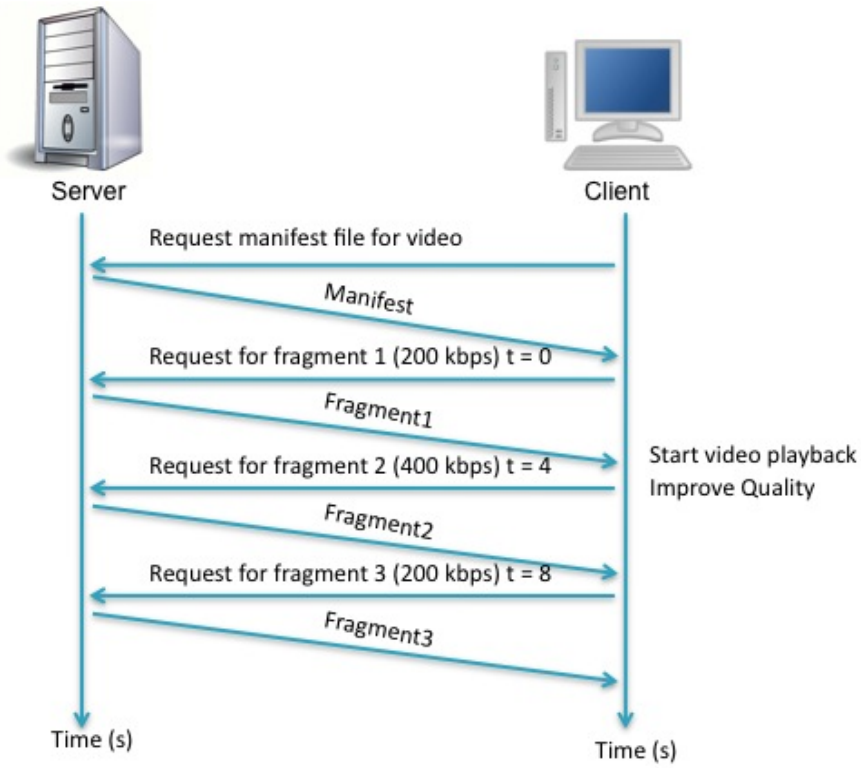


Figure 2.5: HTTP adaptive streaming

contains audio and video file bitrate, resolution, and duration of the requested video. The player generates an HTTP request with content name, bitrate, and segment playback starting timestamp. The client establishes two parallel TCP connections with the server, one for audio content delivery and another for video content delivery [2].

Buffer fullness is defined in seconds. That means it requires three chunks to fill up a 6-second buffer. The maximum buffer size is 20 seconds. This player generally maintains the buffer size in between a lower threshold (12 seconds) and a upper threshold (17 seconds). The player does not request a new segment unless it fully receives the previous segment. Initially, the player tries to fill up the buffer as quick as possible. In this state, the player requests a new segment as soon as the previous segment was downloaded. As long as the player buffer reaches on intermediate state between the lower and upper threshold value, the player enters into steady state. In this state, the player maintains a constant playback buffer and requests segments after certain intervals. The player is conservative in the case of selecting bitrate versions in this state to prevent oscillations across rates. The client measures the bandwidth from the ratio of chunk size and download time of the chunk. In order to get a stable estimation of the bandwidth, it averages 3 samples of the estimation.

The player makes bitrate change decisions based on current buffer size. The bitrate version selection process depends on estimated bandwidth. The player increases the bitrate when the

player current buffer size is higher than upper threshold. On the other hand, the player switches to a lower bitrate version if current buffer size is less than lower bound of the buffer. The player does not request a bitrate version having frame resolutions larger than the resolution of the display window. Control information between client and server is managed by a separate TCP connection.

De Cicco *et al.* [13] described the working principle of the Akamai High Definition Video Streaming Service (AHDVS). It runs over the flash API. When the user selects a video, an HTTP GET request is generated to the server. In response, the server sends a Synchronized Multimedia Integration Language 2.0 (SMIL) compliant file. This file contains base URL of the video (httpBase), the available levels, and corresponding encoding bitrates. Video segments are encoded with the H.264 codec, having 30 frames per second. On the other hand, the audio is encoded encoded with Advanced Audio Coding (AAC), with a bitrate of 128 kbps.

The server encodes the raw video source into different files for different bitrate level. After the initial GET request, all the communication between client and server is carried through POST requests for quality adaptation. POST commands are issued with several parameters like *cmd*, to specify client issued commands to the server or *lvl1*, to specify a feedback variable. Different types of *cmd* can be issued by the client to the server. The *throttle cmd* is issued to specify difference between current buffer and target buffer. The *rtt-test cmd* is issued to request data download in greedy mode. The *SWITCH_UP cmd* and the *BUFFER_FAIL cmd* are issued to increase and decrease bitrate level respectively. The client sends commands with an average interval of 2 seconds. The *lvl1* parameter consists of 12 comma separated feedback variables. These variables include current and desired receiver buffer size, frame rate, estimated bandwidth, data received rate in client, current video level identifier, bitrate, and timestamp.

Netflix is another HTTP adaptive streaming player, which uses the Microsoft Silverlight plugin. According to Akhshabi *et al.* [2], in a similar fashion as Silverlight adaptive streaming, the Netflix player uses two TCP connections with the server. The manifest file format is different from other HDAS players. Initial client-server communication is done over Secure Sockets Layer (SSL) cryptographic protocols. The format of the audio and video segments are Windows Media Audio (WMA) and Windows Media Video (WMV), respectively. Each request to the server is done through a byte range request.

Data transfer starts with the PLAY button click. There is some initial data transfer to check the bandwidth of underlying path. Playback starts when the buffer size fills up to a certain point. If there is any drain in buffer level, playback stops by showing bitrate adjustment message. After a certain level of buffer has filled up, data download resumes.

Finally, Apple HTTP Live Streaming follows a different approach for segment storage. The video content is segmented into several pieces with configurable duration and video quality in the server [13]. Instead of containing all the segment information in a single manifest file, Apple HTTP

Live Streaming maintains separate transport stream files for different bitrate version. There are two levels of transport stream file. The upper level file contains link of the lower level files, which specifies information about a particular bitrate version (e.g. links for individual media segment, sequence number of the segment and others).

2.4.2 OSMF Adaptive Streaming Player

The Open Source Media Framework (OSMF) is a free collection of open-source components for building robust, feature-rich media players and applications for the Adobe Flash Platform. Strobe Media Playback (SMP) is a free, open-source player built with OSMF.[10] In order to get high-quality, full-featured playback experiences, OSMF features easy assembly of pluggable components. As the OSMF player is an open source player, developers can easily change the source code and add different functionality on their own to ensure better user experience. It supports multiple delivery methods including progressive download; plain RTMP and HTTP streaming; RTMP and HTTP Adaptive Streaming; and live streaming with RTMP, HTTP, and RTMFP multicast. The OSMF player has support for most standard media formats, including the following:

- streaming video (FLV, F4V, MP4, MPEG-4, MP4, M4V, F4F, 3GPP),
- progressive audio (mp3), and
- progressive video (FLV, F4V, MP4, MP4V-ES, M4V, 3GPP, 3GPP2, QuickTime/MOV).

When a client makes a request for a particular video in the Adobe OSMF player, a web-page embedded with a SWF file is downloaded in the client. The SWF file provides the control logic for video playback. The control logic is implemented by using Adobe ActionScript, an object-oriented programming language for the Adobe Flash Player[11] and Adobe AIR[12] runtime environments. ActionScript controls the control mechanism of video playback. It loads, play, pause, and seek external video file (FLV or MPEG-4 file). The SWF file is reusable for different video content.[13]

The SWF (Small Web Format) file is a binary file format. It is designed and compressed to be small for efficient delivery over the web.The *.swf* extension is a proprietary format of Adobe Flash. Adobe creative suits products - Adobe Premiere, Adobe Flex, Adobe Captivate and more are capable of generating *.swf* files.

Like other HTTP adaptive streaming players, the OSMF player also contains a manifest file for each media element. Each manifest file is an XML document that represents only a single piece of media. For more than one distinct piece of media file, such as an album of music videos, multiple

[10] `http://www.osmf.com/`, accessed 15-August-2012
[11] `http://www.adobe.com/products/flashplayer.html`, accessed 15-August-2012
[12] `http://www.adobe.com/products/air.html`, accessed 15-August-2012
[13] `http://www.adobe.com/devnet/flash/learning_guide/video/part02.html`, accessed 15-September-2012

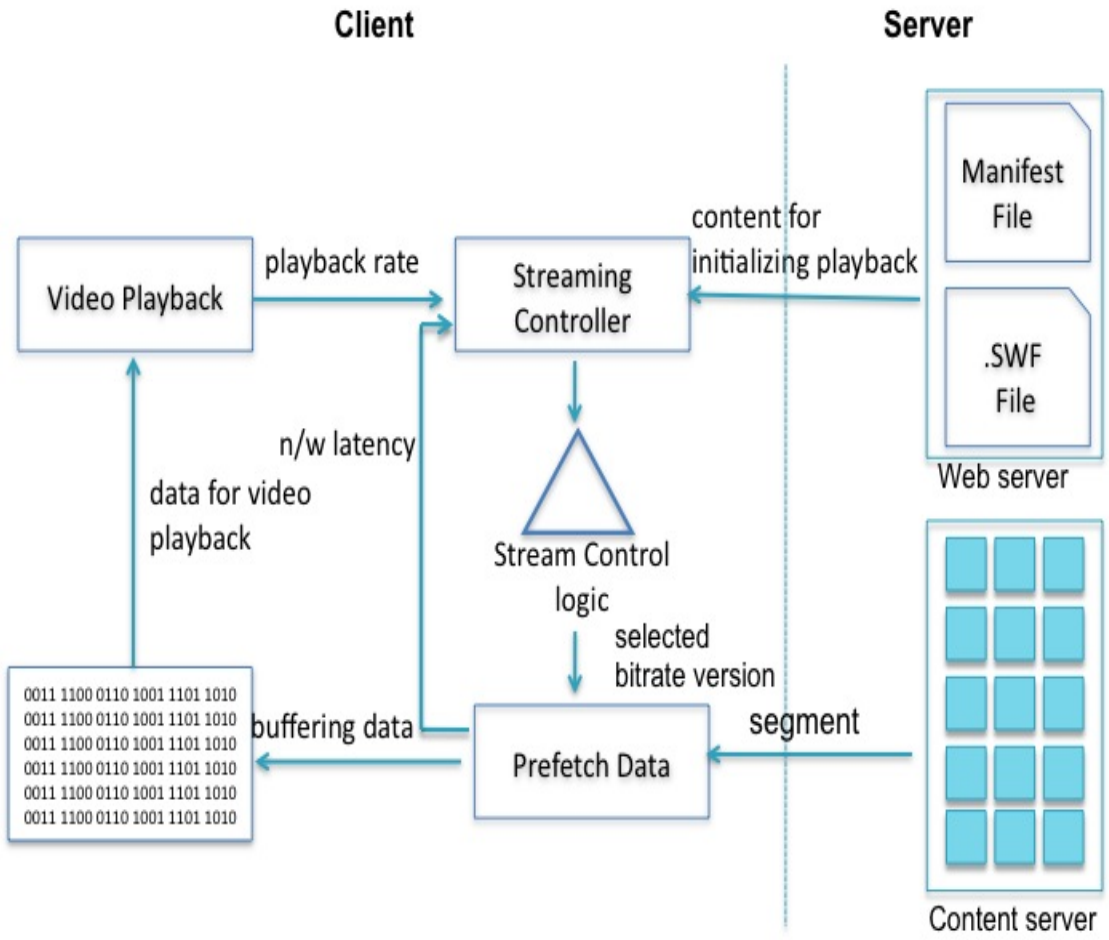


Figure 2.6: Adobe OSMF player version 1.6 stream control mechanism

manifest files or other encapsulation formats like SMIL or RSS/Atom are required. Each manifest file includes the location of the media, duration, DRM authentication information, media bootstrap information, multi-bitrate information (MBR), metadata, and other relevant informations. The manifest file downloaded into the client machine while client selects the media file for playback.

While user selects the play button, the client generates HTTP requests to download video content in the form of segments, from the server. The default segment size is 4 seconds.

Adobe OSMF Player Version 1.6 Stream Control Logic

Figure 2.6 shows the stream control logic of Adobe OSMF player Version 1.6. Initially, a SWF file and a manifest file (F4M file) is downloaded into the client from the web server. At this point, the client gets all the required information for video playback. The client uses this information while making the bitrate selection decision by using stream control logic. The stream control logic comes into play after downloading the manifest file. It makes the bitrate selection decision based on network latency, and playback rate. The stream controller gets the playback rate information

from the player and network latency information for pre-fetched data. While selecting the bitrate, the stream controller takes mainly two parameter values into consideration: Download Ratio (DR) and Switch Ratio (SR).

The Download Ratio (DR) is the ratio between playback time of last downloaded segment and download time for that segment. This ratio defines whether the current network condition is sufficient for supporting playback rate. If the playback rate is higher than the download rate, there will be possibilities of buffer underflow. In this situation, the player may freeze and there is a possible interruption in playback. In order to maintain a smooth playback, stream controller always tries to maintain Download Ratio value greater than 1 by switching lower bitrate versions.

In the OSMF player, the DR value is calculated from the ratio of segment duration and download time of that segment. The OSMF player API provides functionalities to calculate the DR. Segment duration is calculated by an event handler after downloading the segment and download duration is calculated by another event handler, which tracks download start time and end time.

The Switch Ratio (SR) defines whether player can sustain a new bitrate version by maintaining a standard buffer size. SR is calculated from the ratio of current bitrate and another bitrate for which SR is being calculated.

There are separate request-response pairs for each HTTP segment download request generated to the content server. Requests are sent in a sequence and each request is generated after downloading the previous segment. The HTTP requests generated by the client contain only the segment number followed by encoding type and bitrate with corresponding segment location. All the content is delivered through a single TCP connection. If the TCP connection stays idle for long period, a new connection is stablished. After prefetching, data content is buffered into buffer space and the player uses this buffered data for video playback.

Algorithm 1 shows the OSMF player bitrate selection mechanism. Bitrate selection decision is made based on DR and SR values. If the DR value is less than 1, that means the playback time for the segment is less than download time and player needs to decrease the bitrate version. If the DR value is very low (less than SR), the lowest level bitrate is selected; otherwise the player decreases the bitrate value by one level. On the other hand, if DR value is greater than 1, in that case the player increase the bitrate version if a higher bitrate version is available. The player switches to a new bitrate version for which the DR value is still greater than the SR.

The stream controller also checks the frame drop during bitrate selection. If for a particular segment, player experiences a high frame drop, the player switches into lower bitrate version.

There is also another level of control checking during bitrate selection. The stream controller checks the previous history of downloaded bitrates while bitrate up-shifting. If it finds any previous downgrade from a particular bitrate version, the stream controller skips that bitrate version for the next 30 seconds. If downshift occurs thrice for a particular bitrate version, the stream controller

Algorithm 1 ADOBE OSMF Player Version 1.6 bitrate adaptation algorithm

Bitrate down-shift

if $DR < 1$ & $DR < SR[current - 1]$ **then**

switch to lowest rate immediately (even if there's an intermediate that might work)

else if $DR < 1$ & $DR >= SR[current - 1]$ **then**

Bitrate goes down to one level.

end if

Bitrate up-shift

if $DR >= 1$ & $DR < SR[current+1]$

OR no available rate is higher than current **then**

No change in current bitrate version.

else if $DR >= 1$ & $DR > SR[current+1]$ **then**

Player switch to bitrate N where $DR > SR[N]$

end if

blocks bitrate version for next 5 minutes. This is known as bitrate suspension.

After the bitrate selection decision is made by the stream controller, the player goes to prefetch data state. In this state, the player downloads data blocks and buffers it. Buffer occupancy is defined in seconds. There are two buffer parameters defined in the player: initial buffer and additional buffer. The total buffer is the sum of initial buffer and additional buffer. Playback starts while the player fills up the initial buffer. Players generally download data unless the total buffer fills up and pause data download once buffer fills up. In this point, playback continues and consumes the buffer. No additional data is downloaded until buffer goes down to initial buffer value. In the default setting, the initial buffer is set to 2 seconds with 4 seconds additional buffer.

2.5 Summary

In this chapter, energy consumption pattern of different components of the mobile device, especially the network components are described. An overview of MPEG video encoding techniques with a description of different video streaming techniques are provided. HTTP adaptive streaming with a description of different HTTP adaptive streaming players is explained next. In the end, a detailed description of an open source player, the Adobe OSMF player used later in this thesis and its stream control logic is provided to form the context for the implementation work carried out in Chapter 5.

Chapter 3

RELATED WORK

The rapid evolution of mobile computing and mobile multimedia traffic is enabling high quality video streaming on mobile devices. HTTP adaptive streaming is becoming increasingly popular due to its use of HTTP and its rate adaptation functionality with the changing network conditions. Energy efficient data transfer becomes increasingly important with the increasing popularity of multimedia streaming. Related work in both of these areas is reviewed in this chapter.

The remainder of this chapter is organized as follows. Section 3.1 discusses rate adaptation performance in current commercial protocols and gives brief descriptions of recently proposed HTTP adaptive streaming rate adaptation algorithms. Section 3.2 provides an overview of previously proposed energy efficient data transfer mechanisms. Section 3.3 presents previous work focused on energy efficient multimedia streaming.

3.1 Bitrate Adaptation in HTTP Adaptive Streaming

3.1.1 Performance of Rate Adaptation in Commercial HTTP Adaptive Streaming Players

There are different HTTP adaptive streaming players available in the market. Some recent work ([2], [8], [9], [13]) describes the rate-adaptation performance of different HTTP adaptive streaming players.

Akhshabi *et al.* [2] studied the behaviour of two commercial HTTP adaptive streaming players (Microsoft Smooth Streaming and Netflix) and an open source HTTP adaptive streaming player (Adobe Open Source Media Framework (OSMF) version 1.0). The performance of these HTTP adaptive streaming players was evaluated under three operating conditions: unrestricted available bandwidth (available bandwidth is persistent and high), persistent available bandwidth variations (available bandwidth changes after long interval and persists in a certain level for long duration), and short term available bandwidth variations (periodic "spikes" on available bandwidth). *Wireshark*[1] was used for packet tracing and *DummyNet*[2] was used for controlling the network bandwidth.

[1] `http://www.wireshark.org/`, accessed 15-August-2012
[2] `http://info.iet.unipi.it/~luigi/dummynet/`, accessed 15-August-2012

As would be expected, the Smooth Streaming player was found to perform better under persistent and high network bandwidth. In this situation, the player maintains a large buffer and quickly reaches the highest sustainable bitrate. The player was found to be conservative with respect to bitrate switching. It maintains a safety margin between the available bandwidth and its requested bitrate. A long observation period is required before switching the bitrate. The player also avoids large bitrate changes to ensure smooth user viewing experience. The main disadvantage of this conservative bitrate selection process is delayed response during short duration bitrate fluctuations which may cause buffer underflow in the case of sudden drop in bandwidth.

The Netflix player was found to be more aggressive than the Smooth Streaming player. This player always aims to achieve the highest possible video quality even with the cost of additional bitrate changes. Netflix maintains a large buffer (Akhshabi *et al.* [2] measured 300 s worth of buffered video in 75th second) and downloads large chunks of audio and video in advance. If the player buffer is very close to full, Netflix sometimes switches to a higher bitrate than the available bandwidth.

For measuring the performance of Adobe OSMF, OSMF version 1.0 was used. Substantial oscillation in the bitrate version was observed especially when the available network bandwidth was smaller or very close to the highest available bitrate of the media. The authors conjectured that the rate adaptation algorithm is tuned for short variations in the available bandwidth. The reason behind the observed bitrate oscillation may be frame drops by the flash player which trigger changes in bitrate.

DeCicco *et al.* [13] investigated the performance of the Akamai video streaming player under fluctuating network bandwidth. The experiment equipped a client machine with *NetEm*,[3] emulator for controlling network bandwidth and delay; the client received video content from an Akamai HD server. Different network condition was simulated by introducing TCP greedy flow by using *iperf*,[4] a tool for bandwidth measurement. In order to investigate the bitrate adaptation behaviour of the player, three different scenarios were considered: bandwidth capacity changes following a step function (bandwidth capacity increased to higher capacity from a lower capacity after a certain duration), bandwidth capacity varies as a square wave (bandwidth capacity switches between a higher capacity and a lower capacity within a certain interval), and a shared link (a concurrent TCP flow shares a high capacity link). It was found that it takes a large transient time (approximately 150 seconds) to fully adapt to a sudden bandwidth change in a controlled environment. A sudden drop in network bandwidth may cause a short interruption in video playback due to a large actuation delay. The player is also found to be capable of adapting the bitrate version when sharing a link with a concurrent TCP flow.

[3] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem` accessed accessed 25-August-2012

[4] `http://iperf.sourceforge.net/`, accessed 25-August-2012

Benno *et al.* [9] measured the performance of the Rate Determination Algorithm (RDA) used in Microsoft Smooth Streaming. An experimental testbed was designed by using *Dummynet* as network emulator to control bandwidth and delay. The *ttcp*[5] command was used to generate greedy TCP flows and the *Harpoon*[6] was used to generate dynamic TCP traffic.

As the player tries to maintain buffer size between upper and lower thresholds, sometimes it struggles to select the proper bitrate version as the player selects the bitrate version based on predefined advertised bitrates and that may not match the actual bitrate of the video segment. Experimental results found that the player selected quality level has often 20% lower than the actual achievable throughput. The impact of caching for HTTP adaptive streaming was also investigated. Their experiment results showed that, the network can assist to improve HTTP adaptive streaming performance by reducing data transfer delay and by providing achievable throughput information to the client. By using HTTP adaptive streaming aware cache policy, it is possible to improve the cache hit rate, which will reduce delay and help to maintain stability in bitrate.

3.1.2 Proposals for New Rate Adaptation Algorithms

There are some recent works that propose new rate adaptation algorithms in order to better achieve the maximum sustainable bitrate version under different network conditions.

Liu *et al.* [25] described four desirable attributes of rate adaptation algorithms. First, the rate adaptation method should deploy a metric to identify whether the current bitrate version matches with the available bandwidth. Second, the rate adaptation technique should prevent buffer under flow and overflow. Third, the bitrate version should not fluctuate too much, even when the current available bandwidth lies between two adjacent bitrate ranges. Finally, the segment duration should be chosen so as to reduce HTTP overhead.

Liu *et al.* [25] proposed a rate adaptation algorithm for HTTP adaptive streaming that makes bitrate version change decisions based on segment fetch time (SFT). The SFT is the period of time from when the GET request is sent for a specific segment until the last bit is downloaded for that segment. The ratio between media segment playback time and SFT is used to detect network congestion. In order to provide a smooth playback, the SFT should be lower than the segment playback time. Otherwise, the user may experience periodic pauses in video playback. Bitrate increases are step-wise (i.e., go to adjacent bitrates only). Advantages of a cautious step-wise approach include reducing playback interruption, preventing buffer underflow in the case of a sudden bandwidth drop, and reducing initial buffering time. An aggressive decrease in bitrate may occur when the bitrate of the current version doesn't match with end-to-end network capacity. The

[5] `http://www.cisco.com/en/US/tech/tk801/tk36/technologies_tech_note09186a0080094694.shtml`, accessed accessed 25-August-2012

[6] `http://cs.colgate.edu/~jsommers/harpoon/`, accessed accessed 25-August-2012

performance evaluation of the proposed algorithm was done using the ns2 simulator.[7] Background traffic was generated by using an ns2 traffic generator. Simulation results show that this algorithm can efficiently detect network congestion and quickly adopts the optimal bitrate level.

Xiang *et al.* [44] considered rate adaptation of SVC video in wireless networks and modelled the rate adaption problem as a Markov Decision Process (MDP) in an attempt to ensure better user-perceived quality of experience (QoE). A reward function was defined in order to balance the tradeoff between the average video quality and playback smoothness according to a defined weighting parameter. In order to measure QoE, three metrics were defined: interruption ratio, average playback quality, and playback smoothness. Dynamic programming was used in order to solve the MDP problem. A testbed was designed to show the performance improvement of the proposed algorithm over the Rate Adaptive Algorithm proposed by Liu *et al.* [25].

De Cicco *et al.* [14] proposed a Quality Adaptive Controller (QAC) for live adaptive video streaming. The controller tries to maintain the buffer level as stable as possible to match the video bitrate with the available bandwidth. Unlike in other HTTP adaptive streaming protocols, in QAC, measuring, control, and actuation take place at the server instead of the client.

The authors compared their proposed QAC server with Akamai High Definition Video Streaming Service (AHDVS) in four network bottleneck scenarios: 1) available bandwidth changes following a step function in a bottleneck link, 2) available bandwidth varies in a square wave in a bottleneck link, 3) sharing bandwidth with greedy TCP flow in a bottleneck link, and 4) two video streams sharing the same bottleneck link. The experimental results show that the proposed QAC is able to adapt more quickly than AHDVS and achieves a better match with network bandwidth. The main problem of this approach is increased server side complexity as the server needs to maintain the information for each user to perform rate adaptation.

Mok *et al.* [29] proposed a QoE-aware DASH system, named QDASH, composed of two modules: QDASH-abw and QDASH-qoe. QDASH-abw uses probing to detect the highest quality level compatible with current network conditions and is implemented in a measurement proxy. Traffic between the client and the server flows through the proxy. Available bandwidth is measured by RTT variations. QDASH-qoe helps the client to select the most suitable video quality level. Assessment results run on 24 subjects showed that transitioning through intermediate quality levels when a large quality drop is required is favoured by the users.

3.2 Energy Efficient Data Transfer

Reducing mobile device energy consumption during data transfer has been the subject of much research. In the remainder of the section, the work is divided into 2 subsections. In the first

[7] `http://www.isi.edu/nsnam/ns/`, accessed 09-August-2012

subsection, energy saving data transfer mechanisms initiated by the client side are described. The second subsection covers the proxy-based energy saving approaches.

3.2.1 General Client-centric Approaches

Several studies [35], [40], [46] have focused on increasing the burstiness of the traffic by sending zero-sized TCP receive window messages to the server. TCP traffic shaping can be done such that the streaming traffic flowing to a client appears in a predicable pattern, such as periodic bursts. The client can sleep between the bursts.

In a traditional TCP connection, the client acknowledges the the receipt of data by sending a TCP acknowledgement segment. These segments contain the receiver window size which defines the amount of additional data the client can hold in its buffer. Yan *et al.* [46] proposed an approach in which the client initially sends a segment with a receiver window size of zero to the server to delay the data sending process from the server. Later, the client sends a segment with a large receiver window size, so that the server can send the data in a burst.

Tan *et al.* [40] proposed the PSM-throttling protocol for better utilization of 802.11 power saving mode (PSM). During TCP data transmission, the transmission rate of media traffic may be constrained by a server side data transfer mechanism referred to as bandwidth throttling. In PSM-throttling, the client identifies bandwidth throttling by measuring the TCP throughput. In the case where the effective data transmission rate is lower than the available Internet bandwidth due to server side bandwidth throttling, PSM-throttling uses this unused bandwidth to reshape the traffic into periodic bursts with an average throughput the same as the server transmission rate. Packet arrivals can be accurately predicted at the client side in this approach. Raja *et al.* [35] also proposed a similar client-centric energy efficient protocol.

3.2.2 General Proxy-based Approaches

Using a proxy in the access point (AP) for controlling data flow and allowing the client NIC to sleep has been proposed by many researchers [15], [33], [37]. In proxy-based approaches, the proxy is generally used to reshape TCP traffic flow and generate some gaps between data transfers. These gaps allowed the mobile device to enter sleep mode.

In order to support data-oriented and QoS-sensitive applications such as VoIP, the IEEE 802.11e standard includes Unscheduled Automatic Power Save Delivery (U-APSD) [33]. In this approach, QoS-sensitive applications get high priority. Whenever the AP receives any frame from the client device, either an actual data frame or a NULL trigger frame, the AP uses a Transmit Opportunity (TXOP) mechanism to send buffered data with high priority. A TXOP is a bounded time interval during which a station delivers possible highest amount of frames. Perez-Costa and Camps-Mur [33] proposed an adaptive U-APSD approach which extends 802.11e U-APSD method. In this

approach, frame arrivals at the AP are predicted and the client device sends a NULL frame to retrieve the buffered frames.

Catnap [15] aims to combine tiny gaps between data blocks generated due to slow access links in the network into meaningful sleep intervals for mobile devices. A "middlebox" is integrated into the home wireless AP to batch packets and send them in a burst so that the mobile device can sleep between bursts. After receiving a client request, the server sends a response to the Catnap proxy that schedules the best time to initiate wireless transmission to the client and maximize the client sleep time. Generally, the wireless time slot is scheduled as late as possible, ensuring no increase in transfer time. The Catnap scheduler can also operate in batch mode, where the scheduler concatenates multiple small slots into bigger slots at a cost of increased transfer completion time, to decrease energy consumption.

With Network-Assisted Power Management (NAPman) [37], CAM and PSM traffic are isolated from each other and an energy-aware scheduling algorithm is used to minimize energy consumption. Another contribution of NAPman is the idea of a virtualized AP. If there are many clients waiting for incoming packets from the AP and the AP can transmit only one packet at a time, every client may have to wait for a long time to receive their packets. This will cause wastage of energy. To prevent this scenario, the NAPman AP advertises several virtual APs that does not overlap, through beacons. The PSM clients are associated to the appropriate virtual AP and isolate PSM clients from each other.

There are some potential disadvantages in using a proxy based approach. Deploying the proxy causes additional cost. There are also some other disadvantages of having a proxy. One problem concerns the proxy placement. It may be difficult to choose a location for a proxy that could serve large numbers of clients. If there is substantial variance in the distance between clients and the proxy, the rate at which clients receive data will also vary. There may be also some bottleneck links between the proxy and the server, and between the client and the proxy as well. These bottleneck links may hamper the task of achieving high energy efficiency on mobile hand-held devices.

3.3 Multimedia Streaming Approaches

Proxy-based approaches have also proposed for multimedia streaming. Chandra and Vahdat [11] proposed use of both server-side and client-side proxies to batch packets from various streaming applications so that the client WiFi radio can sleep during the intervals between receiving these batched packets. The server-side proxy informs the client-side proxy of the next scheduled data burst. The client-side proxy send the Wireless NIC to a lower power sleep state between scheduled data transfers. Their approach is application dependent and is not compatible with the 802.11 standard as it ignores the beacon interval which is the basis of the standard power saving mechanism.

Mohapatra *et al.* [28] proposed hardware-based architectural optimization techniques together with high level operating system and middleware approaches for achieving power savings. All communication between a mobile device and the server is routed through a proxy server, that can transcode the video stream in realtime based on the residual energy availability information gained from the mobile device. In this architecture, the mobile device measures energy availability information from underlying architecture and sends the information to the proxy. Based on the client's feedback, the proxy performs some middleware adaptation of traffic flow. They also proposed video quality adaptation based on device feedback. Video quality adaptation mechanisms were not described in their work.

Mohapatra *et al.* [28] found that power can be saved if data packets are transmitted in bursts by switching the receiver to energy-saving sleep mode between bursts, but that aggressive use of bursty transmission may adversly impact the performance of other traffic flows. They proposed a suboptimal solution where in a layered multimedia coding scheme is deployed with a packet scheduler providing bursty traffic with adaptive burst length and decreasing priority order of packets in each burst.

Anastasi *et al.* [4] also proposed a proxy-based energy efficient protocol for video streaming named the Real Time and Power Saving (RT-PS) protocol for wireless clients. This streaming protocol requires prior knowledge of audio/video frame lengths, client buffer size, and the wireless network bandwidth. The protocol reacts to network bandwidth changes. The proxy needs to be updated about the network and the playback conditions using communication channel from the client to the proxy that conveys information concerning the throughput as well as available free space in the client buffer from time to time. Due to this additional communication traffic, this approach may cause network overload. The throughput is also dependent on the client buffer size. When the client buffer size is small, the proxy sends shorter data packets to the client. On the other hand, the proxy generates a long traffic burst when the client buffer has a large amount of free space. Background traffic also plays an important role in the RT-PS protocol especially in a congested network. In a network congested by background traffic, energy consumption will increase. A communication bottleneck between the client and the proxy may cause buffer underflow in the client.

Van Antwerpen *et al.* [5] also proposed a proxy-based, client feedback centric energy-aware wireless multimedia data transfer mechanism. Based on client feedback, the proxy performs a feedback-based realtime transcoding of the video streams.

Adams and Muntean [1] proposed Adaptive-Buffer Power Save Mechanism (AB-PSM). In this mechanism, data is buffered at the access point before reaching the mobile device temporarily, so that the mobile device can enter sleep mode in order to achieve high energy savings. Data from the multimedia server is sent in a burst to the client. An additional buffer, referred as the Application

Buffer, is introduced in the AP. The Application Buffer effectively "hides" the packets from the client so that when the client station wakes up at the beacon interval, it doesn't find any waiting traffic and returns to sleep. The client will be acknowledged about the buffer data after the beacon interval has been passed.

Although most of the research work in this area relies on bursty data transfer for energy savings, according to Korhonen *et al.* [21] traffic peaks caused by bursty transmissions are harmful for the network performance. Clustered packet arrivals may cause congestion in routers or overflows in transmitter buffers, leading to packet losses. Bursty data transfer also doesn't guarantee that packets will arrive clustered close together at the receiver due to variation in transport delay, referred to as jitter. Predicting the packet arrival times at the receiver is more difficult in this case. The authors proposed an adaptive energy-saving streaming mechanism that adjusts the burst length based on network congestion. This approach provides a trade-off between power efficiency and congestion avoidance.

Zhu *et al.* [47] proposed Rate-based Bulk Scheduling (RBS) in order to save power and ensure QoS. The RBS scheduler serves or suspends a flow based on the amount of buffered data in the client buffer. Data transfer from the server is suspended for flows with enough buffered data until the buffer is drained so that the wireless NIC can sleep long enough to offset the impact of the state transition delay. When some flows are suspended, other active flows sharing the channel can obtain more bandwidth and fill up this buffers more quickly. In their work the authors used Audio-on-Demand(AoD) as a case study to evaluate the performance of RBS.

Bagchi *et al.* [6] proposed a pull-based multimedia delivery system implemented by using a Fuzzy Adaptive Buffering (FAB) algorithm. In this method, data blocks are prefetched from the server based on several control information including network bandwidth and playback rate. The FAB algorithm is designed using two threads, named the player_thread and the prefetch_thread. The player_thread is responsible for video playback and consumes the data stored in the media buffer. On the other hand, the prefetch_thread retrieves data from the media server and fills up the media buffer. If the playback_thread doesn't find enough available data in the media buffer, it enters into idle mode and causes a pause in playback. The prefetch_thread tries to maintain the buffer such that there will be some sleeping periods in which the buffer has been filled and further data prefetch can be suspended until the buffer drops a certain amount. These sleep intervals are dynamically determined by using the FAB algorithm, using network bandwidth and playback rate for decision making.

Wu *et al.* [43] proposed a two-level framework for cooperative media streaming for mobile handheld devices. In the first level, content is prefetched by mobile devices to form a *virtual prefetching zone* along the direction of the movement of the client device. In the second level, a client is chained to another partner client in which content is prefetched earlier. The requesting client can receive

subsequent segments directly from the partner without server intervention. The dynamic chaining method among the client ensures maximum utilization of cached data.

Earlier prefetching can cause repeated prefetching of the same segment in the case of rapid change of chain due to client movement. Clients may also not be willing to upload content or become a member of the chain, as it may cause a huge energy drain in the client device. Clients may also be concerned about privacy issues in non trusted P2P environments. This method is also prone to failures due to improper management of the chain because of frequent movements of mobile clients and costly repetitive search by the AP.

3.4 Summary

In this chapter bitrate adaptation technique used in different commercial HTTP adaptive streaming protocols are described. It also described recently proposed different rate adaptation algorithms for HTTP adaptive streaming. An overview of previously proposed energy efficient data transfer mechanisms are provided with details description of previously proposed different client-centric approaches and proxy-based approaches. In the end, different proposed energy efficient multimedia streaming techniques are depicted.

Chapter 4

PROPOSED ENERGY EFFICIENT HTTP ADAPTIVE STREAMING PROTOCOL

The main contribution of this thesis is to design, implement, and evaluate performance of an energy efficient HTTP adaptive streaming protocol. This protocol works in the application layer and tries to ensure energy efficiency during video playback. During video playback, the player requests segments from the server not only based on current network condition, but also ensures lower energy consumption by sending network interface card into sleep mode during data transfer.

This chapter addresses the design and architecture of the proposed protocol as well as the techniques used to implement it. Specifically, Section 4.1 describes the design consideration for developing the protocol, and Section 4.2 presents the design of the protocol in details addressing buffer control mechanism, and bitrate selection mechanism, respectively.

4.1 Design Considerations

The HTTP adaptive streaming protocol generally focuses on providing the user with better viewing experience by reacting to network conditions. Depending on underlying network conditions, a player can request different segments. With better network conditions, the user can experience better picture quality with the cost of higher amount of data downloaded. Though player downloads lower bitrate quality versions in slow network connection, data transfer period is still high due to lower transfer rate. On the other hand, if players react on periodic increase in network bandwidth by increasing bitrate versions, amount of data download will also increase. At the same time, frequent change in bitrate versions may hamper the smooth viewing experience by the users.

During device data transfer period, the network interface activates. As discussed in Chapter 2, a longer active time of network interface card causes a higher amount of energy consumption. These higher amounts of data downloaded may cause rapid reduction of battery life in devices where battery size and life is limited, like mobile hand-held devices. As energy efficiency is one of the major challenge in mobile hand-held devices, ensuring energy efficiency is very important for mobile hand-held device applications, especially data transfer intensive applications like video streaming.

As discussed in Chapter 3, several HTTP adaptive streaming performance improvement protocol have been proposed. However, none of them focused on energy efficiency. In this thesis, the goal is to design an energy efficient HTTP adaptive streaming protocol for mobile hand-held devices. The proposed protocol not only considers network conditions while making segment selection decisions, but also tries to ensure sufficient sleep time during data download. It makes conservative bitrate selection decisions by not overreacting to bandwidth increases, while reacting immediately when there is a reduction in network bandwidth. This bitrate version change process is similar to the response to congestion in TCP. During bitrate selection, instead of considering only current network condition, the protocol also considers previous segment download transfer rate to get an overall measure of the network conditions. Restrictions in bitrate increase are implemented if consecutive down-shift have been experienced. During an increase, the bitrate version shift is done stepwise which also helps to increase playback smoothness. An efficient buffer policy is introduced, where content is downloaded such that there are some interval in between data transfer periods. In order to reduce data loss during early playback interruption, buffer size is made adaptive. The buffer size changes, based on the playback duration and the network condition.

Ensuring enough sleep time during the bitrate selection and conservative bitrate selection process reduces the length of the data transfer period during video playback, based on user needs. The NIC awake time is reduced, which increases the battery life. Also, lower bitrate selection reduces probability of buffer underflow and playback interruption. There are several benefits of step-wise bitrate version increase. Rapid change in picture quality is restricted, and initial buffering time is reduced due to slow and steady change in bitrate versions. Also, an adaptive buffer size policy reduces the amount of data wasted when playback is interrupted by the user. It also increases the probability of better video quality in further playback by not filling up player buffer in very beginning of the playback. In order to deploy the proposed energy efficient HTTP adaptive streaming protocol, only client side player version up-gradation is required. There is no change required in the server side.

4.2 Design of Proposed HTTP Adaptive Streaming Protocol

In this section, the design of the proposed protocol is described. Before going into details, some frequently used terms are introduced in Table 4.1. These terms are mostly used during the description of buffer control and bitrate versions selection process.

Figure 4.1 shows the working principles of the proposed HTTP adaptive streaming protocol. Similar to Adobe OSMF player, in the beginning of the playback, a *manifest file* and an *SWF file* is downloaded from the web server. The SWF file and the manifest file were already discussed in Section 2.4. The stream controller controls the buffer using buffer control logic. The stream control

Table 4.1: Terms with descriptions

Term	**Description**
Video Length (V_T)	Total duration of a video
Current Buffer (B_T)	Size of the buffer in seconds
Initial Buffer (B_I)	Minimum amount of buffer that player needs to fill up before playback starts
Maximum Buffer (B_{Max})	Upper limit of buffer time
Minimum Buffer (B_{Min})	Lower limit of buffer time
Playback Time (P_T)	Total duration of video playback
Seek Time (S_T)	Playback time in the duration bar after playback forwarding by the user
Maximum Player Buffer (PB_{Max})	Maximum buffer level at which data download is paused
Minimum Player Buffer (PB_{Min})	Minimum buffer level at which data download is resumed
Maximum Buffer reach time ($BMAX_T$)	Video playback duration for PB_{Max} to reach B_{Max}, when $B_I == B_{Min}$. Defined as percentage of video length
Sleep Threshold (S_thresh)	Threshold for expected sleep possible time
EWMA_DR_UpShift	Exponential Weighted Moving Average (EWMA) of Download Ratio for bitrate up-shift
EWMA_DR_DownShift	EWMA of Download Ratio for bitrate down-shift
Estimated Sleep Time (EST_i)	Estimated sleep time for i-th segment
Suspension Period (SP_t)	Suspension period provided to a particular bitrate version after first failure
Maximum Allowed Failure (F_{Max})	Maximum number of times bitrate failure is allowed before increasing suspension period
Longer suspension period (SP_T)	Suspension period provided to a particular bitrate version when number of failures exceeds F_{Max}

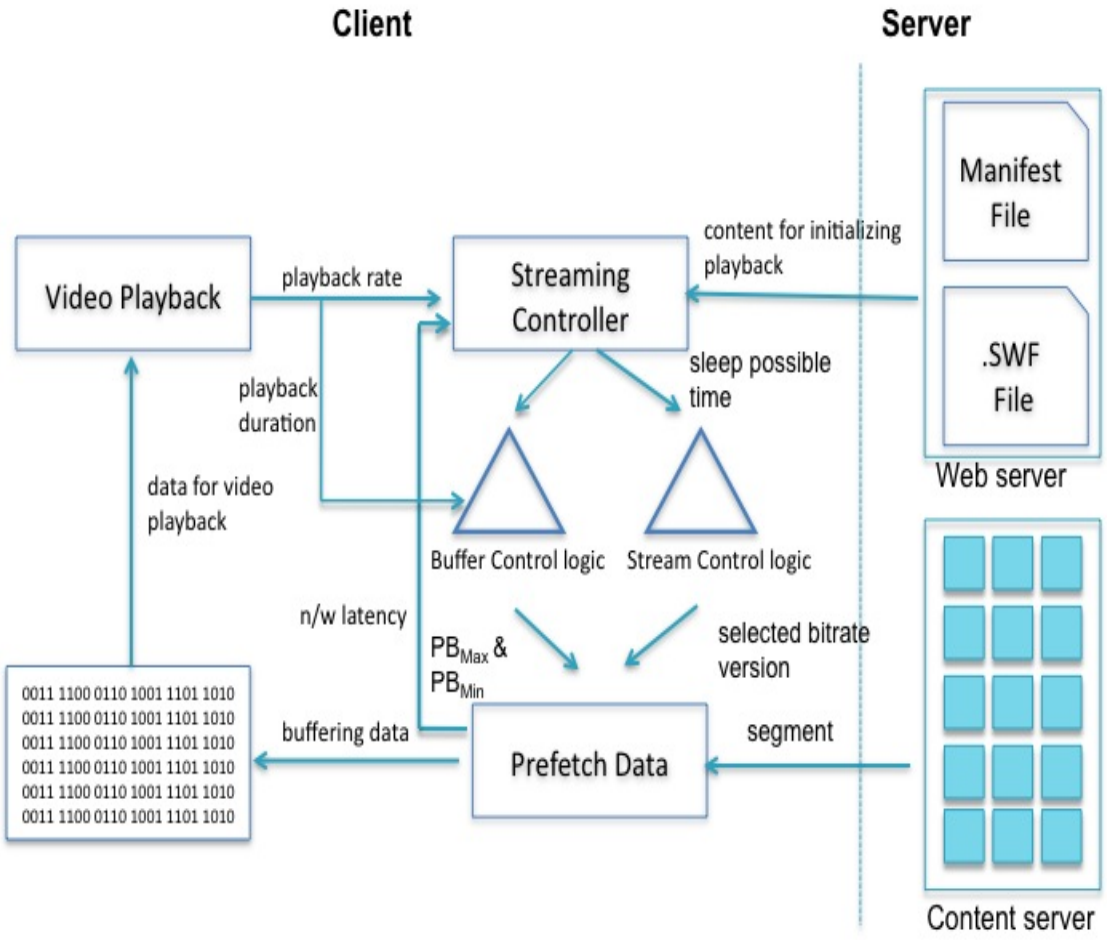


Figure 4.1: Proposed HTTP adaptive streaming protocol

logic is used to select the bitrate version. Playback starts with the lowest available bitrate version and PB_{Max} is set equal to B_I.

In this stage, the stream controller plays an important role by selecting the buffer parameter values using the buffer control logic. Unlike other commercial HTTP adaptive streaming players, the PB_{Max} and the PB_{Min} values are not fixed and are modified depending on current playback duration and network conditions, respectively. The stream controller determine PB_{Max} and PB_{Min} values. The stream controller also selects bitrate version using the stream control logic.

The player starts downloading the selected bitrate version segments to fill up the player buffer and starts video playback when initial buffer fills up. The initial playback delay helps the player to partially fill up the buffer so that the player can consume data from the buffer. The player downloads data until buffer value reaches PB_{Max}. Once it fills the buffer up to PB_{Max}, the player pauses data download. In this stage, only the playback continues, and no data is downloaded until buffer decreases to the PB_{Min} buffer value. Once the buffer reaches the PB_{Min} value, data download resumes and both playback and data download continue simultaneously. This process continues throughout the rest of the playback.

4.2.1 Buffer Control Mechanism

In the proposed energy efficient HTTP adaptive streaming protocol, buffer space is measured in seconds. In other commercial and open-source HTTP adaptive streaming protocols, maximum player buffer time is fixed. In the proposed model, it is adaptive and changes with time. The PB_{Max} value changes based on playback time duration within B_{Max} and B_{Min} buffer values. PB_{Min} also changes based on the network condition.

Different HTTP adaptive streaming players maintain different buffer size. In Microsoft smooth streaming player, the maximum buffer size is 20s [2]. On the other hand Akhshabi *et al.* [2] mentioned a large playback buffer in Netflix player. The Adobe OSMF player version 1.6 maintains a small buffer size of 6s. In all these players, maximum buffer size is fixed throughout the video playback.

In the proposed protocol, the buffer size calculation is done in the beginning of each segment download. PB_{Max} and PB_{Min} are variable. PB_{Max} changes over time until it reaches B_{Max}. Once it reaches B_{Max}, there is no other change in this parameter value unless there is any playback forwarding (resetting the playback point) takes place.

At the beginning of the playback, PB_{Max} is kept comparatively low, still it is bigger than PB_{Min}. Based on video playback, PB_{Max} increases with playback duration. During video playback, if video is played without adjustment to the playback point for a duration of at least $BMAX_T$ (or somewhat earlier if B_I is greater than B_{Min}), then PB_{Max} reaches B_{Max}. If there are any playback forward/backward experiences (user sets the playback point to a new position by forwarding

or backwarding playback bar), B_{Max} is set to B_I again in that particular playback point and again increases from that point considering seek point as new playback start point.

$$PB_{Max} = \min(B_I + \frac{(B_{Max} - B_{Min}) \times (P_T - S_T)}{BMAX_T \times V_T}, B_{Max}) \tag{4.1}$$

In the proposed algorithm, $BMAX_T$ is set to 20%. According to Finamore *et al.* [17], in the case of YouTube, 60% of video playbacks are cancelled at 20% of their duration.

On the other hand, the PB_{Min} also changes based on the network condition. Generally, the PB_{Min} is kept equal to B_{Min} while Download Ratio is greater than 1. On the other hand, when the Download Ratio goes below 1, it is set to $2 \times B_{Min}$

Maximum buffer size is kept moderate sized in the proposed model (30 s). On the other hand, minimum buffer size is very small (2 s), so that there is a comparatively high wait period after buffer fill up.

4.2.2 Bitrate Selection Mechanism

Bitrate selection is important in HTTP adaptive streaming to ensure better viewing experience. In OSMF player version 1.6, the network condition is the main criteria used during bitrate selection [2]. In our proposed model, not only the network condition but also device expected sleep time is considered. During bitrate version selection, the player tries to select the bitrate version for which Sleep Threshold (S_thresh) amount of sleep time is possible. S_thresh defines the minimum amount of expected sleep time during a segment download. It is a tuneable parameter.

During bitrate selection, three steps of estimation are done. In first step, the player selects the bitrate based on network conditions. Then in the second step, possible sleep time is measured for that selected bitrate version. In last step, performance of previous bitrate selection history is considered.

Download Ratio (DR) and Switch Ratio (SR) are used to select the bitrate version, in a similar fashion to the Adobe OSMF player version 1.6. These two terms are discussed in Section 2.4.2. Instead of using only Download Ratio (DR) in the proposed protocol, the Exponential Weighted Moving Average (EWMA) [26] value of the Download Ratio of the same bitrate version is used. Unlike the Adobe OSMF player version 1.6, where the player can up-shift to any bitrate version bigger than current version, the proposed protocol only permits the player to increase bitrate version by one level up at a time.

The moving average smooths out short-term fluctuations and highlights longer-term trends of data. Exponential weighted moving average is a type of moving average in which by changing the weight value, it is possible to emphasize old or recent data of a data set. For a data series Y, Y_t represents data value at time period t and S_t represents EWMA value at time period t. Coefficient α represents the weight value ranging from 0 to 1. A higher weight value provides more importance

on recent data values, while smaller weight value emphasizes old values. As an example, EWMA value for the data series Y is,

$S_1 = Y_1$

$S_2 = Y_1 + Y_2$ / 2

for $t > 2, S_t = \alpha \times Y_t +$ (1 - α) $\times S_{t-1}$, where α is the weighting factor applied to the most recent value.

In the proposed protocol, two separate EWMA values of Download Ratio are calculated. In first one, a lower weight value is used. On the other hand, in the case of second EWMA, a comparatively high weight value is used. The first EWMA value is used during bitrate up-shift decision making and named as EWMA of Download Ratio for bitrate up-shift (EWMA_DR_UpShift). The second EWMA value (EWMA_DR_DownShift) is used while bitrate down-shift decision is made.

After selecting a bitrate version based on the network conditions, in the next step, the player tries to measure estimated sleep time for that segment based on previous segment experienced transfer rate. If that segment is capable of ensuring S_thresh amount of sleep time, the player selects that bitrate version. Otherwise, the player selects another version that has a lower bitrate compared to the previous one but is capable of providing the expected sleep time. For example, if S_thresh is set to 1s, then player will download only those segments for which estimated sleep time is at least 1s. The player checks all the available bitrate versions until it finds a suitable bitrate version for which estimated sleep time is greater than S_thresh. If no other bitrate version found, the lowest bitrate version is selected. Users can select S_thresh based on their preference. With an increase in S_thresh, sleep time also increases.

Let symbol S_l represent the average length of the segments playback time, denoting segment bitrate by S_b and EWMA value of downloaded segment transfer rate, $EWMA_{TR}$, estimated download time for segment i, EDT_i is given by

$$EDT_i = \frac{S_l \times S_b}{EWMA_{TR}} \tag{4.2}$$

Estimated sleep time,

$$EST_i = S_l - EDT_i \tag{4.3}$$

In the last and final step of bitrate selection process, player checks the previous history of of the bitrate version selected in the previous step. If bitrate down-shift occurs more than once for a particular bitrate version within a fixed interval known as wait period after failure (W_t), then it is considered as a bitrate failure. For the first failure, there is a suspension of that particular bitrate version for a fixed period of time, known as suspension period (SP_t). If bitrate failure occurs again after getting the initial suspension, for each additional failure, suspension period get multiplied with number of failures (F_n) for the next three failures. Thus, the suspension period is adaptive and it depends on the number of failures for that specific version. In the proposed model, the wait

period after failure, and the suspension period both are set to 30 s, which is the same as OSMF player version 1.6 suspension period. If the number of failures is more than the Maximum Allowed Failure (F_{Max}), then there will be a longer suspension period of duration min[SP_T, V_T].

Algorithm 2 Bitrate version selection algorithm for proposed protocol

Step 1:

Bitrate version decrease

if EWMA_DR_DownShift < 1 & EWMA_DR_DownShift < $SR[Index_{current}$ - 1] **then**

switch to lowest rate immediately (even if there's an intermediate that might work)

else if EWMA_DR_DownShift <1 & EWMA_DR_DownShift >= $SR[Index_{current}$ - 1] **then**

Bitrate goes down to one level.

end if

Bitrate version increase

if EWMA_DR_DownShift >= 1 & EWMA_DR_UpShift < $SR[Index_{current}$+1] OR no available rate is higher than current **then**

No change in current bitrate version.

else if EWMA_DR_DownShift >= 1 & EWMA_DR_UpShift > $SR[Index_{current}$+1] **then**

Player switch to bitrate N where $N_{s1} = Index_{previous} + 1$

end if

Step 2:

select bitrate version N_{s2}, where $EST_{N_{s2}} > S\_thresh$ & $N_{s2} >= N_{s1}$

Step 3

if up-shift suggested by step 1

if $F_n == 0$ & (previous failure time for $N_{s2} > W_t$) **then**

select N_{s3} where $N_{s3} = N_{s2}$

else if $F_n < F_{Max}$ **then**

$N_{s3} = Index_{previous}$ and suspend N_{s2} for (F_n X SP_t)

else

$N_{s3} = Index_{previous}$ and suspend N_{s2} for min[SP_T, V_T]

end if

Table 4.2 defines the parameter values used during implementation of the proposed protocol.

Table 4.2: Parameter value used in proposed protocol implementation

Parameter	**Value**
V_T	279 s
B_I	6 s
B_{Max}	30 s
B_{Min}	2 s
$BMAX_T$	20%
W_t / SP_t	30 s
lower weight value to calculate EWMA_DR_UpShift	0.1
higher weight value to calculate EWMA_DR_DownShift	0.88
F_{Max}	3
SP_T	300 s

4.3 Summary

This chapter describes the proposed HTTP adaptive streaming protocol with a detail description of adaptive buffer selection mechanism and three step bitrate selection process. In the beginning of video playback, buffer size kept small and it increase gradually with the playback duration. In the proposed protocol, smooth change in bitrate version is ensured by using EWMA of Download Ratio values and making stepwise increase in bitrate versions. During bitrate selection, the amount of possible sleep time for the segment download is considered.

Chapter 5

EXPERIMENTAL METHODOLOGY

The proposed protocol is implemented by modifying the bitrate adaptation and buffer control mechanisms of the Adobe OSMF player version 1.6. The main reasons for selecting this player are that it supports HTTP adaptive streaming, and it is an open source player. The protocol was implemented by using Adobe Flex 4.6[1] software development kit (SDK) and ActionScript 3[2]. The compiled ActionScript code generates an SWF file. The SWF file and the manifest file for video playback was stored in a web server. All video content was delivered from an Akamai HD server.

The main goal of the experiments is to assess the video playback performance and the potential system energy savings when using the proposed protocol. In order to do so, experiments were run under different network conditions in both a lab environment and real world locations. Experiments were also carried out for different alternative protocol designs in order to assess the impact of protocol designs decisions. In additions performance measurements were made for different protocol parameter values and the impact on performance of these parameters is studied. Section 5.1 describes the hardware and software tools used in the experiments. While Section 5.2 gives a description of the experimental environment and setup. The last section presents the performance metrics used for performance evaluation.

5.1 Experimental Tools

5.1.1 Hardware Platform

The experiments utilized a client PC, a network bridge, and content/video servers. Each component is described in the remainder of this section.

Client PC

The client PC is the hardware on which the video is viewed. Generally this could be any type of computing device which provides the functionality to watch streaming video. In the experiments a Macbook Pro laptop (details specification showed in Table 5.1) was used. The reason behind

[1] `http://www.adobe.com/products/flex.html`, accessed 15-August-2012
[2] `http://www.adobe.com/devnet/actionscript.html`, accessed 15-August-2012

Table 5.1: Specification of client device

Property	**Description**
Model Name	MacBook Pro
Manufacturer	Apple Inc.
Processor Name	Intel Core i5
Processor Speed	2.4 GHz
Number Of Processors	1
Total Number Of Cores	2
L2 Cache (per core)	256 KB
L3 Cache	3 MB
Memory	4 GB
Processor Interconnect Speed	4.8 GT/s
Operating System Version	Mac OS X 10.6.8 (10K549)
Kernel Version	Darwin 10.8.0
Wireless Network Interface	IEEE 802.11n

choosing a laptop instead of a device such as a smartphone is to get more experimental flexibility. Also, energy savings are proportional to the amount of sleep time. So policies can be compared based on the NIC sleep time they enable, and the relative energy savings in a laptop will be similar to those in a device such as a smartphone. The absolute energy savings may differ based on the device hardware configuration such the particular type of CPU and NIC card.

During all the experiments, only the basic required applications were run on the client to ensure maximum resource availability for video playback. The client was always connected to the Internet via a WiFi connection during the experiments.

The player is hosted in a MAMP[3] ("MAMP" stands for: Macintosh, Apache, Mysql and PHP) web server running in the client machine. MAMP is a free software package for hosting web sites.

Network Bridge

A network bridge is an easy and inexpensive way to connect local area network (LAN) segments.[4] It can also act as a link-level intermediate device between a single PC and the Internet. In that case, all the incoming and outgoing traffic from that PC will be routed through the network bridge. A desktop PC was configured as a network bridge in lab environment experiments. A network emulator was running on the bridge to control the network bandwidth. Table 5.2 shows

[3] `http://www.mamp.info/en/index.html`, accessed 09-August-2012

[4] `http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/hnw_understanding_bridge.mspx?mfr=true`, accessed 15-August-2012

Table 5.2: Specification of network bridge

Property	**Description**
Model Name	Dell Desktop PC
Manufacturer	Dell Inc.
Processor Name	Intel Core i5
Processor Speed	2.4 GHz
Number Of Processors	1
Total Number Of Cores	2
Memory	4 GB
Operating System Version	Windows 7

the specification of the desktop PC.

Video/Content Server

During the experiments all the video content was delivered from Akamai HD servers.[5] These servers are designed to deliver live and on-demand HD-quality online video. Key features of the Akamai HD network include: use of a highly distributed and high quality network to place content closer to the consumers, support for different open source players, adaptive bitrate streaming support, and secure content delivery. Akamai places video into this Content Delivery Network (CDN) and attempt to deliver content from the server closest to the client [31].

5.1.2 Software Platform

In the experiments different software components have been used for different purposes, in particular for video playback, for network emulation, and for taking network traces.

Video Playback

The proposed protocol is implemented by using Adobe ActionScript. The Adobe Flex 4.6 software development kit (SDK) is used for code compilation. ActionScript compiled files are used on web pages in the form of embedded SWF files. The Adobe Flash player is required to run such a file. The SWF file implemented the proposed HTTP adaptive streaming protocol. The source code of the proposed protocol implementation is provided in the Appendix.

For video playback, an OSMF provided sample webpage was used. The SWF file implementing the proposed protocol was embedded in that webpage, and is downloaded when the webpage is requested. Flash player minimum version 10.1 is required to run the SWF file.

[5] `http://www.akamai.com/html/misc/hdnetwork.html`, accessed 09-August-2012

Figure 5.1: Web page for video playback

Figure 5.1 shows the interface of the webpage in which the SWF file was embedded. In this webpage there are two sections. In the top left, there is a text box to provide the location of the manifest file for the particular video the user wants to play. There is a button on the right to load the manifest file. The user has to load the manifest file before playing the video. The video playback screen resides in the left of the page. Playback position and volume bars are placed at the bottom of the playback screen. There are also some buttons at the bottom of the playback screen for play, pause, stop and bitrate control. For control of the bitrate, there are three buttons. The first one is for auto bitrate selection, the second one is to increase the bitrate and the last one is to decrease the bitrate. If auto bitrate is selected, the bitrate will be selected adaptively based on network conditions. The user can increase or decrease the bitrate manually by clicking the remaining two buttons. In all of the experiments, auto bitrate selection was on. At the bottom left of the page, the current bitrate with its corresponding index value is shown. The right section of the page contains a big read-only text box. This text box shows player-generated log information during video playback. This log information consists of the playback start time, current bitrate version, segment download starting time, downloaded segment size, measured transfer rate, and several other items of information. There is also a save button at the bottom of the text box for saving the log information. This log information was used for further analysis.

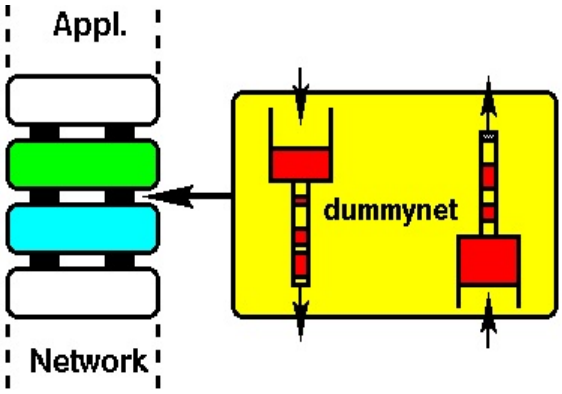


Figure 5.2: DummyNet pipe[7]

During all of the experiments, the same video has been used.[6] This video is provided as an Adobe OSMF video playback sample video and stored in the Akamai HD servers. Among all the sample videos, the video has the longest duration and the highest number of different encodings (8). The video is delivered from the Akamai CDN network and based on the client location the video may be downloaded from different content servers. During video streaming, more than one server may be involved in delivering video segments, although each full segment is delivered from a single server. The video has 70 segments in total, with an average segment playback duration 4 seconds. Total length of the video is 279.76 seconds. There are eight different encodings of the video, with bitrate 250 kbps, 500 kbps, 900 kbps, 1300 kbps, 1700 kbps, 2100 kbps, 2500 kbps, and 3000 kbps.

Network Emulator

DummyNet was used in the experiment as a network emulator. DummyNet is a FreeBSD tool originally designed for testing network protocols. Due to its powerful bandwidth management functionality, it is now one of the most popular WAN emulators. It is possible to emulate different queueing mechanisms; bandwidth limitations, latencies, and packet loss rates using this emulator. In the experiments, DummyNet was run on the network bridge for the purpose of controlling the network bandwidth. Packets flow through a DummyNet network pipe while on their way through the protocol stack (as shown in Figure 5.2).

Network Tracer

Wireshark[8] is a free and open-source packet analyzer. Wireshark was used in the experiments to capture incoming and outgoing packets during video playback. After the video playback, the

[6] `http://zeridemo-f.akamaihd.net/content/inoutedit-mbr/inoutedit_h264_3000.f4m`, accessed 09-August-2012

[7] http://info.iet.unipi.it/ luigi/dummynet/, accessed 15-August-2012

[8] `http://www.wireshark.org/` , accessed 13-August-2012

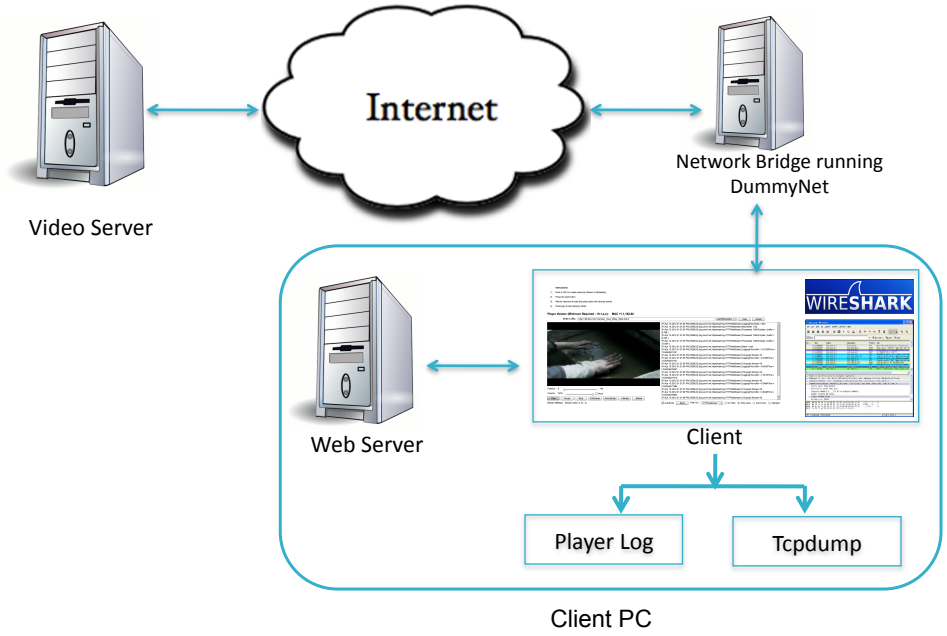


Figure 5.3: Experimental setup: Control environment

relavant packets transferred during video playback were used for further analysis.

5.1.3 Analysis Tool

After a complete video playback, two types of log files were generated. The first one comes from the log data generated from video playback. Another log file was generated from the wireshark network trace. A java parser was implemented to parse the log files and provide data in the desired format.

5.2 Experimental Environment and Setup

In this section, an overview of the experimental environment and corresponding setup is provided. Two types of environment were considered in the experiments: a controlled lab environment, and real world environments.

5.2.1 Controlled Environment

In order to observe the behaviour of the proposed HTTP adaptive streaming protocol and also some alternative designs under known network conditions, a controlled, isolated environment was created to run test cases. The purpose of these tests were to perform some initial studies and performance measurements of the system before going to real world tests.

On the client PC, there were two applications running during each test: a browser, in which the video playback webpage was open and wireshark, to take a network trace. From the video playback page, a playback log was generated and from wireshark a TCP trace was collected. After filtering the dump file to extract the relevant data using the wireshark interface, a video packet trace log file was generated.

The DummyNet network emulator was used to control the downstream available bandwidth to the client PC. DummyNet ran on a network bridge which was placed as an intermediate device between the campus network and the client PC. The main reason for running DummyNet on a separate device was to free up CPU resources on the client PC for use in video playback. When DummyNet limits the network bandwidth, the TCP connections that transfer video and audio streams cannot exceed (collectively) the specified bitrate at any point in time.

5.2.2 Real World Environment

In the real world environments, no network emulator was used to control the available network bandwidth. Instead, the available bandwidth was determined by the dynamically changing network condition and background traffic.

Experiments were run in several locations using WiFi network in which there were competing traffic. In some locations, testing failed due to very low network bandwidth, which was not capable of supporting playback even of the lowest available bitrate version. Also some tests did not succeed due to an unsustainable network connection. Successful test cases were run at three different locations:

Location 1: The first successful real world experiments were run in a fast food restaurant in Saskatoon using their free WiFi service. All the experiments were run consecutively on a weekday afternoon. The restaurant was busy at that time and several customers were connected to the Internet by using the WiFi service. The available bandwidth during the experiments was between 400 kbps to 1200 kbps.

Location 2: The second location at which experiments were successfully run was Saskatoon's John G. Diefenbaker International Airport. The experiments were consecutively on a weekday morning, using the free WiFi service of the airport. During the experiments, the airport was busy and multiple people were connecting to the Internet. The available bandwidth during the experiments varied between 6 Mbps and 20 Mbps.

Location 3: The third set of experiments were run using a home WiFi network with Shaw broadband Internet service. During the experiments several users were connected using the same access point to the Internet, using both wired and wireless connections.

5.3 Performance Metrics

From the results of the experiments, the following performance metrics were calculated:

- *WiFi Sleep Time (WST)*: The WiFi sleep time is the total amount of time the WiFi radio could sleep during video playback. It is calculated by taking the sum of overall packet inter-arrival times, of the maximum of zero and the packet inter arrival time minus the awake mode to sleep mode conversion time, T_c. The value of T_c depends on the device and system software. For example in an investigation[20] on HTC magic phones, it was found that the sleep mode conversion time on this smartphone is 1 second. During experiments, the T_c value was set to 1 s. If T_p denotes the arrival time of the i-th packet, out of N packets of video data received in total, the WiFi Sleep Time (WST) is defined as

$$WST = \sum_{p=1}^{N} \max[0, (T_p - T_{p-1}) - T_c] \tag{5.1}$$

- *Cellular Sleep Time (CST)*: The Cellular Sleep Time (CST) is the total amount of time the cellular radio could sleep during video playback. The CST is calculated in the same means as the WST by using equation 5.1. The HTC magic phone cellular radio remains in high power consumption mode for 12 seconds after completing a data transfer [20]. In this case, the T_c value was set to 12 s.

- *Average Playback Version (APV)*: The Average Playback Version is the average, bitrate versions being used. Each of the available bitrate versions is given an index number, I, provided in ascending order of bitrate starting from a index of 1. For example, if for a particular video there are eight bitrate versions available from 250 kbps to 3000 kbps, each version will be given an index number from 1 to 8, with the 250 kbps version being given an index of 1, and equally the 3000 kbps version being given an index of 8. If for a particular video there are N segments downloaded, with the n'th segment having segment length L_n and index I_n, the average playback version (APV) is given by:

$$APV = \sum_{n=1}^{N} (L_n \times I_n) / \sum_{n=1}^{N} L_n \tag{5.2}$$

- *Playback Smoothness (PS)*: A "run" is a sequence of downloaded segments that are from the same version. Denote the total number of runs by M, and the duration and the bitrate version index of the r'th run by n_r and I_r, respectively. It is expected that a longer run length provides a smoother watching experience. On the other hand, a big jump between index values for consecutive runs may impact playback smoothness and worsen the viewing

experience. The playback smoothness (PS) is defined as,

$$PS = \sqrt{\frac{\sum_{r=1}^{M}(n_r/(I_r - I_{r-1}))^2}{M}} \tag{5.3}$$

5.4 Summary

In this chapter, different hardware and software used during the experiments are described. Experiments were run on both controlled environment, where bandwidth was controlled by using a network emulator and in three different location in real world environment. For performance measurement, four different parameters (WST, CST, APV, and PS) are used.

Chapter 6

EXPERIMENTAL RESULTS

This chapter describes the experimental results obtained using the proposed protocol and several consecutive runs of the protocol at different locations. This chapter is divided into two sections. The first section (Section 6.1) describes how different parameter values and other design choices impact the performance of the proposed protocol. Section 6.1.1 presents the impact of different parameter values used for buffer size selection and Section 6.1.2 presents impact of different design choices during the three steps bitrate selection. The second section (Section 6.2) shows the results of the experiments comparing the performance of proposed protocol at different locations (both in a lab environment and in real world environments).

6.1 Performance Impact of Protocol Design Choices

6.1.1 Buffer Selection

Buffer size selection is an important issue in the HTTP adaptive streaming algorithm. Recall that buffer size is measured in seconds. The amount of consecutive data downloaded is dependent on buffer size. If the buffer size is large, the amount of data downloaded in a row is high. On the other hand, in the case of a small buffer, only a small amount of data will be downloaded at a time until the buffer fills up. Client buffer is used to enable pauses between data download request. The whole data transfer period is divided into several active data transfer periods, in which data is downloaded to fill up the buffer. With the increase in buffer size, the size of each active data transfer period increases.

There are several advantages and disadvantages of both small and large buffers. If playback is interrupted by the user in middle of video playback, data wastage will be less in the case of a small buffer. Also with a small buffer, individual data downloads finish in a small amount of time (a few seconds). It helps the player to adapt the bitrate more frequently when network bandwidth changes. Data need to be requested frequently, since only a small playback time can be served by the buffer. The short interval between requests mean that it is possible to transfer all the data for a complete video using a single TCP connection, which overcomes TCP slow start problem.

There are also several disadvantages of using a small sized buffer. In the case of small buffer,

only a few segments are downloaded in each active data download period. Throughput during data download is also reduced due to the small amount of data transferred in each active data download period. The reason behind lower throughput is investigated later in this chapter. This lower throughput causes higher download time. Also with a small buffer size, the frequent but short intervals between each active data download period, reduce total sleep time of the radio during data transfer.

On the other hand, in the case of a large buffer, between each data download, there are some periodic intervals. With the increase in buffer, the number of intervals is reduced, but the size of the intervals increases. These intervals allows the radio to be put into sleep mode. Also, there are a greater number of segments downloaded in each active data download period. This helps the player to achieve desirable throughput. Thus, in the case of a large buffer, the amount of radio sleep time increases due to its higher throughput and large interval between each active data download period. Large buffers have the same disadvantages as the advantages of small buffers.

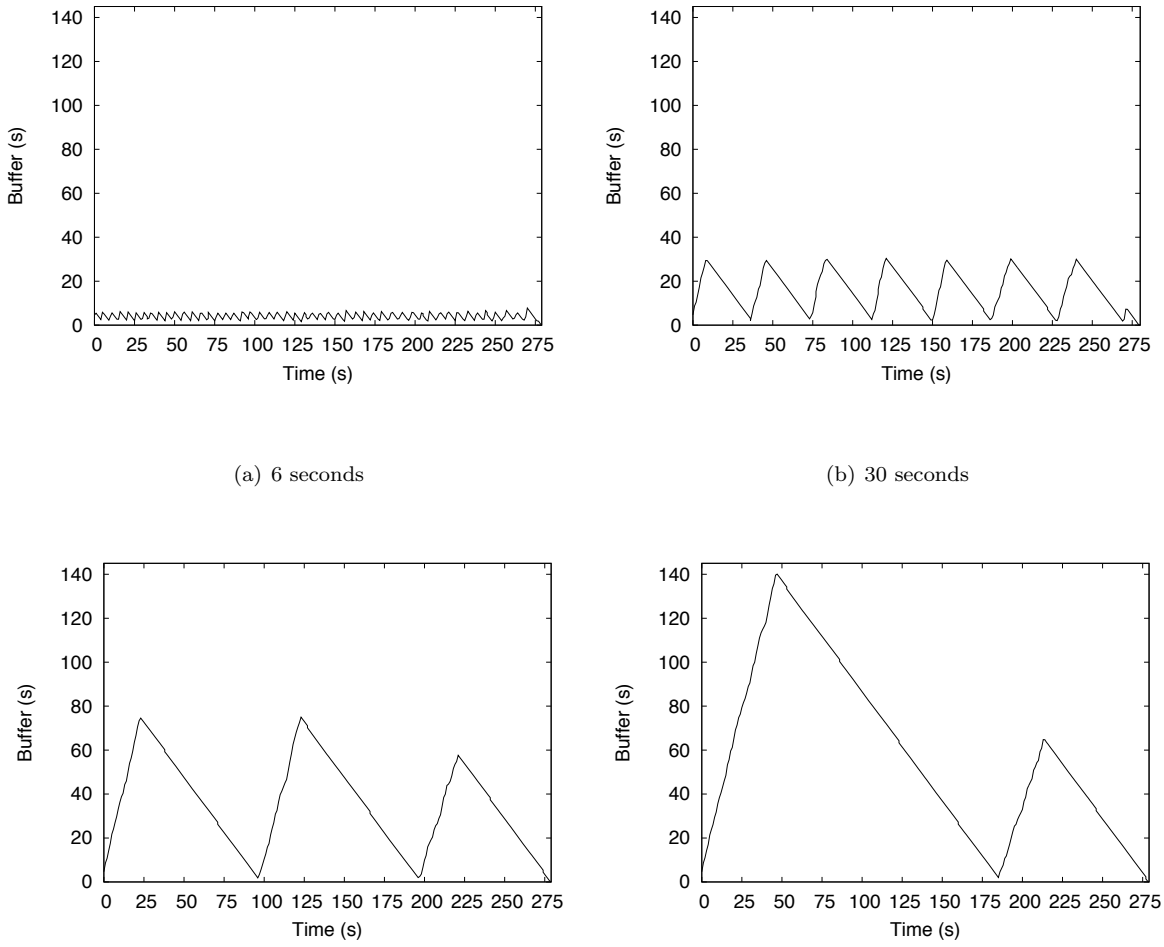
In order to show the buffer size impact on data download, two separate experiments have been run in a controlled environment, described in Table 6.1. In order to make a fair comparison, the bitrate version was kept fixed in both test cases.

Table 6.1: Test case to show the impact of different buffer sizes

Test Case	**Available maximum bandwidth**	**Bitrate version**
Test Case 1	2 Mbps	250 Kbps
Test Case 2	5 Mbps	1700 Kbps

Figure 6.1 shows the change in buffer occupancy during video playback for different buffer sizes. In all the graphs, there are some peaks indicating a maximum value of current buffer occupancy. The upward sloped line shows increase in current buffer. In this condition, both playback and data downloads continue simultaneously. Buffer occupancy increases until the buffer value reaches the peak. Once the buffer fills up, download pauses until the buffer occupancy goes down to minimum value. In this condition, playback consumes the buffer. This causes a downward move from the peak.

Figure 6.1 (a) shows the buffer occupancy change for 6 second maximum buffer which is the buffer size used by Adobe OSMF player version 1.6, for the duration of playback. There are several frequent peaks that appear in this graph. Due to the small buffer size, the buffer fills up quickly which is the reason for frequent appearance of the peaks. Also data download occurs during almost the whole playback duration. Due to this frequent data download, the NIC has to be in “awake” mode during the whole playback duration. Figure 6.1 (b) shows buffer occupancy change for 30 seconds maximum buffer size. There are seven large peaks and one small peak in this graph. In this moderate sized buffer, buffer doesn’t fill up too frequently. After each buffer data download, shown



(a) 6 seconds

(b) 30 seconds

(c) 75 seconds

(d) 140 seconds

Figure 6.1: Test Case 1 - Buffer occupancy over time for different buffer sizes

by a peak, the player pauses data download. Compared with Figure 6.1 (a), in Figure 6.1 (b) the interval between data download pause and download resumption is comparatively long. These long intervals provide comparatively higher amount of possible device sleep time. The remaining two graphs (Figure 6.1 (c) and (d)) show the buffer occupancy change for 75 seconds and 140 seconds buffer, respectively. In both cases, there are fewer download periods and the peak buffer occupancy is higher, as would be expected.

Figure 6.2 shows the throughput during data download for different buffer sizes. Each bar shows throughput per second. Consecutive bars define data download in a active data transfer period. The gap between each active data transfer period indicates a pause in download. Figure 6.2 (a) shows throughput over time for a 6 second buffer. Here data download occurs frequently and

the gap between two consecutive data downloads is very small. Also, the average throughput is comparatively low in the case of 6 second buffer (approximately 800 Kbps) compared to those in 30 second, 75 second, and 140 second buffers (respectively 1080 Kbps) shown in Figure 6.2 (b), (c), and (d). As would be expected from Figure 6.1, in Figure 6.2 (b), (c), and (d), the gaps between each active data transfer period are big and increase with the buffer size.

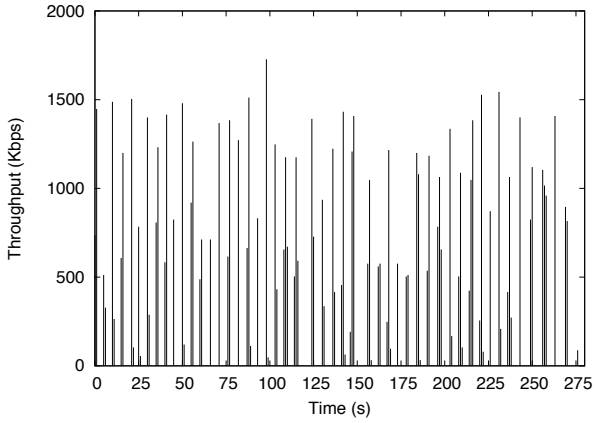
The same experiment was run for the 1700 kbps bitrate version. Figure 6.3 shows the buffer occupancy and Figure 6.4 shows the throughput for the second test case. The result is as the same as previous test case, except the number of peaks and the number of active data download periods increases due to high data transfer rate in Test Case 2. In Figure 6.4 (a), average throughput is comparatively smaller (approximately 2700 Kbps) than in Figure 6.4 (b), (c), and (d) at approximately 3300 Kbps.

The experimental results show that small buffer size reduces data download throughput. There are two possible reasons suspected behind this. The first one is TCP slow start. TCP may show two different types of behaviour during start up after an idle period.[1] If the idle period is longer than the Round Trip Time (RTT) estimate, in some TCP implementations, slow start may take place on the next packet transmission. On some other implementations, the congestion window is not reduced after an idle period. According to the first TCP implementation behaviour, as there are frequent small size gaps between data downloads with a small buffer, these gaps may cause a reduction in the TCP congestion window size and the result is a slow start at the beginning of each data transfer. To check TCP behaviour, the TCP segment download sequence number versus time graph is shown in Figure 6.5 for 6s buffer in Test Case 2. In this graph, there is no evidence of slow start in the data download found after the download interval.

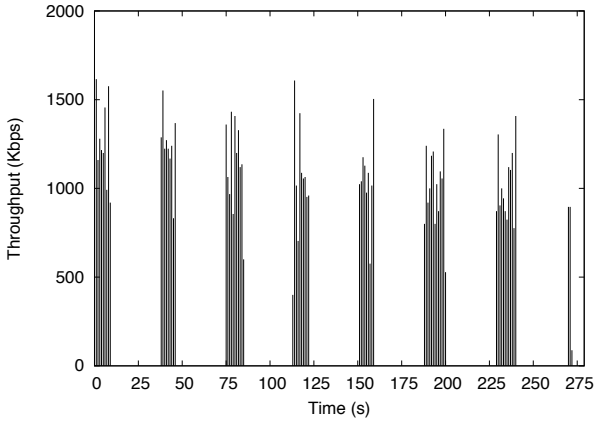
The second possible reason behind reduced throughput is a small player buffer. As the player buffer fills up early, the application thread that is reading data from the TCP buffer reads a very small amount of data and stops reading data while the buffer fills up. On the other hand, if the download request is sent before the buffer fills up, it saturates the TCP receiver buffer. In this situation, if the player stalls reading from the TCP receiver buffer, the receiver stalls the TCP sender until the player starts reading from TCP receiver buffer and space in the TCP buffer frees up again.

Figure 6.6 and Figure 6.7 show the amount of sleep time achieved with a fixed bitrate version for 250 Kbps and 1700 Kbps bitrate versions downloaded with a 5 Mbps maximum bandwidth network connection for WiFi and cellular radio, respectively. For fair comparison, both bitrate version and maximum achievable bandwidth were kept constant for the duration of the video playback. In both the graphs, with the increase in buffer size, possible WiFi and cellular sleep time also increases. In Figure 6.6 from 6 seconds to 30 seconds buffer, there was a monotonic increase in possible WiFi

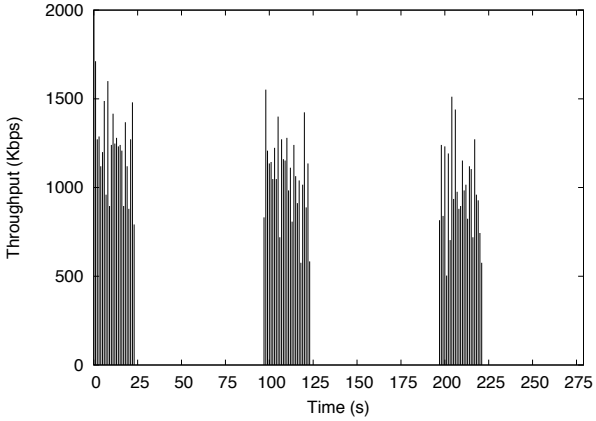
[1] `http://tools.ietf.org/html/draft-handley-tcp-cwv-01`, accessed 12-August-2012



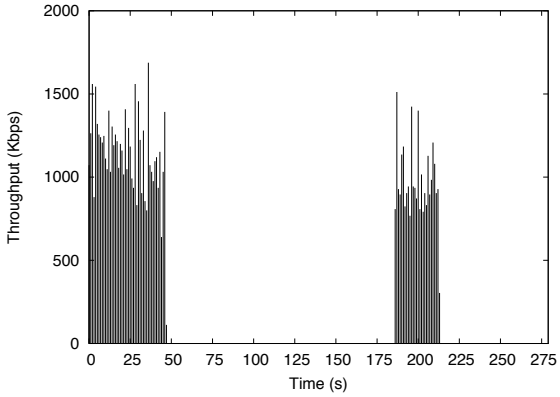
(a) 6 seconds



(b) 30 seconds

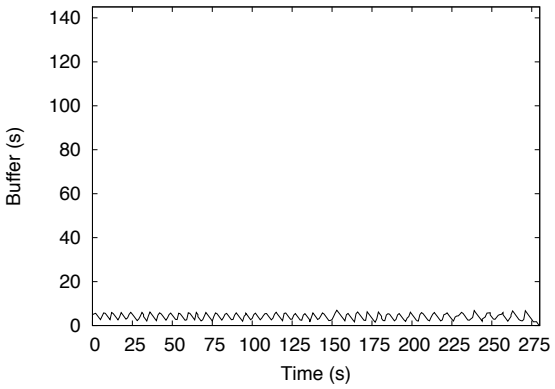


(c) 75 seconds

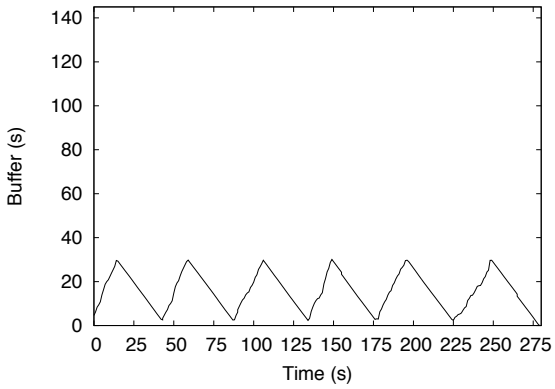


(d) 140 seconds

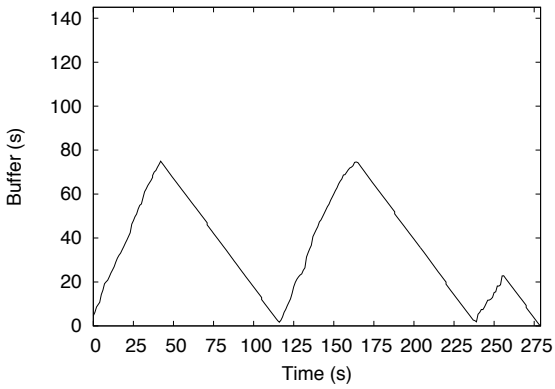
Figure 6.2: Test Case 1 -Throughput for different buffer sizes



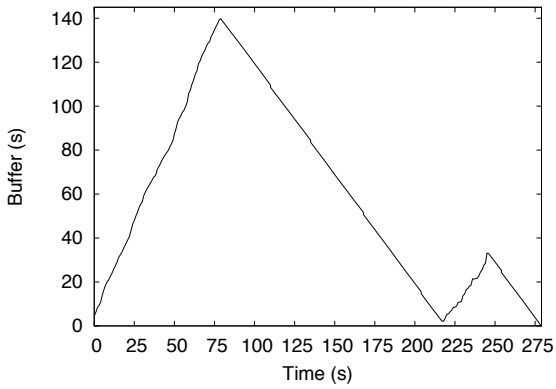
(a) 6 seconds



(b) 30 seconds

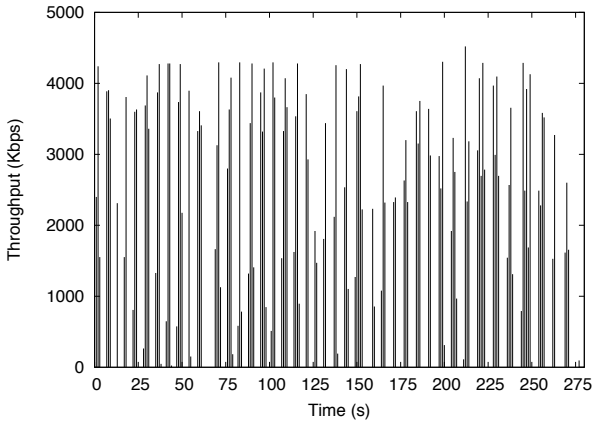


(c) 75 seconds

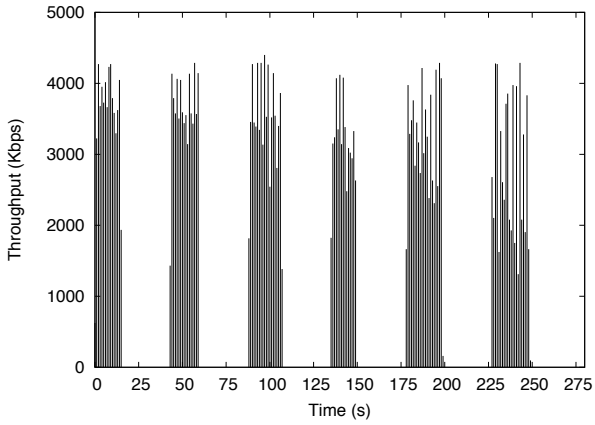


(d) 140 seconds

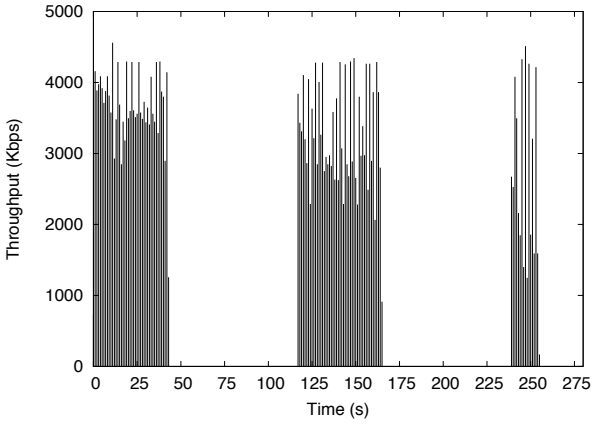
Figure 6.3: Test Case 2 - Buffer occupancy over time for different buffer sizes



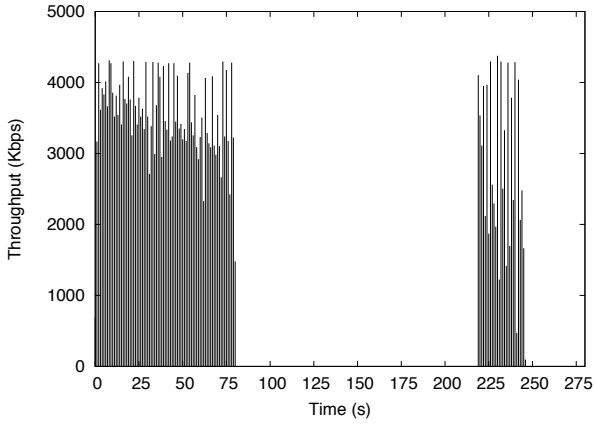
(a) 6 seconds



(b) 30 seconds



(c) 75 seconds



(d) 140 seconds

Figure 6.4: Test Case 2 -Throughput for different buffer sizes

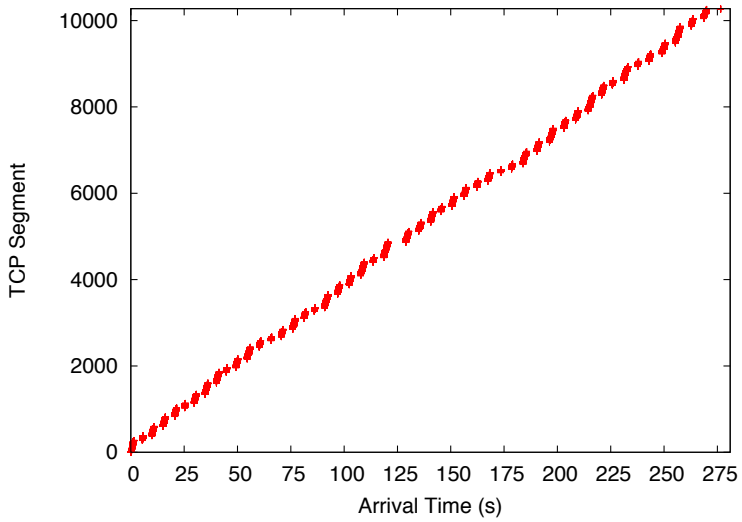


Figure 6.5: TCP segment download sequence over time

sleep time. But after 30 seconds buffer, sleep time did not increase in the same manner.

There are two possible reason found behind comparatively lower sleep time in 6 second and 15 second buffer: 1) low throughput, and 2) time to radio transition between awake mode to sleep mode. In 6 second and 15 second, throughput was low due to small buffer size. These lower throughput values cause higher download time and lower sleep time. Another reason is radio mode transition time. A radio awake-mode to sleep-mode transition time is required for each data download. In the WiFi radio transition time is small (1 s in HTC Magic phone), but in the cellular radio the transition time is big (12 s in HTC Magic phone) [20]. The accumulated total time required for mode transition reduces the total sleep possible time. In Figure 6.7 there was no possible cellular sleep time found for 6 second buffer due to large radio awake-mode to sleep-mode transition period in cellular radio. This large mode transition period also reduces accumulated total possible sleep time in other buffer sizes compared with WiFi possible sleep time.

Experimental results suggest that by deploying a moderate sized buffer, it is possible to ensure higher amount of sleep time for WiFi radio but for cellular radio it is necessary to have a large buffer to achieve more sleep time. Considering the disadvantages of big sized buffer, in the proposed HTTP adaptive streaming protocol, the maximum buffer size is kept moderate.

In the proposed HTTP adaptive streaming algorithm, adaptive buffer sizing policy (described in Section 4.2.1) is used. Playback starts with small buffer. As playback proceeds, buffer size increases gradually. After a certain playback, the buffer reaches its maximum value and maintains that buffer size for the rest of the playback duration.

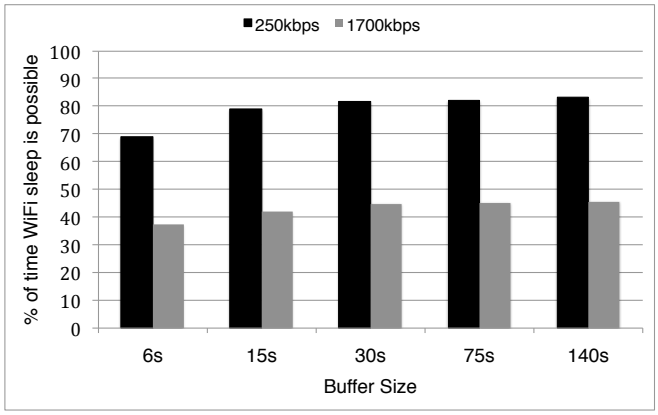


Figure 6.6: WiFi sleep time comparison for 250kbps and 1700 kbps bitrate version

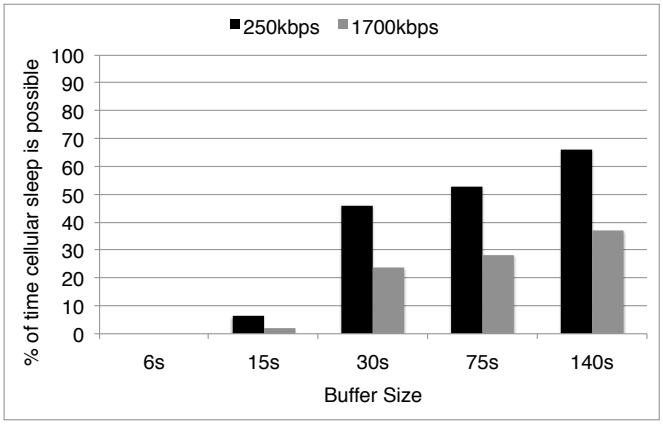


Figure 6.7: Cellular sleep time comparison for 250kbps and 1700 kbps bitrate version

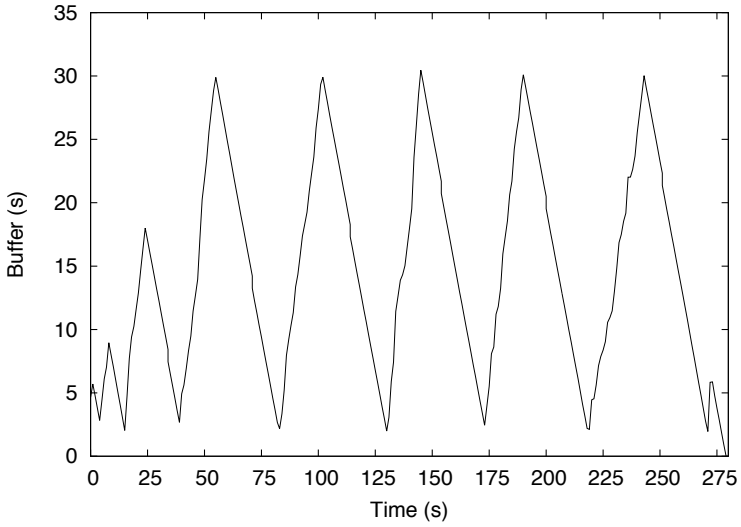


Figure 6.8: Buffer occupancy changes with time using the adaptive buffer sizing policy

Figure 6.8 shows buffer occupancy during playback for adaptive buffer for an experiment with a 4Mbps transfer rate. Playback starts with a 6 second buffer. The player buffer increases gradually and buffer reaches 9 seconds at 8 seconds of playback, 18 seconds at 24 seconds of playback, and the maximum buffer size of 30 seconds at 55 seconds of playback, respectively. In the rest of the playback, the maximum buffer size is maintained at 30 seconds.

Figure 6.9 shows the throughput during data download for the adaptive buffer sizing policy. Between the first two data downloads, there is a small sized gap. The size of the gap increases between the next 2 data downloads. This gap increases gradually for the remaining data downloads. Once the player reaches the maximum buffer, the gap between data download is the same in further downloads.

In Figure 6.10, the bar graph shows the percentage of time both WiFi and cellular sleeping is possible for fixed buffer size of 6 seconds and 30 seconds, and for adaptive buffer size. During these sleep time measurements, the maximum bandwidth was limited to 4 Mbps and the bitrate was fixed at 1700 Kbps. In this graph, for a 6-second buffer, the percentage of WiFi sleep time achieved during video playback is somewhat smaller (37.4%) due to lower throughput during data transfer and higher amount of radio awake-to-sleep mode transition time. For a 30-second fixed buffer, sleep time is increased to 45% due to its higher throughput and fewer radio mode transitions.

On the other hand, for adaptive buffer, total sleep time is 43.8%, which is 6.4% higher than 6 second buffer and slightly lower than 30 second fixed buffer. On the other hand, in the case of cellular radio, there was no sleep time for 6 second buffer. In the case of 30-second fixed buffer,

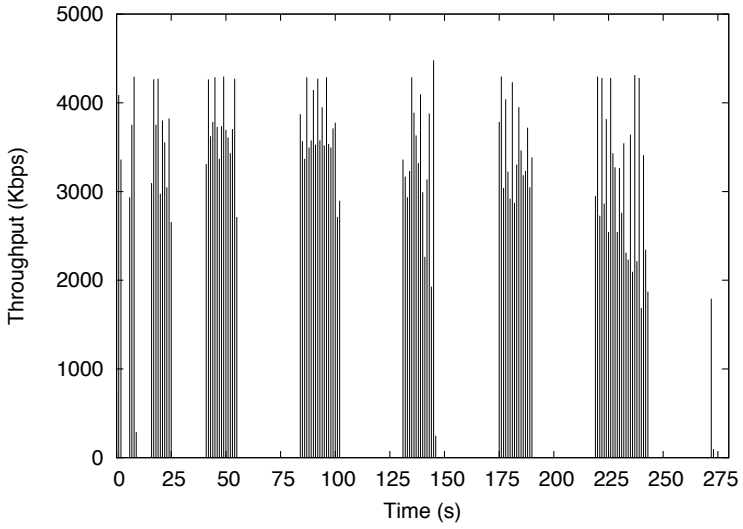


Figure 6.9: Throughput changes with time using the adaptive buffer sizing policy

there was 24% possible sleep time and for adaptive buffer possible sleep time was 22%. In both cases, measured possible sleep time for adaptive buffer size is higher than the 6 second buffer and close to the 30 second fixed buffer.

Figure 6.11 shows the percentage of data wastage for different types of buffer if playback is interrupted by the user. If the buffer size is big, there is a higher amount of data required to fill up the buffer. In the case of small buffer, player requests a small amount of data to fill up the buffer. During video playback, if interruption is to occur, users generally interrupt the playback towards the beginning of the playback [17]. The bar graph shows amount of data wastage for a playback interruption that takes place at 10 seconds and 25 seconds, respectively. The data wastage is calculated from the difference of total downloaded data and amount of data consumed for playback. For a 6-second buffer, if playback is interrupted at the 10th second, there is 39% data wastage found, which increases to approximately 80% for a 30-second buffer. In the case of adaptive buffer, which maintains a smaller buffer in the beginning, there is approximately 44% data wastage reported if playback is interrupted at the 10th second, which reduces data wastage up to 82% over 30-second fixed buffer. On the other hand, if playback is interrupted at 25 seconds, data wastage is found to be approximately 3%, 24% and 4% for 6 second, 30 second and adaptive buffer, respectively.

Figure 6.12 shows the percentage of data downloaded after 25 seconds of playback for the fixed bitrate version of 1700 Kbps, downloaded with bandwidth restricted to 4Mbps. In the graph, for 30 second buffer, a comparatively higher amount of data is downloaded (13%), while for 6 second

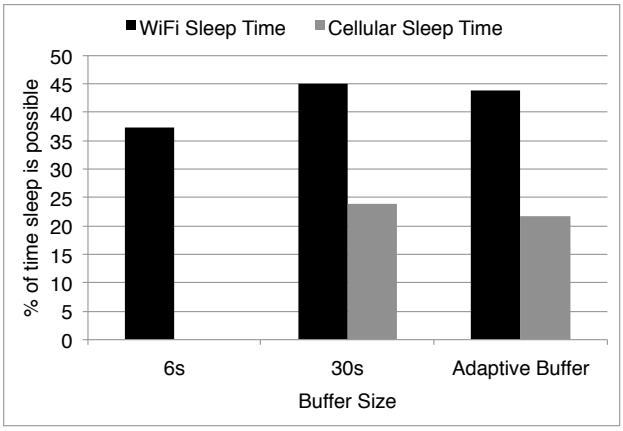


Figure 6.10: Percentage of time WiFi and cellular radio sleeping is possible for different buffer sizing

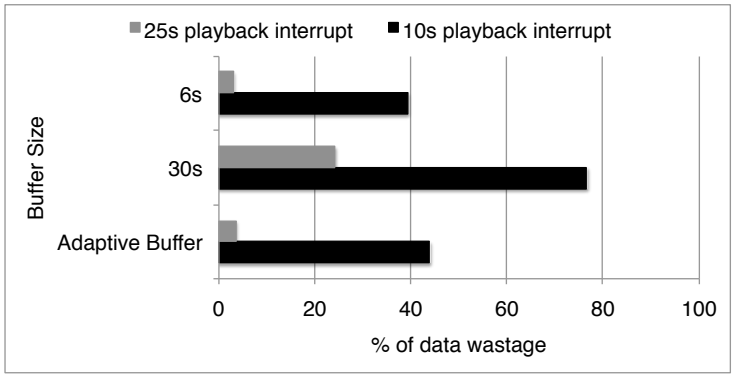


Figure 6.11: Waste of data for different buffer sizing in the case of playback interruption

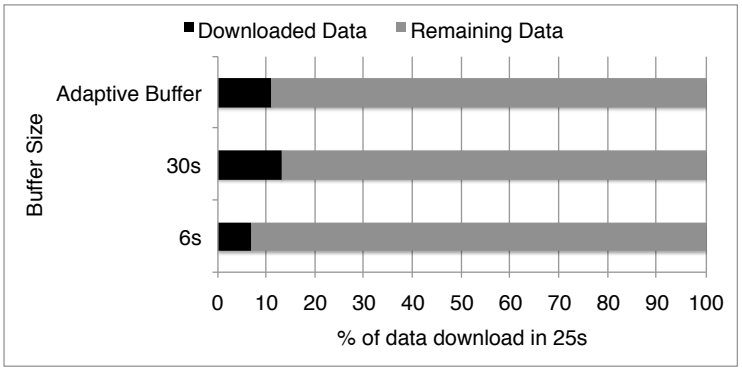


Figure 6.12: Data downloaded after 25 s playback

buffer data download is comparatively lower (7%). In the case of adaptive buffer, 11% of the data is downloaded in the first 25 seconds of playback. This graph shows that the data remaining to be downloaded is higher in the adaptive buffer then the 30 second buffer. With a higher amount of remaining data to be download, in the case of adaptive buffer, the probability of adapting the bitrate with changing network condition is also greater.

6.1.2 Bitrate Selection

As described in Section 4.2.2, there are three steps in the bitrate selection phase of the proposed protocol. The performance improvement achieved by using the proposed design approach compared with the bitrate adaptation mechanism used in Adobe OSMF player version 1.6 and other possible design approaches is described in following sub-sections.

Step 1: Download Ratio for bitrate selection

In the OSMF player version 1.6, the bitrate selection decision is made based on the Download Ratio, DR (ratio between segment playback time and segment download time). There are several factors that impact the DR value, such as network data transfer rate, segment size, and segment duration. The network data transfer rate determines segment download time. With an increased transfer rate, the download time decreases. Similarly, segment playback duration is also important. If a segment has higher playback duration, with lower segment size, the DR value will increase for that segment. The Download Ratio can also experience a sudden increase due to the increase in transfer rate.

There are several disadvantages of using only the DR value for bitrate estimation. As the DR ratio is calculated based on previous segment download experience, fluctuation in available bandwidth will impact the DR value. Also, as the DR depends on segment size and duration, the prediction can go wrong in the case of segments encoded with higher bitrate, but which have small

Table 6.2: Test case to show impact of Download Ratio

Test Case	Available maximum bandwidth	Bitrate version	Buffer Size
Test Case 3	6 Mbps	Adaptive	Adaptive
Test Case 4	Unrestricted	Adaptive	Adaptive

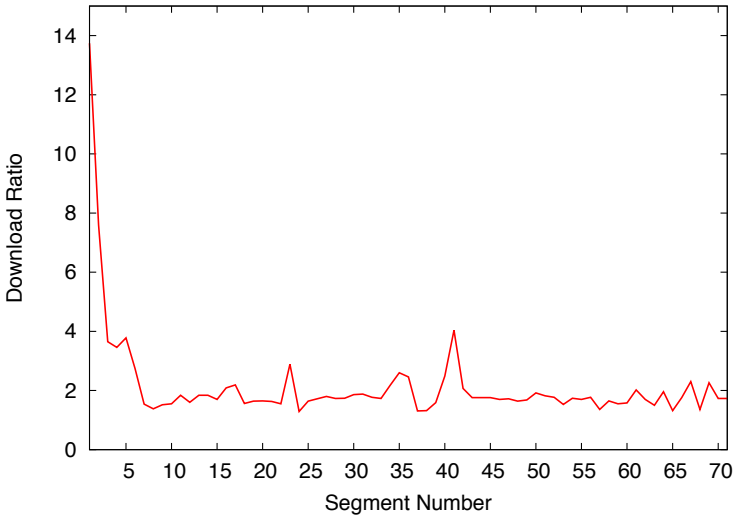


Figure 6.13: Test Case 3: Download Ratio during playback

size due to variable bitrate encoding.

To show the impact of using DR value on bitrate selection decision, two test cases (Test Case 3 and Test Case 4) have been considered, described in Table 6.2. In both cases, buffer size and bitrate version were adaptive.

Figure 6.13 shows the DR value during Test Case 3. In this graph, the DR is high at the beginning of video playback due to segments with lower bitrate version downloaded. With the increase in bitrate versions, the DR gradually decreases and becomes stable value when the bitrate version reached its maximum achievable value. From the 9th segment onwards, the DR value becomes stable, except for a sudden rise for 23rd and 41st segment. The reason is described later in this section.

Figure 6.14 shows the transfer rate for the downloaded segments during the experiment. The transfer rate is stable for all the segments except the 20th and the 40th segment. During each segment download, the transfer rate reached very close to achievable maximum value.

Figure 6.15 shows the size of downloaded segments, while Figure 6.16 shows duration of the downloaded segments. In these figures, despite having a 5 second playback duration, the size of the

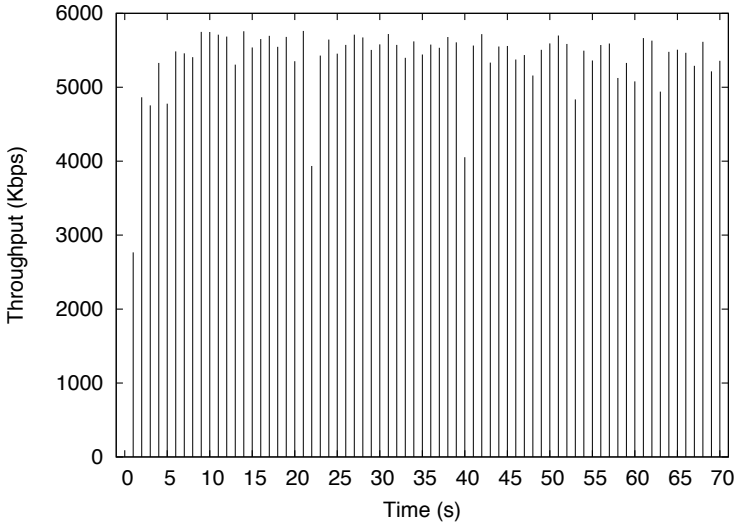


Figure 6.14: Test Case 3: Transfer rate for downloaded segments

41st segment is comparatively small, even though it is encoded with 3500kbps bitrate version. Due to variable bitrate encoding, size of some segments can be smaller than others, causing a sudden increase in DR values.

To mitigate the impact of sudden changes in DR values, the Exponential Weighted Moving Average (EWMA) value of the DR can be used. It smoothes out short-term fluctuations and highlights longer-term trends of data. As discussed in Section 4.4, EWMA value of DR with lower weight value can be used. It puts more weight on previous DR values and does not react immediately in the case of sudden change in DR values.

Figure 6.17 compares unsmoothed DR values with smoothed DR values with high weight value ($\alpha = 0.88$) and low weight value ($\alpha = 0.1$) for Test Case 3. In this graph, EWMA of DR ($\alpha = 0.1$) does not react immediately in the case of sudden raise in DR values, which mitigates temporary increase in DR value problem. There is also a disadvantage of using EWMA of DR ($\alpha = 0.1$). In the case of a sudden fall in DR values, it reacts slowly. In that case, if only EWMA of DR ($\alpha = 0.1$) is used for bitrate selection, it may happen that player reacts slowly when changing bitrate version despite a sudden drop in network bandwidth. In the graph, though EWMA of DR ($\alpha = 0.1$) doesn't react to a sudden increase in DR value for the 23rd, 34th, 35th, 36th, and 41st segments, it reacted very slowly for a sudden drop in DR value for the 24th, 37th, 38th, and 39th segments. Due to the drop of DR values of 37th, 38th, and 39th segment, the EWMA of DR ($\alpha = 0.1$) reacted slowly by the time, when the DR values went up. This late reaction may cause a sudden drop of buffer occupancy and player may suffer a buffer-underflow.

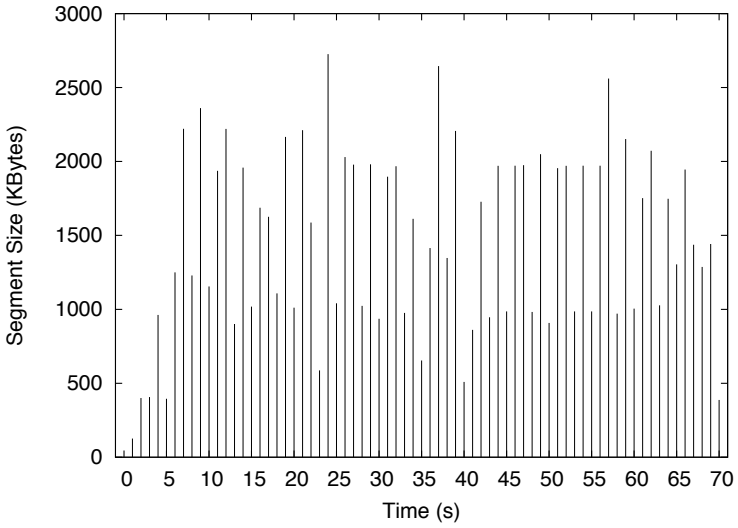


Figure 6.15: Test Case 3: Segment size for downloaded segments

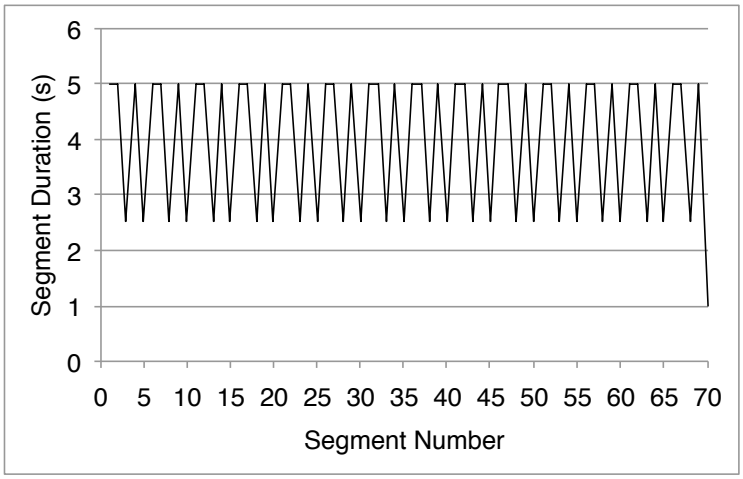


Figure 6.16: Test Case 3: Duration of the segments

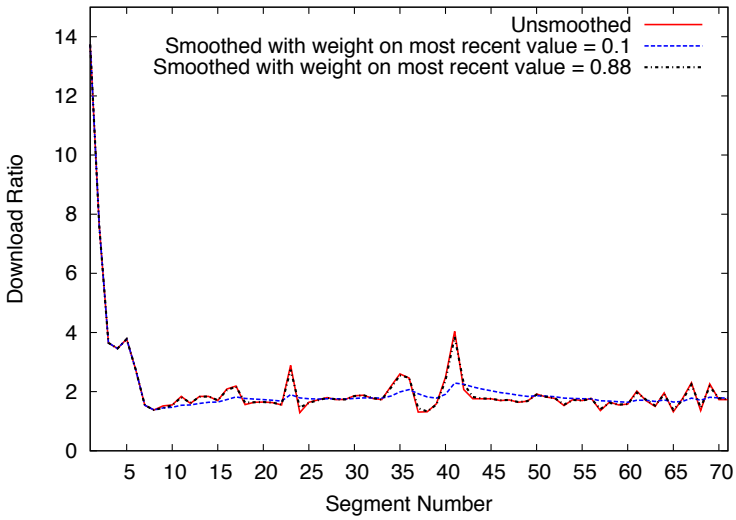


Figure 6.17: Test Case 3: Comparison of smoothed and unsmoothed Download Ratio values

On the other hand, there are several benefits of using EWMA value of DR calculated by using higher weight value. EWMA of DR ($\alpha = 0.88$) focuses more on recent DR values and reacts quickly with DR value change. It helps to reduce the bitrate version immediately when available bandwidth goes down which reduces the probability of buffer underflow when available bandwidth is low. The problem of using EWMA of DR ($\alpha = 0.88$) to calculate bitrate is that, during sudden increase of network bandwidth or increase in DR value due to small segment size, EWMA of DR ($\alpha = 0.88$) reacts immediately and may increase bitrate version. This may also cause player buffer underflow when there is a sudden but not sustained increase in available bandwidth and the playback bitrate can not be supported. In Figure 6.17, EWMA of DR ($\alpha = 0.88$) increased with sudden increase in DR value for 23rd and 41st segment.

In the proposed protocol, both EWMA of DR ($\alpha = 0.1$) and EWMA of DR ($\alpha = 0.88$) are used. The player checks the current network conditions by using EWMA of DR ($\alpha = 0.88$) and it uses EWMA of DR ($\alpha = 0.1$) to make the bitrate increase decision. During bitrate downgrade, the system uses EWMA of DR ($\alpha = 0.88$) to make the decisions.

In the next experiment (Test Case 4), there was no restriction of the bandwidth. The experiment was run on a home environment (described in Section 5.3.2).

Figure 6.18 compares unsmoothed DR values with smoothed DR values with high weight value ($\alpha = 0.88$) and low weight value ($\alpha = 0.1$) for Test Case 4. Due to fluctuating network conditions, there are several ups and downs in the DR value up to the 43rd segment. The DR values stabilized from the 44th segment until the end of the playback. Similar to Test Case 3, in this case sudden

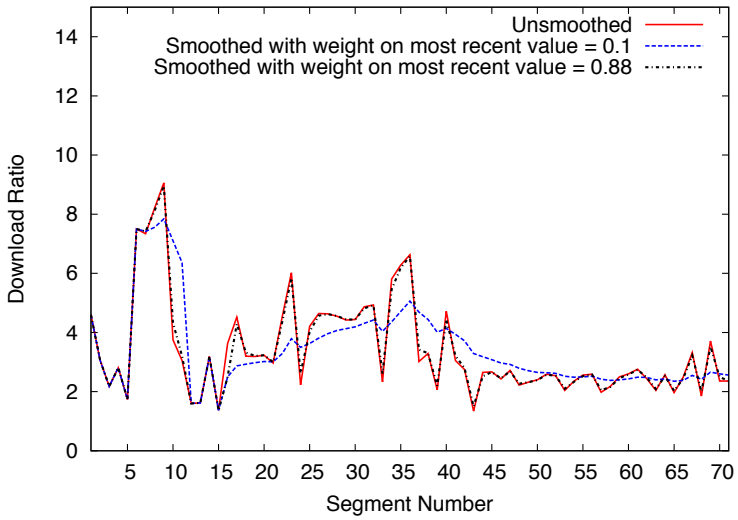


Figure 6.18: Test Case 4: Comparison of smoothed and unsmoothed Download Ratio values

increases in DR values (17th, 23rd, and 36th segment) due to sudden increases in network bandwidth or smaller segment bitrate versions were not caused an immediate reaction. Similarly, when the DR value went down (15th and 43rd segment) due to lower network bandwidth, DR value reacted immediately by shifting into a comparatively lower bitrate version.

Figure 6.19 shows the transfer rate measured during segment download. Transfer rates fluctuates up to the 43rd segment and stabilizes after that. Figure 6.20 shows size of the downloaded segments. Similar to Test Case 3, the small size of the 41st segment caused a sudden increase in DR values.

The amount of WiFi sleep time during video download is also increased with a cost of selecting lower bitrate versions. Table 6.3 shows the WiFi Sleep Time (WST), Cellular Sleep Time (CST), and Average Playback Version (APV) for the sample video played in 4 Mbps restricted bandwidth for different DR values. The amount of possible WST is 43% when using unsmoothed DR for selecting bitrate version (in the OSMF player version 1.6). The amount of possible WiFi sleep time increases to 47% by using two smoothed DR values with high and low weight values for bitrate selection (in the proposed protocol). Similarly CST value is also high for both smoothed DR values.

Step 2 : Estimate sleep possible time

In the proposed model, based on previous segment transfer rate, sleep time is estimated for the next segment. Accuracy of sleep time estimation is calculated for three outdoor locations. The experimental results show that the proposed protocol is able to predict sleep possibility to a high

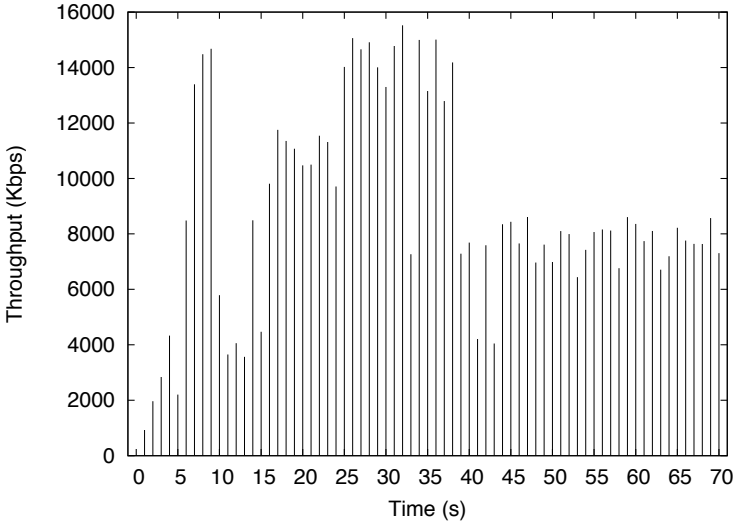


Figure 6.19: Test Case 4: Transfer rate for downloaded segments

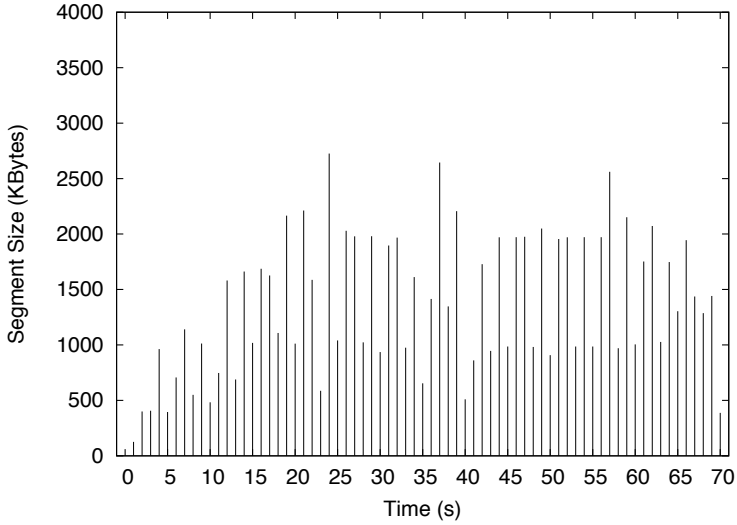


Figure 6.20: Test Case 4: Segment size for downloaded segments

Table 6.3: WNIC sleep time and APV values when using smoothed and unsmoothed Download Ratio values

DR values used	WST	CST	APV
Unsmoothed DR	119.61	44.34	4.24
Smoothed with weight on most recent value = 0.1	130.68	48.73	3.97
Smoothed with weight on most recent value = 0.88	124.8	46.21	4.11
Both smoothed with weight = 0.1, and with weight = 0.88	132.79	51.32	3.91

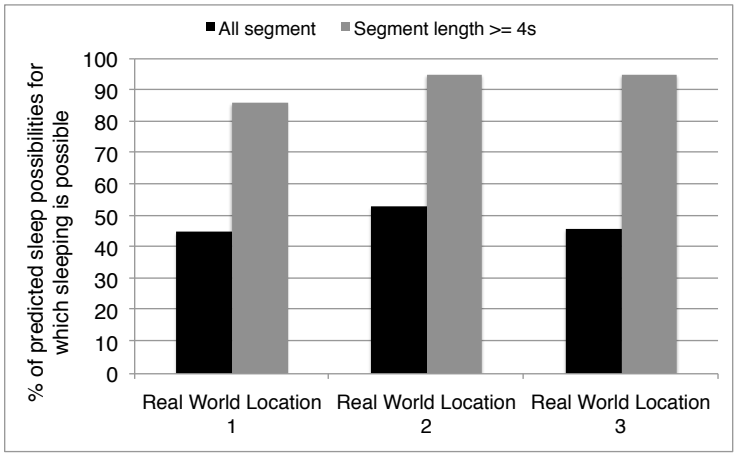


Figure 6.21: Accuracy of sleep time prediction

degree (86% in Location 1, and 95% each in Location 2 and Location 3) in the case of segments with comparatively higher playback duration (at least 4 seconds). On the other hand, for segments with lower duration, accuracy of prediction is comparatively low (45% in Location 1, 53% in Location 2, and 46% in Location 3), showed in Figure 6.21.

Step 3 : Adaptive bitrate suspension

In the proposed protocol, instead of fixed bitrate suspension period (time period in which a particular bitrate version is prevented from being selected) of Adobe OSMF player version 1.6, adaptive suspension period is applied in the case of bitrate failure. Bitrate failure occurs when the player must downgrade the bitrate version due to lack of resources, either network or server bandwidth. In the Adobe OSMF version 1.6, bitrate version suspended for 30 second for each failure. The bitrate version may fail again after the suspension period has elapsed. In the worst case, a particular bitrate may fail every 30 seconds. In the proposed model, an adaptive suspension period is applied. This reduces the probability of repetitive failure for a particular bitrate version. In fixed suspension, there is 5%-7% bitrate failure observed; this is reduced to 2%-3% in the case of adaptive suspension period.

6.2 Performance Measurement

In order to measure performance of the proposed energy efficient HTTP adaptive streaming protocol, several experiments were run in both a controlled environment and real world environment. The purpose of testing the protocol in the lab environment was to do performance analysis with different controlled restricted bandwidths. Performance was measured in both low and high bandwidth by repetitive testing. The experiment was also run in three different outdoor places for measuring the performance in real world scenario.

Performance measurements were taken to measure video playback quality, playback smoothness, and amount of sleep time for video playback by using both WiFi and radio. The impact of long pauses between segments downloads was studied.

6.2.1 Controlled Environment

For measuring the performance in a controlled environment, two different test cases with restricted bandwidth were considered. During this experiment, only client side bandwidth was controlled. There were no restrictions on server side data transfer.

Behaviour under low available bandwidth

In order to measure the performance of the proposed protocol, 20 playback operations of the video were performed using a player developed based on the proposed protocol. During video playback, the available bandwidth was restricted to 2 Mbps by using the DummyNet network emulator.

Figure 6.22 shows the data transfer rate and downloaded bitrate versions over time for a sample video playback with 2 Mbps restricted bandwidth for Sleep Threshold (S_thresh) (mentioned in Table 4.1) value of 3s. In this figure, each vertical bar defines the data transfer rate during segment download and the triangle defines the bitrate version of each downloaded segment. In this figure, during video playback, the measured transfer rate is consistent and very close to maximum achievable transfer rate except in the beginning of the first segment download and in the beginning of data download after a long pause.

Table 6.4 shows how the S_thresh value effects the major performance metrics. With the increase of S_thresh, Average Playback Version (APV) value decreases, but WiFi Sleep Time (WST) and Cellular Sleep Time (CST) increases. With the increase of the S_thresh value of 1 s to 3 s, WST increased to 68% from 38%, when CST value increased to 28% from 16%. This table indicates that in order to achieve higher amount of sleep time, the player need to compromises with respect to playback version. On the other hand, the Playback Smoothness (PS) value does not show any specific relationship with the change of S_thresh value. For S_thresh value of 1.5 s, there is higher PS value. The probable reason could be that the current bandwidth fits best with the bitrate

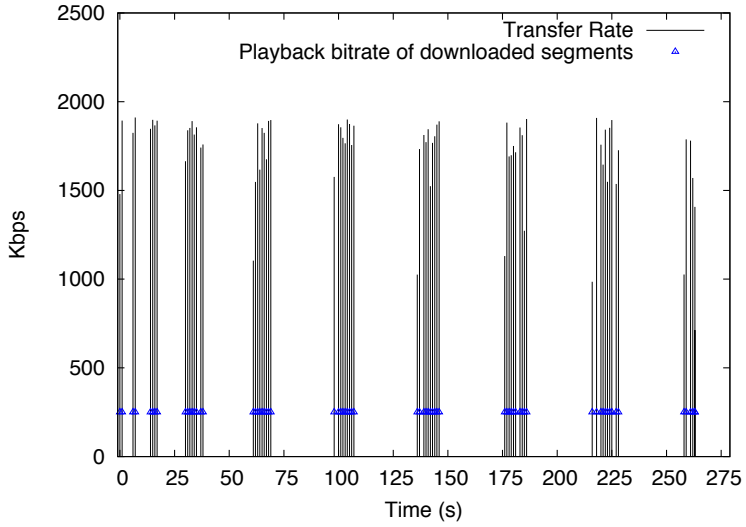


Figure 6.22: Transfer rate and playback bitrate over time (2 Mbps bandwidth, S_thresh = 3 s)

version selected with S_thresh value of 1.5 s. There is also a very high PS for S_thresh value 3 s. This is because the player selected the lowest available bitrate in all the cases to meet the high S_thresh value.

Table 6.4: Performance metrics versus S_thresh (2 Mbps bandwidth)

S_thresh	APV	PS	WST	CST
1 s	2.89	34.2	105.49	43.91
1.5 s	2.87	110.98	117.65	44.73
2 s	2.19	44.93	146.28	62.36
3 s	1.01	196.93	195.27	79.2

In Figure 6.22, at the beginning of each data download, the transfer rate is reduced for the first segment. There could be two possible reasons. Generally, if the idle period in between TCP data downloads is long, TCP terminates the connection with the server. In that case, when data transfer resumes after a long pause, a new TCP connection is created. If the data is downloaded from a CDN that transfers data from the available closest server, when a new TCP connection is created. The server may also change during video playback. This may also introduce additional high Time To First Byte (TTFB) values. It defines the time required to download the first byte after sending the HTTP request. The TTFB value may also be high in the same server while creating a new TCP connection, if the server gets busy with other tasks and the handshake takes some time to

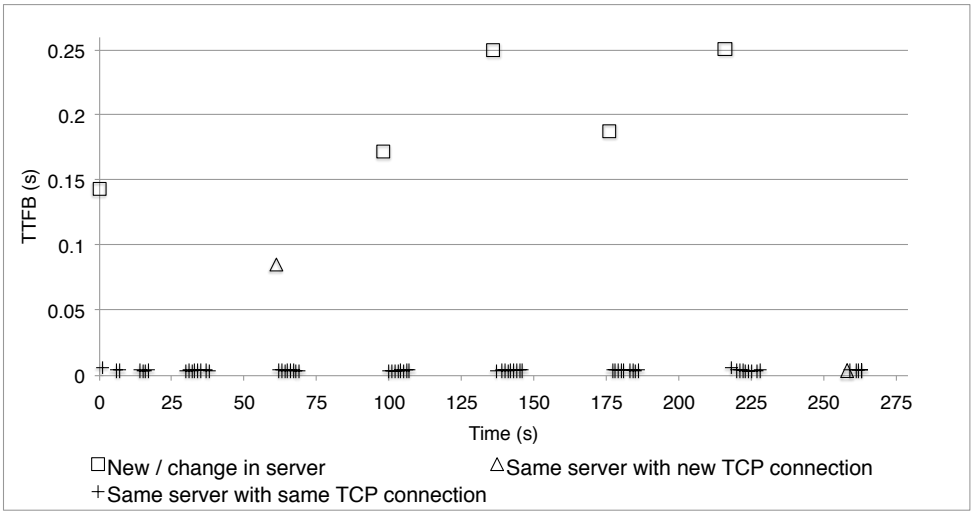


Figure 6.23: TTFB for downloaded segments over time (2 Mbps bandwidth, S_thresh = 3 s)

establish the connection. The second possible reason could be TCP slow start.

Figure 6.23 shows the TTFB values during segment download. There were two servers active during data download. At 100s, 136s, 176s, and 216s of playback, respectoively, the server was changed. In the graph, TTFB value is comparatively high when server is changed. At 62s, the TTFB is also high when the segment is delivered from the same server with a new TCP connection, though it is comparatively lower than the case when segment is downloaded from a new server. The results of the experiments show that, the TTFB value is comparatively high during a server change. This high value also affected transfer rate in the corresponding seconds in Figure 6.22.

Behaviour under high available bandwidth

In order to measure performance of proposed protocol under comparatively high transfer rates, DummyNet was used again to limit the bandwidth to 6 Mbps. Figure 6.24 shows the measured transfer rate and downloaded bitrate versions. In this graph, the transfer rate is consistent in most of the cases and was very close to the maximum available bandwidth. The downloaded bitrate version also increased gradually towards the maximum available bitrate version. The bitrate version remained consistent thereafter.

Table 6.5 shows average performance metrics values for different S_thresh values measured through repetitive experiments. For higher available bandwidth, the APV value is also high compared with the previous case. Playback smoothness is also increased as the player achieved the maximum available bandwidth value within a few seconds of playback duration and was consistent with that bitrate version in most of the cases. PS value is comparatively low for S_thresh values

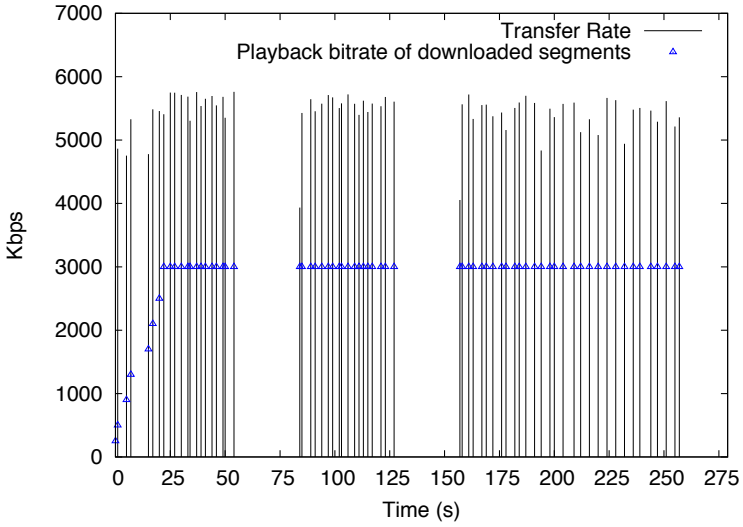


Figure 6.24: Transfer rate and playback bitrate over time (6 Mbps bandwidth, S_thresh = 1 s)

Table 6.5: Performance metrics versus S_thresh (6 Mbps bandwidth)

S_thresh	APV	PS	WST	CST
1 s	7.4	66.92	97.99	38.65
1.5 s	6.6	29.4	115.68	41.35
2 s	4.65	35.7	155.15	51.76
3 s	2.72	32.46	201.05	83.66

greater than 1s. In all three cases, the bitrate value fluctuates between two consecutive bitrate versions to ensure expected sleep time. Unlike low available bandwidth, the PS value is low in this case, as the player switched to different achievable bitrate version instead of staying with lowest available bitrate. The WST and the CST values also increased with increasing S_thresh. With the increase of the S_thresh value 1s to 3s, WST reached to 72% from 35%, when CST value reached 30% from 14%. The measured WST and CST value for different S_thresh values is within 5% of the WST and CST values measured for low available bandwidth.

Similar to Figure 6.22, in Figure 6.24, there is a comparatively low transfer rate measured after a long pause in active download. Investigation results show that this is because of high TTFB values similar to previous case. Figure 6.25 shows the TTFB values for downloaded segments. The TTFB value is high for initial few segments. It may be due to a huge workload on the server or bottleneck connection between the server and the client. These high TTFB values also reduced the transfer rate in corresponding seconds. Similar to previous case, there is a comparatively high

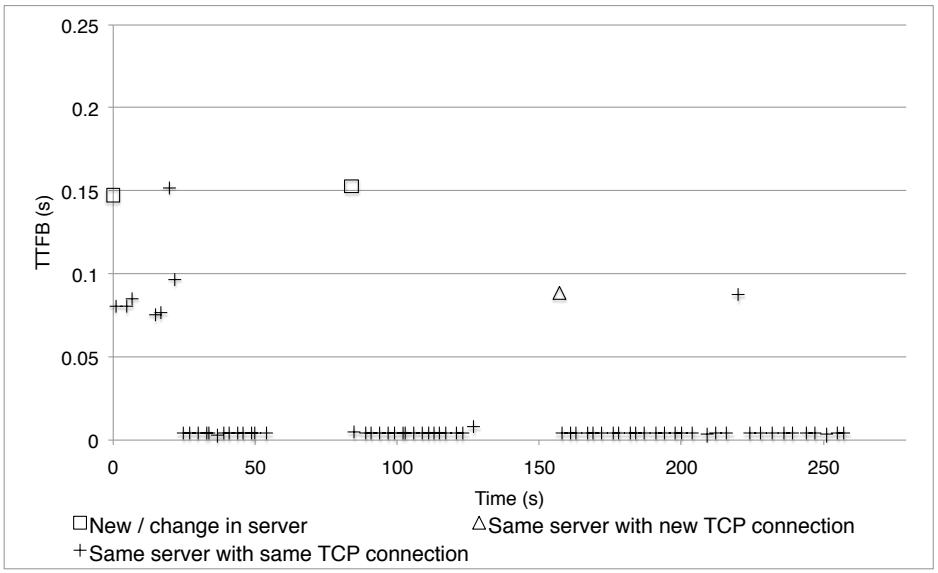


Figure 6.25: TTFB for downloaded segments over time (6 Mbps bandwidth, S_thresh = 1 s)

TTFB value found when player connects with a new server after a long pause between segment download.

6.2.2 Real World Environment

Performance measurement was also done in three different outdoor locations (described in Section 5.2.2) to analyze the performance of the proposed protocol in a real world environment.

Performance measurement on Location 1

In Location 1, the experiment was run 20 times. During the experiment, the available bandwidth was low but consistent.

Figure 6.26 shows the measured transfer rate during segment download with corresponding downloaded bitrate versions for a sample experiment ran on real world location 1. During the experiment, the available bandwidth of the location was approximately 1400 kbps. The player tried to utilize the maximum available bandwidth. Due to some cross traffic generated by other devices active during the experiment, the available bandwidth decreases at some points during the playback. The downloaded bitrate version varies between 250 kbps and 500 kbps versions.

Table 6.6 shows the change in performance measurement metrics values with changing S_thresh values. APV, PS, WST, and CST values show the same behaviour in this case similar to previous cases. Due to low available bandwidth of the test location, the APV value is small in all cases. The bitrate version fluctuates between the two lowest available bitrate versions during the experiment,

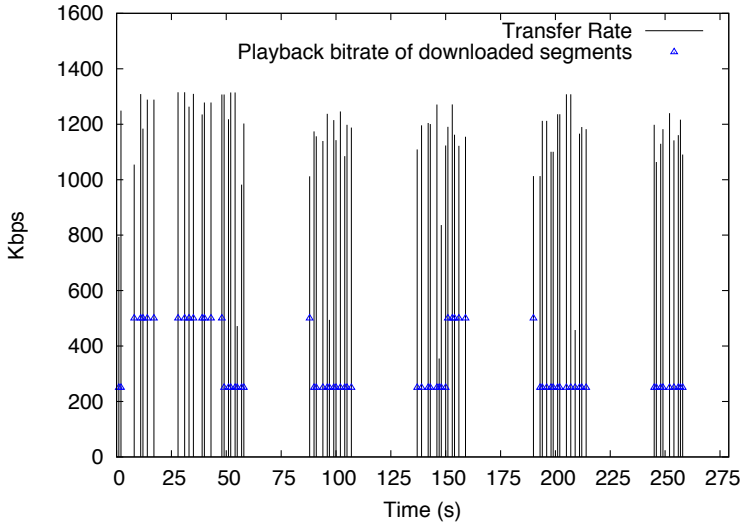


Figure 6.26: Transfer rate and playback bitrate over time (Location 1, S_thresh = 1.5 s)

Table 6.6: Performance metrics versus S_thresh (Location 1)

S_thresh	APV	PS	WST	CST
1 s	1.78	48.84	104.23	63.1
1.5 s	1.68	68.17	116.77	68
2 s	1.25	45.64	131.74	73.68

which provides comparatively high PS values. With the increase of S_thresh values, WST and CST also increases. For S_thresh value of 1s, there was 37% sleep time observed for WiFi and 23% sleep time observed for cellular data transfer. For WiFi data transfer sleep time increases to 42% for S_thresh value of 1.5s and 47% for S_thresh value of 2s. On the other hand, for cellular data transfer, sleep time found for S_thresh value of 1.5s and 2s were 24% and 26%, respectively.

In Figure 6.26, there is also a reduction in transfer rate observed in the beginning of data download (at 1s, 8s, 88s, and 190s) after a long pause in segment download. Upon further investigation, TTFB and TCP slow start both were found to explain the lower transfer rate.

Figure 6.27 shows the initial 100 TCP segment arrival sequence numbers for buffer data download chunk downloaded during the 88th second, at the beginning of the 4th data download period. In this graph, the X-axis defines the arrival time of TCP segments started from HTTP GET request generation time. First TCP segment download starts with an interval of 0.15 seconds after player sends the HTTP GET request. In the beginning of segment download, there are some periodic intervals between TCP segment arrival. Due to TCP slow start, the initial TCP receiver buffer

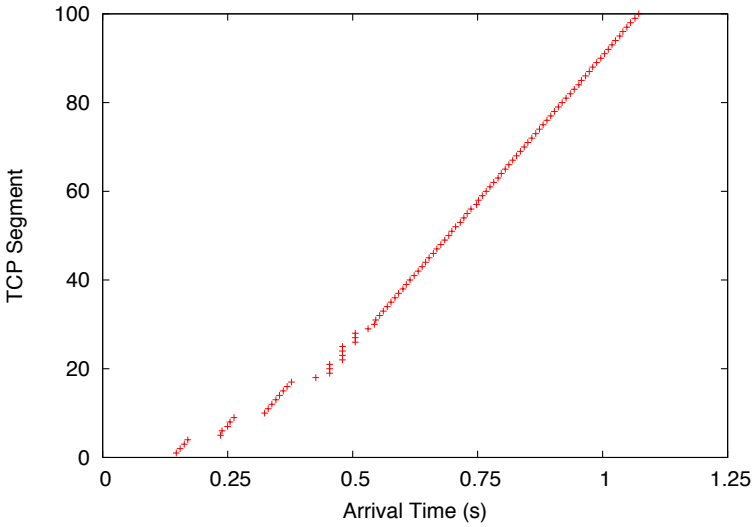


Figure 6.27: Arrival of the first 100 TCP segments containing data for the video segment requested in the 88th second (Location 1, S_thresh = 1.5 s)

size is small and it fills up early by TCP sender buffer.[2] This creates some intervals between TCP segment receptions in client side.

Figure 6.28 shows TTFB values during segment downloads. In this figure, unlike previous cases, TTFB value is comparatively high for all the downloaded segments, on average 0.09s. Above that, it increases in some cases, especially in the beginning of the data download after a long pause in segment download at 1s, 8s, 88s, and 190s due to new TCP connection establishment. From Figure 6.27 and Figure 6.28, it is clear that both high TTFB values and TCP slow start are responsible for comparatively low transfer rate at the 88s mark. Investigation results also found the same reason behind lower transfer rates at 1s, 8s, and 190s, respectively.

Performance measurement at Location 2

Real world experiment 2 was run at an airport. Due to huge competitive traffic and large WiFi hotspot area, fluctuations in transfer rate were observed. The available average bandwidth was high (approximately 14 Mbps).

Figure 6.29 shows the transfer rate measured during segment download with corresponding bitrate versions of downloaded segments. Due to high available bandwidth, the player reached the maximum available bitrate version (3000 kbps) very quickly (in 8th segment) and sustained that bitrate version for the remaining segments.

[2] `http://tools.ietf.org/html/rfc793`, accessed 24-August-2012

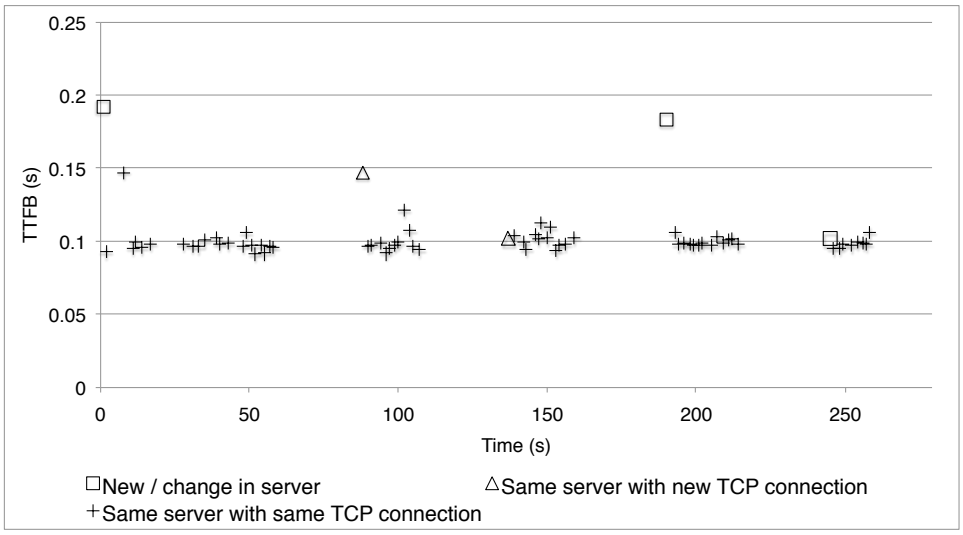


Figure 6.28: TTFB for downloaded segments over time (Location 1, S_thresh = 1.5 s)

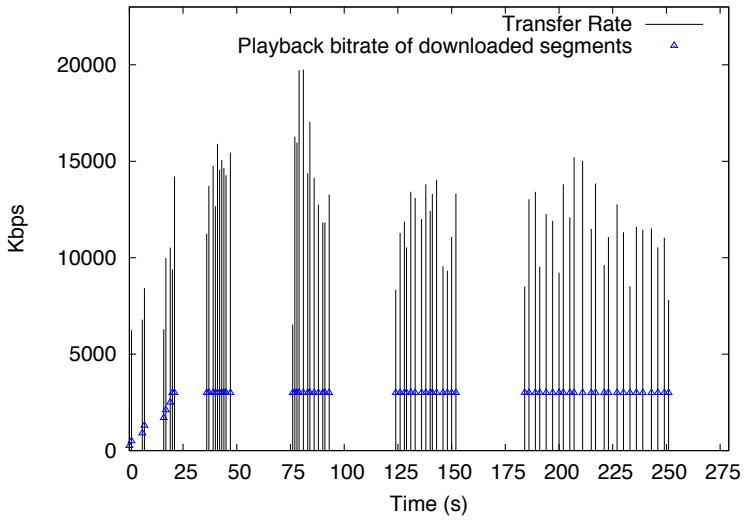


Figure 6.29: Transfer rate and playback bitrate over time (Location 2, S_thresh = 1.5 s)

Table 6.7 shows performance measurement metric values for different S_thresh values. Due to high available bandwidth, the APV and PS values are high for S_thresh value of 1s and 1.5s. The player maintains the highest available bitrate version most of the time. For S_thresh values of 1.5s and 2s, the APV and PS values reduced due to player-provided constraints on bitrate versions. With the increase in S_thresh values s to 3s, the WST increases from 53% to 70% and the CST increases from 27% to 38%.

Table 6.7: Performance metrics versus S_thresh (Location 2)

S_thresh	APV	PS	WST	CST
1 s	6.44	53.4	148.82	74.47
1.5 s	6.25	46.29	157.84	83.63
2 s	5.03	25.67	166.4	88.8
3 s	3.26	20.18	195.81	107.28

Similar to the previous cases, there were reductions of the transfer rate measured in the beginning of data download after a long pause in segment download. Investigation result shows that TTFB doesn't have much impact on transfer rate reduction. Instead, TCP slow start is mostly responsible.

Figure 6.30 shows the initial 100 TCP segment sequence number for segment downloaded in 124th second with bitrate version 3000 kbps. In this figure, it is clear that TCP slow start is affecting the transfer rate. It took approximately 0.75 seconds to start back to back TCP segment downloads.

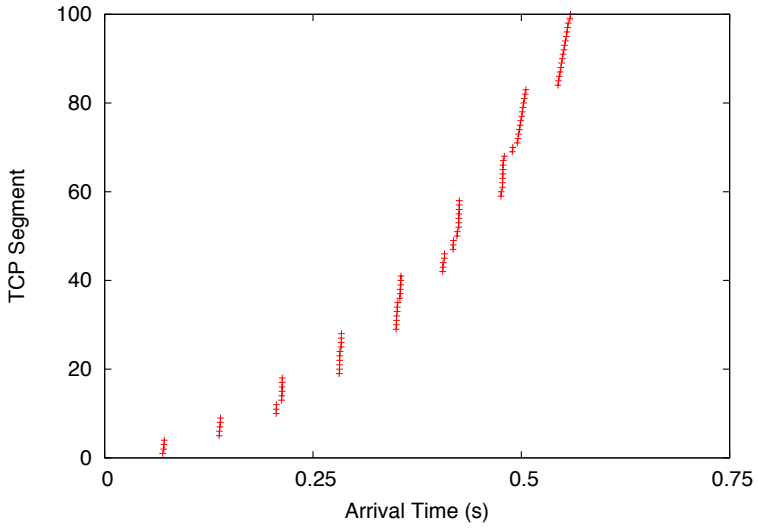


Figure 6.30: Arrival of the first 100 TCP segments containing data for the video segment requested in the 124th second (Location 2, S_thresh = 1.5 s)

Figure 6.31 shows TTFB values for the downloaded segments. In this figure, the average TTFB value found for downloaded segments is more than 0.05 seconds. Initially, there is a high TTFB found for the first segment. The TTFB value is also high for next few segments. These delays may be due to server side workload. At 75 seconds, the content started to download from another

server with a new TCP connection. There were also new TCP connections created at 124 seconds and 184 seconds. There were no high TTFB values found during the new TCP connection, though there are few increase in TTFB value found during middle of the data download. From this graph, it is clear that TTFB value doesn't have much impact on lower transfer rate in beginning of data download after a long pause in segment download in this scenerio. Rather TCP slow start is mainly responsible for reduction of transfer rate on 76 seconds, 126 seconds, and 184 seconds.

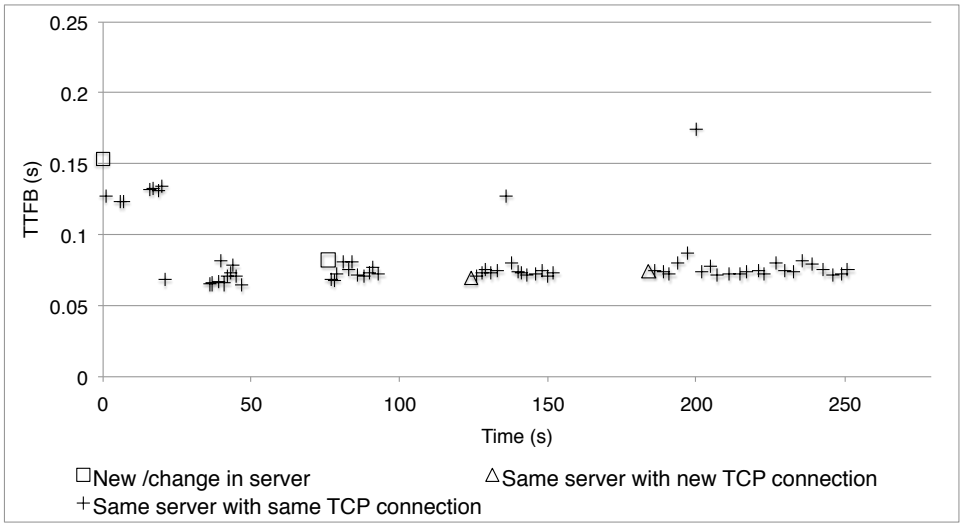


Figure 6.31: TTFB for downloaded segments over time (Location 2, S_thresh = 1.5 s)

Performance measurement in Location 3

Real world experiment 3 was conducted at a residential location using a home WiFi network connection. During the experiment, the available bandwidth was very high, though it varied due to competitive traffic generated by other devices.

Figure 6.32 shows the transfer rate and playback bitrate of downloaded segments. Due to high available bandwidth, the player achieved maximum available bitrate in the 9th segment and sustain in that bitrate version during the rest of the playback.

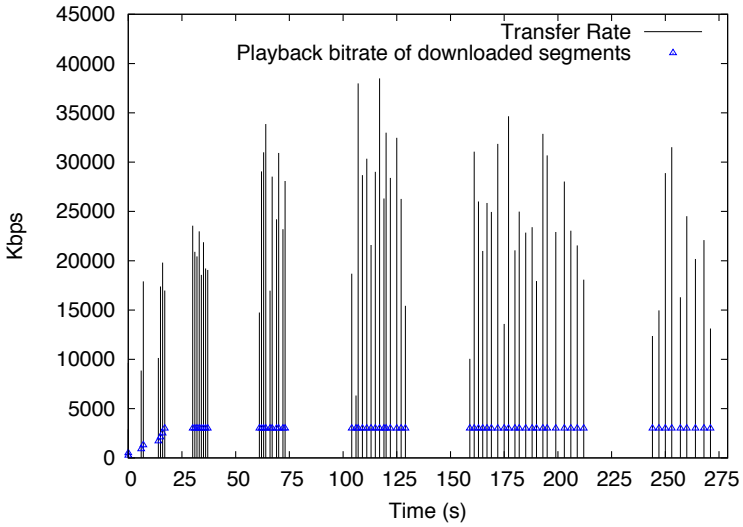


Figure 6.32: Transfer rate and playback bitrate over time (Location 3, S_thresh = 1.5 s)

In Location 3, due to high available bandwidth, it took a very short amount of time to download the segments. For S_thresh values of 1 s, 1.5 s, and 2 s, the APV value is same and it reached the maximum achievable value for the proposed protocol. The PS value is also the same due to stable choice of bitrate versions. The achieved WST and CST values are also very close for different S_thresh values. For S_thresh value of 3 s, the APV value was reduced. The PS value is also low as the bitrate version fluctuates among three bitrate versions to achieve expected sleep time during segment download. In Location 3, the player achieved a very high amount of sleep time during playback. For S_thresh value of 1 s, 1.5 s, and 2 s WST is more than 60% of video duration which reached more than 70% in the case of S_thresh value of 3 s. Similarly, the CST is also high in all the cases (26% for S_thresh value of 1 s and 38% for S_thresh value of 3 s).

Table 6.8: Performance metrics versus S_thresh (Location 3)

S_thresh	**APV**	**PS**	**WST**	**CST**
1 s	7.58	82.91	172.46	72.21
1.5 s	7.58	82.91	175.5	74.01
2 s	7.58	82.91	175.55	74.41
3 s	3.7	19.56	196.94	107.36

There was also reduced transfer rate observed in the beginning of data download after a long pause in segment download, shown in Figure 6.32. Figure 6.33 shows TTFB values during segment download. Figure 6.34 shows the initial 100 TCP segments downloaded for the video segment

downloaded in 162th second. In this scenario, only TCP slow start was found to be responsible for reduced transfer rate.

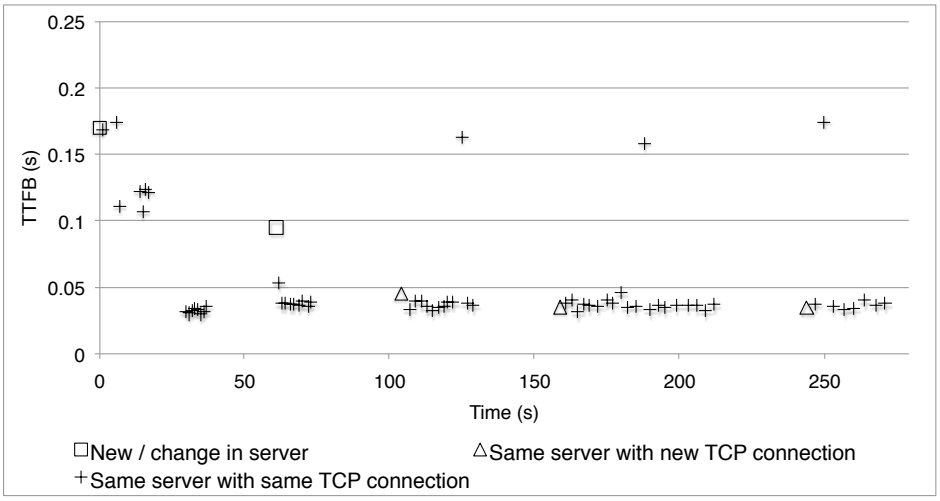


Figure 6.33: TTFB for downloaded segments over time (Location 3, S_thresh = 1.5 s)

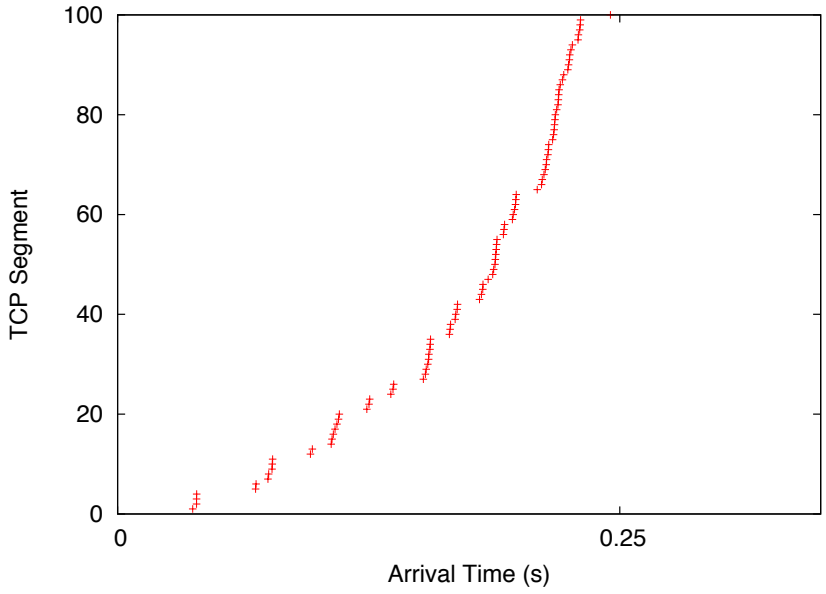


Figure 6.34: Arrival of the first 100 TCP segments containing data for the video segment requested in the 162th second (Location 3, S_thresh = 1.5 s)

6.3 Summary

In this chapter, validation of proposed protocol is done by showing how different parameter values impacting the performance of the protocol. Different alternative designs was also evaluated. The advantage of using dynamic buffer over small and large buffer is shown. By using dynamic buffer, possible WiFi sleep time increased by up to 6.4% of the data transfer time, while for cellular radio achieved possible sleep time is 22% over no sleep for small buffer of 6 second used in Adobe OSMF version 1.6. Adaptive buffer also reduce data wastage up to 82% in the case of playback interruption in 10th second by user over 30-second fixed buffer. The performance improvement achieved by using three step bitrate selection mechanism is also described. The experiment results depict that it is possible to estimate sleep possibilities before downloading the segments more accurately for large segments, 86% in Location 1, and 95% each in Location 2 and Location 3. Performance analysis of the proposed protocol is also done in different test locations with varying network condition. In all the test locations, the proposed protocol achieved a high amount of possible sleep time (up to 70% for WiFi and more than 35% for cellular radio) with the sacrifice of playback bitrate versions. High playback smoothness is found in most of the test locations. The impact of long pause between segment download is also examined. TCP creates a new connection when there is a long pause between segment download either with same server or with a new server. In this case, measured transfer rate can be reduced due to high TTFB values and TCP slow-start especially when a new TCP connection is created with a new server.

Chapter 7

SUMMARY AND CONCLUSION

Energy efficiency is a great challenge for the battery powered and size constrained mobile devices nowadays. Wireless network interfaces have been identified as among the most energy intensive components of these devices. On the other hand, with the rapid growth in popularity of multimedia, multimedia applications (e.g. video streaming) have become common on mobile devices. Due to the large amount of data downloaded in multimedia applications, the wireless NIC is kept awake for long time periods, which causes a large energy consumption. There are several previous studies that have focused on reducing energy consumption during multimedia data transfer. None of these works has considered energy efficiency in the context of HTTP adaptive streaming. In this thesis, an energy efficient HTTP adaptive streaming protocol has been proposed. This protocol is implemented by modifying an open source HTTP adaptive streaming player. Performance analysis was done by running experiments in both a lab environment and in real world environments. By introducing energy efficient buffer management and bitrate selection mechanisms, this approach is able to achieve radio sleep time during video playback (by more than an estimated 70% for WiFi and 35% for 3G/EDGE) during data transfer.

7.1 Thesis Summary

There are several recent research projects that have measured the performance of HTTP adaptive streaming bitrate adaptation mechanisms used in existing players. There is also some work that has proposed new bitrate adaptation mechanisms for HTTP adaptive streaming. All the existing and previous proposed HTTP adaptive streaming methods carry out bitrate adaptation based on network conditions and device capacity. No prior work has explored the opportunity of reducing energy consumption by requesting a comparatively lower bitrate version in order to gain higher energy efficiency. Also, instead of considering only the current network conditions during bitrate selection, the proposed protocol considers the possible sleep time if a candidate bitrate version is chosen. During bitrate selection, the previous bitrate failure history is also considered to avoid repetitive failure of the same bitrate version.

It is possible to reduce device energy consumption during data transfer by deploying an efficient

buffer management policy. With a small buffer, the amount of data that can be downloaded in a burst is small and there can be only small duration gaps between periods of data transfer. On the other hand, with a big buffer, there will be more data wastage if the user interrupts playback early. A big buffer also increases the observed delay until video quality improves, when improving network conditions allow a higher bitrate version to be selected. The buffer could also be flushed and higher quality content downloaded, but that is a significant waste of bandwidth. In the proposed protocol, the buffer size is kept moderate and it increases gradually with the playback duration. Experimental results show that it is possible to achieve long radio sleep times during data transfer by using a moderate sized player buffer.

The proposed protocol is implemented by modifying the Adobe OSMF player version 1.6, an open source HTTP adaptive streaming player. The Adobe OSMF player version 1.6 requests segments based on observations of the current network conditions. The ratio between segment playback duration and download duration, named the Download Ratio, is used to make bitrate selection decisions. In the implementation of the proposed HTTP adaptive streaming protocol, two separate exponential moving averages of the Download Ratio (DR), one smoothed using a lower weight value for the most recent measurement and another with a higher weight value, are used for bitrate upshift and downshift decisions, respectively. Adobe ActionScript is used for the implementation of the proposed protocol.

In order to measure the performance of the proposed protocol, a number of experiments were run in both a lab environment and three real world environments. In the lab environment, the available bandwidth was restricted by using DummyNet, a network emulator, and the performance of the proposed protocol was measured under both high and low available network bandwidth conditions. In both cases, the proposed protocol was able to achieved high amounts of sleep time during data transfer. Experiments were also run using three public WiFi networks to investigate the performance of the proposed protocol in different real world scenarios. In all three locations, the proposed protocol was able to achieve high amounts of sleep time. By tuning a protocol parameter, S_thresh it is possible to achieve high sleep time at the cost of lower video quality.

The impact of long pauses between segment downloads was also observed and quantified. The achieved initial transfer rate was found to be lower when data transfer was resumed after a long pause, owing to (in increasing order of their impact on transfer rate) TCP slow start using the same TCP connection, creation of a new TCP connection on the same server, and a change to a new server.

7.2 Thesis Contribution

The main contributions of this thesis are as follows.

- An energy efficient HTTP adaptive streaming protocol is designed. This protocol includes an efficient buffer size selection mechanism that enables long radio sleep times, while also reducing the data wastage when playback is interrupted by the user. The protocol also includes an efficient three step bitrate selection process. This process ensures conservative bitrate selection and only gradual changes in bitrate versions, so as to increase the potential radio sleep time during data download and provide smoother variation in picture quality.

- The proposed protocol is implemented. Experiments were run to assess the performance improvement achieved with different parameter value choices. Experimental results shows that compared to using a small buffer of 6 s, the proposed adaptive buffer sizing increases the possible WiFi sleep time during data transfer by up to 6.4% of the data transfer time and possible cellular sleep time by up to 22% of the data transfer time. Big buffer sizing also increases the amount of data wastage in the case of playback interruption by user. In the proposed protocol, the adaptive buffer sizing reduces data wastage up to 82%, over a 30-second fixed buffer, in the case of playback interruption in 10 second video playback. The proposed protocol is also able to predict sleep possibility up to 95% cases for large segments.

- Performance evaluation of the proposed protocol is done by running experiments under different network conditions in both a lab environment and real world environments. Three different real world test locations were used, with differing network conditions due to varying network capacity and concurrent traffic. The experimental results show that it is possible to achieve WiFi sleep time during video playback by more than 70%, and by more than 35% for cellular, at the cost of somewhat reduced video quality. The amount of possible energy savings depends on network conditions.

- The impacts of long pause between segment downloads were also observed and quantified. It was found that a particularly high cost is incurred when a new server delivers the segments following the pause period.

7.3 Discussion

The main goal of this thesis is to design and implement an energy efficient HTTP adaptive streaming protocol for mobile devices. However, rather than using a mobile device such as a smartphone for the performance evaluation, a laptop is used. The reason behind using a laptop is to get more experimental flexibility. In this thesis, instead of providing device specific experimental results, the possibility of achieving higher amount of sleep time is shown for any kind of mobile device. Experimental results achieved for a smartphone on a cellular network may vary because of more variable network conditions and long wake up mode to sleep mode conversion times.

Experiments were run in both a controlled lab environment and three real world environments with different network conditions in each. The network conditions in the four test locations do not reflect all the possible network conditions. The experimental results may differ for other possible network conditions.

With the rapid growth of technology broadband Internet speed is getting faster (for e.g., in 4G[1] standard). The faster Internet speed will not only reduce segment download time but will also increase the amount of possible sleep time in the proposed protocol.

In the proposed protocol all the modifications are done in the client side. During experiment it was expected that there are no implications on the server side and data will be transferred from the server at the possible highest rate. If there are any constraints provided in the server side (for example pacing for controlling data flow [3]), the amount of possible sleep time will be reduced.

There are several previous research works focused on reducing energy consumption during video streaming. None of these works investigated the possibility of reducing energy consumption in the context of HTTP adaptive streaming. Thus in this thesis, the experimental results were not compared with those of any previous work.

7.4 Future Work

In this thesis, experiments were performed only for a laptop client. In the future, experiments can also be carried out for different types of mobile devices, such as smartphones. Also, only four test locations were used. Experiments can be run in several other environments to measure the performance of the proposed protocol under different network conditions.

For the experiments in this thesis, all the segments were downloaded from Akamai HD servers using a video provided as a sample for the Adobe OSMF player. These segments are small in size and playback duration. Experiments could also be carried out for other videos with longer segments.

Finally, in this thesis, the quality of the viewing experience was measured using two parameters, Average Playback Version (APV) and Playback Smoothness (PS). In the future, human subjects could be used to provide qualitative assessments.

[1] `http://chronosmedia.net/what-is-4g/`, accessed 18-December-2012

Appendix: Code for Implementing the Proposed Protocol

```
/*****************************************************************
 The following section describes the Buffer Selection mechanism
 of proposed protocol
 *****************************************************************/

// Setting the buffer parameters
// OSMFSettings is the configuration class
override public function set bufferTime(value:Number):void{
    super.bufferTime = value;
    _desiredBufferTime_Min = Math.max(OSMFSettings.hdsMinimumBufferTime, value);
    _desiredBufferTime_Max = _desiredBufferTime_Min +
     OSMFSettings.hdsAdditionalBufferTime;
    _initialBufferTime_Max = _desiredBufferTime_Min +
    OSMFSettings.hdsAdditionalBufferTime;
    _buffer_Min = OSMFSettings.MinBufferTime;
    _buffer_Max = OSMFSettings.MaxBufferTime;
    _buffer_MaxReachTime = (OSMFSettings.MaxBufferReachTime)/100;
    dynamicBuffer = OSMFSettings.dynamicBuffer;
    _max_min_difference = _buffer_Max - _buffer_Min;
    }
// Called in the beginning of each segment download
private function onBeginFragment(event:HTTPStreamingEvent):void{

    if (_initialTime < 0 || _seekTime < 0 || _insertScriptDataTags ||  _playForDuration
        >= 0)
    {
        if (_flvParser == null){
            if (_enhancedSeekTarget >= 0 || _playForDuration >= 0){
            _flvParserIsSegmentStart = true;
            }
            _flvParser = new FLVParser(false);
        }
        _flvParserDone = false;
    }

    var _url:String = event.url;
    var _bitRate:String = _url.substr(_url.lastIndexOf("_")+ 1, _url.length);
    new_brate = Number(_bitRate);

// Dynamic buffer selection mechanism in proposed protocol
```

```
    if(dynamicBuffer)
    {
    // Check whether dynamicBuffer selection process is on or not
    var _proposedBufferTime:Number;
    _arrayLogBitRate[_counterBitRate] = new_brate;
    _arrayLogBufferSize[_counterBitRate] = _desiredBufferTime_Max;
    _counterBitRate++;

        if(_counterBitRate > 1) {
            // Calculate the proposed Maximum buffer
            _proposedBufferTime = Math.round( _initialBufferTime_Max +
            ((_max_min_difference * (_lastValidTimeTime-_seekTarget)) /
                    (_buffer_MaxReachTime * video_length)));
            //Check whether the proposed Maximum buffer is larger
            //than the maximum buffer or not
            if(_proposedBufferTime > _buffer_Min && _proposedBufferTime <
            _buffer_Max)
                    _desiredBufferTime_Max = _proposedBufferTime;
            else if (_proposedBufferTime <= _buffer_Min)
                    _desiredBufferTime_Max = _buffer_Min;
            else if (_proposedBufferTime >= _buffer_Max)
                    _desiredBufferTime_Max = _buffer_Max;

            if(metrics.downloadRatio > 1) {
            // Calculate minimum buffer
                    _desiredBufferTime_Min = _buffer_Min + 2;
            } else
                    _desiredBufferTime_Min = _buffer_Min;
        }
    }
}



/*****************************************************************
 The following section describes the bitrate selection: step 1 and
 step 2 of proposed protocol
 *****************************************************************/

override public function getNewIndex():int
{

    //Bitrate selection Step1 in proposed protocol

    // The downloadRatio is
    //        "playback time of last segment downloaded" /
    //        "amount of time it took to download that whole segment, from
    //             request to finished"
    // The switchRatio[proposed] is
    //        "claimed rate of proposed quality" /
    //        "claimed rate of current quality"
    //
    // There are two download ratio, one is smoothed with high weight value,
    //  named downloadRatio_down and another is smoothed with
```

```
// low weight value, named downloadRatio_up.
// There are exactly four cases need to deal with:
// 1. The downloadRatio_down is < 1 and < switchRatio[current-1]:
//        Bandwidth is way down, switch to lowest rate immediately
//        (even if there's an intermediate that might work).
// 2. The downloadRatio_down is < 1 but >= switchRatio[current-1]:
//        Player should be able to keep going if it go down one level.
// 3. The downloadRatio_up is >= 1 but < switchRatio[current+1]
//    OR no available rate is higher than current:
//        Steady state. Select the ner version as previous.
// 4. The downloadRatio_up is >= 1 and > switchRatio[current+1]:
//        Player can switch up to rate N where N = Current Index + 1
//        downloadRatio is still > switchRatio[N]

var proposedIndex:int = -1;
var switchRatio:Number;
var dr_array:Array = getEMAvgdownloadRatio().split(" ");
// Download Ratio smoothed with lower weight value;
// EWMA for weight value= 0.1
var downloadRatio_up:Number = Number(dr_array[0]);
// Download Ratio smoothed with higher weight value
// EWMA for weight value= 0.88
var downloadRatio_down:Number = Number(dr_array[1]); //
var SleepPossible_EMA:Number;


if (downloadRatio_down < 1.0)
{
        // First checking if the player is able to switch down.
    if (httpMetrics.currentIndex > 0) {
          switchRatio = getSwitchRatio(httpMetrics.currentIndex - 1);

        if (downloadRatio_down < switchRatio) {
                // Case #1, switch to the lowest index
                proposedIndex = 0;
        } else {
                // Case #2, down by one
                proposedIndex = httpMetrics.currentIndex - 1;
        }
    }
} else
{
    // Cases #3 and #4

    // First check to see if it is able to switch up.
    if (httpMetrics.currentIndex < httpMetrics.maxAllowedIndex)
    {
        switchRatio = getSwitchRatio(httpMetrics.currentIndex + 1);
        if (downloadRatio_up < switchRatio)
        {
         // Case #3, no need to change bitrate version
        }
        else
        {
```

```
        // Case #4, increase bitrate version to one level
                        proposedIndex = httpMetrics.currentIndex + 1;

        }
    }
}


// Bitrate selection Step2 in proposed protocol

// Calculate the expected sleep time from proposed bitrate from step1,
// and check whether it is greater than the  Sleep Threshold (S_thresh), threshold
    sleep
// time for each segment download


var index_SleepPossible_check:Number;
// Proposed Index -1 defines player selected previous index again for
//  bitrate version
if (proposedIndex == -1)
     index_SleepPossible_check = httpMetrics.currentIndex;
else
    index_SleepPossible_check = proposedIndex;

// Checking sleep possible value for step1 suggested bitrate version
var sleepPossible_System:Number = getSleepTime(index_SleepPossible_check);
var sleepPossibe_deducted:Number;
var less_dn_wifiSleepInitiateTime:Boolean = false;
var selected_Index_sleepPossible:Number;
var S_thresh:Number = getS_thresh();
CONFIG::LOGGING
{
        logger.debug("System Suggested Index = " + index_SleepPossible_check + "
            System Suggested Index_Sleep Possible = " + sleepPossible_System);
}


// Comparing step1 suggested sleep possible time with S_thresh
if (sleepPossible_System < S_thresh)
{
    // If sleep possible for this version is less than S_thresh
    // look for lower bitrate version and check
    while (--index_SleepPossible_check > 0) {
            sleepPossibe_deducted = getSleepTime(index_SleepPossible_check);
            CONFIG::LOGGING
            {
                    logger.debug("Index = " + index_SleepPossible_check + "
                        Index_Sleep Possible = " + sleepPossibe_deducted);
            }
            if (sleepPossibe_deducted > S_thresh) {
            // Found expected amount of sleep possible value, now break
                    break;
            }
    }
```

```
            proposedIndex = index_SleepPossible_check;
            less_dn_S_thresh = true;

        }

        if(less_dn_S_thresh == false)
             selected_Index_sleepPossible = sleepPossible_System;
        else
            selected_Index_sleepPossible = sleepPossibe_deducted;

        CONFIG::LOGGING
        {
                logger.debug("Selected Index = " + index_SleepPossible_check + " Selected
                    Index_Sleep Possible = " + selected_Index_sleepPossible);
        }

        return proposedIndex;
    }

    // get S_thresh value
    private function getS_thresh():Number
    {
        return OSMFSettings.S_threshold;
    }

    // Get switch ratio value, same process as OSMF player version 1.6
    private function getSwitchRatio(index:int):Number
    {
        return httpMetrics.getBitrateForIndex(index) / httpMetrics.getBitrateForIndex(metrics
            .currentIndex);
    }

    // Calculate the expected sleep possible time
    private function getSleepTime(index:int):Number
    {
        // Calculate Expected Sleep Time from bitrate and EWMA of transferrate
        var EstimateDownloadDuration:Number = (Avg_Fragment_Playback_Duration * httpMetrics.
            getBitrateForIndex(index)) / getEMATransferRate();
        var EstimatedSleepTime:Number = 0;

        CONFIG::LOGGING
        {
            logger.debug( " Estimate Sleep Possible = "+ httpMetrics.fragmentDownloadDuration
                 + " " + httpMetrics.getBitrateForIndex(index) + " " + httpMetrics.
                fragmentSize);
        }

        // Check whether download duration of the index is bigger than the
        // playback duration, if yes, it proposes the lowest bitrate version
        if (EstimateDownloadDuration < Avg_Fragment_Playback_Duration)
        {
                EstimatedSleepTime = (Avg_Fragment_Playback_Duration -
                    EstimateDownloadDuration);
```

```
            return EstimatedSleepTime;
        }
        else return 0;
    }


    // Calculating EWMA value of transfer rate
    private function getEMATransferRate():Number
    {

        var transferRate:Number = httpMetrics.fragmentSize/httpMetrics.
            fragmentDownloadDuration;
        var ema_multiplier:Number = 0.888;

        if (est_counter == 0)
                ema_transferRate = transferRate;
        else if (est_counter == 1)
                ema_transferRate = (transferRate + last_transferRate)/2;
        else ema_transferRate = (transferRate - ema_transferRate) * ema_multiplier +
            ema_transferRate;

        last_transferRate = transferRate;
        est_counter++;
        return ema_transferRate;
    }


    // Calculate EWMA value of Download Ratio
    private function getEMAvgdownloadRatio():String
    {
        var _i:Number;
        var current:int = getTimer();
        var total_downloadRatio:Number = 0;
        var total_downloadRatio1:Number = 0;
        var total_downloadRatio_f4:Number = 0;
        var counter_downloadRatio:int = 0;
        var counter_downloadRatio1:int = 0;
        var Tp:Number;
        var Tp1:Number;
        var ema_multiplier:Number;
        var ema_multiplier1:Number;
        var ema_downloadRatio_initial:Number = 0;

        index_value[index_counter] = httpMetrics.currentIndex;
        downloadRatio_values[index_counter] = httpMetrics.downloadRatio;
        index_counter++;

        if( downloadRatio_values.length > 0)
                _i = downloadRatio_values.length;
        else _i = 0;

        if (index_counter > 1 && index_value[index_counter-1] == index_value[index_counter -
            2])
        {
```

```
        if (starting_index == 1000) {
        starting_index = index_counter-2;
        SMA = (downloadRatio_values[index_counter - 1] + downloadRatio_values[
            index_counter - 2])/2;
        ema_downloadRatio = SMA;
        ema_downloadRatio1 = SMA;
        } else {
        ema_multiplier_low = 0.181;
        ema_downloadRatio_up = (downloadRatio_values[index_counter-1] - ema_downloadRatio
            ) * ema_multiplier_low + ema_downloadRatio_up;
        ema_multiplier_high = 0.888;
        ema_downloadRatio_down = (downloadRatio_values[index_counter-1] -
            ema_downloadRatio1) * ema_multiplier_high + ema_downloadRatio_down;
        }
        return ema_downloadRatio_up + " " + ema_downloadRatio_down;

    } else
    {
        starting_index = 1000;
        return httpMetrics.downloadRatio + " " + httpMetrics.downloadRatio;
    }

}


// Calculate Download Ratio, same as OSMF player version 1.6
getDownloadRatio(){

    if (!isNaN(fragmentDuration) && !isNaN(downloadDuration) && downloadDuration > 0)
    {
        downloadRatio = fragmentDuration / downloadDuration;

    }
    return downloadRatio;
}



/*****************************************************************
 The following section describes the bitrate selection: step 1 and
 step 2 of proposed protocol
 *****************************************************************/
// Most of the code of this section is kept same as Adobe OSMF
// version 1.6 except canAutoSwitchNow(newIndex) method

// Check the switching conditions; modified in proposed protocol
protected function canAutoSwitchNow(newIndex:int):Boolean
{
    var failed_index_lowest:Number = 0;
    var max_up_failed_index_lowest:Number = 0;

    if (dsiFailedCounts[newIndex] >= 1)
    {
        var current:int = getTimer();
```

```
            // Check the time interval between last two failure; the restriction period
            // after failure increases with the number of failure;
            // implemented for proposed protocol

            if (current - failedDSI[newIndex] < (dsiFailedCounts[newIndex] *
                DEFAULT_WAIT_DURATION_AFTER_DOWN_SWITCH)) {
                CONFIG::LOGGING
                {
                        logger.debug("canAutoSwitchNow() - ignoring switch request because
                            index has " + dsiFailedCounts[newIndex]+" failure(s) and only "+
                            (current - failedDSI[newIndex])/1000 + " seconds have passed
                            since the last failure. Wait Period = " + (dsiFailedCounts[
                            newIndex] * DEFAULT_WAIT_DURATION_AFTER_DOWN_SWITCH)/1000 + "
                            seconds.");
                }
                return false;
            }
        }
        // Check whether the failure count is more than 3 or not;
        // if yes, there will be a higher amount of restriction period
        else if (dsiFailedCounts[newIndex] > DEFAULT_MAX_UP_SWITCHES_PER_STREAM_ITEM)
        {
                return false;
        }

        return true;
    }

    // Initialize fail count counter; same as Adobe OSMF version 1.6
    private function initDSIFailedCounts():void
    {
        if (dsiFailedCounts != null)
        {
                dsiFailedCounts.length = 0;
                dsiFailedCounts = null;
        }

        dsiFailedCounts = new Vector.<int>();
        for (var i:int = 0; i < dsResource.streamItems.length; i++)
        {
                dsiFailedCounts.push(0);
        }
    }

    // Increment counter if failure occurs; same as Adobe OSMF version 1.6
    private function incrementDSIFailedCount(index:int):void
    {
        dsiFailedCounts[index]++;

        // Start the timer that clears the failed counts if one of them
        // just went over the max failed count
        if (dsiFailedCounts[index] > DEFAULT_MAX_UP_SWITCHES_PER_STREAM_ITEM)
        {
            if (clearFailedCountsTimer == null) {
```

```
                clearFailedCountsTimer = new Timer(DEFAULT_CLEAR_FAILED_COUNTS_INTERVAL,
                    1);
                clearFailedCountsTimer.addEventListener(TimerEvent.TIMER,
                    clearFailedCounts);
            }

            clearFailedCountsTimer.start();
        }
    }

    // Reset counter; same as Adobe OSMF version 1.6
    private function clearFailedCounts(event:TimerEvent):void
    {
        clearFailedCountsTimer.removeEventListener(TimerEvent.TIMER, clearFailedCounts);
        clearFailedCountsTimer = null;
        initDSIFailedCounts();
    }
```

References

[1] J. Adams and G. M. Muntean. Power save adaptation algorithm for multimedia streaming to mobile devices. In *Proc. IEEE International Conference on Portable Information Devices*, pages 1 –5, Orlando, FL, May 2007.

[2] S. Akhshabi, A. C. Begen, and C. Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proc. 2nd ACM International Conference on Multimedia Systems*, pages 157–168, San Jose, CA, February 2011.

[3] S. Alcock and R. Nelson. Application flow control in YouTube video streams. *SIGCOMM Comput. Commun. Rev.*, 41(2):24–30, April 2011.

[4] G. Anastasi, M. Conti, E. Gregori, L. Pelusi, and A. Passarella. An energy-efficient protocol for multimedia streaming in a mobile environment. *Pervasive Computing and Communications*, 1:301–312, March 2005.

[5] H. V. Antwerpen, N. Dutt, R. Gupta, S. Mohapatra, C. Pereira, N. Venkatasubramanian, and R. V. Vignau. Energy-aware system design for wireless multimedia. In *Proc. IEEE International Conference on Design, Automation and Test in Europe*, volume 2, page 21124, Paris, France, February 2004.

[6] S. Bagchi. A fuzzy algorithm for dynamically adaptive multimedia streaming. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 7(2):11:1–11:26, March 2011.

[7] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc. 9th ACM SIGCOMM Internet Measurement Conference*, pages 280–293, Chicago, IL, November 2009.

[8] A. Begen, T. Akgul, and M. Baugher. Watching video over the web: Part 1: Streaming protocols. *Internet Computing, IEEE*, 15(2):54 –63, March-April 2011.

[9] S. Benno, J. O. Esteban, and I. Rimac. Adaptive streaming: The network has to help. *Bell Labs Technical Journal*, 16(2):101–114, September 2011.

[10] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. USENIX Annual Technical Conference*, pages 21–21, Boston, MA, June 2010.

[11] S. Chandra and A. Vahdat. Application-specific network management for energy-aware streaming of popular multimedia formats. In *Proc. USENIX Annual Technical Conference*, pages 329–342, Monterey, CA, June 2002.

[12] Y. Cui and V. Roto. How people use the web on mobile devices. In *Proc. 17th ACM International Conference on the World Wide Web*, pages 905–914, Beijing, China, April 2008.

[13] L. De Cicco and S. Mascolo. An experimental investigation of the Akamai adaptive video streaming. In *Proc. 6th Springer-Verlag International Conference on HCI in Work and Learning, Life and Leisure: Workgroup Human-Computer Interaction and Usability Engineering*, pages 447–464, Klagenfurt, Austria, April 2010.

[14] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback control for adaptive live video streaming. In *Proc. 2nd ACM International Conference on Multimedia Systems*, pages 145–156, San Jose, CA, February 2011.

[15] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *Proc. 8th ACM International Conference on Mobile Systems, Applications, and Services*, pages 107–122, San Francisco, CA, June 2010.

[16] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. 10th ACM Internet Measurement Conference*, pages 281–287, Melbourne, Australia, November 2010.

[17] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao. YouTube everywhere: impact of device and infrastructure synergies on user experience. In *Proc. 11th ACM Internet Measurement Conference*, pages 345–360, Berlin, Germany, November 2011.

[18] T. Friedman, R. Caceres, and A. Clark. RTP Control Protocol Extended Reports (RTCP XR). 2003.

[19] T. Hossain. Analyzing mobile youtube data traffic and corresponding energy consumption pattern. In *Course Project CMPT 842, Dept. of Computer Science, University of Saskatchewan*, Saskatoon, Saskatchewan, Canada, April 2011.

[20] T. Hossain. Reducing wifi energy consumption in smartphone. In *Graduate Student Symposium, Dept. of Computer Science, University of Saskatchewan*, Saskatoon, Saskatchewan, Canada, April 2011.

[21] J. Korhonen and Y. Wang. Power-efficient streaming for mobile terminals. In *Proc. 15th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 39–44, Stevenson, WA, June 2005.

[22] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. *Wireless Networks*, 11:135–148, January 2005.

[23] R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of TCP-based rate-control algorithms for adaptive internet streaming of H.264/SVC. In *Proc. 1st ACM International Conference on Multimedia Systems*, pages 157–168, Phoenix, AZ, February 2010.

[24] R. Kuschnig, I. Kofler, and H. Hellwagner. Evaluation of HTTP-based request-response streams for internet video streaming. In *Proc. 2nd ACM International Conference on Multimedia Systems*, pages 245–256, San Jose, CA, February 2011.

[25] C. Liu, I. Bouazizi, and M. Gabbouj. Rate adaptation for adaptive http streaming. In *Proc. 2nd ACM International Conference on Multimedia Systems*, pages 169–174, San Jose, CA, February 2011.

[26] J. M. Lucas and M. S. Saccucci. Exponentially weighted moving average control schemes: properties and enhancements. *Technometrics*, 32(1):1–29, February 1990.

[27] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *IEEE Journal on Selected Areas in Communications*, 25(4):722 –737, May 2007.

[28] Shivajit Mohapatra, Radu Cornea, Nikil Dutt, Alex Nicolau, and Nalini Venkatasubramanian. Integrated power management for video streaming to mobile handheld devices. In *Proc. 11th ACM International Conference on Multimedia*, pages 582–591, Berkeley, CA, November 2003.

[29] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang. QDASH: a QoE-aware DASH system. In *In proc. 3rd ACM Multimedia Systems Conference*, pages 11–22, Chapel Hill, NC, February 2012.

[30] Vinod Namboodiri and Lixin Gao. Towards energy efficient VoIP over wireless LANs. In *Proc. 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 169–178, Hong Kong, China, May 2008.

[31] G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *ACM Communications*, 49(1):101–106, January 2006.

[32] P. Pancha and M. El Zarki. Mpeg coding for variable bit rate video transmission. *Communications Magazine, IEEE*, 32(5):54–66, May 1994.

[33] X. Perez-Costa and D. Camps-Mur. AU-APSD: Adaptive IEEE 802.11e unscheduled automatic power save delivery. In *Proc. IEEE International Conference on Communications*, volume 5, pages 2020–2027, Istanbul, Turkey, June 2006.

[34] M. Prangl, I. Kofler, and H. Hellwagner. Towards QoS Improvements of TCP-Based Media Delivery. In *Proc. 4th IEEE International Conference on Networking and Services*, pages 188–193, Guadeloupe, France, March 2008.

[35] A.N. Raja, J. Zhihua, and M. Siekkinen. Energy efficient client-centric shaping of multi-flow tcp traffic. In *Proc. IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, pages 260 –267, Hangzhou, China, December 2010.

[36] G. Ravindra, J. Thaliath, and I. D. Chakeres. In-network optimal rate reduction for packetized mpeg video. In *Proc. 4th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, pages 55–62, Vancouver, British Columbia, Canada, October 2008.

[37] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu. NAPman: network-assisted power management for WiFi devices. In *Proc. 8th ACM International Conference on Mobile Systems, Applications, and Services*, pages 91–106, San Francisco, CA, June 2010.

[38] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. 2003.

[39] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). 1998.

[40] E. Tan, L. Guo, S. Chen, and X. Zhang. PSM-throttling: Minimizing energy consumption for bulk data communications in WLANs. In *Proc. IEEE International Conference on Network Protocols*, pages 123 –132, Beijing, China, October 2007.

[41] John Watkinson. *MPEG Handbook*. Butterworth-Heinemann, Newton, MA, 2001.

[42] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560 –576, July 2003.

[43] S. Y. Wu, J. Hsu, and C. M. Chen. Headlight prefetching and dynamic chaining for cooperative media streaming in mobile environments. *IEEE Transactions on Mobile Computing*, 8(2):173 –187, February 2009.

[44] S. Xiang, L. Cai, and J. Pan. Adaptive scalable video streaming in wireless networks. In *Proc. 3rd ACM International Conference on Multimedia Systems*, pages 167–172, Chapel Hill, NC, February 2012.

[45] Y. Xiao, R. S. Kalyanaraman, and A. Yla-Jaaski. Energy consumption of mobile youtube: Quantitative measurement and analysis. In *Proc. 2nd IEEE International Conference on Next Generation Mobile Applications, Services, and Technologies*, pages 61–69, Cardiff, Wales, UK, September 2008.

[46] H. Yan, R. Krishnan, S. A. Watterson, D. K. Lowenthal, K. Li, and L. L. Peterson. Client-centered energy and delay analysis for TCP downloads. In *Proc. 12th IEEE International Workshop on Quality of Service*, pages 255 – 264, Montreal, Canada, June 2004.

[47] H. Zhu and G. Cao. On supporting power-efficient streaming applications in wireless environments. *IEEE Transactions on Mobile Computing*, 4(4):391 – 403, July-August 2005.