Addis Ababa University
School of Graduate Studies
Addis Ababa Institute of Technology

Caching Scheme in Patch Streaming Multimedia-On-Demand System: Design and Implementation

A thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

By: Dinkisa Aga Bulti

Advisor: Dr. Kumudha Raimond

August, 2011
Addis Ababa

Addis Ababa University

School of Graduate Studies

Addis Ababa Institute of Technology

Caching Scheme in Patch Streaming Multimedia-On-Demand System: Design and Implementation

By: Dinkisa Aga Bulti

Approval by Board of Examiners

Dr.-Ing. Getahun Mekuria Chairman, Department of Electrical and Computer engineering	_____ Signature
Dr. Kumudha Raimond Advisor	_____ Signature
Ato Fitsum Assamnew External Examiner	_____ Signature
Ato Yoseph Abate Internal Examiner	_____ Signature

ACKNOWLEDGMENT

I would like to thank all those who supported me in this thesis work. First, my deepest thank goes to my advisor Dr. Kumudha Raimond for her proper guidance and continuous comments.

My heartfelt gratitude also goes to Mr. Fekadu Aga and his wife and to all members of our family for their honest support all the way through my undergraduate and graduate studies. They have always been a source of my inspiration, and shared my burden to keep me going with their continuous love and care.

Last but not the least, many friends and colleagues deserve my gratitude in every aspect of their effort and support. I gratefully acknowledge these individuals; here are a few of them listed in alphabetical order: Abadi, Adisu, Biniyam, Gemechisa, Huda, Musa, Seifu, and Taye.

ABSTRACT

Multimedia streaming is a technique that allows users to play media content as it is being received without waiting for the entire file to download. It can be live-streaming or On-Demand. In On-Demand multimedia streaming systems, streaming techniques are usually combined with proxy caching to obtain better performance. A number of caching schemes are proposed and some are optimized for a conventional multicast or batch streaming system. These caching schemes reduce the startup latency of this technique. However, patch streaming, which has no startup latency inherent to it, requires extra bandwidth to deliver the media data in patch streams.

This thesis work proposes a caching technique which aims at reducing the bandwidth cost of patch streaming technique. It uses the neural networks' reservoir computing (RC) approach for the popularity prediction in the optimization of media prefix design and selection. The system is implemented and the performance of the proposed caching scheme is compared with the popularity and prefix aware interval caching (2PIC) scheme [prefix part] and patch streaming with no caching using an extensive simulation on a synthetically generated media server workload. The bandwidth saving, hit ratio and concurrent number of clients are used to compare the performance in which the proposed scheme is found to perform better for different caching capacities of the proxy server. The bandwidth saving of as high as 36% can be obtained from the proposed scheme while 32% from 2PIC compared to the no caching scheme for proxy caching capacity of about 16.7% of the total media size on the server. Higher hit ratio is obtained using proposed scheme than the 2PIC scheme. The number of concurrent clients that can be served is large when using proposed scheme followed by 2PIC and no caching schemes.

Key words

Access Pattern, Batch Streaming, Multimedia, Patch Streaming, Proxy Caching, Reservoir Computing, Synthetic Workload Generator, Popularity Prediction

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AC	Admission control
ARMA	Auto-Regressive Moving Average
CBR	Constant Bit Rate
CMA	Cumulative Moving Average
CSRC	Contributing Source Identifier
ESN	Echo-State Network
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IPTV	Internet Protocol Television
MOD	Multimedia-On-Demand
MPEG	Moving Picture Experts Group
LSM	Liquid-State Machine
P-stream	Patch Stream
RC	Reservoir Computing
RFC	Request For Comment
RNN	Recurrent Neural Network
RR	Ridge Regression
RRSE	Root Relative Squared Error
R-stream	Regular Stream
RTSP	Real-Time Streaming Protocol
RTCP	Real-Time Transport Control Protocol
RTP	Real-Time Transport Protocol
SSRC	Synchronization Source identifier
TCP	Transmission Control Protocol
T-stream	Transition Stream
UDP	User Datagram Protocol
UGC	User Generated Content

VBR	Variable Bit rate
VCR	Video-Cassette-Recorder
VOD	Video-On-Demand

1. BACKGROUND and INTRODUCTION

1.1 Motivation

The recent advancements in the broadband computer networks, data compression techniques, and availability of large and efficient mass storage systems have facilitated different Multimedia-On-demand (MOD) services such as TV broadcasting over Internet (IPTV), video-on-demand (VOD), and learning-on-demand and digital libraries. Videos from news, sports, and entertainment sites are more popular than ever. Media servers are being used for educational and training purposes by many universities [19]. A number of social media sites are also common these days. Content delivery technologies and different streaming techniques which make efficient use of available bandwidth are driving forces supporting the possibility of video, audio and other multimedia data (like: animation videos, interactive games, training tutorials and lecture slides, …) streaming. With an efficient standardized data compression technology such as MPEG (Moving Picture Experts Group), huge multimedia files become manageable in computer systems and transferable over networks [1]. An MOD system facilitates playing back of a media object or file from a large collection of media stored on multimedia media server. The multimedia server delivers the streams of media data to the requesting client. The MOD system can be in a download-playback mode or a streaming mode.

Continuous media data has unique characteristics that are not common in discrete (or text based) data. Its data access requires high Disk IO rate and has large memory requirement to store. These media data also needs the server resources to be reserved in advance to meet the real-time transfer requirement. In addition, it tends to last for a relatively long period of time to watch or listen which consumes large amount of resources. The other most important characteristic of continuous data is the interactivity requirement to play, pause, and stop among others. These continuous media data characteristics are the causes for the difficulties of media content delivery. In solving these problems, different mechanisms are proposed and being used. To mention some of them: Data replication, data compression, data and resource sharing. Data replication uses different replicated media servers located at different geographic area serving the same media in coordination whereas data compression is used to make the bulk of media object manageable. Data and resource sharing are usually achieved through caching and

streaming techniques. Caching uses the reference locality in the media request pattern to serve subsequent clients and is usually done at the proxy or local server. Hence, the performance of a resource sharing technique depends on media server workload characteristics. In other words, a resource sharing system which can efficiently predict or capture the access pattern is the one which benefits the most from caching and streaming techniques. A media streaming technique is another resource sharing method to playback the media while receiving. Downloading the whole media is not required to start the playback unlike the download and playback technique. An overview of major streaming techniques is presented in the next section.

1.2 Overview of Multimedia streaming techniques

Mostly available content delivery technologies rely on broadcasting, multicasting and unicasting. Multicasting serves as a compromise between high resource demand of unicasting and needless network congestion problem of broadcasting. Sometimes, multicasting and unicasting are combined to achieve the requirements of streaming from different perspectives. These streaming techniques are discussed below.

Batch streaming:

In Batch streaming multicast system [10], requests by multiple clients for the same video arriving within a short time interval are batched together and serviced using a single multicast stream. The batching window is designed to group more clients while reducing the startup latency. It makes the clients arriving at different times share a multicast stream, which may incur a long service latency (or waiting time) causing some clients to renege.

Piggybacking:

Piggybacking [9] is a streaming technique in which the later stream is allowed to speed up its streaming bit rate so as to catch up the previous stream and join it. Usually up to 5% speed up is used to have an acceptable playback speed. The resource sharing of this technique depends on the gap between two arrivals. Long time gap cannot be satisfied by the allowed speed up percentage.

Patch streaming:

Patching, introduced in [29], is also a bandwidth-saving approach which can provide real-time service to a client request without start up latency. It initially uses a full multicast stream and then a new unicast patch streams for each subsequent request. When a first request for a video arrives, the server sets up a multicast stream so that later users can also join the stream and receive the missed portion of the media in the patch stream as shown in Fig.1.1. The complete multicast connection is known as *multicast or regular stream* (R-stream) and the unicast connection is called *patch stream* (P-stream). It reduces the request waiting time but requires additional server bandwidth and buffer space at the client than the batch streaming.

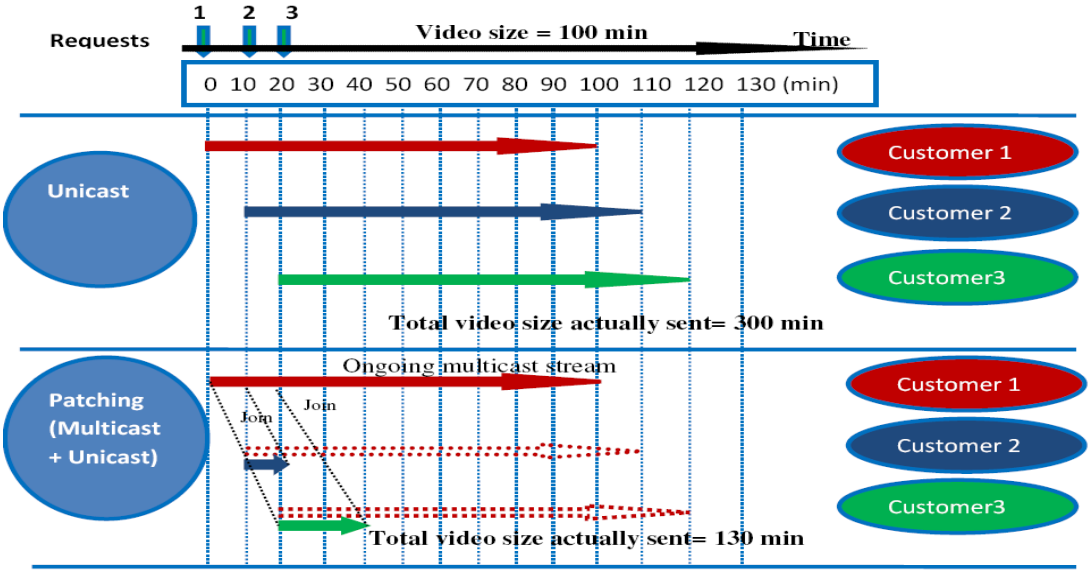


Fig.1.1 Comparison between Unicasting and Patching [3]

1.3 Problem Statement

Usually caching and streaming techniques are combined to enhance the performance of MOD systems. A number of caching techniques are proposed and some are optimized to work with batch streaming system. These caching techniques are mostly used as a means of decreasing the startup latency since batch streaming is characterized by large inherent latency. In patch streaming system, there is no inherent startup latency except network delay. The problem with patch streaming is the extra bandwidth requirement to deliver the P-streams. Therefore, the caching technique in patch streaming MOD system needs to be designed from the point of view of bandwidth requirement minimization. The

existing caching schemes place bandwidth requirement minimization as a secondary performance measures, primarily aiming at the reduction of startup latency.

This work focuses on design and implementation of a caching scheme which tries to minimize the extra bandwidth requirement in patch streaming system by caching media prefixes based on their popularity. To cache the media prefixes in an efficient manner, the knowledge of media server workload and the popularity of the media play a vital role because the media prefix size design and selection depends on the relative popularity of each media. Furthermore, in the real world MOD streaming system, the ability to capture the temporal reference locality in the client access pattern from the real-time media access traffic is also important. The media request arrival rates in the near future need to be estimated to reserve server resources in advance and hence, enhance the caching and streaming scheme performance. This requires a prediction technique for predicting the arrival rate for the next few hours.

1.4 Objectives

The general objective of this work is to design and implement a caching scheme in patch streaming MOD system.

The Specific objectives include:

I. To review the media server workload characteristics: access pattern, popularity distribution models and arrival processes.
II. To develop a synthetic workload generator
III. To design a caching scheme in patch streaming system
IV. To analyze and design popularity prediction technique
V. To implement, evaluate and compare the performance of the proposed scheme with other schemes

1.5 Methodology

The following methodology was used in design and implementation of the caching scheme in patch streaming MOD system.

Literature survey: Existing literatures on patch streaming, caching techniques, media server workload characteristics and models, popularity prediction scheme are reviewed.

Data collection: By reviewing media server workload characteristics and models, a synthetic workload generator was developed. The synthetic workload is used as a test data set for various metrics of caching performance analysis and prediction subsystem.

System design and implementation: A proxy caching scheme proposed in this work was designed and implemented in the patch streaming MOD system. The 2PIC [prefix] scheme was also implemented in the same system as an independent module to facilitate selection between these schemes. The RC technique was used for the prediction of hourly arrival rates; the values used in making optimal selection of media prefixes and resource reservation.

Performance analysis: The scheme is tested by developing a simulation environment for the designed system which simulates the streaming time. The test dataset was (as discussed above) generated from the workload models that better represent the media popularity distribution and access pattern. Performance of the proposed technique was compared with other techniques.

1.6 Scope and limitations of the work

The scope of this thesis is limited to client-server architecture. A central server with a number of media files accepts and responds to client requests either directly or through local (proxy) server. Peer-to-peer media streaming systems are not under the scope of the work and no data replication system is assumed. The test workload assumes a general media server workload. News and other special user interactive media types and the effect of web links are not considered. This work does not try to take the effect of network delay into account and the entire system is tested on the same machine.

1.7 Contribution

This thesis work proposes a new design approach for caching scheme in patch streaming MOD system and introduces a possible implementation mechanism.

1.8 Thesis Outline

The rest of the report is organized as follows: In chapter two, related works are presented. Chapter three reviews the media server workload characteristics and models, and the need for synthetic workload generation. Patch streaming technique is discussed, the cost of patching and the scheme to select media prefixes is designed in chapter four

of the report. The next chapter analyzes and designs a prediction scheme used in the implementation of the Proxy caching patch streaming system. The system design and implementation, the development of synthetic media workload generator are presented in chapter six. Simulation environment setup and performance analysis of the system is in chapter seven and the last chapter presents the conclusions and recommendations.

2. RELATED WORKS

The random access pattern that is reported to exist in discrete file (like texts, images and web objects) access is not common for continuous media files. A sequential access is common for continuous media files. Furthermore, media objects are many times larger than a web or text file and caching a media object as a whole is impracticable. The usual approach in On-Demand media streaming is to cache a selected portion of a media, mostly called hot or frequently accessed portion, at the proxy server. There are immense literatures on this issue and this chapter presents only those closely related caching schemes to this work.

Due to the large size of media objects and clients' partial viewing patterns [15], [16], [20], [21], *selective* or *partial* caching is always being used. These techniques can be divided into two categories: interval-based [2], [4], [5], [7] and prefix and interval-based [5], [6], [18] caching.

Interval-based schemes: Interval-based caching schemes include interval caching (IC) [2], popularity-aware interval caching (PIC) [4], and Client-assisted interval caching (CIC) [7]. All caching techniques which are based on ***Interval*** are characterized by caching intervals between two consecutive requests on a single media object. Fig.2.1 shows this scheme. Stream requests are S_1, S_2, S_3 and intervals are I_{23} and I_{12}. For I_{12} the request S_1 is called ***preceding*** and S_2 is the ***following*** request. The IC scheme introduced in [2] sorts the intervals in increasing size and cache the shortest. The largest interval is the victim of replacement. The goal of IC scheme is to optimize the number of concurrent streams served from the cache. The memory requirement of an interval is proportional to the length of the interval and the playback rate of the stream involved. The PIC scheme [4] uses the reference popularity of multimedia

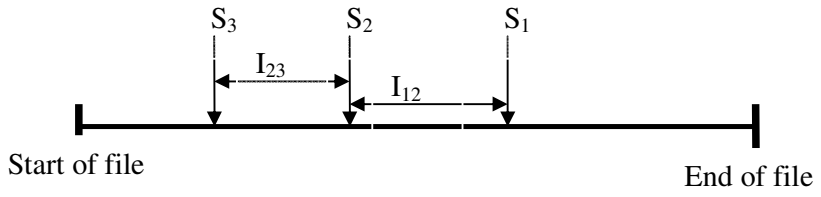


Fig.2.1 Interval Caching

objects as well as the time interval between two consecutive requests to predict the time

of next request and include it in the candidates of caching. The interval formed by the predicted request is called virtual interval (VI) and the request itself is Virtual request or stream as shown in fig.2.2a. Exponential averaging is used in calculating the VI.

Prefix and interval-based schemes: Popularity and Prefix aware Interval Caching (2PIC) [5], Optimal Prefix caching and data sharing (OPC-DS) [18], and dynamic buffer allocation [6] are designed to minimize startup latency and improve bandwidth utilization. As it can be seen from fig.2.2a, the sliding window of a predicted interval proceeds over time and the prefix of popular media objects may not be cached resulting in high startup latency. To solve this problem Kwon et al. in [5] proposed a 2PIC scheme. The 2PIC scheme (shown in fig.2.2b) uses the reference popularity information to decide the targets of prefix caching and calculates the prefix interval as in equ.2.1 below

$$PI_n = \alpha I_{n-1} + (1-\alpha) Avg_{n-1} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (2.1)$$

Where, α-constant between 0 and 1

Avg_{n-1}- the $(n-1)^{th}$ average interval from 1^{st} to $(n-1)^{th}$ request

I_{n-1} – the $(n-1)^{th}$ real interval

In 2PIC, buffer space is allocated to intervals (including prefix interval) by the increasing order of the interval size.

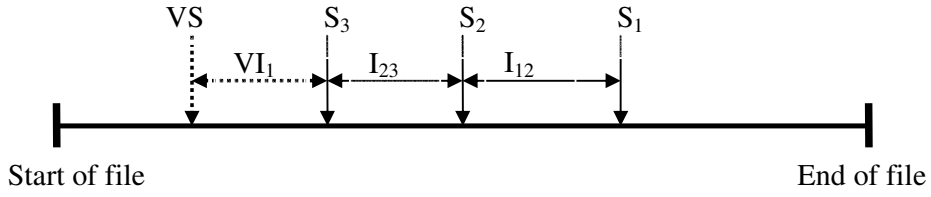


Fig.2.2a Popularity Aware Interval Caching

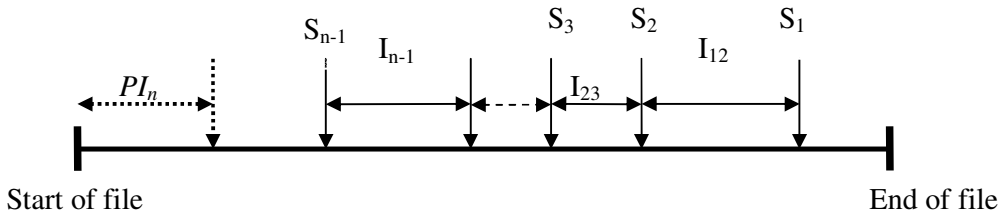


Fig.2.2b Popularity and Prefix Aware Interval Caching

From equ.2.1, it can be observed that the more popular the media object is the smaller the prefix interval and this assures that the prefix of popular files remain in the cache since 2PIC scheme caches the smallest interval. However, for those media objects that have prefix cached, less cache blocks are allocated for popular files than less popular files and with poor prediction capability of the technique, its performance will not be optimal because of the high number of placement and replacement actions. The prefix interval obtained is not optimized in terms of the back-bone network bandwidth consumption. In addition to this, the cached size for each media object is not investigated from patch streaming point of view which transmits the patch stream frequently to achieve minimal startup latency.

Li et al. in [18] designed an optimal prefix size that aims at minimizing the bandwidth requirement on the back-bone network. This scheme caches the optimal video prefix obtained by dynamic programming, at the same time, dynamically caches the interval data consumed by the adjacent displays referencing the same stream in the proxy. Assumes the streaming technique that does not involve any caching at the client and also the client receives only a single stream. However, in patch streaming, a client is intended to receive at least two streams: Regular multicast stream (R-stream) and patch stream (P-stream). Minimizing the data in the p-stream reduces bandwidth usage. And also the amount of data blocks that the client must buffer is minimum compared to its capacity (i.e. these days clients have sufficient capacity to buffer large portion of a media indicating client buffering capacity is no more a problem). Also the cost of simultaneously receiving multiple streams decreases with proper caching.

The number of cached streams increases with an increase in cache size [8] when using interval based stream caching. On the other hand, cache memory is also relatively expensive the same as bandwidth and one cannot simply increase the cache size without cost. Wujuan et al. in [7] proposed a client-assisted interval caching (CIC) which is based on the cost model that tries to balance cache memory and IO bandwidth cost. The CIC scheme uses the cache memory available in the clients to serve the first few blocks of streams so as to reduce the demand on the IO bandwidth of the server.

In [6], a dynamic buffer allocation algorithm was designed for the prefix based on the popularity of the media objects. More cache blocks for most popular videos and less

cache blocks for less popular videos. This scheme makes use of batching technique to serve multiple streams from a single cache improving the service efficiency for most popular videos since they can be batched relatively easily.

To summarize, caching tries to use the reference locality in media access requests. The interval-based caching techniques caches intervals between two consecutive requests to serve the following request and make use of the sequential access pattern. The existing prefix caching schemes gives first priority to the reduction of a start up latency and some of them are optimized for the Batch Streaming system which has an inherent start up latency. The Patch Streaming system which is the focus of this work has no start up latency and the caching for such system (especially the prefix caching) should be designed from the point of view of bandwidth reduction as a number one priority. This is because the Patch Streaming requires extra bandwidth to deliver media data in P-streams in order to reduce the start up latency.

3. STREAMING MEDIA WORKLOAD CHARACTERISTICS and MODELS

Different resource reservation schemes like caching, bandwidth reservation, disk scheduling and others should take into consideration the access pattern and media popularity distribution to achieve a better performance. In other words, studying popularity distribution models, access pattern and arrival processes assist in understanding the benefits that can be gained from caching and streaming. It also helps in generating a synthetic workload which is controllable and suitable for evaluating the performance of caching, streaming and other resource sharing techniques. Since, caching uses the temporal access pattern of media requests to maximize the hit ratio, the knowledge of popularity distribution, media segment popularity or access pattern is the first step in designing best performance caching. To mention some of media workload characteristics: access frequency or popularity, arrival processes, segment popularity and session time. ***Access frequency or popularity*** indicates the total number of reference to a particular media file in a given observation period while the ***arrival process*** models the diurnal and hourly changes in the request rates or inter-arrival times. Segment popularity (internal popularity) and session times are studied under a general term called ***Access pattern.***

This chapter makes a thorough review on these media workload characteristics and models which are used to represent popularity distribution, arrival process and access pattern.

3.1 Media popularity distributions and arrival processes

In this section, different media popularity distributions and characteristics are reviewed. The differences and similarities between short-term and long-term popularity distributions, the daily media request series and hourly arrival rate process are also presented.

Like Web objects, media objects' popularity distributions were used to be approximated by Zipf-like distribution (i.e. the total access of the i^{th} most popular movie is, $y_i = c/i^{(\theta)}$ with skew factor θ and normalization constant c) [2], [25]. However, recent studies [20], [24], [26], [27], [28] showed that long-term media popularity cannot be captured by a

simple Zipf-like distribution. In [24], [27] and [28] Zipf-Mandelbrot and K-transformed Zipf-like distributions are used to approximate the long-term distributions. The Zipf-Mandelbrot [24] is a generalized Zipf-like distribution represented by equ.3.1

$$y_i(x_i) = \frac{C}{(x_i+k)^\theta} \quad \text{...} (3.1)$$

Where y_i – total access for the i^{th} media in the observation period

x_i – Popularity rank of the i^{th} media

θ – Skew factor

C –constant of normalization

k --scaling parameter on the x-axis

K-transformed Zipf-like distribution is another form of Zipf-Mandelbrot distribution representation that includes more parameters and transforms the distribution into the Zipf-like distribution by scaling the y-axis and x-axis with parameters K_y and K_x.

$$y_k = \frac{y+K_y-1}{K_y} \text{..} (3.2)$$

$$x_k = \frac{x+K_x-1}{K_x} \text{..} (3.3)$$

The relation between the scaled variables y_k and x_k follows the simple Zipf-like distribution where as the relation between y and x is Zipf-Mandelbrot distribution given by equ.3.4 [24]

$$y = \frac{C_k K_x^\theta K_y}{(x+K_x-1)^\theta} + 1 - K_y \text{.......................................} (3.4)$$

The stretched Exponential [26] is also proposed for capturing long-term streaming media popularity distributions.

3.1.1 Short-term versus Long-term popularity distributions

Different studies on media workload characteristics show that the differences in popularity distribution are also affected by the time scale. The study on enterprise media

service [20] and educational media service [27] show that the popularity distribution of media objects can be captured by Zipf-like distribution or concatenation of two Zipf-like distributions for short-term observation period (i.e. observation time of up to a month on one of the servers and up to six months on another). The long-term popularity distribution cannot be approximated by a simple Zipf-like distribution. In [24], [26], [27], [28], the long-term popularity distributions are approximated by different distributions like Zipf-Mandelbrot, stretched exponential, and concatenation of two Zipf-like distributions.

3.1.2 Daily and hourly media request arrival rates

Popularity distributions described above show only the total number of views for each media objects within a total observation period. However, streaming media requests exhibits diurnal or seasonal patterns [25]. These diurnal or seasonal patterns can be weekly, daily or hourly patterns. Some media objects tend to remain popular for a long time while other's popularity dies out immediately after its introduction. However, the idea of "rich-get-richer" effect cannot be generalized to model media popularity over a passage of time [26]. This idea tells that if $\boldsymbol{k}$ users have fetched an object, the rate of other users fetching it is proportional to $\boldsymbol{k}$. It implies that a popular object can maintain its popularity continuously, rather than becoming unpopular over time. Nevertheless, this argument is not valid for media objects since they are immutable unlike the web objects which are regularly updated and hence continues to draw more attention. Though "rich-get-richer" idea is not general, the correlation between the number of access in the early and later days of a media object is large as reported in [22] for User Generated Contents (UGC).

In [23], an Auto-Regressive Moving average (ARMA) was used to model and predict the daily arrival rate series of media requests on YouTube media server. The ARMA models can capture the correlation and persistence in the series. The ARMA has two parts: the Auto-Regressive and the Moving Average parts. An ARMA model of order (p, q) is given in equ.3.5.

$$Y_t = \sum_{i=1}^{p} \alpha_i Y_{t-i} + \in_t + \sum_{j=1}^{q} \theta_j \in_{t-j} \quad \text{..... (3.5)}$$

Where, Y_t--is number of requests for a media on t[th] day, $\in_t$,...,$\in_1$ are a Gaussian error terms, α_i and $\theta_j$are the parameters of the models

The hourly pattern of number of requests in a given day is generally divided as less busy hours and high peak hours. Usually high peak hours are few and the rest are less busy hours [3] [24]. Three hours of peak loads are commonly used in different synthetic workload generators [24], [25].

3.2 Inter-arrival, segment popularity and session time

Inter-arrival time:

Session inter-arrival time measures the temporal locality of reference or request arrivals on the same media object. Here, all arrivals to all media are assumed to start the playback. Existence of temporal locality indicates that recently accessed objects are likely to be accessed in the near future. A number of studies have shown that high temporal locality exist in media access pattern. In [24], the study on inter-arrival time distribution shows that it can be captured by exponential distribution for every hour on each media object. This shows that a popularity prediction technique which can capture temporal locality is required for efficient caching. It is also reported that for a daylong inter-arrival time distribution, it is difficult to approximate by exponential distribution because of variation in hourly pattern.

Session time and segment popularity:

Media segment popularity is found to follow a Zipf-Mandelbrot distribution in a study on internal popularity [21]. This shows that most requests to a media object accesses the initial portion or media sessions are mostly incomplete. Hence, caching the media prefixes increases data and resource sharing and enhances media streaming performance as reported in a number of literatures (discussed in chapter two). The session time and internal segment popularity are related if user interactivity is limited to playing and stopping the media playback. i.e. a request is either to ***start*** the playback or to ***terminate*** the playback.

3.3 Synthetic Workloads

In evaluating the performance of resource reservation and resource sharing system, test dataset is required. For media streaming system, this dataset can be a real world media access trace or a synthetically generated workload. If integrating a particular streaming or caching technique to an existing system is possible, its performance can also be evaluated on the real system itself. In the absence of such real system or difficulty of experimentation on it, simulation on media access trace or synthetic workload is the only option. In this work, because of the lack of appropriate real world media access trace, a synthetic workload generator is developed and used. Most of the time, it is difficult to obtain a real trace that contains all workload characteristics altogether at the same time. However, in synthetic workloads, most of these characteristics can be easily included. Furthermore, a synthetic workload facilitates the possibility of controlling so many variables involved and their complex relationships. A synthetic media server workload generator was developed, similar to Generator of Internet Streaming Media Objects and Workloads, (GISMO) in [25], workload model in [24] and MediSyn in [28].

This synthetically generated workload was used in evaluating the performance of the caching and prediction schemes. The models used in developing the synthetic workload try to capture as many properties as possible in the user access pattern that are very important in the performance of the system. In other words, the models are intended to represent the necessary media workload characteristics essential for caching. The request inter-arrival and session times are the basis for obtaining the playback ***start*** and ***termination*** times which can actually simulate a large number of clients. Therefore, the final goal of the synthetic workload generator developed was to get the media playback ***start*** and ***termination*** times. The implementation is discussed in chapter six of this report.

4. PATCH STREAMING and CACHING

As described in earlier chapters, Proxy caching of media prefixes reduces start up latencies and is especially important in Batch Streaming system. These schemes are not primarily designed to solve the extra bandwidth requirement of the Patch Streaming technique. Since start up latency is not a problem in Patch Streaming, a caching scheme in such system should aim at minimizing the bandwidth cost related to the delivery of the Patch streams. In this chapter, first the Patch Streaming technique is reviewed. Then an analytical model for the Patching cost is presented, and finally a caching scheme design which tries to minimize this Patching cost is discussed.

4.1 Patch streaming and patch window size

Patching [29] is a streaming mechanism that offers smallest startup latency compared to any other multicasting schemes. The only startup delay is because of network delay. It enables an ongoing regular multicast channel (R-stream) to serve new additional clients dynamically (Fig1.1 in chapter one). A joining request receives the missed portion of the data through a unicast or patch stream (P-stream). The patch window size determines the time for which a subsequent request is allowed to join the R-stream. The performance of patching scheme depends on the size of patching window used. Different techniques of determining patching window size are discussed in [14]. In *Greedy patching* there is no multicast channel initiated as long as there is an existing multicast channel for the same video making the patch window size effectively equal to the media playback duration. This method has two main drawbacks. The first is that the client side should have cache capacity to store the full length of the media object in case it joins at a time close to the end of the ongoing R-stream. The other problem is that it can result in less data sharing. The other scheme is called *Grace patching*. Here, the window size is determined by cache capacity of the requesting clients and it uses patching stream for new client only if it has enough cache capacity, otherwise a new regular multicast stream is scheduled.

Factors that influence window size are video length (playback time), client buffer size, request rate (inter-arrival time) and others. In [12], optimal patch window size is introduced so as to minimize the bandwidth required to deliver the media. Cai et al. in [14] also proposed an optimal patching scheme considering client buffer, server bandwidth, arrival rate and video length. In [12], the arrival process is assumed to be

Poisson distributed with an average arrival rate of λ_i to derive the optimal patch window size and the corresponding minimum bandwidth requirement as shown in equ.4.2 and equ.4.3 below respectively. They found the total bandwidth required for both the R-stream and P-stream of a given media as:

$$BW = \frac{\left(T_i+\lambda_i\omega_i T_i\times\frac{(\omega_i T_i)}{2}\right)}{\left(\omega_i T_i+\frac{1}{\lambda_i}\right)} \quad \text{..... (4.1)}$$

Where, BW – bandwidth required

T_i – media i playback duration

λ_i - request arrival rate for media *i* [Poisson arrival process]

ω_i – patch window size as a ratio of *Ti*

Since their aim was to minimize the server bandwidth requirement, the window size that gives minimum required server bandwidth was obtained by differentiating BW by ω_i and equating it to zero. This gives

$$\omega_i = (\sqrt[2]{2\lambda_i T_i + 1} - 1)/\lambda_i T_i \quad \text{..... (4.2)}$$

By inserting the expression of ω_i in equ.4.1, the minimum required bandwidth is found to be [12]:

$$BW_{optimal} = \sqrt[2]{2\lambda_i T_i + 1} - 1 \text{..... (4.3)}$$

Where, $BW_{optimal}$ – minimum required bandwidth

As it is presented in this section, the optimization of patch window size was to minimize the server bandwidth requirement. In this work, with the aim of further minimizing the bandwidth requirement, a proxy caching scheme in patch streaming system is proposed. The next sections present the analytical model for the total data blocks to be delivered in the P-streams or the patching cost and a caching scheme which tries to minimize this patching cost.

4.2 Cost of Patching

The P-stream uses solely a unicast channel and its increase in number means less resource sharing. In this work, the cost of patching is defined as the total data size that has to be delivered in P-streams (Fig.4.1 below) which is directly related to the extra bandwidth requirement of the technique.

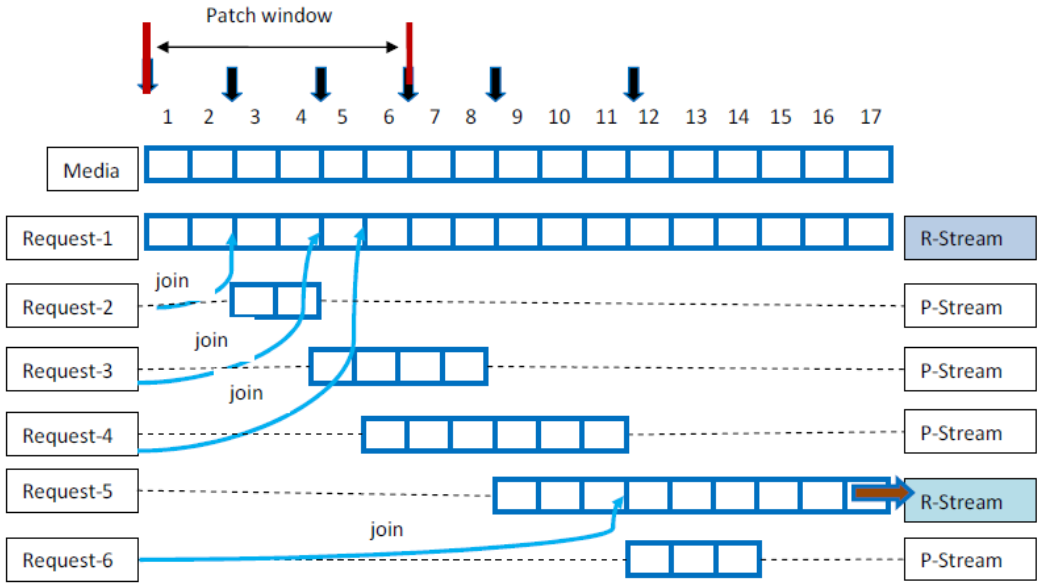


Fig.4.1 Cost of Patching

If a constant bit rate is assumed, the number of data blocks in each P-stream is related to the time gap between the arrival time of the request and the start time of recent R-stream. Number of blocks in P-streams of one R-stream (B_i) in seconds (assuming a streaming time of one block of data equivalent to one second) is:

$$B_i = \sum_{n=1}^{\frac{Pw_i}{s_i}} ns_i = \frac{1}{s_i} Pw_i(Pw_i + s_i)/2 \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (4.4)$$

Where, Pw_i—Patch window size in seconds

s_i—average inter-arrival time for media- i in seconds

n—the n^{th} request that can join the existing R-stream

ns_i—arrival time for n^{th} request

During one playback time of a media object, the number of R-streams to be established is equal to the ratio of playback time to the patch window size. This gives the total data size in P-streams for a duration of one playback time as in equ.4.5

$$B_i = \frac{T_i}{Pw_i} * \sum_{n=1}^{\frac{Pw_i}{s_i}} ns_i = \frac{T_i}{2*s_i}(Pw_i + s_i) \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (4.5)$$

The plot in Fig.4.2 below shows the patch window size and patching cost for constant playback duration of one hour with increasing arrival rate (numerical values are found in Appendix 1). It can be seen that the cost of patching increase with the arrival rate indicating more popular media objects have higher cost than less popular media objects. In other words, as a popularity of a media object increases its patching cost also increases.

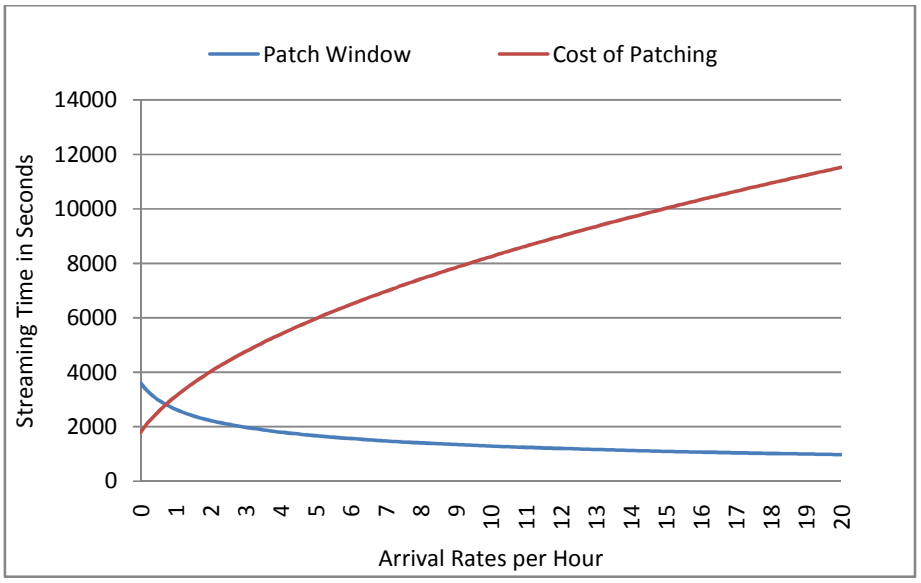


Fig.4.2 Patch Window and Cost of Patching Vs Arrival Rate

As a way of reducing this patching cost or data in the P-stream, *Transition patching* and *Recursive patching* are proposed [17]. Transition patching takes the patching technique one step further by allowing a new client to cache video data not only from R-stream, but also from a nearby in-progress patching channel (P-stream) as well to further reduce resource consumption. Once a new transition stream (T-stream) is initiated, the server tries to patch the T-stream instead of the R-stream. After the client completes to playback the P-stream, it patches the T-stream and the R-stream, respectively. Recursive Patching enables a client to patch a series of T-streams on upper levels to reduce the video data in a P-stream and hence decrease the patching cost. However, the computational expense of recursive patching makes it difficult to implement in real-time streaming media. The size of media data that needs to be delivered in P-streams can also be reduced if the prefixes of media objects are cached at the local/proxy server. The following section presents the proxy caching scheme which also tries to minimize the cost of patching.

4.3 Caching in patch streaming System

As described above the P-stream uses solely a unicast channel and increase in number of P-streams leads to less resource sharing. Proxy caching can reduce the bandwidth requirement increasing the resource sharing. If a few blocks of data are cached at the proxy server, the main media server has to stream only the portion that is not cached as shown in fig.4.3 below and subsequent requests can be served from the cached prefixes. Efficient caching should minimize the data in the P-streams. Since the cost of patching depends on the popularity of the media object, the caching scheme should be able to select prefixes of the most popular media. The selection is based on the cost of patching when few blocks of the media are cached at the proxy. Modifying equ.4.4 to include the prefix size (Cp_i), of media object cached at the proxy server, the size of data blocks, B_i to be delivered in P-streams for each R-stream of media-i assuming a constant bit-rate can be obtained as in equ.4.6.or equ.4.7 below.

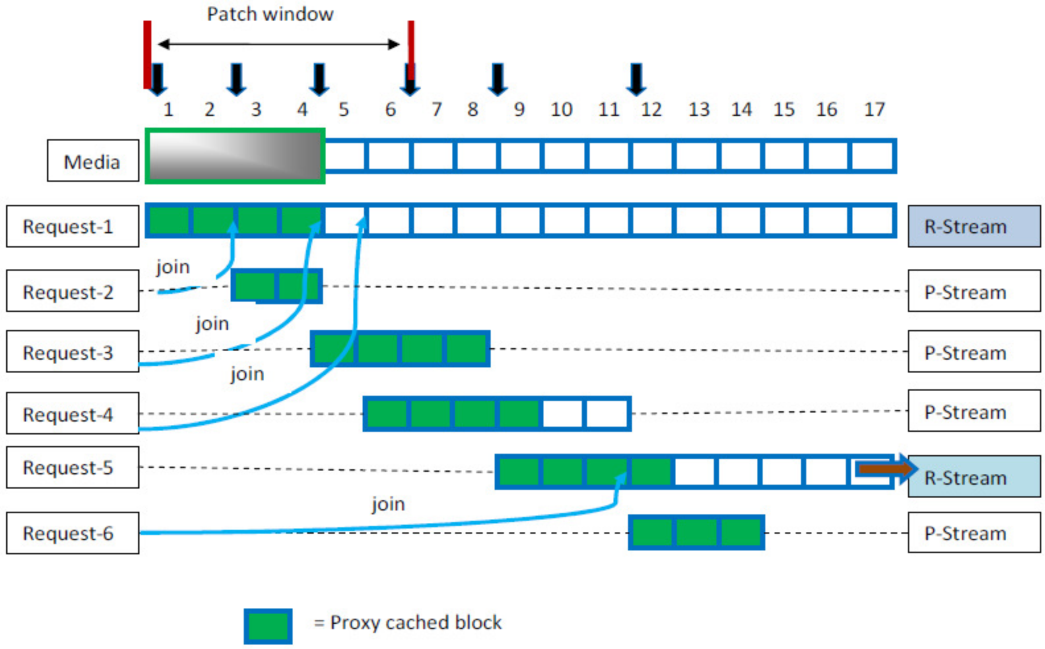


Fig.4.3 Patching and Caching

$$B_i = \sum_{n=\frac{Cp_i}{s_i}+1}^{\frac{Pw_i}{s_i}} (ns_i - Cp_i) \qquad (4.6)$$

$$B_i = (\frac{1}{2s_i})\left[Pw_i(Pw_i + s_i) + \left(Cp_i{}^2\right) - Cp_i(s_i + 2Pw_i)\right] \text{......... (4.7)}$$

B_i, Pw_i, Cp_i, are in terms of the time required to stream the data blocks in each media streams.

For M media objects, total media data B_T in P-streams that has to be delivered in each of the respective patch window sizes of media objects from the main media server in terms of the time it requires is:

$$B_T = \sum_{i=1}^{M} B_i = \sum_{i=1}^{M}(\frac{1}{2s_i})\left[Pw_i(Pw_i + s_i) + \left(Cp_i{}^2\right) - Cp_i(s_i + 2Pw_i)\right] \text{.. (4.8)}$$

Effect of patching window size on the number of R-streams, *R_i*

In the time duration of one R-stream (or playback duration), there would be a number of R-streams established based on the arrival rate as discussed in section 4.1 above. Corresponding to a media file, the number of R-streams to be created is given as

$$R_i = T_i/Pw_i \text{.. (4.9)}$$

The number of P-streams and hence data to be transmitted in P-stream can be approximately calculated as an integral multiple of this number, R_i.

Therefore, the number of blocks in P-stream in seconds for media-*i* can be found as follows:

$$B_i = (\frac{T_i}{2Pw_i s_i})\left[\left(Cp_i{}^2\right) - Cp_i(s_i + 2Pw_i) + Pw_i(Pw_i + s_i)\right] \text{........ (4.10)}$$

For M media objects, the total number of blocks in seconds:

$$B_T = \sum_{i=1}^{M}(\frac{T_i}{2Pw_i s_i})\left[\left(Cp_i{}^2\right) - Cp_i(s_i + 2Pw_i) + Pw_i(Pw_i + s_i)\right] \text{..... (4.11)}$$

Effect of different bit-rate

Media files are usually encoded either at variable bit rate (VBR) or at constant bit rate (CBR). For streaming media, the CBR is very common. Assuming CBR for each media

at different bit rates, the media encoded at a higher bit rate requires a higher bandwidth. Compression techniques are used to minimize the high bit rate. From the bandwidth saving point of view caching the high bit rate media can significantly reduce the bandwidth requirement. If the i^{th}-media bit rate b_i is normalized to a reference of 1Mbps bit rate, then the total data to be delivered in P-streams corresponding to a particular media object is a multiple of this normalized bit rate. For M media, the total number of blocks in seconds becomes:

$$B_T = \sum_{i=1}^{M} \left(\frac{T_i b_i}{2Pw_i s_i * 1Mbps}\right)\left[\left(Cp_i^{\,2}\right) - Cp_i(s_i + 2Pw_i) + Pw_i(Pw_i + s_i)\right] \ldots \quad (4.12)$$

Equ.4.12 shows the total cost of patching when Cp_i blocks of each media object is cached at the proxy server.

4.4 Fractional Knapsack Problem and Greedy Algorithm

The number of blocks to be cached to obtain minimum bandwidth cost is equal to the sum of patch window and half the inter-arrival time. This can be obtained by differentiating equ.4.12 above with respect to Cp_i and setting it to zero.

$$\frac{dB_T}{dCp_i} = 0 \ldots\ldots \quad (4.13)$$

If the inter-arrival time is assumed much less than the patch window size, the number of blocks needed to be cached is equal to Pw_i . This shows that if all data blocks in the P-stream are cached, there is no need for explicit unicast connection and the only remaining data to be streamed is the R-stream which is already in the resource sharing scheme. However, this is hardly achievable for prefix of all media objects because of caching capacity limitation (equ.4.14 below).

$$\sum_{i=1}^{M} Cp_i \le C_{max} \ldots\ldots \quad (4.14)$$

C_{max}-- Maximum caching capacity of the proxy

The problem is can we cache media prefixes up to the maximum capacity and still minimize the size of total blocks of data for all media in the P-stream?

i.e. minimize $B_T = \sum_{i=1}^{M} B_i$

With $\sum_{i=1}^{M} Cp_i \leq C_{max}$ ..... (4.15)

The problem can be seen as a fractional knapsack problem by enabling caching of portion of the data in the P-stream which minimize the cost. The Greedy Algorithm [37] is well suited for solving such problems since the cost associated to each size of cache blocks can be found as in equ.4.12. The approach is to select prefixes of media objects with higher costs and cache them at the proxy server. Greedy algorithm is also simple for implementation and its computational cost is very small. Before proceeding to the greedy algorithm (algorithm 4.1 below), the media objects is sorted in the decreasing order of their costs when there is no caching (Cp_i=0) for all media. Including the effect of differences in bit rates in equ.4.5, this cost is given in equ.4.16 as below.

$$B_i = \frac{T_i * b_i}{2 * s_i * 1Mbps}(Pw_i + s_i) \text{.....} \quad (4.16)$$

Algorithm 4.1: Greedy Algorithm

```
GREEDY_ALGORITHM (S, n, Cmax)
// S is already sorted by the decreasing cost values of each media
//F is an array to hold the number of cache blocks to be assigned to each media or fi
//Cmax- the maximum allocated caching capacity of the proxy
//n - the number of media objects
    R = Cmax; i = 1;
    WHILE ((R > 0) and (i ≤ n)) DO
        IF(R ≥ pwi) THEN fi=pwi;
        ELSE fi=R;
        R = R - fi
        i + +;
    END WHILE
    RETURN F;
```

The algorithm selects media prefixes with higher patching costs that are also related to the request rate. The prefix is equal to the patch window of respective media or a portion

of it. The algorithm also makes sure that all media prefixes would be cached provided that the proxy caching capacity is enough to contain.

4.5 Placement, Replacement and Request service process

The proxy server regularly updates the candidate media prefixes for caching. Prefixes of popular media are selected for caching and those media which are losing their popularity are removed from cache and replaced by the prefixes of popular media. When request arrives, if prefix is already cached the streaming starts from the proxy. If the prefix of the requested media is not cached the streaming starts from the main media server. If the media is popular and is in the selected list of caching, the proxy caches the prefix of the media while the data is being forwarded to the client by joining the R-stream or P-stream channels.

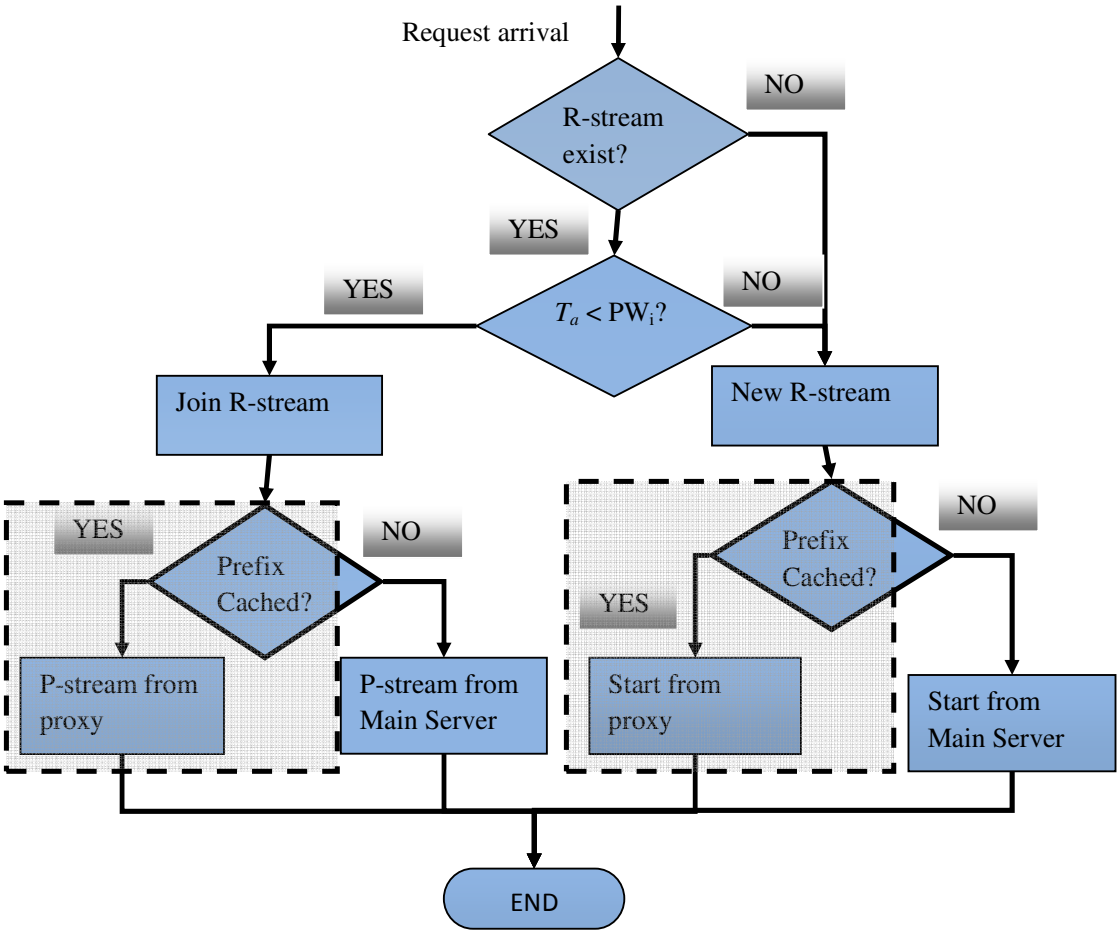


Fig.4.4 Flowchart of Request Service

The process of admitting and servicing a client's request by the main media server is shown in fig.4.4 above. For every request the server checks whether there is an ongoing R-stream or not. If there is an ongoing R-stream, it compares the patch widow size with the request arrival time gap (T_a) and the client joins the R-stream if T_a is less than the patch window size. A new R-stream is established if T_a is greater than patch window size and this is also the case if there is no ongoing R-stream. The data stream in either of the P-stream and R-stream begin from main media server if there are no cached prefixes at the proxy and begin from proxy if there are cached blocks.

4.6 Conclusion

In the first section of this chapter, the patch streaming and the optimal patch window size are reviewed. The optimal patch window size minimizes the server bandwidth requirement. Since, P-stream uses a unicast channel its increase in number decreases resource sharing. The next section presented an analytical model for the cost of patching that is defined as the total media data to be delivered in P-streams. The cost of patching is found to increase with the request arrival rate and also popular media files have larger costs than less popular ones. Caching a few blocks of media prefix reduces the bandwidth cost and the optimal prefix size is found to be equal to the patch window size of respective media. Since, the proxy caching capacity is limited the Greedy Algorithm is used for optimal selection of media prefixes. In the last section, the placement, replacement and the request service processes are presented.

5. PREDICTION OF THE ARRIVAL RATES

As described in previous chapter, the optimal patch window size, the cost of patching, and the selection of media prefixes depends heavily on the popularity or the arrival rate of requests to a media object. Therefore, the implementation of caching in patch streaming system requires prediction of arrival rates from the real-time traffic and updating the patch window size and cached media prefixes regularly according to these predicted values. This chapter presents the prediction scheme used in this thesis work. First the time scale for prediction is discussed, required characteristics of the predictor and then the selected prediction technique is presented.

5.1 Prediction Time Scale

What time scale has to be used for prediction: daily arrival rate series or hourly arrival rate series? The choice can be based on different factors: the distribution of the arrival process and temporal locality of references. As explained in section 4.1 of this report, the optimal patch window size is obtained by assuming a Poisson request arrival process. However, the study in [24] showed that only the hourly request arrival process on a single media object can be approximated by Poisson arrival process or an exponentially distributed inter-arrival time. This implies that the optimal patch window size which assumes Poisson arrival process needs to be updated every hour. In addition to this, the temporal locality of reference shows that a portion of media object may not need to remain cached for long time. Caching for a full day may be only for a few media. Some of the media prefixes need to be cached for few hours and removed from cache for the rest of the hours in the day. This also enforces predicting request arrival rates every hour and updating the caching accordingly.

5.2 Criteria for the Predictor

Short-term memory: One decisive factor for the predictor is that it should be able to capture the temporal locality in the request pattern. Though there is a general correlation in the long-term popularity, media access changes from time to time as described in section 3.1. A recent few hours of arrival rate history has a strong effect on the next few hours arrival rate than the long-term history. This requires a predictor with a short-term or fading memory.

Fast computation: As it is discussed in the previous section, the prediction needs to be performed every hour. This places limit on the prediction time because of the real-time media streaming requirement. The media server is busy in streaming the media data and hence prediction should be accomplished in a reasonably short time.

Ease of implementation: The other factor for selecting a predictor is the simplicity of implementation.

Based on the above criteria, a Reservoir Computing technique is selected and implemented in this work. The next sections present this technique.

5.3 Reservoir Computing

Reservoir Computing (RC), [36] is a recurrent neural network (RNN) approach used for time series prediction, pattern generation, classification, and nonlinear control. It is derived from RNN but avoids the iterative parameter optimization and uses a very simple training approach. RNNs are able to solve temporal problems such as the modeling of non-linear processes. The dynamical complexity of the interaction between recurrent nodes is for a large part the reason why those networks suffer from slow training speeds and convergence problems [34]. To avoid the training problem, the Liquid State Machine (LSM) and the Echo State Network (ESN) have been introduced independently. The ESN which is used in this work will be described in more detail in this chapter.

In reservoir computing, the input signal is fed into a fixed dynamical system called ***reservoir***. This dynamical reservoir is a recurrent network of nodes with weighted connections generally left unchanged or remain untrained. Instead, a separate read-out connection weight is trained on the response of the reservoir to the input signals or using a supervised learning approach. RC has been used for media popularity prediction in [30].

The recurrent delayed connections inside the reservoir have a form of short-term memory since its current state depends only on the very recent previous state. This is one of its attractive advantages [30], its fading memory, distinguishes it from analytical models that usually assign the same importance to every historical observed access no matter how far they occurred in the past. The attention media object received recently will have a larger impact on the future than what happened in the beginning of its

appearance. For instance, an online video that did not get many views after it was introduced on the server but receives a lot of attention recently, probably will also continue the high attention for a while.

The other main advantage of RC is that training is only performed for the read-out connection weight. The simplicity of the training is mainly due to the randomly initialized and untrained reservoir. A random network is constructed with certain topology and the read-out weight is trained on the data set using a supervised learning.

5.4 Echo State Network

The echo state network (ESN) [33] and liquid state machine (LSM) [35] models are the common ones in RC. Here, the ESN model is presented and also implemented for arrival rate prediction. The property of ESN is that the very recent inputs to the network determine the current state of the reservoir. The network is thus state forgetting or has a fading memory. The diagram of the ESN is shown in fig.5.1 and the current state of the reservoir X_n is found as in equ.5.1 below. ESNs use simple sigmoid neurons, where the nonlinear function $f(,)$ is a sigmoid, usually the *tanh(·)* function.

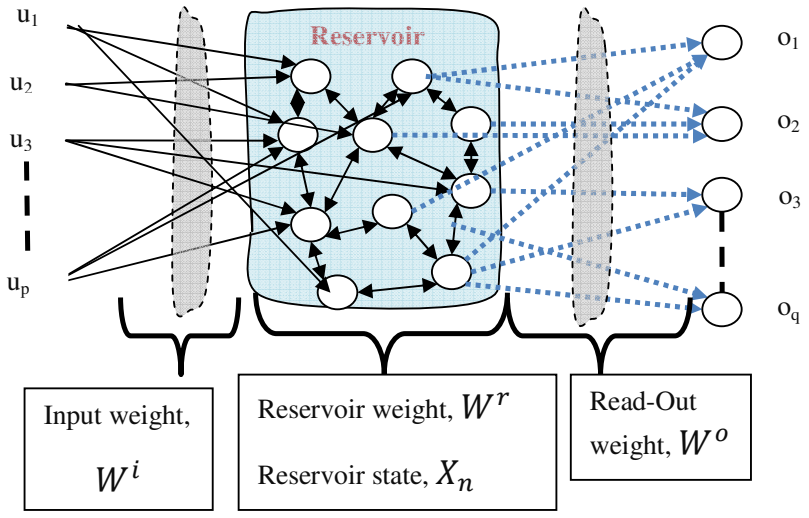


Fig.5.1 Echo State Network

$$X_n = f\left(W^r X_{n-1} + W^i U_n\right) \quad \text{..... (5.1)}$$

Where, the function $f(,)$ is non-linear activation function typically *tanh*. W^r-- Reservoir weight matrix, W^i-- input connection matrix, U_n--the n[th] input vector of size *p* and *n*—current time

The output of the network O_n depends only on the current reservoir state and the trained read-out weight, W^o as in equ.5.2.

$$O_n = W^o X_n \quad (5.2)$$

As is described previously, the input and reservoir weights are created randomly and fixed. To guarantee the stability of the reservoir, the values of these connection weights have to be scaled before training the network. In scaling W^r the largest eigen-value $|\lambda|_{max}$ and the spectral radius *r* of W^r must satisfy that $|\lambda|_{max} < 1$ and $0 < r < 1$. Therefore, after initializing the random reservoir, it is scaled as:

$$W^r_{new} = \frac{W^r}{|\lambda|_{max}} \quad (5.3)$$

5.4.1 Training

The supervised training in ESN tries to minimize the error between the estimated and target outputs in the least-square sense. Least-square and ridge-regression (RR) methods are the two approaches used to train the read-out weight matrix [34]. The least-square is mostly used for on-line training while the RR is for the off-line batch training. In this work, the RR training is used and hence this sub-section discusses this approach. The RR is introduced in ESN training to avoid the problem of near collinearities between the reservoir states X which might result in near singularity making the read-out weight sensitive to random errors. Several methods have been developed in order to reduce this sensitivity, and one of them is RR, which means that a number δ is added to the elements on the diagonal of the matrix to be inverted, yielding a modified estimator of the form in equ.5.4

Hence, solution exists even when $X^T X$ is singular, i.e., has zero eigen values and when $X^T X$ is ill-conditioned (nearly singular), the RR solution is still more robust.

$$W^o = (X^T X + \delta \mathrm{I})^{-1} X^T Y \quad \text{..... (5.4)}$$

Where, I—identity matrix, δ—regression parameter, X—concatenation of all reservoir state vectors, Y—concatenation of desired output vectors

Steps in training

- Create random weight matrices
  - Input to reservoir W^i
  - Reservoir to reservoir W^r
- Scale weight matrices for stability (equ.5.3)
- Feed input samples (previous hourly media request arrival rates) to the reservoir and drop initial warm up states to avoid the problem of randomization
- Collect the rest of reservoir states X_n into matrix X, and desired outputs Y_n into Y
- Train read-out weight using RR as in equ.5.4

5.4.2 Testing

During testing, the network is excited with previous inputs and the reservoir state X_n is calculated from equ.5.1. The trained read-out connections remain unchanged and are used to obtain the predicted output as in equ.5.2. The performance of the predictor is evaluated as root relative squared error (RRSE) between the predicted and the desired (actual) arrival output as shown in equ.5.5 below.

$$\varepsilon = \sqrt{\left(\frac{1}{N}\sum_{n=1}^{N}\left(\frac{O_n - Y_n}{Y_n}\right)^2\right)} \quad \text{..... (5.5)}$$

Where, O_n—predicted output, Y_n—desired output, N—sample size used for testing.

6. SYSTEM DESIGN AND IMPLEMENTATION

The design and implementation of On-Demand media streaming system involves a number of issues. Two of these issues are caching technique and the prediction of arrival rates as discussed in the previous chapters. Other issues include system architecture, streaming protocols, and admission control (AC) strategy. To obtain a working system, all these components should be put together in an efficient way and with an effective communication between them. This chapter first presents the general architecture, the role of each system components and an overview of the streaming protocols. Then the AC strategy used in this work is presented. Lastly, the implementation of the different modules of the system and synthetic workload generation are discussed.

6.1 System Architecture

The design and implementation of the system assumes the general client-server architecture as shown in fig.6.1. Large repository of media files are at the central main media server. A number of local/proxy servers are at the edge of the backbone network each serving clients' requests as an intermediary between the main media server and clients. The selected portions of multimedia objects are cached at the proxies and buffering is used at clients to support the patch streaming. This section presents the general design and architectural functions of the different nodes in the system.

- ***Client nodes***

The architectural design of the media streaming client is the same as it is described in [29]. A client performs two main tasks in a patch streaming systems: playback or buffering with playback.

I. A client simply plays back the byte streams it receives from the server, if it is the first to access the media or is only receiving the R-stream.
II. A client plays back the media stream it receives from the P-stream while it buffers the data being received from R-stream if it is receiving both the R-stream and P-stream at the same time.

To do these tasks, the client needs three threads of control: a thread to receive from the P-stream, a thread to receive from the R-stream and a thread for the player. Since caching is basically used as a means for resource sharing and performance improvement between the main server and the proxy, the client design is not affected by the way the

caching is implemented. Therefore, the discussion of client design is very limited in this work.

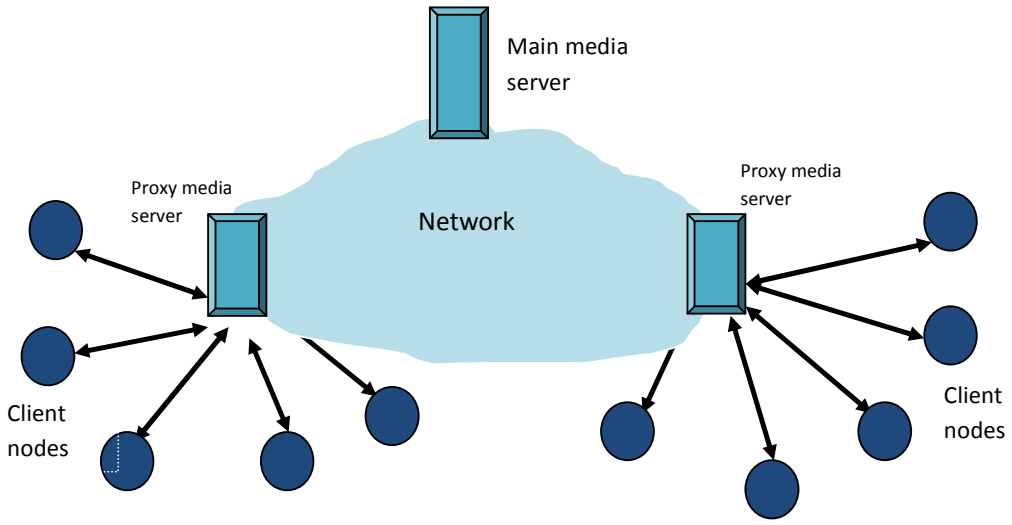


Fig.6.1 Client-Server Architecture

- ***Main media server***

Here, the design of the main media server is different from the server described in [29] because of the cached media portions at the proxy servers. The main media server has to complete the following tasks:

- Serve a request in different data streams (R-stream, P-stream and T-stream if it supports)
- Collecting and monitoring request arrival rates from real-time media access traffic
- Predicting hourly arrival rates
- Calculates the patch size and cost
- Use cache information from proxy and server resources to accept or reject a service request.
- Communicates the patch size and patching cost information with the proxy.

The different modules and their implementations are discussed in detail later in this chapter.

- ***Proxy server***

Proxy server acts as an intermediary between the main server and client like a web proxy server. Receives client requests and forwards it to the main server, forwards responses from the main server to the client, caches selected media prefixes, and also streams media data from its cache as needed. The number of blocks of media prefixes to be cached is obtained based on the cost of patching (section 4.2 and 4.3) and the Greedy algorithm presented in section 4.4.

In general, the proxy is required to have the ability to serve a request in R-stream or P-stream, making optimal selection of media prefixes to be cached, caching and managing the cache, accepting and responding to the client's requests, and communicating with the server.

The different modules of the proxy server which performs these tasks are presented in the following sections.

6.2 Components and Modules of the Architecture

The generic client-server interaction of proxy caching for streaming media over the internet as presented in [13] is used and modified to support the patch streaming technique together with caching, AC and media popularity predictions. See fig.6.2 below. The functions of each system component are grouped into different modules. The modules in the client, main media server and proxy server have to communicate effectively to achieve effective streaming system. The functions of these modules are presented in this section.

Server manager: this module is the main manager of the media server which is in charge of all the scheduling, streaming control, and managing the communication and different threads of control. The server manager communicates with the proxy manager using stream control protocols, the Real-Time Streaming Protocol (RTSP) and Real-Time Transport Control Protocol (RTCP), monitors the real time traffic and run the AC module. This module also calculates the patch window size and patching cost regularly based on the arrival rate prediction obtained from the prediction module. Using the calculated patch window size for each media object and available bandwidth resource, it determines whether a requesting client is to be served in R-stream or in both R-stream and P-stream as shown in fig.4.4.

Admission control: the server manager runs this module to decide on whether to accept or reject a request. The AC strategy used in this work is discussed later in this report.

Prediction module: the function of this module is predicting the hourly arrival rates for each media objects based on past history of hourly arrival series. The RC prediction technique presented in chapter five of this report is used. It is implemented on Microsoft .Net platform as a dynamic link library and integrated into the server as discussed in subsection 6.5.2 below. The server manager calls this module after every fixed period of time.

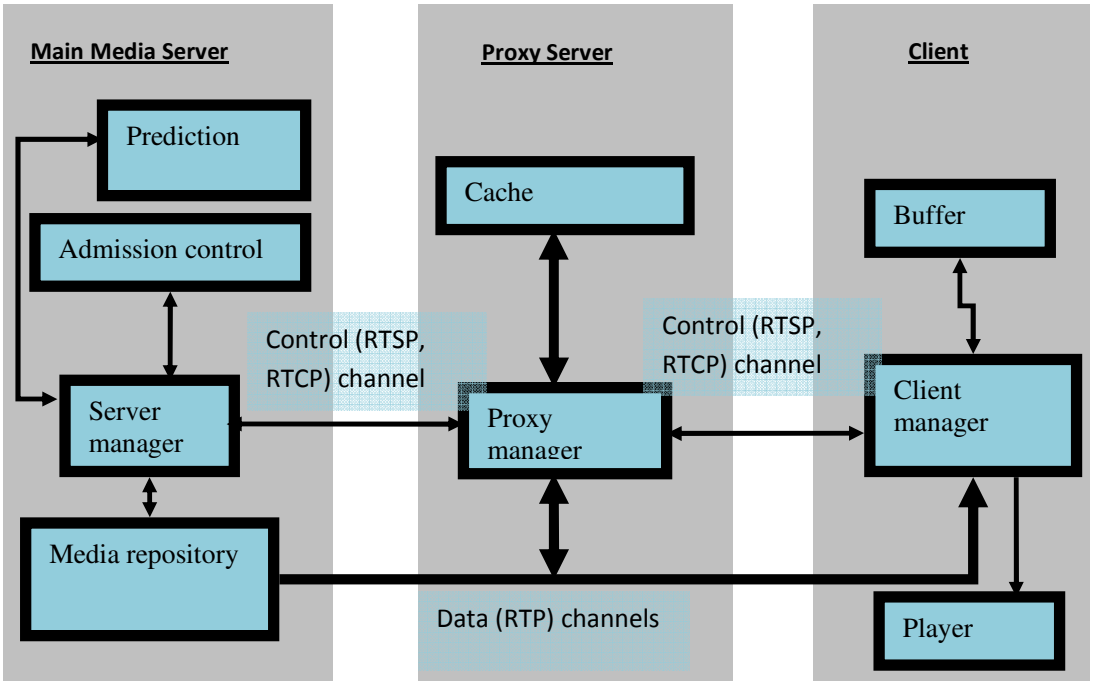


Fig.6.2 Client-Proxy-Server Interaction for Proxy Caching Patch Streaming System

Media repository: the multimedia database on permanent memory or disk.

Proxy manager: controls all the communication between main server and client. Caches media prefixes and also streams the cached media prefixes. Selects media prefixes to be cached and caches the selected ones. For communication, RTSP and RTCP are used for stream controls and Real-time Transport Protocol (RTP) for data transfer.

Cache: this module manages the caching of media sections. The proxy manager calls this module after every fixed time interval to make replacements.

Client manager: the player and buffer modules are controlled by the client manager. If the client receives from two streams (R-stream and P-stream), the data stream from R-stream is buffered and the data stream from P-stream is played back.

6.3 Overview of Streaming Protocols

- **RTP and RTCP**

Real-time transport protocol (RTP, RFC 3550 [31]) is real-time transport protocol which provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. The services provided by RTP include payload type identification, sequence numbering, time stamping and delivery monitoring. It can be used for one-way transport such as video-on-demand as well as interactive services such as Internet telephony. RTP relies on the UDP data transport protocol because packet retransmission in TCP is not crucial for real-time data.

The RTP header format contains different fields. This includes version (V) indicator bits, padding bit (P), extension (X), Contributing Source identifier (CSRC) count (CC), and marker (M) bit at the beginning and other fields which are presented below. The format is shown in Fig.6.3.

Payload Type: 7 bits, providing 128 possible different types of encoding; e.g. PCM, MPEG2 video, etc.

Sequence Number: 16 bits; random number incremented by one for each RTP data packet sent; used to detect packet loss

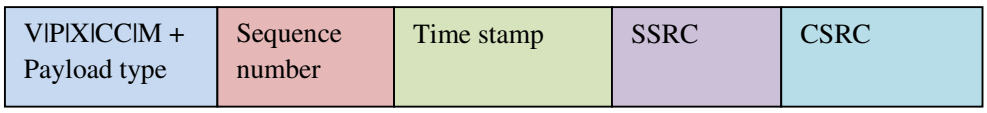


Fig.6.3 RTP Header Format

Timestamp: 32 bytes; gives the sampling instant of the first audio/video byte in the packet; used to remove jitter introduced by the network

Synchronization Source identifier (SSRC): 32 bits; an identifier for the source of a stream; assigned randomly by the source

Miscellaneous fields: Contributing Source identifier (CSRC)

RTP is designed to work in conjunction with the auxiliary control protocol RTCP to get feedback on quality of data transmission and information about participants in the on-going session. The RTCP specifies report packets exchanged between sources and destinations of multimedia information. Reports contain statistics such as the number of packets sent, number of packets lost, and inter-arrival jitter. This reports are used to modify sender transmission rates and for diagnostics purposes.

- **RTSP**

The Real Time Streaming Protocol (RTSP, RFC2326 [32]) is an application-level protocol for control over the delivery of data with real-time properties. The protocol is used for establishing and controlling media sessions between end points. It is designed to support VCR-like operations, such as play and pause, that clients issue for real-time control of the playback. RTSP is supposed to be used in conjunction with various other protocols. It can be used with either TCP or UDP as an underlying transport protocol. The data transfer protocol that is mostly used with RTSP, is RTP which is used for media stream delivery. Though, similar in some ways to HTTP, the RTSP defines control sequences useful in controlling multimedia playback which cannot be accomplished using a stateless server, the case of HTTP. Multimedia presentations are identified by URLs, using a protocol scheme of RTSP. The hostname is the server containing the presentation; while the port (the default is 554) indicates which port the RTSP control requests should be sent to. Some of the requests are made similar with those requests supported by HTTP and others not. Some of the basic RTSP requests are: *OPTIONS, DESCRIBE, SETUP, PLAY, PAUSE, RECORD,* and *TEARDOWN*.

6.4 Admission Control

In a continuous media delivery system, the decision for accepting or rejecting of a new request or client is based on the available resources to serve the request. The overall performance of the system and the quality of service (QoS) should also be maintained to guarantee the sustained service of the ongoing streams. Qazzez in [11] classified the AC into four general categories as: deterministic server, statistical server, predictive server and best-effort server. In deterministic server, the system resources like CPU time, server

bandwidth and other resources are allocated in a way which guarantees all the accepted requests are served with full QoS requirement while statistical server guarantees the QoS requirement with certain probability. In predictive server, the server predicts the resource utilization from past history and uses it in resource allocation while the best-effort server makes no guarantee for the sustainability of the service.

In this work, the bandwidth resource is allocated using the deterministic approach. Patch streaming technique as described in section 4.1 requires additional bandwidth to serve the missing portions of the media. This is also called *residual* bandwidth requirement. When a request arrives and joins an existing R-stream there should be enough bandwidth to serve the p-stream. However, with proxy caching the need to serve the p-stream directly from the main server is not always the case. So, the AC should check whether the media prefix is cached or not and how much. For the current request, if the whole data blocks in the p-stream are cached at the proxy and the client can join the R-stream, the request can be accepted without additional resource. Otherwise, there should be bandwidth available to sustain the requested media's playback speed. The following algorithm (algorithm 6.1) shows the process of accepting or rejecting a request using deterministic strategy.

Algorithm 6.1: Deterministic Admission Control

FOR EACH *request i for media m*

IF *cached size (m) = patch window (m) and i can join existing R-stream*

THEN *Accept request i*

Increment number of clients

ELSE IF *bit rate (m) <= available Bandwidth* ***THEN***

Accept request i

Increment number of clients

Decrement available bandwidth by bit rate (m)

ELSE

Reject request i

END

When stream ends or a termination request arrives, the server restores all the resources allocated.

6.5 Implementation

The implementation of the system requires a number of modules to work together effectively to obtain a working system. Before having the system run, the media access workload is required to train the predictor and also as a test dataset to evaluate the performance of overall system. Therefore, first the synthetic workload generator is developed. Then the predictor subsystem is implemented. Finally, the process of communication and session setup, the main server and proxy server components are implemented. The following subsections present these procedures step-by-step.

6.5.1 Generation of Synthetic Workload

This subsection presents synthetic workload generation steps used in this work to generate test data set for various performance measurements of the proxy cached patch streaming MOD system. The main differences between the generator developed here and the generators developed in [24], [25], and [28] are the methods used in the generation process and the scope.

The method used in this work is to generate the long-term popularity distribution and then proceed to the generation of the daily arrival series, hourly arrival series, and finally to the generation of inter-arrival time and session time as shown in Fig.6.4. The aim is to include both the generation of media popularity distribution and individual access traces together. The scope is limited to support a few user interactivity functions (only PLAY and STOP or TERMINATE are assumed). The generator is implemented in Microsoft .Net platform and the user can generate different workloads by varying the parameters of the workload.

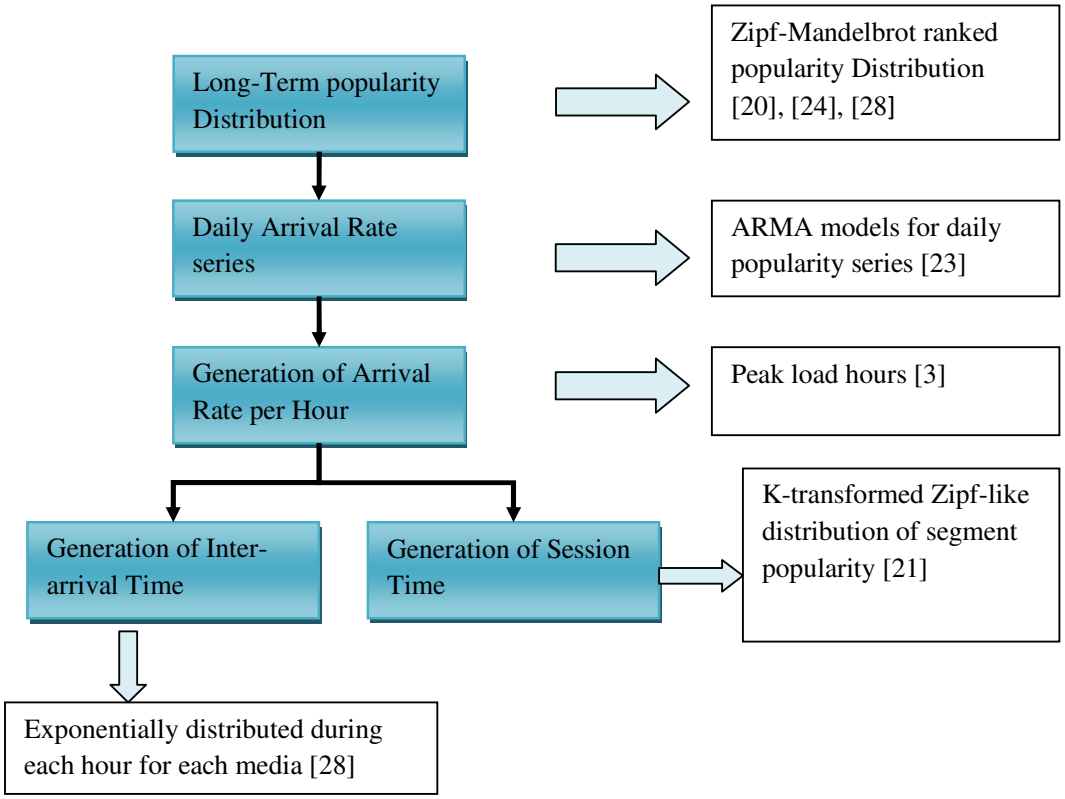


Fig.6.4 Diagram of Synthetic Workload Generation

Generation of Long-term popularity:

As described in chapter three, the long-term popularity distribution is assumed to follow K-transformed Zipf-like distribution (equ.3.4). Skew factors of 0.7 [25], 0.729[2] are reported. Skew factor was also shown to range between 1.4 and 1.6 for Zipf-like distribution [20]. Here, a skew factor of 0.729 is used as it is the most reported and used in literatures. Other model parameters and the skew factor can be adjusted as needed using the developed GUI for the synthetic workload generator which is shown in App.2.1 of Appendix 2. The following figure shows the sample generated long-term popularity distribution.

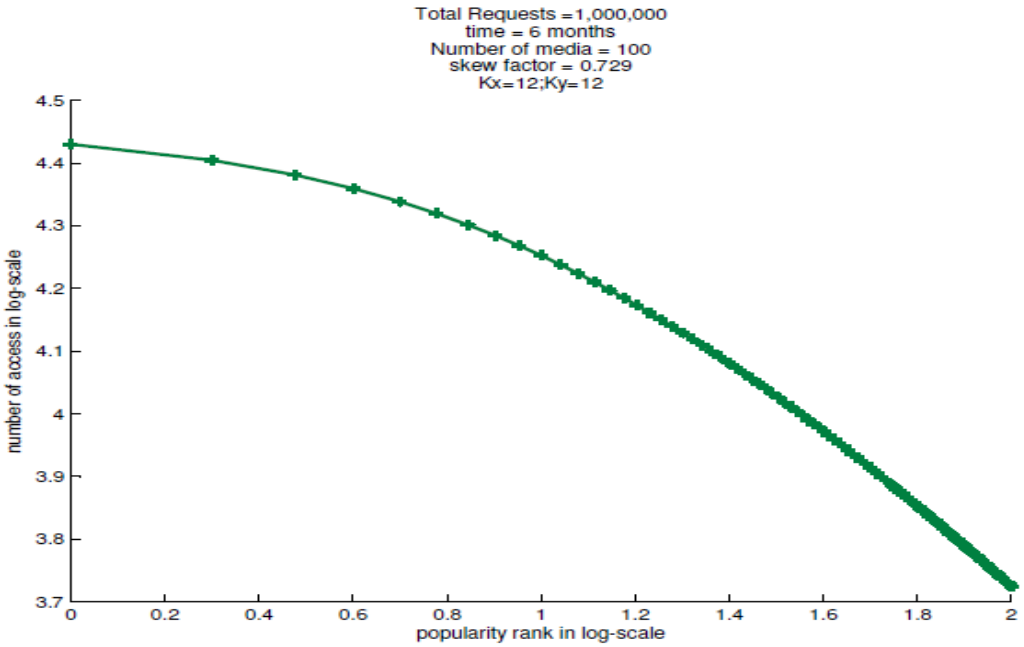


Fig.6.5 Long-Term Ranked Popularity Distribution

Daily arrival series:

The daily arrival rate series for each media object is generated from the long-term total access assuming the ARMA model. The order of the ARMA is chosen to be (7, 7) as is reported in [23]. Regression and moving average parameters are selected to give a total sum of less than one so as to insure the stability and wide-sense stationary property of the series. The following figure shows the sample for two media objects.

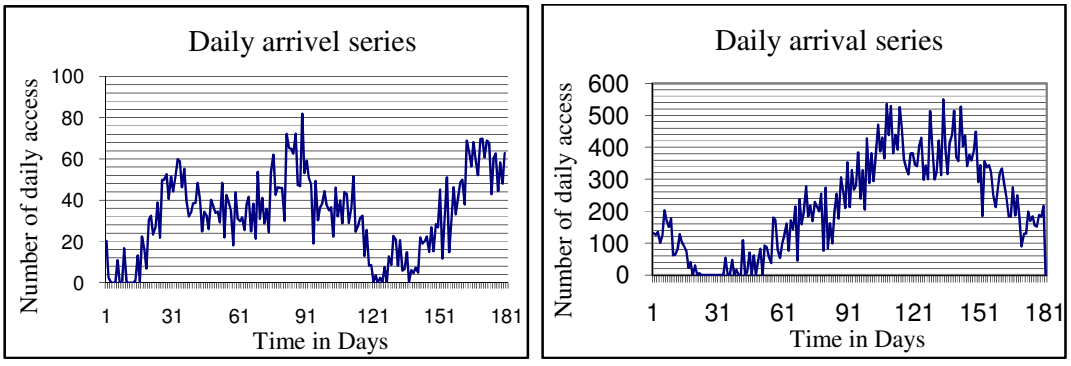


Fig.6.6 Daily Series

Hourly arrival rate:

After obtaining the daily arrival series, the hourly arrival series can be generated assuming a few peak hours in the day. The number of peak load hours can be adjusted to a desired value using the GUI for the workload generator (App.2.1 of appendix 2). Here, three peak hours are used as in [24] and [25]. The peak hours arrival rates are a multiple of some factor of the less busy hour's arrival rates. To smooth out the random fluctuations, Cumulative Moving Average (CMA) is used. Fig.6.7 shows sample hourly arrival rate series for one media object and more samples are included in App.2.2 of appendix 2.

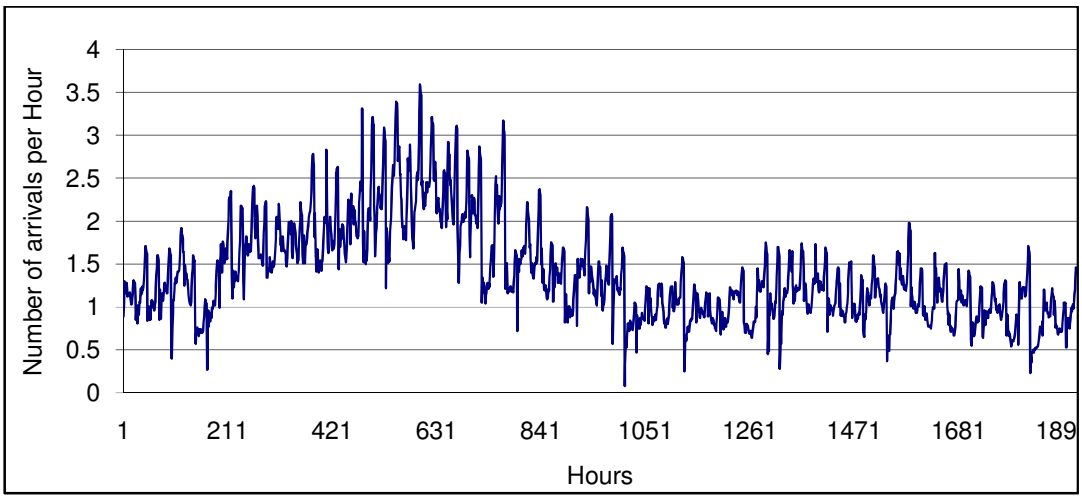


Fig.6.7 Hourly Arrival Series

Generation of inter-arrival and session times:

As it is discussed in section 3.2, the hourly number of arrival follows a Poisson process and hence, the inter-arrival time is exponentially distributed. Therefore, the inter-arrival time for each media object is generated by inverse transformation technique using the hourly mean number of arrival obtained previously. The request arrival time is obtained by concatenating these inter-arrival times of each media objects.

Since the segment popularity follows a Zipf-Mandelbrot distribution [21], and assuming the media request is either for playback *start* or *termination*, the session time is generated by inverse transformation of the Zipf-Mandelbrot distribution. The difference between the start time and termination time is the session time. As described in chapter three,

these two request types: playback *start* and *termination* times represent requesting clients and constitute the real-time traffic. Table.6.1 shows sample generated playback ***start*** and ***termination*** times.

Table 6.1 Sample Playback S*tart* and *Termination* Times

day: 101 [partial] media Id	start time	termination time
3156029	0:00:23	0:29:22
3156084	0:00:31	0:16:37
3156072	0:00:37	0:03:18
3156072	0:00:41	0:01:14
3156067	0:01:13	0:07:00
3156047	0:01:17	0:03:36
3156028	0:01:24	0:10:14
3156093	0:01:46	0:01:51
3156072	0:01:46	0:05:32
3156119	0:01:59	0:05:09
3156102	0:02:32	0:43:52
3156098	0:02:30	0:03:27
3156080	0:02:44	0:03:20
3156117	0:02:56	0:28:45
3156098	0:03:06	0:06:02
3156075	0:03:07	0:21:15
3156116	0:03:05	0:07:16
3156026	0:03:10	0:03:22
3156113	0:03:28	0:16:32
3156098	0:03:29	0:15:21
3156054	0:03:47	0:04:36
3156113	0:04:06	0:33:53

6.5.2 Implementation of the predictor module

The RC prediction technique used in this work has two stages. The first stage is to train the network. The next stage is using the trained network for the prediction of the next hour's arrival rates in the overall system. The hourly arrival rate series obtained from the synthetic workload generator is used for training, the process which is to be presented in

detail in this subsection. The use of the predictor is based on the real-time requests that are collected by the server manager.

RC Network configuration: Before starting the training, the number of input nodes (*p*), output nodes (*q*) and reservoir nodes (*r*) should be configured and the input and reservoir weight connection matrix has to be initialized. The numbers of input nodes indicate the number of previous hour's arrival rates, to be used to predict the next hour's arrival rates, which is equal to the number of output nodes. The developed user interfaces are used to enter these parameters (see App.3.1 and App.3.2 in appendix 3). The input and reservoir weight matrices are initialized with random values between 0 and 1. Then the reservoir weights are scaled to ensure stability as shown in equ.5.3.

Training: The generated hourly arrival rate series is divided into two as training and testing datasets. From the training dataset the initial samples are used as initial warm up samples. Usually this warp up sample is chosen to be between 5% - 10% of the training sample size. After the warm up run, the reservoir states are collected from each iterations to X_n and the corresponding actual arrival rates into Y_n. At the end of the iterations, the read-out weight matrix is obtained from the RR as:

$$W^o = \left(\sum_n X_n X_n^T + \delta I\right)^{-1} \sum_n Y_{n+1} X_n^T \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (6.1)$$

The other version of this equation is given in equ.5.1 in its matrix form.

Testing is required to make best selection of the number of inputs, outputs, and reservoir nodes. The network was evaluated a number of times with different combinations of input, output and reservoir nodes to find the best topology. The parameter δ is also optimized manually. This selected topology is used in the overall system.

Usage of the predictor module: the testing dataset is the request arrival for playback *start*. It starts from the next day after the training dataset. For the generated 6-months (180 days) workload, if 170 days are used for training the 171th day workload can be used for the evaluation of the overall system. After training the selected topology, the main media server is set to run and receive client's request through the proxy server. The main media server collects the number of *start* request arrivals to a particular media object each hour from the real-time traffic. This collected arrival rates are the actual

arrivals used as an input to the predictor. The output of the predictor is the next hour's arrival rates that the media server uses to calculate the patch window size and patching costs.

6.5.3 Session setup and Communication process

Communication between the server, proxy and client involve a number of message exchanges as shown in Fig.6.8. In this work these messages are designed to be UDP packets similar in functionality to the RTSP discussed above.

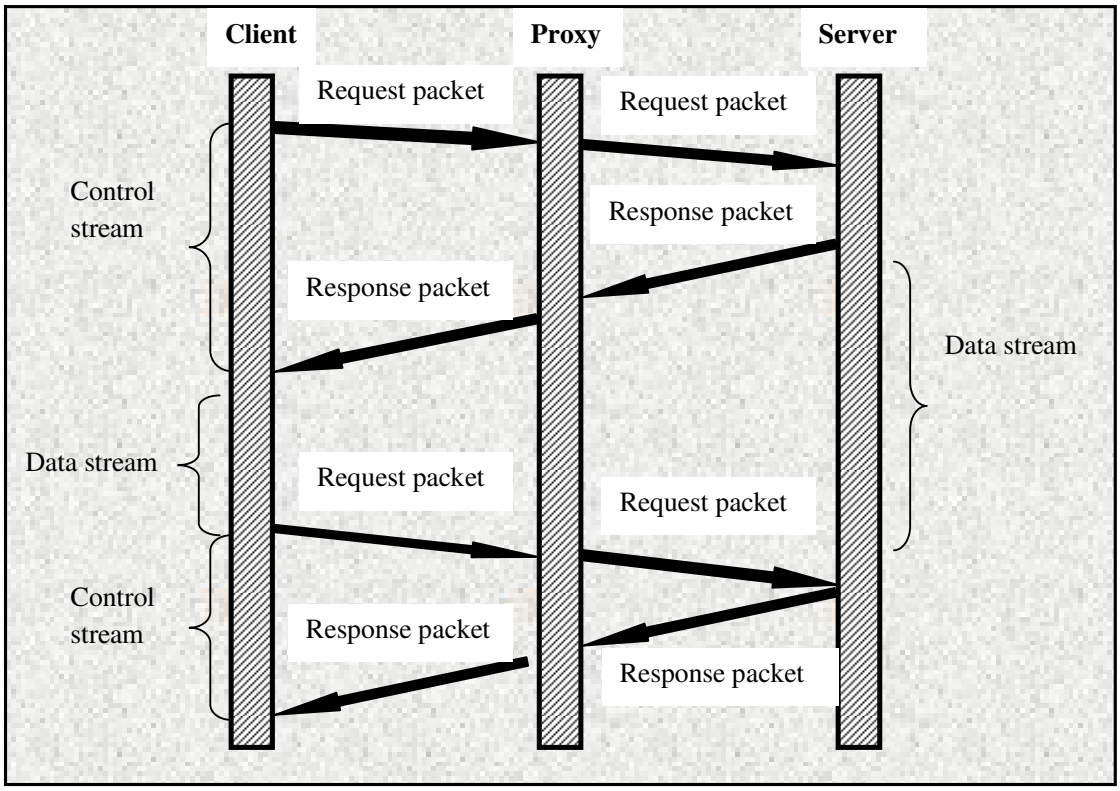


Fig.6.8 Communication Process

- **Request packets**

  Request packets contain the setup, play or terminate functions of the RTSP. Though the implementation of these functions is independent of the RTSP request format, similar functionalities are maintained. A request packet of a client contains the endpoint for client (CLIENT_EP), the media identifier (MEDIA_ID), or a file URL, and the request type (REQ_TYPE). The request type can be either of "SETUP", "PLAY" or "TERMINATE".

// procedure for the Client request packet

ClientRequestPacket (CLIENT_EP, MEDIA_ID, REQ_TYPE)

//……

At the proxy, the request identifier ("REQ_ID"), "TIME" and "CACHE" information are added to these packets, and forwarded to the main media server.

// procedure for the request packet at proxy server

ProxyRequestPacket (CLIENT_EP, MEDIA_ID, REQ_TYPE, REQ_ID, TIME, CACHE)

//……

The "TIME" field indicates the time of setup or playback initiation. This is only required in the simulation mode (to be discussed in chapter 7) since in the real mode, the system time can be used for same purpose. The "CACHE" field contains the size of data blocks, prefix size in this case, that is cached at the proxy server.

- **Response packets**

The response from the server can be either accept or reject response. An accept response packet contains eight fields: STREAM_TYPE, REQ_ID, P_STREAM_IP, PORT_1, PORT_2, patch window size (PW), patch cost (PC) and response type (R_TYPE).

// procedure for the server response packet

ServerResponsePacket (STREAM_TYPE, REQ_ID, P_STREAM_IP, PORT_1, PORT_2, PW, PC, R_TYPE)

//……

"STREAM_TYPE" field informs the proxy server or client whether the request is to be served from R-stream or also from the P-stream. If the stream type field indicates a regular stream, the client joins the R-stream on "PORT_1", in which case P_STREAM_IP and PORT_2 are null, and if it indicates a patch stream, the client joins the R-stream on "PORT_1" and receives the P-stream on IP address "P_STREAM_IP" and "PORT_2".

It is assumed that the multicast IP address for the R-stream is known by both the main media server and proxy server. "PW" and "PC" contain the patch window

size and patching cost information respectively. The "R_TYPE" field indicates whether the request is accepted or rejected. For rejected requests, other fields can be null.

The proxy server extracts patch window size, patch cost and stream type information from the server response and constructs a proxy response packet containing STREAM_TYPE, MEDIA_ID, P_STREAM_IP, PORT_1, PORT_2, PW and R_TYPE and send it to the client.

```
// procedure for the proxy response packet
ProxyResponsePacket (MEDIA_ID, STREAM_TYPE, P_STREAM_IP,
PORT_1, PORT_2, PW, R_TYPE)
//……
```

6.5.4 Implementation of proxy and server managers

Proxy Manager: as described earlier, the role of proxy manager is to facilitate the communication between the client and the main media server. It also updates its cache regularly based on the caching scheme being used and serves requests from the cache. The proxy has three threads of control:

I. A thread which listens to client requests
II. A thread which listens to the server response
III. Threads which are responsible for media data streams in cases a request is to be served from proxy cache.

For each client request, the first thread adds cache information and forwards it to the server. Since the response of the server is asynchronous to the request, the second thread is required to handle the response from the server. Information like on which port to join the on-going R-stream or to receive a P-stream data, patch window size and patching costs are extracted from the server response packet. The patch window size and patching cost information are used by the Cache Manager to make best selection of media prefixes to cache. The third thread is responsible to stream data blocks in the cache when required. When the proxy completes streaming all the cached blocks, the remaining data blocks are streamed from the main media server.

The proxy can run under three options: without caching, with caching using the 2PIC scheme, and caching using the proposed scheme. For all three cases, the proxy can be in real mode or simulation mode. In real mode, the proxy listens to actual client requests

and in the simulation mode it reads the synthetic playback *start* and *termination* time file and forwards requests to the main media server. The developed GUI for the proxy server is shown in App.3.3 of Appendix 3.

Server manager: the server manager establishes three main thread types:

I. A thread which listens to client requests and respond
II. A thread to run the predictor module and regularly update the patch window size and patching cost information
III. Threads to control data stream delivery

The first thread calls the AC every time a request arrives. If the request is accepted a data stream thread will be started if the blocks requested are not cached at the proxy. A response packet is also constructed as described in the subsection 6.5.3 and sent to the proxy.

When the server accepts a request, it either establish a new R-stream or the client joins an existing R-stream and receives the missing data in P-stream. Either of these cases requires a thread of control and network connections. R-stream is established on a multicast channel on unique port, here a port indicated by PORT_1 in the response packet. The media data packet in RTP packets are streamed to the proxy or client in this port. The thread which controls this data stream continues to send the data at a specified bit rate until the file ends.

P-streams are unicast streams and the threads of control established for this purpose controls the unicast RTP packets delivery on a unicast endpoint, “P_STREAM_IP” and “PORT_2”. For the prefix part of the media object, the data blocks that are cached at the proxy, are streamed from the proxy. The server decides from where to stream based on the “CACHE” field of the request packets. The data in the P-stream is delivered until end or until the *termination* request arrives for the session. The developed GUI for the main media server is shown in App.3.2 of Appendix 3.

7. SIMULATION AND PERFORMANCE EVALUATION

This chapter presents the performance evaluation method and results obtained in this thesis work. First section talks about how the streaming system discussed in chapter six is simulated. Then the evaluation of the prediction subsystem is presented. The last section discusses the overall system performance comparing streaming system with and without caching. Various metrics are also used to compare the proposed scheme with the 2PIC scheme.

7.1 Simulation setup

For a media server supposed to have significant number of media files in its media repository and which serves very large number of client requests, the resource requirement is huge and difficult to set up in the available computer laboratory environment. However, by modeling important variables it is possible to simulate the system on a single machine and evaluate its performance. The main module of the architecture which needs to be simulated is the data stream module because of its high resource demand. The technique used here is to simulate the streaming time without actual data transfer.

A resource demand of a media object is related to its playback speed during streaming. The bandwidth requirement of a particular media is equal to its streaming speed which is related to its encoding bit rate. A streaming time and storage requirement are also related to playback speed. For instance, a media of 100MB and one hour playback duration has a playback speed of 0.222Mbps assuming CBR encoding. Therefore, a playback time of a block (a segment) of a media can be found from its streaming speed. i.e. if the streaming speed is equal to the encoding bit rate of the media for CBR encoding, the time required to stream a block of media can be simulated. Therefore, the media repository, the Data stream channels and the cache modules are not required and Fig.7.1 shows the simulation setup of the system architecture. And also the clients are represented by the generated real-time traffic (media playback ***start*** and ***termination*** times) of the synthetic workload.

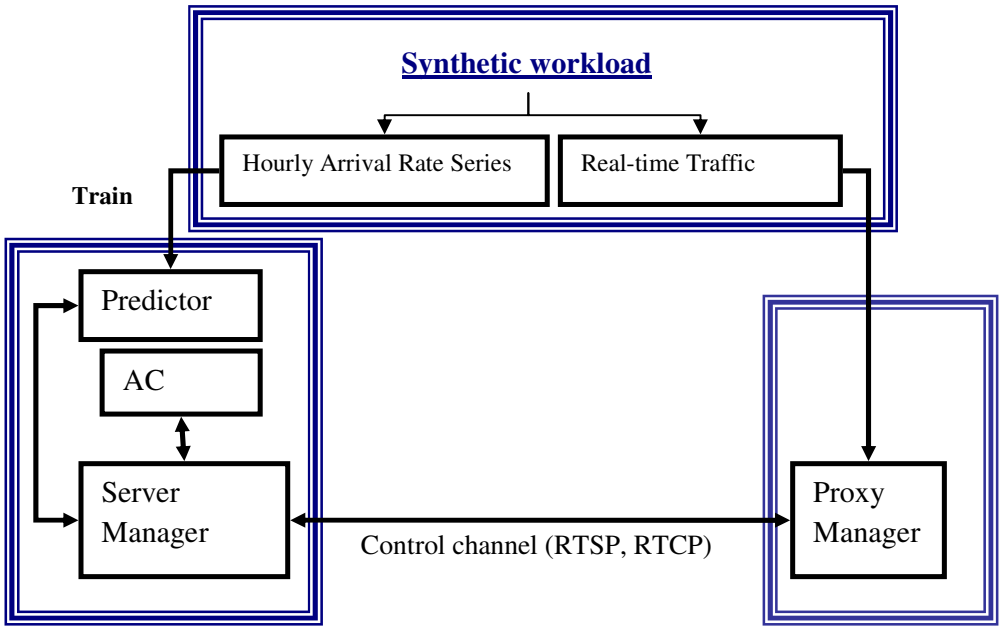


Fig.7.1 Simulation Setup for the System Architecture

In addition to reducing the resource requirement, the simulation facilitates the scaling of experimentation time. With actual data transfer, it takes 24 hours to measure a single day's media streaming. In the simulation which is used here the streaming time can be scaled so as to finish the experimentation within a much shorter time.

As described in chapter six, an accepted request either served in R-stream or in both P-stream and R-stream. Each of these requires RTP connection and a thread of control. In the simulation, a new R-stream reserves a bandwidth equal to a particular media bit rate for a time equal to a playback duration of the same media object. These threads are put in sleep mode for the specified time effectively simulating a background working of data stream threads. The code for these stream simulator threads is shown in the Appendix 5. When the stream ends, the resources reserved are released and the corresponding thread terminates. Similarly, if the P-stream is to be established a bandwidth equal to the bit rate is reserved for a time equal to the interval between current arrival time and start time of the recent R-stream for the same media which the request is to join. For the cached portions, there is no resource reservation between the server and proxy. At the end of stream or arrival of *termination* request for the same stream, the resource reserved is released and the corresponding thread terminates.

Simulation parameters

In the simulation, a media presentation is assumed to have single stream (e.g. video stream). Table 7.1 summarizes the values of parameters and constants used for the synthetic workload generation, media object characteristics and training and testing datasets.

Table 7.1 Values of Parameters and Constants

Media popularity and arrival processes		**Session time**	
workload	180 days	Kx	10
Number of media objects	100	Ky	50
Skew factor	0.729	Skew factor	1.56
K_x	12		
K_y	12		
Peak-load hour multiplying factor	1.5		
Peak-load hours	From5:00PM - 8:00PM		

Media information		**Datasets for the Predictor**	
Playback duration	3600 sec	Training dataset	170 days
Playback bit rate	340Kbps	Testing dataset	10 days

7.2 Evaluation of the prediction subsystem

The performance of the prediction module is important for the performance of media streaming and caching schemes. The best topology for the RC network is selected by testing the performance of the predictor with different number of input, output and reservoir nodes. The metrics used are the training, testing time and RRSE stated in chapter 5. As indicated in table 7.1, the 170 and 10 days hourly arrival rate series are used for training and testing respectively. Evaluation of the predictor is shown in Table 7.2

Table 7.2 Evaluation of the Predictor

No.	Number of input nodes	Number of output nodes	Number of reservoir nodes	Training time in seconds	Testing time in seconds	RRSE
1	24	24	50	10	0.257	1.6
2	24	12	50	15.7	0.36	1.05
3	24	8	50	23.13	0.485	1.01
4	24	4	50	44.4	0.91	0.9
5	24	1	50	170.7	3.4	0.8
6	24	24	100	25.6	0.32	1.7
7	12	24	50	9.6	0.2	2
8	12	12	50	16.8	0.34	1.07
9	12	8	50	24.9	0.31	1.12
10	12	4	50	48.9	0.89	0.72
11	12	1	50	147.8	1.86	0.71
12	12	12	100	40.5	0.53	1.09
13	8	8	50	18.5	0.27	0.93
14	8	4	50	33.93	0.48	0.7
15	8	1	100	498.4	5.25	0.69
16	8	1	50	135.8	1.77	0.69

For the regression parameter(δ), a value of 0.1 is used as it gives smallest RRSE. From table 7.2, it can be seen that the topology with 8-input nodes, 1-output node and 50-reservoir nodes (or 8-1-50) gives the smallest RRSE with an acceptable training and testing time relative to others. Increase in the number of reservoir nodes reduces the prediction error while it increases the training and testing time. Fig.7.2 (a-d) shows graphs indicating the predicted and actual hourly arrival rate series of a particular media file for four different topologies of table 7.2 above. As can be seen the prediction error reduces as the number of output nodes decreases. In this case, the smallest prediction error is obtained when only one-step (next hour arrival rate) is predicted. Hence, the topology (8-1-50) is selected and used in the overall system.

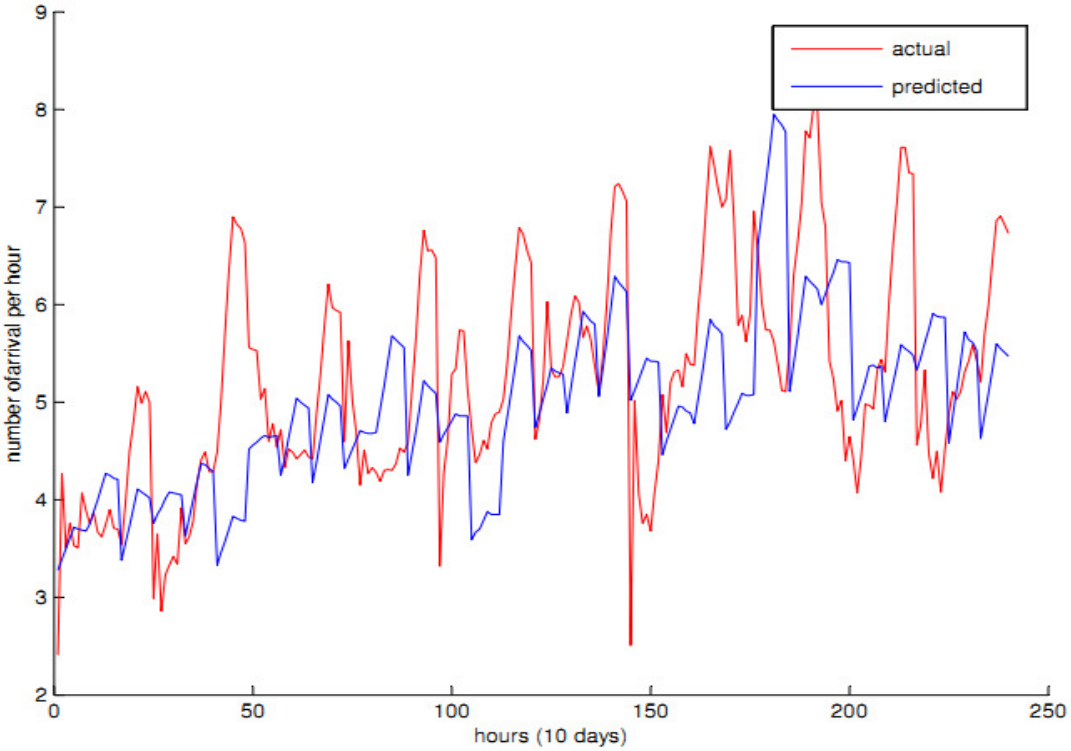


Fig.7.2a Predicted & Actual Rate Series for Topology No. 1 (24-24-50)

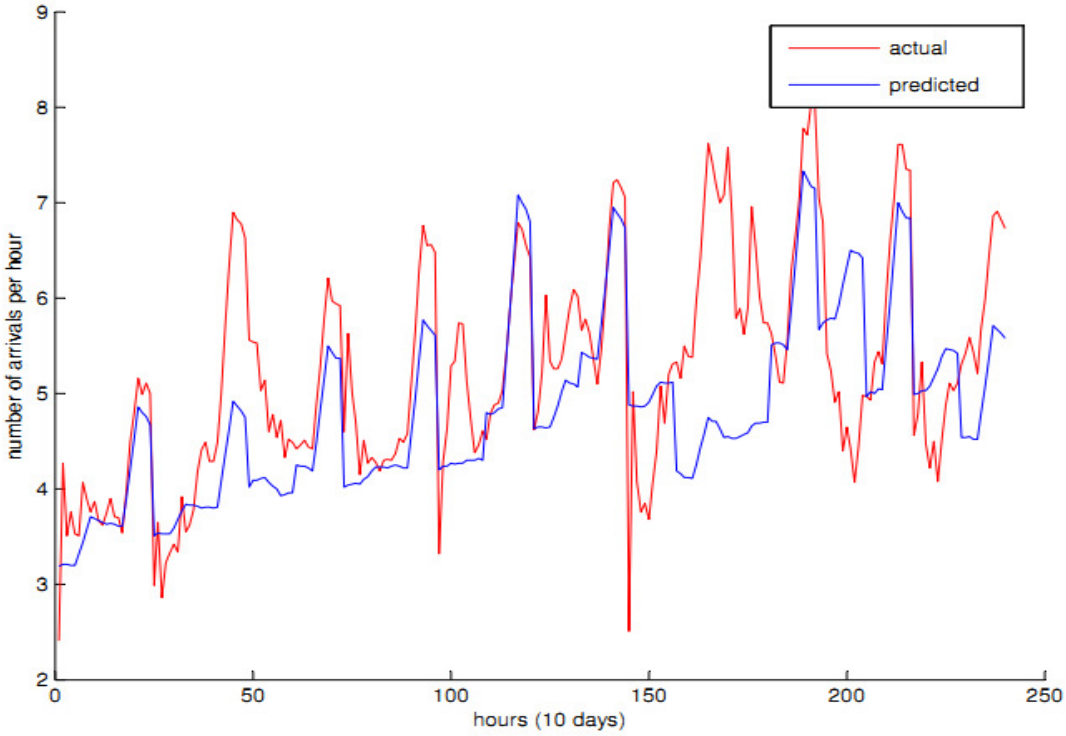


Fig.7.2b Predicted & Actual Rate Series for Topology No. 4 (24-4-50)

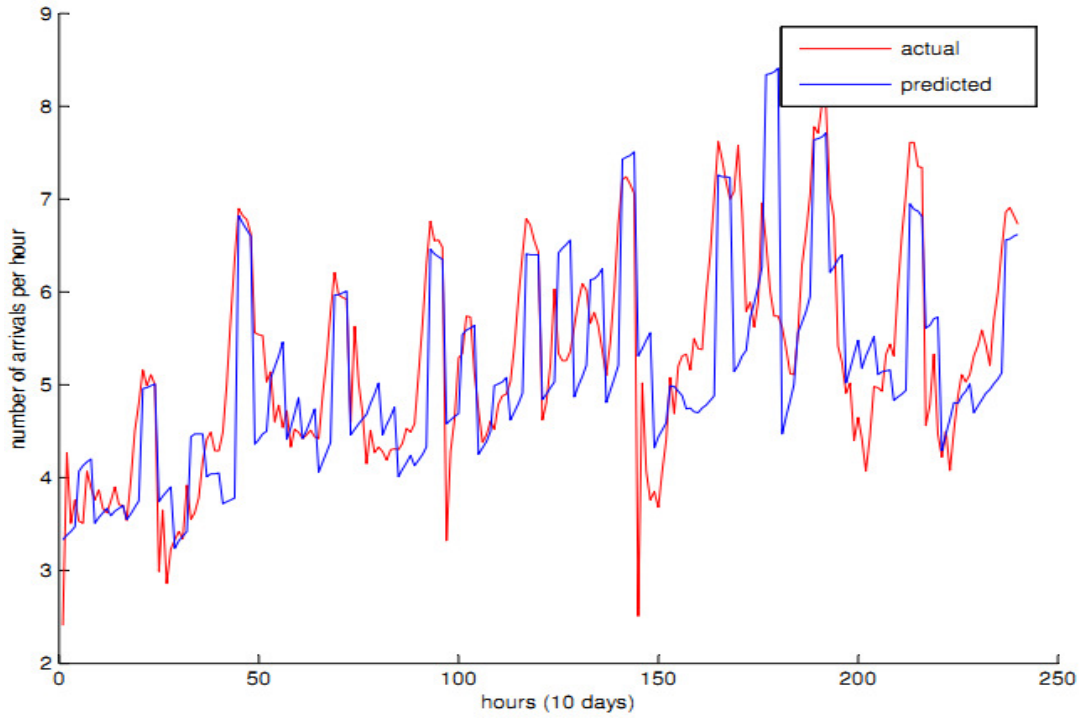


Fig.7.2c Predicted & Actual Rate Series for Topology No. 13 (8-4-50)

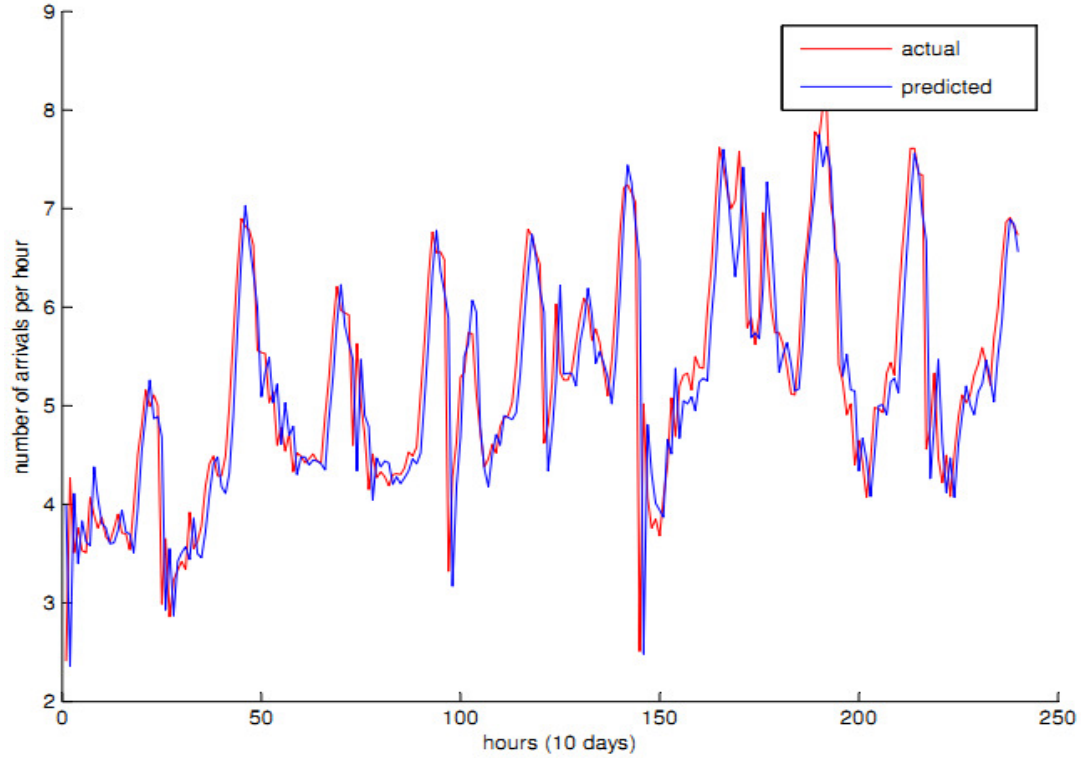


Fig.7.2d Predicted & Actual Rate Series for Topology No. 16 (8-1-50)

7.3 Performance of the overall system

7.3.1 Bandwidth savings

Caching is meant to save bandwidth and hence bandwidth saving is the performance metric for a caching scheme. Here, the bandwidth saving is defined as the average normalized difference between the bandwidth required by two schemes (for instance, with caching and without caching) as given by equ.7.1. The reference required bandwidth is chosen to be of a patch streaming system without caching.

$$BW_{saving} = \frac{1}{N}\sum_{i=1}^{N}\left\{\frac{BW_{yi}-BW_{xi}}{BW_{yi}}\right\} \quad \text{..... (7.1)}$$

Where, N —sample size (number of instants for which the required bandwidth is measured, in this case it is equal to the number of playback *start* requests in one day workload), BW_{yi}, BW_{xi}—required bandwidth for scheme *y* and *x* respectively at instant *i* (for each instant of request arrival). Here, scheme *y* is the reference required bandwidth when there is no caching and the scheme *x* can be either the 2PIC [prefix] or the proposed scheme.

The maximum required bandwidth using each technique is measured by fixing the bandwidth capacity too large such that there is no media request rejected. The caching capacity of the proxy is varied from 500MB to 2500MB (that is equivalent to 3.35% - 16.7% of total media size on the Main Server). These maximum required bandwidth for the patch streaming without caching (or no caching), 2PIC [prefix part] and proposed schemes are measured for each accepted requests in 24-hour time or one day workload. Fig.7.3 (a and b) shows the maximum required bandwidth by these schemes at each instance of *start* request arrival for two proxy caching capacities (1000MB and 2000MB).

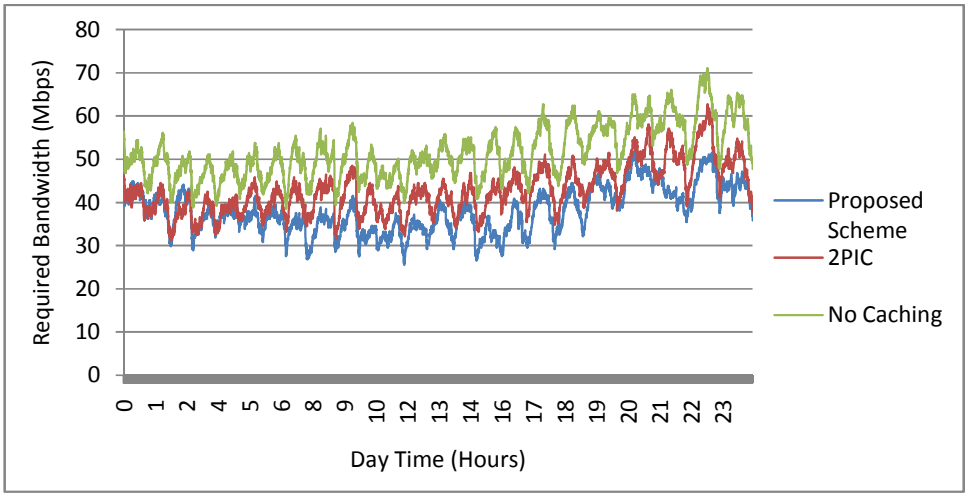


Fig.7.3a Maximum Required Bandwidth with Proxy Cache Capacity of 1000MB

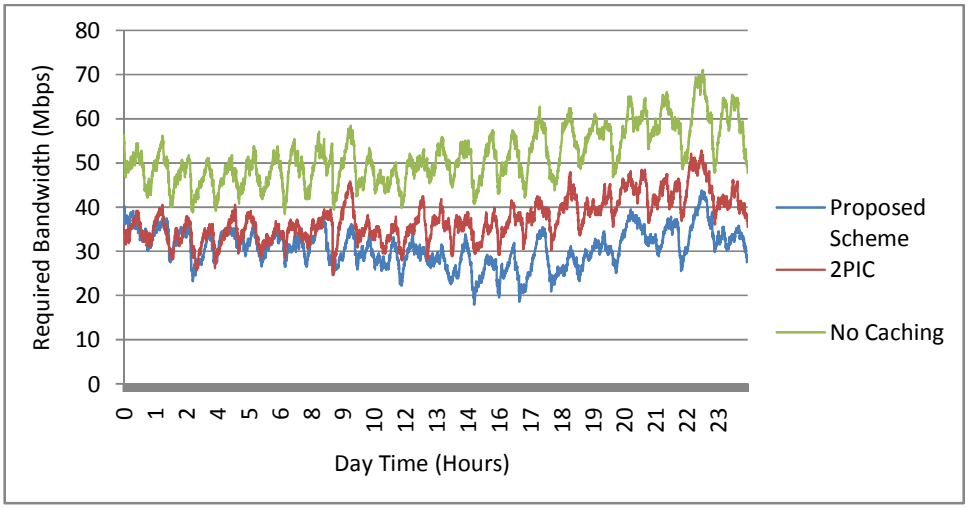


Fig.7.3b Maximum Required Bandwidth with Proxy Cache Capacity of 2000MB

The bandwidth saving is calculated for each proxy caching capacities as in equ.7.1 and is shown in fig.7.4 below (see App.4.1 in appendix 4 for numerical results). The bandwidth saving of the proposed scheme is larger than the 2PIC scheme for all cache sizes. As can be seen from fig.7.4, bandwidth saving of up to 36% and 32% can be obtained from proposed scheme and 2PIC schemes respectively for proxy caching capacity of 2500MB. The bandwidth saving of the proposed scheme even increase during the peak-load hours as can be seen from fig.7.3 above which is the result of increased number of P-streams which can be served from proxy cache.

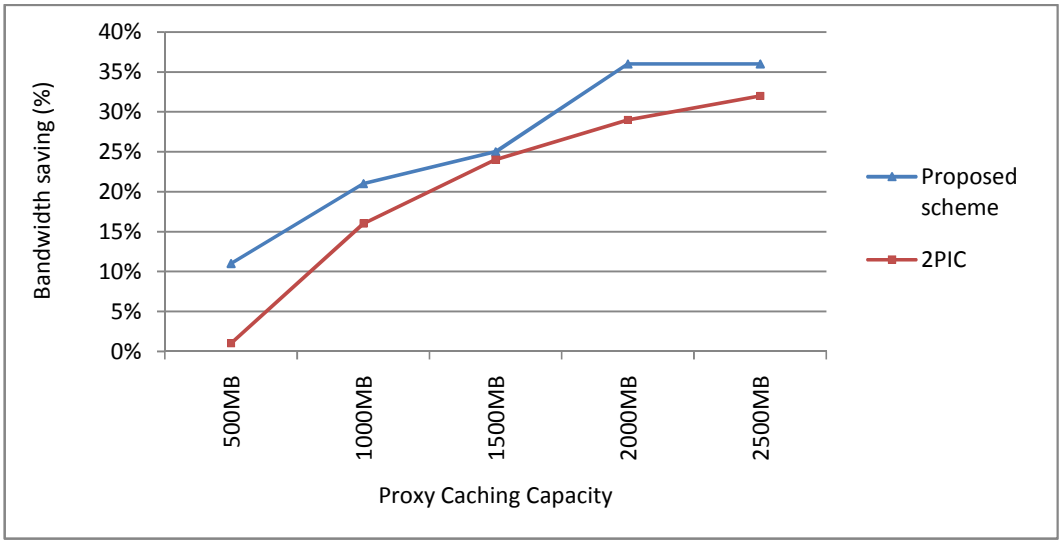


Fig.7.4 Bandwidth Savings

7.3.2 Hit ratio

A cache hit is said to occur if a requested block or segment is found cached at the proxy. Therefore, a hit ratio is defined as the ratio of total cache hit to the total number of blocks to be streamed. With each request there is cached blocks information as discussed in chapter six (section 6.5.3), and hence the hit ratio can be calculated from it.

The hit ratio for different caching capacity of proxy server was obtained by running the simulation so many times. The following figure shows the hit ratio for both caching techniques (proposed scheme and 2PIC [prefix]). The proposed scheme gives much higher hit ratio than the 2PIC scheme for each of the cache capacities (fig.7.5). Numerical results are in App.4.2 of appendix 4.

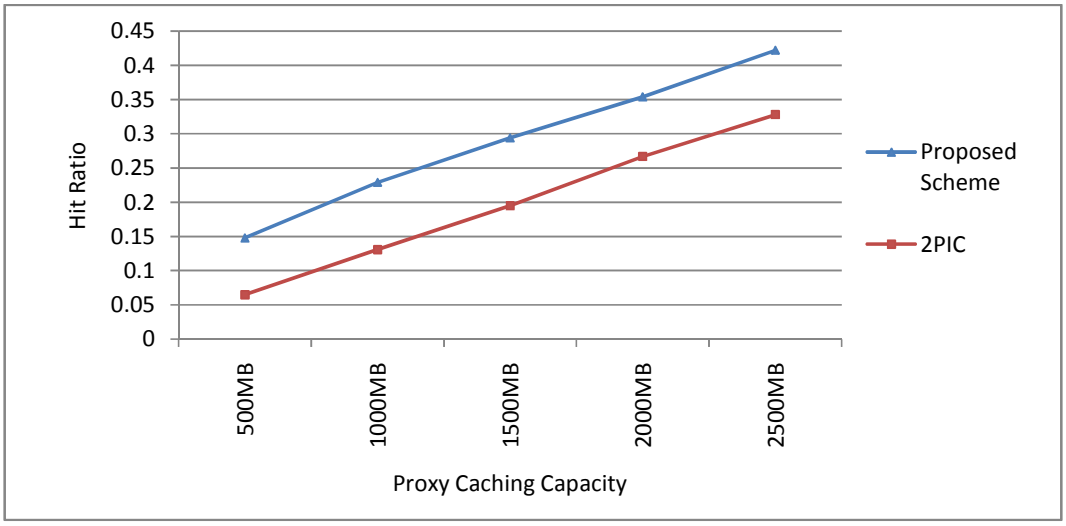


Fig.7.5 Hit Ratio

7.3.3 Maximum number of concurrent clients

Maximum number of concurrent clients that the AC can accept depends on the available resources when the request arrives. Resource consumption of each scheme is different. Here, the cache capacity of the proxy is fixed and the maximum network bandwidth is varied from 25Mbps-50Mbps. The same day media access workload is used for all schemes and the maximum number of concurrent clients is recorded and shown in Fig.7.6 (see App.4.3 in appendix 4 for numerical result). The result shows that the proposed scheme admits more clients followed by the 2PIC scheme. In general, patching without caching accepts fewer requests as it is expected.

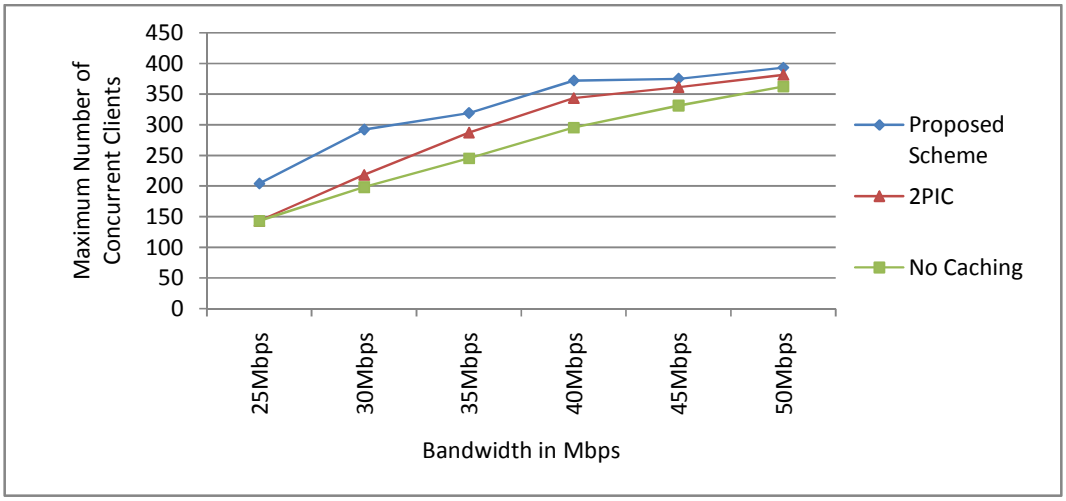


Fig.7.6 Maximum Concurrent Clients

7.4 Conclusion

In this chapter, the simulation setup and performance evaluations are presented. The first section explained how the streaming time is simulated for evaluating the performance of the system. Next, the topology of the RC network is evaluated and the best topology is selected and used in the developed proxy caching patch streaming system. The simulation for different performance metrics has shown that the proposed scheme's performance is better than the 2PIC and no caching schemes.

8. CONCLUSION, RECOMMENDATION AND FUTURE WORKS

8.1 Conclusions

As the objective of this thesis work is to design and implement a caching scheme in Patch Streaming MOD system, there are sequences of steps and procedures required to achieve it.

The first is reviewing of the media server workload characteristics which is required for two purposes: for the caching scheme design and synthetic workload generation. In general, the long-term media popularity distribution is found to be approximated by Zipf-Mandelbrot, stretched exponential or concatenation of two Zipf-like distributions. For the individual media access the ARMA model can capture the daily series and the hourly arrival rate is divided as less busy and peak load hours. An exponential distribution can be used to approximate the inter-arrival time distribution in an hour for each media objects. Media segment popularity follows the Zipf-Mandelbrot distribution. The synthetic workload generator is developed and implemented on Microsoft .Net platform.

Next, after studying the media workload characteristics and reviewing of the patch streaming technique, the analytical model for patching cost is obtained. The patching cost is found to increase with the media request rate. Based on this cost function the prefix cache size selection is made using the Greedy Algorithm. Because the arrival rate varies from hour to hour, the patch window size and cost of patching should be updated every hour. For this purpose, the prediction of hourly arrival rate is analyzed, designed and implemented in the system. The selected prediction technique is the RC approach and is evaluated on the generated synthetic hourly arrival rate series and the best topology is selected and used in the overall system.

Thirdly, the system as a whole is designed and implemented. The system design and implementation issues are the architecture, the interaction between the architecture modules, the streaming protocols and the AC strategy. The communication messages are implemented as UDP data packets.

To evaluate the performance, the system is simulated by replacing the data stream module with respective required streaming time of media segment. Bandwidth saving, hit

ratio and maximum number of concurrent clients are the performance metrics used. The proposed scheme is compared with the 2PIC and no caching and found to perform better based on the above performance metrics. The results show that bandwidth saving of as high as 36% can be obtained from the proposed scheme while 32% from 2PIC [prefix] for proxy caching capacity of about 16.7% of the total media size on the sever compared to the no caching scheme. The hit ratio obtained from the proposed scheme is much higher than that can be obtained from the 2PIC scheme for various proxy caching capacities. Also, larger concurrent numbers of requests are accepted using proposed scheme than the other two schemes.

8.2 Recommendations and Future Works

- **Workload study**

  Though the proposed caching scheme is found to perform as expected, further studies needs to be made on the synthetic workload. This includes the extension of the synthetic workload to add more VCR-like operations, introduction of new media files, the effect of web-links, media types, and testing the scheme on a real world environment.

- **Peer-to-Peer**

  Media streaming system architecture are also becoming more peer-to-peer oriented than the client-server architecture. Therefore, further work is required to integrate the patch streaming and caching in the peer-to-peer architecture.

- **RC Network topology and parameter optimization**

  Parameter and topology optimization in the RC network has to be further investigated to achieve an adaptive topology which can capture more features as user access pattern changes over time.

- **Interval Caching**

  Combining the proposed caching scheme with interval caching can further improve the system performance particularly with an increased support for VCR-like operations. Interval caching is well suited for such operations.

REFERENCES

[1] http://encyclopedia.jrank.org/articles/pages/6833/Multimedia-Servers.html, accessed in October 2010

[2] Asit Dan, Daniel M. Dias, Rajat Mukherjee, Dinkar Sitaram and Renu Tewari, "Buffering and Caching in Large-Scale Video Servers", IEEE, 1995

[3] Joonho Choi, Myungsik Yoo, Biswanath Mukherjee "Efficient Video-on-Demand Streaming for Broadband Access Networks" Proc. Of IEEE "GLOBECOM", 2008

[4] Taeseok Kim, Hyokyung Bahn and Kern Koh, "Popularity-aware interval caching for multimedia streaming servers", ELECTRONICS LETTERS, Vol. 39, No.21 October 2003

[5] Ohhoon Kwon, Hyokyung Bahn and Kern Koh, "Popularity and Prefix Aware Interval Caching for Multimedia Streaming Servers", IEEE, 2008

[6] T.R.GopalaKrishnan nair (Dr.), P Jayarekha, "A Strategy to enable Prefix of Multicast VoD through dynamic buffer allocation", International Journal of Computer Science Issues (IJCSI), Vol. 7, Issue 1, No. 2, January 2010

[7] Lin Wujuan, Law Sie Yong, and Yong Khai Leong, "A Novel Interval Caching Strategy for Video-on-Demand Systems", IEEE, 2006

[8] Sun-Euy Kim Anand Sivasubramaniam Chita R. Das, "Analyzing Cache Performance for Video Servers", IEEE, 1998

[9] Leana Golubchik, John C.S. Lui, Richard R. Muntz, "Adaptive Piggybacking: A novel technique for data sharing in Video-On-Demand Storage Servers", SIGMETRICS/performance '95,

[10] Huadong Ma, Kang G. Shin, "Multicast Video-On-Demand Services", ACM Computer Communication Review, 2002

[11] Bahjat Mohammad Khaleel Qazzez, "Admission Control and Media Delivery Subsystems for video on Demand proxy server", University of Autonomy of Barcelona, Computer Science Department, 2004

[12] Derek Eager, Mary Vernon, "Minimizing Bandwidth Requirements for On-Demand Data Delivery", IEEE Transactions on Knowledge and Data Engineering, Vol. 13, NO. 5, September/October 2001

[13] Jiangchuan Liu, "Proxy Caching For Media Streaming Over the Internet", IEEE Communications, August 2004

[14] Ying Cai, Kien A. Hua, Khanh Vu, "optimizing patching performance", School of Computer Science, University of Central Florida, Orlando, U.S.A

[15] Bin Chang, Liang Dai, Yi Cui, Yuan Xue, "On Feasibility of P2P On-Demand Streaming via Empirical VoD User Behavior Analysis", IEEE, 28th International Conference on Distributed Computing Systems Workshops, 2008

[16] Frank T. Johnsen, Trude Hafsøe and Carsten Griwodz, "Analysis of Server Workload and Client Interactions in a News-on-Demand Streaming System", Proc. of IEEE, 8th International Symposium on Multimedia (ISM'06), 2006

[17] Y. W. Wong, Jack Y. B. Lee, "RECURSIVE PATCHING An Efficient Technique for Multicast Video Streaming", Databases And Information Systems Integration, ICEIS, 2003

[18] Kaihui Li, Changqiao Xu, Yuanhai Zhang, Zhimei Wu, "Optimal Prefix Caching and Data Sharing Strategy", IEEE International Conference on Multimedia & Expo, ICME, IEEE, 2008

[19] Kun-Lung Wu, Philip S. Yu, Joel L. Wolf, "Segmentation of Multimedia Streams for Proxy Caching", IEEE Transactions on Multimedia, Vol. 6, NO. 5, OCTOBER 2004

[20] Ludmila Cherkasova, Member, IEEE, and Minaxi Gupta, "Analysis of Enterprise Media Server Workloads: Access Patterns, Locality, Content Evolution, and Rates of Change", IEEE/ACM Transactions on Networking, Vol. 12, No. 5, OCTOBER 2004

[21] Jiang Yu, Chun Tung Chou, XuDu, TaiWang, "Internal popularity of streaming video and its implication on caching", Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA'06), 2006

[22] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon, "Analyzing the Video Popularity Characteristics of Large-Scale User Generated Content Systems", IEEE/ACM Transactions on Networking, Vol. 17, No. 5, OCTOBER 2009

[23] Gonca Gursun, Mark Crovella, Ibrahim Matta, "Describing and Forecasting Video Access Patterns", Department of Computer Science, Boston University, http://cs-people.bu.edu/goncag/papers/youtube_papers.pdf , accessed on 21- Dec-2010

[24] Wenting Tang, Yun Fu, Ludmila Cherkasova, Amin Vahdat, "Modeling and generating realistic streaming media server workloads", Computer Networks (2007) 336–356

[25] Shudong Jin, Azer Bestavros, "GISMO: A generator of internet streaming media objects and workloads", Technical report BUCS-TR-2001-020, Department of Computer Science, Boston University, October 2001.

[26] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang, "The Stretched Exponential Distribution of Internet Media Access Patterns", ACM, PODC'08, August, 2008.

[27] Jussara M. Almeida Jeffrey Krueger Derek L. Eager Mary K. Vernon, "Analysis of Educational Media Server Workloads", In the proceedings of, NOSSDAV, June, 2001

[28] Wenting Tang, Yun Fu, Ludmila Cherkasova, Amin Vahdat, "MediSyn: A Synthetic Streaming Media Service Workload Generator", in the proceedings of NOSSDAV'03, ACM, June, 2003.

[29] Hua, K., Cai, Y. and Sheu, S., "Patching, A Multicast Technique for True Video-on-Demand Services," ACM Multimedia, 1998.

[30] Tingyao Wu, Michael Timmers, Danny De Vleeschauwer, Werner Van Leekwijck, "On the Use of Reservoir Computing in Popularity Prediction", IEEE Computer Society, Second International Conference on Evolving Internet, 2010

[31] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", Network Working Group, Request for Comments: 3550, July 2003

[32] H. Schulzrinne, A. Rao, R. Lanphier, "Real Time Streaming Protocol (RTSP)", Network Working Group, Request for Comments: 2326, April 1998

[33] H. Jaeger, "Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach", GMD Report 159, German National Research Center for Information Technology, 2002

[34] Tanguy Mezzano, "Echo State Networks application on maze problems", Katholieke Universiteit Leuven, Faculty of Engineering, M.Sc. Thesis, 2007

[35] Stefan Kok, , Dr. Marco A. Wiering, Drs. Leo Pape, Dr. Ignace T.C. Hooge, "Liquid State Machine Optimization", Thesis, Utrecht University, 2007

[36] http://reservoir-computing.org/, accessed in December, 2010

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, Second Edition ,pp#370-384, The MIT Press, Cambridge, 2002

[38] http://www.codeproject.com/KB/recipes/simpleRGN.aspx, accessed in April, 2011

APPENDIXES

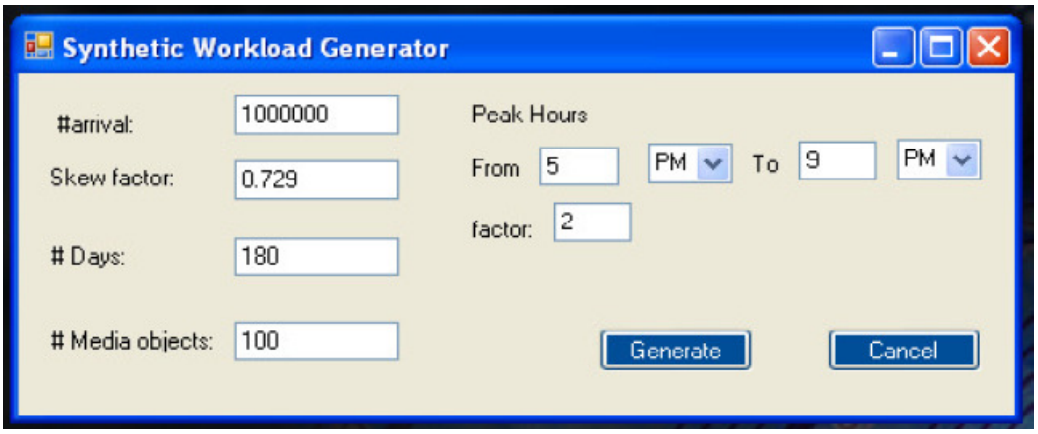
Appendix 1: Patching window size and Cost of Patching Vs arrival Rate

Table App.1.1 Patch window size and Cost of Patching Vs arrival rate

Arrival/Hour	Patch Window, (seconds)	Cost of Patching (seconds)
0	3600.00	1800.00
0.1	3436.02	1971.80
0.2	3297.89	2129.79
0.3	3178.93	2276.84
0.4	3074.77	2414.95
0.5	2982.34	2545.58
0.6	2899.44	2669.83
0.7	2824.42	2788.55
0.8	2756.03	2902.41
0.9	2693.28	3011.98
1	2635.38	3117.69
1.1	2581.71	3219.94
1.2	2531.73	3319.04
1.3	2485.02	3415.26
1.4	2441.21	3508.85
1.5	2400.00	3600.00
1.6	2361.13	3688.90
1.7	2324.37	3775.71
1.8	2289.52	3860.57
1.9	2256.42	3943.60
2	2224.92	4024.92
2.1	2194.89	4104.63
2.2	2166.20	4182.82
2.3	2138.76	4259.58
2.4	2112.48	4334.97
2.5	2087.27	4409.08
2.6	2063.05	4481.96
2.7	2039.76	4553.68
2.8	2017.35	4624.28
2.9	1995.74	4693.83
3	1974.90	4762.35
3.1	1954.78	4829.91
3.2	1935.33	4896.53
3.3	1916.52	4962.26
3.4	1898.31	5027.13
3.5	1880.67	5091.17
3.6	1863.56	5154.42
3.7	1846.97	5216.90
3.8	1830.86	5278.64
3.9	1815.21	5339.66
4	1800.00	5400.00
4.1	1785.21	5459.67
4.2	1770.81	5518.70
4.3	1756.79	5577.10
4.4	1743.13	5634.89
4.5	1729.82	5692.10
4.6	1716.84	5748.74
4.7	1704.18	5804.83
4.8	1691.82	5860.38
4.9	1679.76	5915.40
5	1667.97	5969.92
5.1	1656.45	6023.95
5.2	1645.19	6077.50
5.3	1634.18	6130.58
5.4	1623.41	6183.20
5.5	1612.87	6235.38
5.6	1602.55	6287.13
5.7	1592.44	6338.45
5.8	1582.54	6389.37
5.9	1572.84	6439.88
6	1563.33	6489.99
6.1	1554.01	6539.72
6.2	1544.87	6589.08
6.3	1535.90	6638.07
6.4	1527.09	6686.70
6.5	1518.46	6734.98
6.6	1509.98	6782.92
6.7	1501.65	6830.52
6.8	1493.47	6877.79
6.9	1485.43	6924.74
7	1477.53	6971.37
7.1	1469.77	7017.69
7.2	1462.14	7063.71
7.3	1454.64	7109.43
7.4	1447.26	7154.86
7.5	1440.00	7200.00
7.6	1432.86	7244.86
7.7	1425.83	7289.44
7.8	1418.91	7333.76
7.9	1412.10	7377.80
8	1405.40	7421.59
8.1	1398.79	7465.12
8.2	1392.29	7508.40
8.3	1385.89	7551.42
8.4	1379.57	7594.21
8.5	1373.35	7636.75
8.6	1367.22	7679.06
8.7	1361.18	7721.14
8.8	1355.22	7762.99
8.9	1349.35	7804.61
9	1343.56	7846.02
9.1	1337.85	7887.20
9.2	1332.21	7928.18
9.3	1326.65	7968.94
9.4	1321.17	8009.49
9.5	1315.76	8049.84
9.6	1310.42	8089.99
9.7	1305.14	8129.94

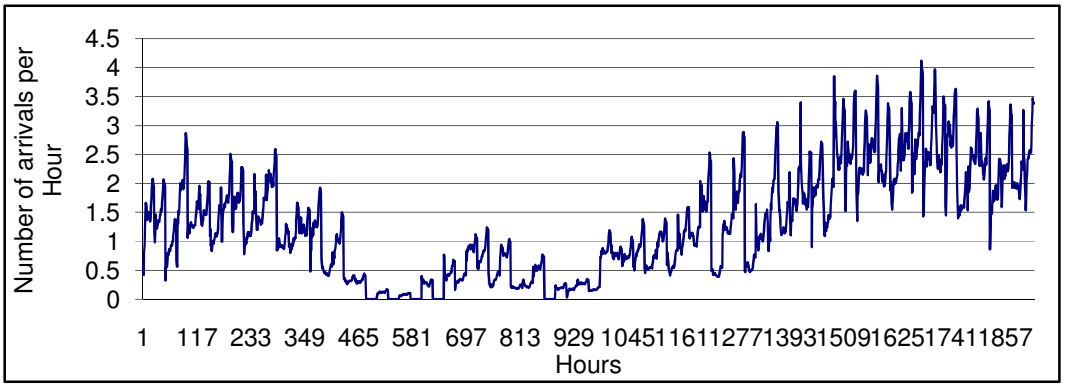
9.8	1299.94	8169.70	13.3	1151.34	9456.43	16.8	1046.18	10587.92
9.9	1294.80	8209.26	13.4	1147.85	9490.63	16.9	1043.61	10618.47
10	1289.73	8248.64	13.5	1144.40	9524.70	17	1041.05	10648.94
10.1	1284.72	8287.82	13.6	1140.98	9558.66	17.1	1038.52	10679.33
10.2	1279.77	8326.82	13.7	1137.59	9592.50	17.2	1036.00	10709.62
10.3	1274.88	8365.64	13.8	1134.23	9626.21	17.3	1033.51	10739.83
10.4	1270.05	8404.28	13.9	1130.91	9659.81	17.4	1031.03	10769.96
10.5	1265.29	8442.75	14	1127.61	9693.30	17.5	1028.57	10800.00
10.6	1260.57	8481.04	14.1	1124.35	9726.66	17.6	1026.13	10829.96
10.7	1255.92	8519.15	14.2	1121.12	9759.92	17.7	1023.71	10859.83
10.8	1251.32	8557.10	14.3	1117.91	9793.06	17.8	1021.31	10889.63
10.9	1246.77	8594.88	14.4	1114.73	9826.09	17.9	1018.92	10919.34
11	1242.27	8632.50	14.5	1111.59	9859.01	18	1016.55	10948.97
11.1	1237.83	8669.95	14.6	1108.47	9891.81	18.1	1014.20	10978.52
11.2	1233.44	8707.24	14.7	1105.38	9924.52	18.2	1011.87	11008.00
11.3	1229.09	8744.37	14.8	1102.31	9957.11	18.3	1009.55	11037.39
11.4	1224.80	8781.34	14.9	1099.27	9989.59	18.4	1007.25	11066.71
11.5	1220.55	8818.16	15	1096.26	10021.98	18.5	1004.97	11095.95
11.6	1216.35	8854.83	15.1	1093.28	10054.25	18.6	1002.70	11125.11
11.7	1212.20	8891.34	15.2	1090.32	10086.43	18.7	1000.45	11154.19
11.8	1208.09	8927.71	15.3	1087.39	10118.50	18.8	998.21	11183.20
11.9	1204.02	8963.93	15.4	1084.48	10150.47	18.9	995.99	11212.14
12	1200.00	9000.00	15.5	1081.59	10182.34	19	993.79	11241.00
12.1	1196.02	9035.93	15.6	1078.73	10214.11	19.1	991.60	11269.78
12.2	1192.08	9071.71	15.7	1075.90	10245.78	19.2	989.43	11298.50
12.3	1188.19	9107.36	15.8	1073.08	10277.35	19.3	987.27	11327.14
12.4	1184.33	9142.87	15.9	1070.29	10308.83	19.4	985.12	11355.70
12.5	1180.52	9178.24	16	1067.53	10340.21	19.5	982.99	11384.20
12.6	1176.74	9213.47	16.1	1064.78	10371.50	19.6	980.88	11412.62
12.7	1173.00	9248.57	16.2	1062.06	10402.69	19.7	978.78	11440.98
12.8	1169.30	9283.53	16.3	1059.36	10433.79	19.8	976.69	11469.26
12.9	1165.64	9318.37	16.4	1056.68	10464.80	19.9	974.62	11497.48
13	1162.01	9353.07	16.5	1054.03	10495.71	20	972.56	11525.62
13.1	1158.42	9387.65	16.6	1051.39	10526.54			
13.2	1154.86	9422.10	16.7	1048.78	10557.27			

Appendix 2: Synthetic workloads

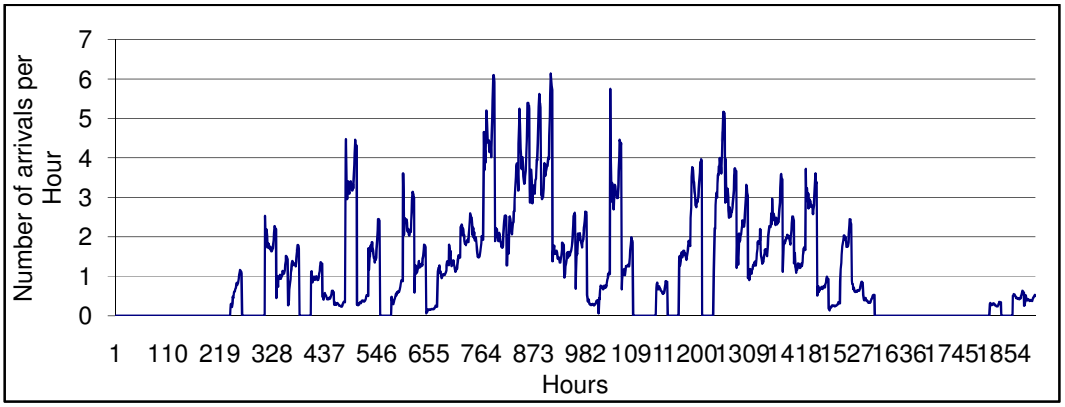


App.2.1 GUI for workload generator

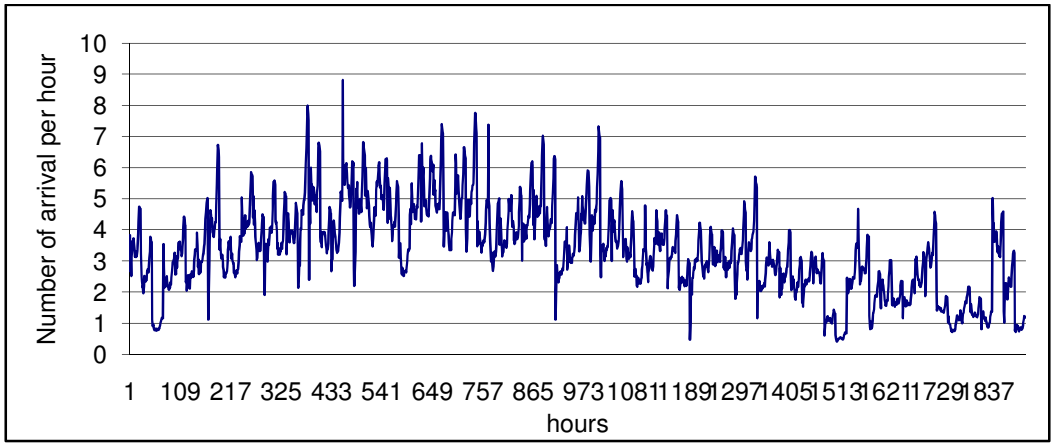
Samples from the Generated Hourly Rate series



(a) Hourly arrival rate series for media 3156026



(b) Hourly arrival rate series for media 3156027

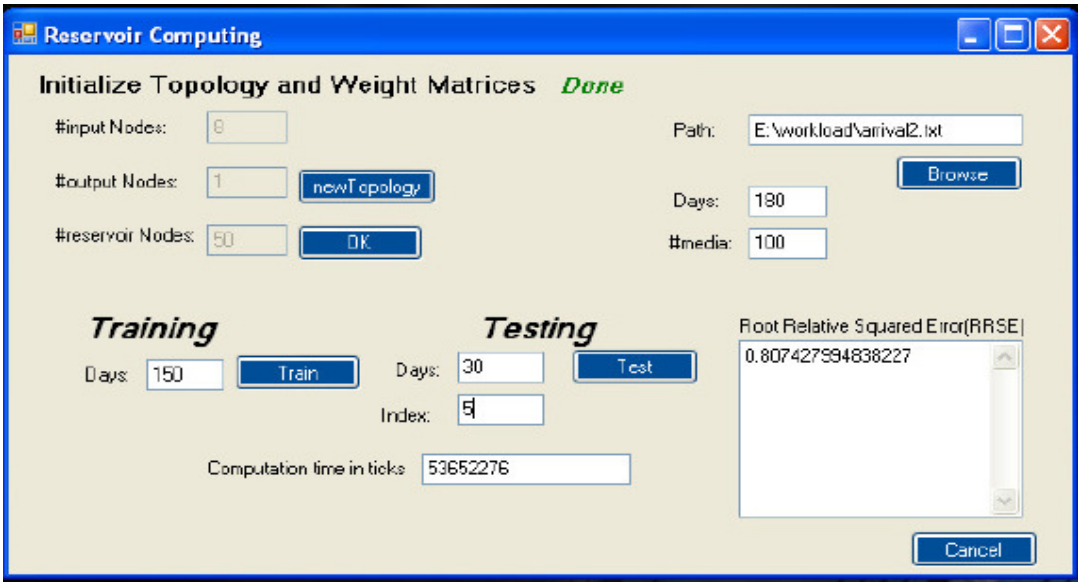


(c) Hourly arrival rate series for media 3156028

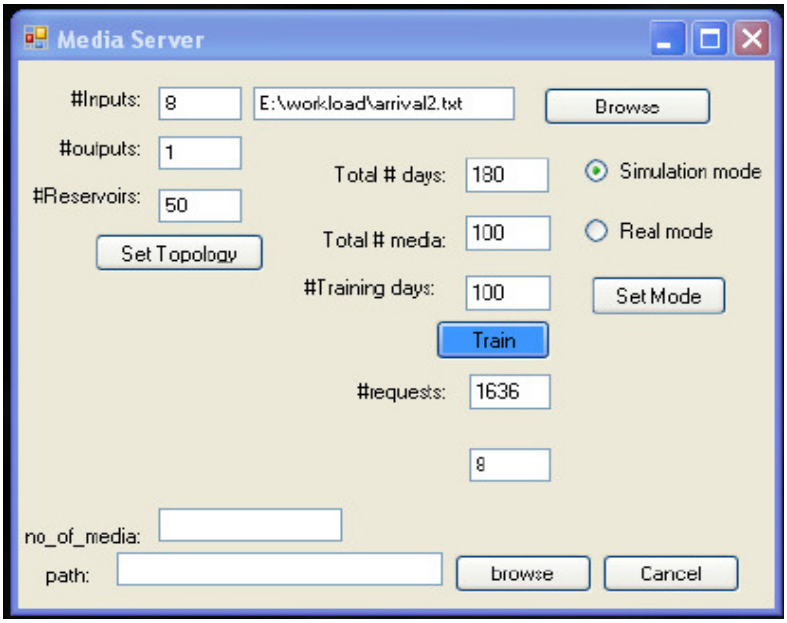
App.2.2 Sample Hourly Arrival Rate Series

Appendix 3: GUIs

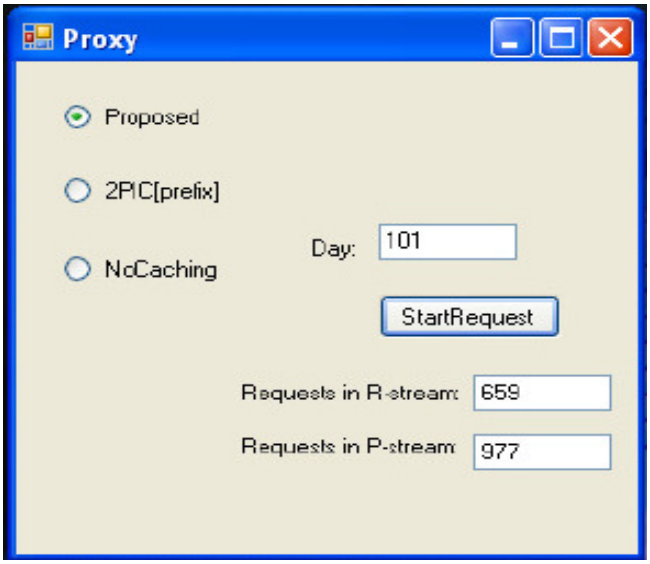
GUI for the RC used in the selection of best topology for the hourly arrival rate prediction



App3.1 GUI for RC



App.3.2 GUI for the main media server



App.3.3 GUI for the Proxy/local server

Appendix 4: Results

App.4.1 Bandwidths saving to the reference patch streaming without caching

Cache Capacity	Bandwidth savings	
	PROPOSED SCHEME	2PIC
500MB	11%	1%
1000MB	21%	16%
1500MB	25%	24%
2000MB	36%	29%
2500MB	36%	32%

App.4.2 Hit ratio for different cache capacity and for both schemes

Cache Capacity in MB	PROPOSED SCHEME	2PIC
500	0.148	0.065
1000	0.229	0.131
1500	0.294	0.195
2000	0.354	0.267
2500	0.422	0.328

App.4.3 Maximum number of concurrent clients (cache capacity=500MB)

BW (Mbps)	PROPOSED SCHEME	2PIC	NO CACHING
25Mbps	204	143	143
30Mbps	292	218	198
35Mbps	319	287	245
40Mbps	372	343	295
45Mbps	375	361	331
50Mbps	393	381	362

Appendix 5: Source codes

The source code for all the work requires large space and cannot be included. A few important methods are added below.

K-Transformed Zipf-like Distribution for Long-term popularity Distribution

```
//*************************************************************************************
**
    public void K_Zipf(int N, int M, double a,double Kx,double Ky, out int[] fre, int[] rank)
    {
       double sum1 = 0;
       for (int r = 1; r <= M; r++)
       {
          sum1 = sum1 + Math.Pow(r + Kx - 1, -1 * a);
       }
       int C =(int) ((N - M*(1-Ky)) / (int)(Ky * Math.Pow(Kx, a) * sum1));
       fre = new int[M];
       for (int r = 1; r <= M; r++)
          fre[rank[r - 1]] = (int)(Ky * C * Math.Pow((r + Kx - 1) / Kx, -1 * a) - Ky + 1);
    }
```

A code to generate the Daily arrival series using ARMA model

```
//*************************************************************************************
***
    public void SeriesDaily(int popularity, out double[] daily_series, int order,int num_days)
    {
       double average = (double)popularity / num_days;
       double[] coef1;
       double[] coef2;
       Random_coeff(out coef1, order);
       Random_coeff(out coef2, order);
       double[] rand_init;;
       double[] output_init = new double[order];
       for (int j = 0; j < order; j++)
          output_init[j] = 1;
       doubleRand(order, out rand_init);
       daily_series = new double[num_days];
       double sum1,sum2=0, next1,next2;
       for (int i = 0; i < num_days; i++)
       {
          sum1 = 0;
          for (int k = 0; k < order; k++)
              sum1 += coef1[k] * output_init[k] + coef2[k] * rand_init[k];
          next1 = g_normal.NextNormal();
          next2=sum1 + next1;
          if (next2 < 0)          next2 = 0;
          daily_series[i] = (next2) * average;
          sum2 += daily_series[i];
          output_init = insert(output_init, next2);
          rand_init = insert(rand_init, next1);
       }
       double scale_factor = (double)popularity / sum2;
       for (int i = 0; i < num_days; i++)
          daily_series[i] = daily_series[i] * scale_factor;
    }
```

```
//*****************************************************************************
***
```

A code to generate hourly arrival rate

```
//*****************************************************************************
***
public void RatePerHour(out  double[] hourly_rate, int daily_media_popularity)
      {
          double temp=hourly_dist[0];
         for (int i = 0; i < 24; i++)
         {
            if (i >= time1 && i <= time2)
            {
               temp = (hourly_dist[(i + 1) % 24] + (i + 1) * (temp*factor)) / (i + 2);
               hourly_dist[i] = temp;
            }
            else
            {
               temp = (hourly_dist[(i + 1) % 24] + (i + 1) * temp) / (i + 2);
               hourly_dist[i] = temp;
            }
         }
         double sum = 0;
         for (int i = 0; i < 24; i++)
            sum += hourly_dist[i];
         for (int i = 0; i < 24; i++)
         {
            hourly_dist[i] = hourly_dist[i] / sum;
            hourly_rate[i] = (hourly_dist[i] * daily_media_popularity);
         }
      }
//*****************************************************************************
```

A code to generate exponentially distributed inter-arrival time based on George Marsaglia's MWC (multiply with carry) generator code from [38]

```
//*****************************************************************************
public class G_Exponential
   {
      private int count;
      private int size;
      private double[] Expo_sample;
      public G_Exponential()
      {
         SimpleRNG.SetSeed(8756, 654);
      }
      public void sample(double number)
      {
          count = 0;
         size = (int)number;
         Expo_sample = new double[size + 1];
         double sum = 0;
         for (int i = 0; i <= size; i++)
         {
            Expo_sample[i] = SimpleRNG.GetExponential(number);
            sum += Expo_sample[i];
         }
      }
```

```
//*************************************************************************************
```

A code to generate the session time

```
//*************************************************************************************
 public void sessionTime(out int[] sessionTime,int N)
      {
         int[] rank = new int[4800];
         for (int i = 0; i < 4800; i++)
            rank[i] = i;
         double Kx = 10;
         double Ky = 50;
         double skewFactor = 1.56;
         int[] fre;
         K_Zipf(N, 4800, skewFactor, Kx, Ky, out fre, rank);
         int count1 = 0, x = 0, temp;
         int[] arr = new int[N];
         for (int i = 0; i < 4800; i++)
         {
            x = 0;
            temp = fre[i];
            while (x < temp)
            {
               arr[count1 + x] = i;
               x++;
            }
            count1 += x;
         }
          sessionTime = new int[N];
         for (int i = 0; i < N; i++)
            sessionTime[i] = arr[rand_int.Next(N)];
      }
//*************************************************************************************
```

Reservoir computing topology initialization

```
//*************************************************************************************
      private void initialization()
      {
         int i;
         int j;
         newRandom = new Random();
        for (i = 0; i < this.Num_Reservoir_states; i++)
            for (j = 0; j < this.Num_Reservoir_states; j++)
               reservoir_weight[i, j] = 2*(newRandom.NextDouble()-.5);
         for (i = 0; i < this.Num_Reservoir_states; i++)
            reservoir_weight[i, i] = 0;
         double[,]  eigenVector;
         double[,] eigenValue;
         double[] eigenValueArray;
         Matrix.Eigen(reservoir_weight.toArray, out eigenValue,out eigenVector);
         eigenValueArray = Matrix.TwoD_2_OneD(eigenValue);
         Array.Sort(eigenValueArray);
         int lastIndex = eigenValueArray.Length - 1;
         reservoir_weight = Matrix.ScalarMultiply(.01, Matrix.ScalarDivide(eigenValueArray[lastIndex],
reservoir_weight));
         for (i = 0; i < this.Num_Reservoir_states; i++)
            for (j = 0; j < this.Num_inputs; j++)
               input_weight[i, j] = 0.05*(newRandom.NextDouble()-0.5);
         for (i = 0; i < this.Num_outputs; i++)
            for (j = 0; j < this.Num_Reservoir_states; j++)
```

```
                read_out_weight[i, j] = newRandom.NextDouble();
            for (i = 0; i < this.Num_Reservoir_states; i++)
                for (j = 0; j < this.Num_outputs; j++)
                    output_weight[i,j] = newRandom.NextDouble();
        }
//*****************************************************************************************
```

A code to find reservoir state and make prediction

```
//*****************************************************************************************
        public Matrix Reservoir_state(Matrix input,Matrix previous_state)
        {
            previous_state =this.reservoir_weight * previous_state +this.input_weight * input;
            previous_state = Matrix.Bi_Sigmoid(previous_state);
            return previous_state;
        }

        public Matrix Predict(Matrix state)
        {
            output_data = read_out_weight * state;
            return output_data;
        }
//*****************************************************************************************
```

A code to train the network

```
//*****************************************************************************************
        public void train(out Matrix a, int days)
        {
             int count1 = this_topology.Num_Inputs;
            int count2 = this_topology.Num_Outputs;
            int offset1 = 0;
            int offset2 = count1;
            Matrix Ax_mat = new Matrix(this_topology.Num_Reservoir_States,
this_topology.Num_Reservoir_States);
            Matrix temp = new Matrix(this_topology.Num_Reservoir_States, 1);
            Matrix gain = new Matrix(this_topology.Num_Reservoir_States, 1);
            Matrix error = new Matrix(count2, 1);
            Matrix desired = new Matrix(count2, 1);
            Matrix xx = new Matrix(1, 1);
            int i;
            double[,] factor = new double[1, 1];
            factor[0, 0] = .5;
            Ax_mat = this_topology.Reservoir_Weight;
            a = new Matrix(this_topology.Num_Reservoir_States, 1);
            Matrix I =new Matrix(Matrix.Identity(this_topology.Num_Reservoir_States));
            Matrix productSumState = new Matrix(this_topology.Num_Reservoir_States,
this_topology.Num_Reservoir_States);
            Matrix productSum = new Matrix(this_topology.Num_Outputs,
this_topology.Num_Reservoir_States);
            while ((offset2 + count2) <= days * 24)
            {
                i = 0;
                while (i < dataParser.Num_media)
                {
                    this_topology.Input_Data = dataParser.inputMatrix(dataParser.Media_ID(i), offset1, count1);
                    this_topology.Internal_State = this_topology.Reservoir_state(this_topology.Input_Data,
this_topology.Internal_State);
  if ((offset2 + count2) > 5 * 24)
                    {
```

```
                productSumState += this_topology.Internal_State *
Matrix.Transpose(this_topology.Internal_State);
                desired = dataParser.inputMatrix(dataParser.Media_ID(i), offset2, count2);
                productSum += desired * Matrix.Transpose(this_topology.Internal_State);
              }
              i++;
            }
            offset1 += count2;
            offset2 += count2;
            a = this_topology.Internal_State;
          }
          this_topology.Read_out_Weight = productSum * Matrix.Inverse(productSumState + .1 * I);
//*************************************************************************
```

A code for testing and evaluating the predictor

```
//*************************************************************************************
      public void Test(out Matrix a, int days,int index)
      {
         int count1 = this_topology.Num_Inputs;
         int count2 = this_topology.Num_Outputs;
         int offset1 = (dataParser.Days - days) * 24 - count1;
         int offset2 = offset1 + count1;
         Matrix error = new Matrix(this_topology.Num_Outputs, 1);
         Matrix desired = new Matrix(count2, 1);
         a = new Matrix(count2, 1);
         int i;
         double scale = (double)days * 24 / count2 * dataParser.Num_media;
         while ((offset2 +count2)<= dataParser.Days * 24)
         {

            i = 0;
            while (i < dataParser.Num_media)
            {
               this_topology.Input_Data = dataParser.inputMatrix(dataParser.Media_ID(i), offset1, count1);
               this_topology.Internal_State = this_topology.Reservoir_state(this_topology.Input_Data,
this_topology.Internal_State);
               this_topology.Output_Data = this_topology.Predict(this_topology.Internal_State);
               desired = dataParser.inputMatrix(dataParser.Media_ID(i), offset2, count2);
               if (i == index)
               {
                  double thisAverage = 0;
                  for (int j = 0; j < count2; j++)
                  {
                     double[,] arr = desired.toArray;
                     thisAverage += arr[j, 0];
                  }
               }
               double sum1 = 0;// 0.00001;
               for (int h = 0; h < count2; h++)//averaging the normalization
                  sum1 += (double)desired[h, 0];
               double d = sum1 / count2;
               if (d != 0)//to avoid division by zero
                  error += Matrix.Power((this_topology.Output_Data - desired) / d, 2);
                i++;
            }
            offset1 += count2;
            offset2 += count2;
         }
```

```
            error = Matrix.ScalarDivide(scale, error);
            a =Matrix.sqrt(error);
        }
    }
//*************************************************************************************
```

A timer event handler to make prediction and update media information

```
//*************************************************************************************
  void timer1_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
      {
         Predictor.update(NewArrivalRate);
         prediction_timer++;
         for (int i = 0; i < NewArrivalRate.Length; i++)
            NewArrivalRate[i] = 0;
         this.indicateRun(++updateCount);
         if (prediction_timer == outputNodes)
         {
            Predictor.predict();
            this.PatchSize();
            DS.UpdateMediaInfo(media_info1);
            prediction_timer = 0;

         }
      }
//*************************************************************************************
```

A thread which send data stream to the client

```
//*************************************************************************************
      public void Run()
      {
         this.server = new PeerClass(LOCAL_PORT, GROUP_IP, RTP_PORT);
         FileStream fis = new FileStream(fileName, FileMode.Open, FileAccess.Read);
         int Data_size = 32768;
         byte[] frame = new byte[Data_size];
         long media_size = fis.Length;
         long pos = 0;
         byte[] packet;
         bool streaming = true;
         while (streaming)
         {
            if (pos <= (media_size - Data_size))
            {
               int len = fis.Read(frame, 0, Data_size);
               rtppacket = new RTPpacket(PType,frameN, time, frame, Data_size);
               rtppacket.getpacket(out packet);
               pos = fis.Position;
               this.server.Send(frame);
            }
            // wait one second
            Thread.Sleep(500);
            if (pos >= media_size)//stop if end of file
               streaming = false;
         }

      }
 //*************************************************************************************
```

A code to calculate the patch window size and patch cost

```
//*****************************************************************************************
      public void PatchSize()
      {
         short j;
         for (int i = 0; i < media_info1.Length; i++)
         {
            double sum = 0;
            double[] rate = Predictor.NexthoursTrace(media_info1[i].media_id);
            for (j = 0; j < rate.Length; j++)
               sum += rate[j]/3600;
            double rate1 = sum / j;
            if (rate1 > 0)
           media_info1[i].patchSize = (int)((Math.Sqrt(2 * rate1 * media_info1[i].playbackDuration + 1) -
1) / rate1);
             else
                media_info1[i].patchSize = media_info1[i].playbackDuration;
     media_info1[i].patchCost = (int)(media_info1[i].playbackDuration * (media_info1[i].patchSize *
rate1 + 1) / 2);
         }
      }
//*****************************************************************************************
```

Structures to represent media information, request and response packets

```
//*****************************************************************************************
   struct media_info
   {
      public int media_id;
      public string fullpath;
      public double playbackRate;//in Mbps
      public int playbackDuration;//in seconds
      public int patchSize;
      public int patchCost;//in terms of time it takes to deliver[unit=seconds]
   }
   struct Request
   {
      public int type;//request type: start playing=1;stop=0
      public int media_id;
      public int request_id;
      public double timeStamp;
      public int cachedSize;

   }
struct response
   {
      public int channelType;
      public int request_Id;
      public int media_id;
      public int portRegular;
      public int portPatch;
      public int Patchwindow;
      public int patchCost;
      public int ResponseType;
   }
//*****************************************************************************************
```

RTP packet constructor

```
//*****************************************************************************************
public RTPpacket(uint PType, uint Framenb, int Time, byte[] data, int data_length)//, out byte[] header1)
      {
```

```
        Version = 2;
        Padding = 0;
        Extension = 0;
        CC = (uint)CC_const.E7;
        Marker = 0;
        Ssrc = 0;
        SequenceNumber = Framenb;
        TimeStamp = Time;
        PayloadType = PType;
        header = new byte[HEADER_SIZE];
        uint temp2 = Version;
        temp2 = temp2 | (Padding << 2);
        temp2 = temp2 | (Extension << 3);
        temp2 = temp2 | (CC << 4);
        temp2 = temp2 | (Marker << 8);
        temp2 = temp2 | (PayloadType << 9);
        temp2 = temp2 | (SequenceNumber << 16);
        string temp1;
        byte[] byteArray1 = BitConverter.GetBytes(temp2);
        temp1 = BitConverter.ToString(byteArray1);
        for (short j = 0; j < 4; j++)
           header[j] = byteArray1[j];
        byte[] byteArray2 = BitConverter.GetBytes(TimeStamp);
        temp1 = temp1 + BitConverter.ToString(byteArray2);
        for (short j = 4; j <= 7; j++)
           header[j] = byteArray2[j - 4];
        byte[] byteArray3 = BitConverter.GetBytes(Ssrc);
        temp1 = temp1 + BitConverter.ToString(byteArray3);
        for (short j = 8; j < 12; j++)
           header[j] = byteArray3[j - 8];
        payload_size = data_length;
        payload = new byte[data_length];
        payload = data;
     }
//*****************************************************************************************
```

The code for the Admission control module

```
//*****************************************************************************************
  class AC
  {
     private double max_BW;
     private double active_BW;
     private int No_activeRequests;
     private StreamWriter sw1;
     private StreamWriter sw2;
     public AC(double  max_bw)//, double a_bw,int requests)
     {
        max_BW = max_bw;
        active_BW = 0.0;
        No_activeRequests = 0;
     }
     public bool isAcceptable(double playbackRate, int timeDuration,Request req,out bool cacheCheck)
     {
        int temp = (int)req.timeStamp % 24;
        if (timeDuration <= 0)
        {
           //addmit request Ri;
           No_activeRequests++;
```

```
            cacheCheck = true;
            return true;
        }
        else if ((max_BW - active_BW) >= playbackRate)
        {
            //addmit request Ri;
            active_BW += playbackRate;
            No_activeRequests++;
            cacheCheck = false;
            return true;
        }
        else
        {
            cacheCheck = false;
            return false;
        }
    }
    public void req_Complete(double playbackRate,int count)
    {
        if (active_BW >= playbackRate)
        {
            active_BW = active_BW - playbackRate;
            No_activeRequests -= count;
        }
    }
    public void req_Complete(double playbackRate)
    {
        if (active_BW >= playbackRate)
        {
            active_BW = active_BW - playbackRate;
        }
    }
}
//*****************************************************************************
```

```
//a method for streaming simulator
    public void Simulate_streaming(Request req)//, int id, int time, string threadName)
    {
        double hitRatio = 0;
        if (req.type == 1)
        {
            if (Number_of_R_stream == maxThread)
                Number_of_R_stream = 0;
            if (P_streamThread_id == maxThread)
                P_streamThread_id = 0;
                int index;
                if (this.R_contains(req.media_id,req, out index))
                {

                    if(AddimisionCtrl.isAcceptable(this.playbackSpeed(req.media_id),GetPatchSize(req.me
                    dia_id)-req.cachedSize,req,out cacheCheck))
                    {
                        cachehit += req.cachedSize;
                        missPlushit += this.GetPatchSize(req.media_id);
                        hitRatio = cachehit / missPlushit;
                        sw.WriteLine(hitRatio.ToString("#0.000"));
                        sw.Flush();
```

```
                    ServerRespone.Send(GetPacket(10, req.request_id,req.media_id,
RTP_PORT_Regular, RTP_PORT_Patch, GetPatchSize(req.media_id), GetPatchCost(req.media_id), 1));
                    Timer = (int)((req.timeStamp - regularList[index].StartTime) * 3600);
                    regularList[index].joinCount++;
                    regularList[index].port = RTP_PORT_Patch++;
                    Number_of_P_stream++;
                    mediaID = req.media_id;
                    P_streamThread[P_streamThread_id] = new Thread(new
ThreadStart(P_streamSimulator));
                    P_streamThread[P_streamThread_id].Start();
                    P_streamThread_id++;
                  }
                  else//if rejected.....
                  {
                    ServerRespone.Send(GetPacket(0, req.request_id,req.media_id, 0,
(int)GetPatchSize(req.media_id), GetPatchCost(req.media_id), 0));

                  }

              }
              else
              {
                if (AddimisionCtrl.isAcceptable(this.playbackSpeed(req.media_id),
this.PlaybackDuration(req.media_id) - req.cachedSize,req,out cacheCheck))
                {
                  cachehit += req.cachedSize;
                  missPlushit += this.PlaybackDuration(req.media_id);
                  hitRatio = cachehit / missPlushit;
                  sw.WriteLine(hitRatio.ToString("#0.000"));
                  sw.Flush();
                  ServerRespone.Send(GetPacket(20, req.request_id,req.media_id, RTP_PORT_Regular,
GetPatchSize(req.media_id), GetPatchCost(req.media_id), 1));
                  Timer = this.PlaybackDuration(req.media_id);
                  regularList[Number_of_R_stream].media_id = req.media_id;
                  regularList[Number_of_R_stream].StartTime = req.timeStamp;
                  regularList[Number_of_R_stream].index = Number_of_R_stream;
                  regularList[Number_of_R_stream].port = RTP_PORT_Regular++;
                  regularList[Number_of_R_stream].request_id = req.request_id;
                  regularList[Number_of_R_stream].joinCount = 1;

                  mediaID = req.media_id;
                  reqIndex = Number_of_R_stream;
                  R_streamThread[Number_of_R_stream] = new Thread(new
ThreadStart(R_streamSimulator));
                  R_streamThread[Number_of_R_stream].Start();
                  Number_of_R_stream++;
                }
                else //if rejected
                {
                  ServerRespone.Send(GetPacket(0, req.request_id,req.media_id, RTP_PORT_Regular,
(int)GetPatchSize(req.media_id), GetPatchCost(req.media_id), 0));

                }

              }

            /* }
          }
```

```
        else //if the request is to terminate
        {
          AddimisionCtrl.req_Complete(this.playbackSpeed(id));
          P_streamThread[P_streamThread_id].join();
           //.....................................
        }
      }

//*************************************************************************************
```

A code to simulate the R-stream and P-stream

```
//*************************************************************************************
    public void R_streamSimulator()
     {
        Thread.BeginCriticalRegion();
        int time = Timer * 1000 / 24;//in milliseconds
        int id = mediaID;
        int reqId = reqIndex;
        bool cached = cacheCheck;
        Thread.EndCriticalRegion();
        if(time>0)
        Thread.Sleep(time);
        if (!cached)
           AddimisionCtrl.req_Complete(this.playbackSpeed(id), regularList[reqId].joinCount);
        regularList[reqId].joinCount = 0;
     }
     public void P_streamSimulator()
     {
        Thread.BeginCriticalRegion();
        int time = Timer * 1000 / 24;//in milliseconds
        int id = mediaID;
        bool cached = cacheCheck;
        Thread.EndCriticalRegion();
        if (!cached)
        {
           if (time > 0)
              Thread.Sleep(time);
           AddimisionCtrl.req_Complete(this.playbackSpeed(id));
        }
     }
//*************************************************************************************
```

The code to select media prefixes both for PROPOSED SCHEME and 2PIC

```
/*************************************************************************************
   class CacheM
   {
      private int maxCache;
      private int usedCache;
      private int maxNo_media;
      private cacheInfo[] cacheList;
      public CacheM(int cacheSize, int maxNoOfMedia)
      {
         maxCache = cacheSize;
         usedCache = 0;
         maxNo_media = maxNoOfMedia;
         cacheList = new cacheInfo[maxNo_media];
         for (int i = 0; i < maxNo_media; i++)
```

```
            cacheList[i] = new cacheInfo();
        }
        public void InitialCache()
        {
            int media_id = 3156020;
            int portion= 1024 * 8 * maxCache / (maxNo_media * 340);
            for (int i = 0; i < maxNo_media; i++)
            {
                cacheList[i].media_id = ++media_id;
                cacheList[i].portion = portion;
            }
         }

        public void insert(response res)
        {
            int index = 0;
            bool set = false;
            while (index < maxNo_media)
            {
                if (cacheList[index].media_id == 0)
                    break;
                if (res.media_id == cacheList[index].media_id)
                {
                    cacheList[index].patchWindow = res.Patchwindow;
                    cacheList[index].patchCost = res.patchCost;
                    set = true;
                }
                index++;
            }
            if (!set)
                if (index < maxNo_media)
                {
                    cacheList[index].media_id = res.media_id;
                    cacheList[index].patchWindow = res.Patchwindow;
                    cacheList[index].patchCost = res.patchCost;
                }
        }

        public void insert(int id,double arr_time)
        {
            int index = 0;
            bool set = false;
            while (index < maxNo_media)
            {
                if (cacheList[index].media_id == 0)
                    break;
                if (id == cacheList[index].media_id)
                {
                    //PI=alpha*PIn-1+(alpha-)*IntervalTime [2PIC-prefix formula]
                    cacheList[index].gap = 0.5 * cacheList[index].portion + 0.5 * (cacheList[index].gap +
arr_time - cacheList[index].Previous_time);
                    cacheList[index].Previous_time = arr_time;
                    set = true;
                }
                index++;
            }
            if (!set)
                if (index < maxNo_media)
```

```
            {
               cacheList[index].media_id = id;
               cacheList[index].gap = 0.5 * cacheList[index].gap + 0.5 * (arr_time -
cacheList[index].Previous_time);
               cacheList[index].Previous_time = arr_time;
            }
      }
      public void PC_CacheSize()
      {
         int len = cacheList.Length;
         int cost, id, window, portion, x;
         for (int j = 1; j < len; j++)
         {
            cost = cacheList[j].patchCost;
            id = cacheList[j].media_id;
            window = cacheList[j].patchWindow;
            portion = cacheList[j].portion;
            x = j;
            while ((x > 0) && (cacheList[x - 1].patchCost > cost))
            {
               cacheList[x].patchCost = cacheList[x - 1].patchCost;
               cacheList[x].media_id = cacheList[x - 1].media_id;
               cacheList[x].patchWindow = cacheList[x - 1].patchWindow;
               cacheList[x].portion = cacheList[x - 1].portion;
               x--;
            }
            cacheList[x].patchCost = cost;
            cacheList[x].media_id = id;
            cacheList[x].patchWindow = window;
            cacheList[x].portion = portion;
         }
         int R = maxCache * 1024 * 8 / 340;
         int i = maxNo_media - 1;
         while (i >=0)
         {
            if (R >= cacheList[i].patchWindow)
               cacheList[i].portion = cacheList[i].patchWindow;   //or fi = 1;
            else
               cacheList[i].portion = R;                        //or fi=R/pwi;
           if(R>0)
            R = R - cacheList[i].portion;
            i--;
         }
      }
      public void Cache2PICPrefix()
      {
         //start with insertion sort
         int len = cacheList.Length;
         int  id, portion, x;
         double gap,pre_time;
         for (int j = 1; j < len; j++)
         {
            gap = cacheList[j].gap;
            id = cacheList[j].media_id;
            pre_time = cacheList[j].Previous_time;
            portion = cacheList[j].portion;
            x = j;
            //loop through our array inserting each value
```

```
            while ((x > 0) && (cacheList[x - 1].gap > gap))
            {
                cacheList[x].gap = cacheList[x - 1].gap;
                cacheList[x].media_id = cacheList[x - 1].media_id;
                cacheList[x].Previous_time = cacheList[x - 1].Previous_time;
                cacheList[x].portion = cacheList[x - 1].portion;
                x--;
            }
            cacheList[x].gap = gap;
            cacheList[x].media_id = id;
            cacheList[x].Previous_time = pre_time;
            cacheList[x].portion = portion;
        }
        int R = maxCache * 1024 * 8 / 340;
        int i = 0;
        double sum = 0;
        for (int j = 0; j < cacheList.Length; j++)
            sum += cacheList[j].gap;
        double share = R / sum;
        while (i < maxNo_media)
        {
            if (R >= (int)cacheList[i].gap)
                cacheList[i].portion = (int)(cacheList[i].gap * share);
            else
                cacheList[i].portion = R;
          if(R>0)
            R = R - cacheList[i].portion;
            i++;
        }
      }
    }
//*************************************************************************************
```

Declaration

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr. Kumudha Raimond, has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

_____

Dinkisa Aga Bulti

This thesis has been submitted for examination with my approval as a university advisor

Advisor,

_____

Dr. Kumudha Raimond

Department of Electrical and Computer Engineering,

Addis Ababa Institute of Technology (AAiT), Addis Ababa University,

Addis Ababa,

August, 2011