UNIVERSITY OF OSLO
Department of Informatics

RBC: A Relevance Based Caching Algorithm for P2P Access Patterns

Master thesis

Kristoffer Høegh Mysen

9th July 2007

Preface

This Master Thesis is written at the Distributed Multimedia Systems Research Group at the Department of Informatics, University of Oslo, between August 2006 and August 2007.

I want to give my special thanks to both of my supervisors, Vera Goebel and Karl-André Skevik. They have both given me outstanding advices and shown great patience throughout this process.

Kristoffer Høegh Mysen
University of Oslo
July, 2007

Abstract

The emerging of content providers such as YouTube induces a rapidly increasing demand for multimedia streaming, which augments the network resource consumption. Current content distribution networks however are not suited for the high quality video streaming we can expect in the future. Nevertheless, Peer-to-Peer (P2P) networking ensures a high degree of scalability and is a possible solution. On the other hand, P2P also imposes higher resource requirements on the end users. The end users have to use disk access time and CPU resources in order to serve requests. This resource consumption can reduce the playback quality, and it is therefore desirable to reduce the resource cost of P2P networking. One method is to employ caching.

While P2P networking seems promising with respect to scalability issues, it also creates traffic patterns that make current caching strategies insufficient. This thesis examines different caching techniques and their performance with P2P traffic patterns. These differ from regular patterns since clients request individual blocks of a file from multiple providers, instead of downloading the file as a whole from one provider alone.

In this thesis we show that existing caching algorithms are inefficient in combination with P2P multimedia streaming. Multiple difficulties associated with P2P traffic patterns have been detected. To solve these problems, we propose a new and improved caching technique called Relevance Based Caching (RBC). RBC uses prefetching, in which the most relevant blocks are cached. The caching algorithm identifies the P2P access pattern, and together with the popularity of the individual blocks and the files as a whole, it calculates relevance values for each block. We show that by using this algorithm, we obtain a good performance, without exerting too high resource demands on the end users.

Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Problem Description 2

1.3 Outline 3

2 Distributing Multimedia Content **5**

2.1 Multimedia Content 7

2.1.1 Multimedia Characteristics 7

2.1.2 General Multimedia Use and End User Resources 8

2.2 Streaming 9

2.2.1 Protocols Used for Multimedia Streaming 9

2.2.2 Streaming Requirements 10

2.3 Network Architectures 12

2.3.1 Content Distribution Networks 12

2.3.2 Peer-to-Peer 14

2.3.3 The SPP Architecture 18

2.4 Caching 19

2.4.1 Caching Techniques for Streaming Media 20

2.4.2 Memory Caching 21

2.5 Summary 23

3 Existing Caching Algorithms **25**

3.1 Block-level Caching 25

3.1.1 First-In First-Out Cache Replacement 25

3.1.2 Least Frequently Used Cache Replacement 26

3.1.3 Least Recently Used Cache Replacement 26
3.1.4 Least/Most Relevant for Presentation Cache Replacement 27
3.2 Stream-Dependant Caching 30
3.2.1 Interval Caching/Generalized Interval Caching 30
3.2.2 The BASIC Cache Replacement Algorithm 31
3.3 Summary 32

4 Performance Analysis and Evaluation of Existing Caching Algorithms 33
4.1 Simulation Description 33
4.1.1 Implementation 34
4.1.2 Evaluation Method 34
4.1.3 Metrics 35
4.1.4 Factors 35
4.1.5 Workloads 36
4.2 Simulations 38
4.2.1 Access Pattern 39
4.2.2 Analysis of The Random Eviction Algorithm 40
4.2.3 Analysis of the LFU algorithm 44
4.2.4 Analysis of the LRU algorithm 46
4.3 Summary 49

5 Design of a P2P Multimedia Streaming Caching Algorithm. 51
5.1 Design Objectives 51
5.2 General Design 52
5.3 Cost Prediction 57
5.4 Summary 60

6 Performance Analysis and Evaluation of the RBC Algorithm. 61
6.1 Simulation Description 61
6.1.1 Implementation 61
6.1.2 Metrics 62
6.1.3 Factors 62
6.1.4 Workloads 63
6.1.5 Performance Goal 64

6.2 Simulations 64
6.2.1 Results from Default Setup 65
6.2.2 Experiments with Cache Sizes 67
6.2.3 Experiments with Relevance R 69
6.2.4 Experiments with Relevance A 71
6.2.5 Experiments with Window Sizes 73
6.2.6 Experiments with Combinations of Relevance Values . . . 75
6.2.7 Verification of Results using Workload 3 77
6.3 Summary 79

7 Conclusions **81**
7.1 Contributions 81
7.1.1 Design and Implementation 81
7.1.2 Evaluation 82
7.2 Critical Assessment 83
7.3 Future Work 83

Bibliography **84**

A List of Abbreviations **89**

B Running The Simulations **91**

C Source Code **93**

D The CD **101**

List of Figures

2.1 Network traffic increase. 5
2.2 Network factors. 11
2.3 High Level Factors [25]. 14
2.4 Decentrialized P2P achitecture. 15
2.5 Client-Server request pattern. 15
2.6 P2P request pattern. 16
2.7 SPP architecture [32]. 19
2.8 Client-Server interaction. 21
2.9 P2P interaction. 22

3.1 LRU cache replacement. 27
3.2 L/MRP Client - Server interaction. 28
3.3 L/MRP 28
3.4 Results from simulations with Q-L/MRP [17]. 29
3.5 Interval Caching 30

4.1 A section of the trace file for Workload 1. 37
4.2 A section of the log file for Workload 2. 38
4.3 Showing how the client gets a block through a SCC. 38
4.4 Showing an access pattern on one client from Workload 1. 39
4.5 Showing an access pattern on one client from Workload 2. 40
4.6 Workload 1, cache size=102, Random Eviction. 41
4.7 Workload 1, cache size=204, Random Eviction. 42
4.8 Workload 2, cache size=102, Random Eviction. 43
4.9 Workload 1, cache size=102, LFU. 44

4.10 Workload 1, cache size=204, LFU. 45
4.11 Workload 2, cache size=102, LFU. 46
4.12 Workload 1, cache size=102, LRU. 47
4.13 Workload 1, cache size=204, LRU. 48
4.14 Workload 2, cache size=102, LRU. 49

5.1 Relevance values connected to blocks in the cache. 53
5.2 A sorted list with the relevance values R_g 54
5.3 Five different peers with different connection speeds. 55
5.4 Request pattern from five peers. 56
5.5 Cache replacement in RBC. 57
5.6 Detailed description of the RBC algorithm. 58
5.7 Steps to calculate R_{total}. 58

6.1 Workload 1, default setup, RBC. 66
6.2 Workload 2, default setup, RBC. 67
6.3 Workload 1, Cache Size=204, RBC. 68
6.4 Workload 2, Cache Size=204, RBC. 69
6.5 Workload 1, Relevance R=2, RBC. 70
6.6 Workload 2, Relevance R=2, RBC. 71
6.7 Workload 2, Maximum Relevance A=20, RBC. 72
6.8 Workload 2, Relevance A Increase=0,10, RBC. 73
6.9 Workload 2, Window Size=40, RBC. 74
6.10 Workload 2, Window Size=10, RBC. 75
6.11 Workload 2, Relevance R only, RBC. 76
6.12 Workload 2, Relevance R and G, RBC. 77
6.13 Workload 2, Relevance R and A, RBC. 78
6.14 Workload 3, default setup, RBC 78
6.15 Workload 3, Window Size=10, Relevance A Increase=0.10, RBC. 79

7.1 Cache replacement proposal. 84

B.1 An example of a Gnuplot script. 92

List of Tables

2.1 Characteristics of typical multimedia streams [12] 7

5.1 Cost Analysis of RANDOM, LFU, LRU and RBC. 60

6.1 Formulas describing when we have cache misses with RBC. . . . 79

Chapter 1

Introduction

1.1 Background and Motivation

The demand for web content is increasing every day as people begin to use the Internet as part of their daily lives. While people register the increasing availability of the technology around them, they do not see the limitations of the Internet as it is composed today. As users become more and more aware of the possibilities Internet offers, an ever-increasing number of people start to use it. We make our own homepages where we upload our photos, videos and other multimedia data, which subsequently are distributed to large consumer populations. This imposes new requirements on the underlying distribution architecture.

The main issue with today's Internet is that the architecture is not built for the heavy load many servers are experiencing today. All user requests for a single web page or any other content are generally handled by a single server. This approach is not scalable. When a large user population is requesting the same content from one site at the same time, server resources like CPU capacity or bandwidth (BW) become the bottleneck. Another problem is the distance between a web server and a web client. When a user is requesting some content, he/she is not aware of its actual location because this is transparent. If the content is located at a web server in the USA and the user is located in Europe, the distance leads to significant delays. This happens even if the server has enough resources to handle all its incoming requests.

Different approaches have been taken to meet the increasing demand of web content. One obvious approach, albeit a costly one, would be to simply increase the available BW or CPU resources, or add other physical measures to secure a good quality of service (QoS). These steps however, entail a high cost for the content providers resulting in higher prices for the content consumers. Consumers will generally choose a best-effort service if it is cheaper.

Today several evolutionary steps have been taken to ease the server side's bottleneck, all at which seem to work. One step is to distribute load at a centralized server site, by establishing server farms where each web server shares the burden of serving requests. This is normally handled by a load balancer which dispatches the request to a server with enough available resources. Another step that is currently under intense investigation is distributing the content and employing centralized services. The basic idea is to move the content closer to the user by either server replication or Web caches. Web caches work as a temporary storage space for requested content and therefore reduce network traffic and retrieval time. The final step for solving the increasing demand for Web content is to use Peer-to-Peer (P2P) distribution architectures. P2P distribute the content and the services among the end users, having them participating with their resources. By doing this, the resource load is spread over a much larger user population, which results in a very scalable architecture [29].

1.2 Problem Description

Multimedia content is increasing in popularity [24]. The demand for this type of content is growing every day, although the current content distribution networks are not suited for the heavy high quality multimedia streaming we can expect in the future. P2P networking is one solution to this problem. By using end user resources, it is possible to achieve a high degree of scalability.

Compared to servers or proxy caches, end users have a much more limited supply of resources. While a server or proxy cache is only concerned with serving requests, end users have to simultaneously run playback with a proper quality. However, the bottleneck at an end user is not the high CPU resource consumption enforced by playback, but the disk retrieval times imposed by serving requests. We measure disk access in ms, while we measure memory access in ns, which is a ratio of 1:1000000. Disk I/O takes 1000000 times longer, and is more CPU consuming. Therefore end user P2P applications must keep the relevant data close to hand, stored in the memory.

There already exist multiple caching strategies, developed to decrease disk I/O resource requirements, and to decrease response time. However, P2P multimedia streaming creates new traffic patterns which make current caching strategies insufficient. This thesis' objectives are:

- **1.** Identify shortcomings of existing caching strategies when used with P2P multimedia streaming.

- **2.** Propose a new caching strategy especially designed for P2P traffic patterns.

We will identify the shortcomings of existing caching strategies by implementing a simulation environment for each of the selected strategies. Then we will evaluate the results with respect to how much disk I/O we are able to avoid. Combining these results with the learnings from the theoretical part of this thesis, we will design, implement and analyze a new caching strategy, streamlined for use with P2P multimedia streaming.

1.3 Outline

We begin the work for this thesis by creating a theoretical basis for designing, implementing and evaluating a new P2P caching algorithm. Chapter 2 starts by identifying different contents available in the Internet today, with a special focus on multimedia content, as this is the content type we will work with throughout this thesis. Then we give an overview of streaming characteristics and requirements, and use this as a basis for evaluating different network architectures developed for content distribution. Within the presented architectures, we will emphasize P2P. Finally, we explain caching and different caching techniques used in combination with multimedia streaming.

In Chapter 3, we explain a selection of already existing caching algorithms designed for Client-Server architectures. We introduce block-level caching algorithms and stream-dependant caching algorithms. Finally, we identify shortcomings of the algorithms when these are used together with P2P multimedia streaming.

In Chapter 4, we give a performance analysis and evaluation of existing algorithms. First, we explain how we implemented the different algorithms, and illustrate special design decisions we have made. Then we introduce the evaluation method, metrics, factors and workloads, followed by the different simulations and a summary where the obtained results are evaluated and discussed.

In the subsequent chapter, Chapter 5, we present our design of a caching algorithm for use with a P2P-based architecture, which we call Relevance Based Caching (RBC). The chapter starts by explaining the general ideas underlying our concept, then we give a small example along with further description and cost prediction. The chapter is concluded by a short summary.

Then, in Chapter 6, we provide the performance analysis and evaluation of the RBC algorithm. Here as well, we start by explaining how we implemented the algorithm, then we illustrate the metrics, factors, new workloads and the performance goal. Next, we show the results from the different simulations and conclude with a summary of the chapter.

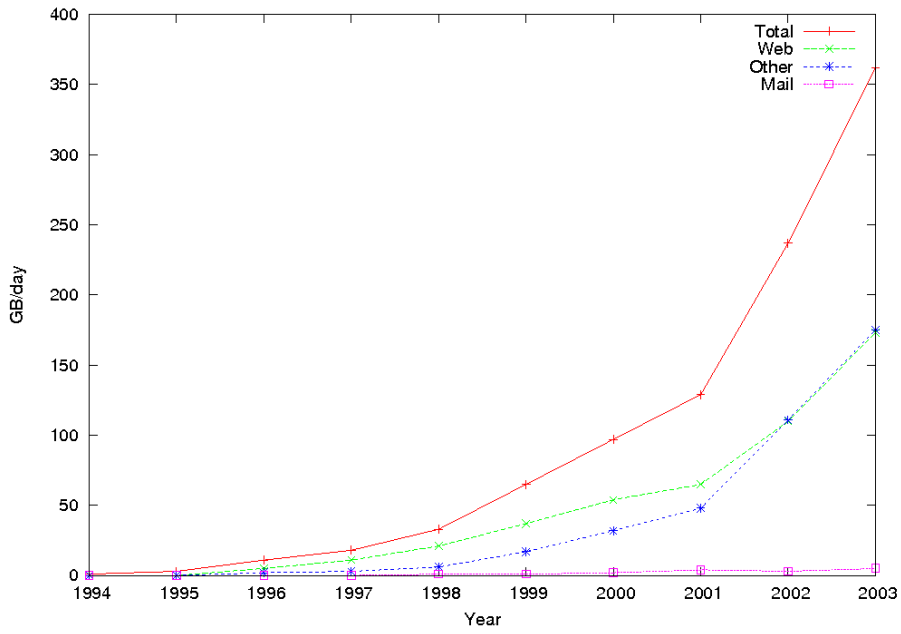
Finally, in Chapter 7 we draw some conclusions based on our results, summarize the work on the thesis, and subject it to critical assessments. We also discuss future work and contributions to this thesis.

Chapter 2

Distributing Multimedia Content

As multimedia content increases in popularity, it brings new challenges to the distribution architectures and the individual parts that compose them. Throughout this chapter we will present different aspects which are challenging in order to distribute multimedia content. We start by giving a brief description of different content available, for then to narrow our presentation to multimedia content and streaming. Then we discuss different network architectures and caching.

Figure 2.1 Network traffic increase.



In the Internet, we have different types of content. We see new types of content

emerging as the available resources grow. In Figure 2.1, we show the increase in BW usage for three different content categories. The graph is based on data gathered in an article describing the current Internet traffic growth [24]. It shows a sample of the increase in network traffic at the University of Waterloo. We use this table as an indication of how much the different traffic increases with respect to BW consumption. The categories are explained below:

- **E-mail:** *E-mail* has become an important communication channel for many people. We have seen the creation of several *E-mail* services such as *Hotmail* [3], *Gmail* [2] and many more. These service providers offer *E-mail* accounts with enough storage space for multiple GB, and covers the need for most users. From Table 2.1, we see that *E-mail* typically utilize approximately 1,5 % of the total BW. We also see that the increase is almost linear, telling us that even today, *E-mail* would only utilize about 1,5 % of the available BW.

- **World Wide Web:** The *World Wide Web* (WWW) is a system of interlinked, hypertext documents accessed via the Internet. This is traffic generated by using regular web browsers. From Table 2.1, we see that this traffic has traditionally made up for over 50 % of the total BW usage. However, we see that the *'Other'* traffic is taking over as the biggest BW consumer.

- **Other:** The other category involves both *File Sharing* and *Multimedia Streaming*. *File sharing* can be done by many means. The methods for file sharing are constantly changing. However, we see that P2P architectures have dominated during the last years. The *P2P file sharing* applications have evolved from being *Client-Server* based, like Napster [31], to having a *decentrialized* architecture like we see with BitTorrent [11]. *Streaming media* lets users watch or/and listen to multimedia content directly from a content provider. Today most television broadcasters offer services where a user can watch either live or stored content. We have also seen the immense popularity of sites like *YouTube* [6], where users can upload homemade videos for others to watch. This type of content is also included in the 'other' tab of Table 2.1. Together with *File sharing*, this is the content which increases the most today with regards to total BW usage in the Internet.

We see a large increase in the BW consumption of multimedia streaming. Multimedia is often larger and has more requirements than other content such as *HTML*. This implies new requirements on content providers. In order to look at the requirements imposed by this increase in multimedia streaming, we look at multimedia characteristics and end user behavior in the next section.

2.1 Multimedia Content

From the previous section, we see that *File sharing* and *Streaming media*, has the highest total BW increase of the content. This thesis is directed towards streaming media, and we lay our emphasis on this subject. However, most ideas and practices discussed throughout this paper, can also be applied to file sharing. To better understand what the increase in multimedia streaming implies, we illustrate multimedia characteristics next. When we write about a *stream* in this section, this refers to a stream of data from disk to an application of some sorts.

2.1.1 Multimedia Characteristics

Multimedia content is a combination of content forms such as text, audio, animation and video. Multimedia content is *continuous* and *time based*. We can say it is continuous because the content is represented as sequences of discrete values that replace each other over time. For example, when watching a movie, you actually watch an image array where each image in the array is presented continuously. It is time based, because it matters at which time an element in a stream is played. If some elements are played too early or too late, the multimedia object is no longer valid. This means the multimedia object is no longer in its original form.

	Data rate (approximate)	Sample or frame size	frequency
Telephone speech	64 Kbps	8 bits	8000/sec
CD-quality sound	1,4Mbps	16 bits	44 000/sec
Standard TV video (uncompressed)	120 Mbps	up to 640 x 480 pixels x 16 bits	24/sec
Standard TV video (MPEG-1 compressed)	1.5 Mbps	vaiable	24/sec
HDTV video (uncompressed)	1000-3000 Mbps	up to 1920 x 1080 pixels x 24 bits	24 -60/sec
HDTV video (MPEG-2 compressed)	10-30 Mbps	variable	24-60/sec

Table 2.1: Characteristics of typical multimedia streams [12]

Table 2.1 shows the data rate at which an application has to move data in order to give a correct presentation. The rates range from 64 Kbps to 3000 Mbps, and this shows us that all types of multimedia content, except telephone speech, demand

large amounts of resources. Not only do they put a heavy demand on either disk I/O or network BW[1], they are also CPU intensive as the content data has to be transformed to fit the media at which it is displayed. The heavy demand imposed by high data rates are often solved by compression methods like MPEG-1, MPEG-2 and MPEG-4 [4]. However, compression puts extra demands on the CPU as we now have to decompress the data.

2.1.2 General Multimedia Use and End User Resources

When a user watches multimedia data, indifferent to whether the content is stored locally or is streamed[2], he/she expects VCR-like behavior. For example, a user may request various interactive services such as fast forward or jump to skip uninteresting parts of a movie [21]. However, users spend most of their time in playback mode while watching/listening to multimedia content [14].

In order to watch a multimedia stream, the stream has to first be loaded into main memory. When streaming over the Internet, the data is stored directly in the main memory, while if the stream is stored on disk, the CPU has to load the stream into main memory. When the data is in the main memory, the CPU resources used to play the data, are dependent on the compression type. The CPU has to decompress each frame before it is shown. The CPU cost per second can then be expressed as the number of frames to be shown each second multiplied with the decompression cost, as shown in Formula 2.1. From Coulouris, Dollimore and Kindberg et al [12]. we have that playing a standard TV video stream requires at least 10 % of the CPU capacity of a 400 MHz PC.

$$Fps * DecompressionCost \tag{2.1}$$

This is for a scenario with a regular *Client-Server* architecture, where all clients access the same server. The clients only concern is to receive, decompress and play the data. However, with the high demand on BW from multimedia data, we get scalability issues using such an architecture. If one server has to serve high quality multimedia data to thousands of clients, the outgoing BW would soon be saturated. To solve this scalability issue, *P2P*[3] technology is now used in multiple multimedia streaming solutions such as GnuStream [20], Joost [22] and SPP [32][4]. However, with P2P the clients have to participate with their own resources.

If a user should both run playback and serve requests from other clients in a P2P fashion, the CPU resources would not impose a bottleneck. With today's CPUs,

[1] Depends on whether you are playing the content locally or from a content provider on the Web.
[2] Streaming refers here to retrieval of data from a location on the Web.
[3] P2P will be covered later in this chapter.
[4] SPP is covered later in this chapter.

which have multiple GHz, a user can easily saturate their uplink without spoiling the playback. While *disk I/O* does not impose any CPU bottleneck, it imposes a great increase in response time. If a node gets a request, it would be significantly slower to serve the request from the disk, rather than from main memory. This is due to access time differences. We measure access time to disk in ms, while we measure access time to main memory in ns [34], which is a ratio of 1:1000000. Disk access is one million times slower than access to main memory. If a client is serving other clients in a P2P fashion, and the client starts to get page faults due to too high main memory usage, the *disk I/O* can actually 'spoil' the playback. If a page fault occurs, and the disk is busy reading some multimedia content, we get a large delay in playback as the CPU has to wait for the page to get loaded into main memory from disk.

We see that multimedia content has a high BW demand, and calls for new distribution architectures such as P2P. With P2P, the clients have to participate with their own resources, and this transmits efficiency bottlenecks such as CPU and disk I/O from the server. However, the large bottleneck is disk I/O, as this limits the outgoing BW. This BW is needed by other clients to get a proper playback. To further analyze multimedia streaming, we look at the concept of streaming in the next chapter.

2.2 Streaming

In this section, we will describe what streaming is, and what requirements streaming imposes on a network architecture or application. Streaming in this section and throughout the rest of the thesis, refers to retrieving a stream of data through the network interface, and not from disk. We start by giving a brief overview of protocols used for multimedia streaming, and subsequently we address the requirements of multimedia streaming.

2.2.1 Protocols Used for Multimedia Streaming

Several protocols have been developed to meet the requirements of *multimedia streaming*, and one solution is to use *multicast* [28]. *Multicast* is a method for sending content to many different receivers at the same time. To support this, portions of the IP address space are reserved for multicast purposes, known as Class D Internet addresses. This method has never had its breakthrough, for several reasons. The most important reason is that it is very complex to deploy and manage multicast at the network layer.

Instead of multicast, a combination of *RTP, RTCP and RTSP* is often used. The *real-time transport protocol* (RTP) which is the bearer channel, and the *RTP control protocol* (RTCP), which is a separate signaling channel. Finally, we have the *real-time streaming protocol* (RTSP) which is used to select and control a stream. This protocol actually uses the *RTP* protocol to get the media stream it controls. While the *RTSP protocol* is very similar to HTTP/1.1, it applies a number of new methods and has a different protocol identifier. All the mentioned protocols often work together when a user wants streaming media.

When a user watches or listens to a media stream, he/she is oblivious to the fact that the media often consist of multiple streams. These streams have to be synchronized or else the media experience will be unsatisfactory. We can have one stream for video, one for text and one for audio to make up a presentation. In order to time these three streams so that the complexity seems transparent for the user, we need something to control layout and time. To do this the *synchronized multimedia integration language* (SMIL) is often used. It is based on XML, and allows users to create presentations including audio, video, image and animation features.

2.2.2 Streaming Requirements

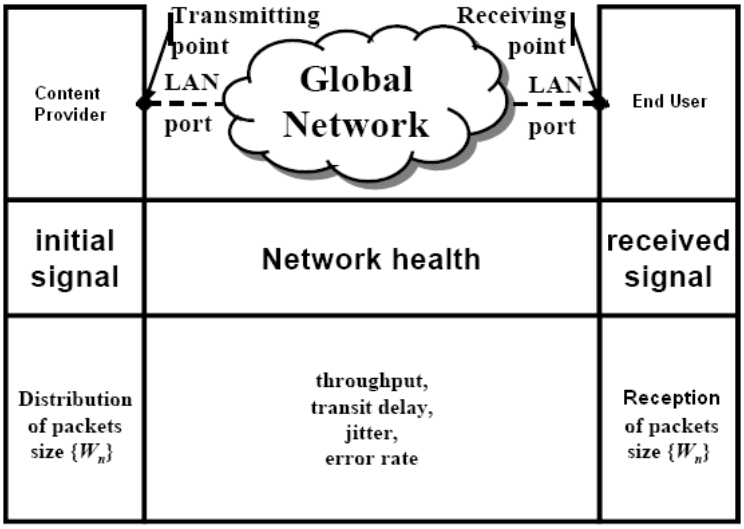
Streaming refers to multimedia data with time constraints and continuous data flow such as audio or video transmissions. Streaming has many requirements to ensure a minimum quality. It is time critical, it needs support for random access or time based access, and it needs high BW. From Table 2.1 we see the different data rate requirements for the different media formats. A tendency is that as the available end user BW increases, the content providers offer more content with higher quality and higher requirements.

We can divide the streaming requirements in two categories, i.e., *application related* requirements, and *network related* requirements. Application related requirements are requirements such as *start-up delay* and *interactivity*. The *delay* a user tolerates before leaving a system, is variable. A solution to this is to broadcast video with the knowledge of the user delay preference, which is proposed in [8]. With the amount of available BW today, the users become more demanding. New and improved methods shorten start-up delay and set new standards for fast deliverance of multimedia data. However, most users are aware of quality differences, and often accept more delay when waiting for a high quality movie. Start-up delay is often correlated with *jitter*. The faster an application can start presenting the content, the less time it has to fill a buffer. The smaller buffer an application has, the more likely it is that a user experiences *jitter*. A user expects the same *interactivity* as he/she is used to from watching a movie with their DVD player, or listening to their song with a CD player. Both the streaming application and the receiving

application need support for this, and we have protocols like *RTSP* to offer this functionality.

With application related factors, a provider often has control both at the providing end, and at the end user. However, the content provider has no control of the intermediate network between the provider and the end user. Multimedia streaming imposes requirements that have to be met by this intermediate network. We have adapted Figure 2.2 from [33] and made some adjustments to fit a scenario with a content provider and an end user. As we see in Figure 2.2, there are several factors that influence the received signal. *Throughput, transit delay, jitter* and *error rate* are all factors that have to meet the demands imposed by the users receiving streaming content.

Figure 2.2 Network factors.



- **Throughput:** The network has to offer the required BW in order to give an end user a proper playback. Without enough throughput, the end user may experience *transit delay* and *jitter*.

- **Transit Delay:** In theory, data should be transmitted instantly between one point and another. However, several factors influence the transmission, resulting in *transit delay*. First, we have the delay imposed by the medium which the data is transmitted through. For example, optical fiber limits the propagation to the speed of light. Finally, intermediate routers and other processing impose a further delay. Each encountered router uses time to examine and possibly change the header of a packet.

- **Jitter:** *Jitter* is the fluctuation of end-to-end delay from one packet to the next packet within the same packet stream. This is very annoying for the end user. In fact, an article discussing network requirements for multimedia streaming states: *'end-user perception of audiovisual quality is more sensitive to changes in jitter than to changes in delay and loss'*[33].

- **Error Rate:** The transmission of multimedia content is *time critical*. The application presenting the content to an end user needs to get the packets in correct order to give a meaningful presentation to the user. This can be made more resilient with buffering mechanisms. A buffer is filled with blocks before the playback is started. By doing this, the application has time to rearrange the blocks in the correct order, before starting playback.

In this and the previous section, we identify the characteristics of both multimedia content and streaming. We further identify important distribution requirements, both at the application layer, and at the network layer. The network layer requirements have to be solved by the intermediate network architecture, and brings us to the next section, which describes different architectures used for distributing multimedia content.

2.3 Network Architectures

Today, we have a wide range of network architectures streamlined for content distribution. It all began with basic *Client-Server* based architectures and has evolved to specialized *Content Distribution Networks* and *P2P networks*. In this section, we will address three different approaches, i.e., *Content Distribution Networks, P2P networks*, and finally a novel architecture called *SPP*.

2.3.1 Content Distribution Networks

The precise characterization of a CDN and Content are described in [25]. *"The term Content Distribution Network (CDN) implies a networked infrastructure that supports the distribution of content. Content in this context consists of encoded data or multimedia data, e.g. video, audio, documents, images, web pages, and metadata, i.e., data about data. Metadata allows identifying, finding and managing the multimedia data, and also facilitates the interpretation of the multimedia data. Content can be pre-recorded or retrieved from live sources; it can be persistent or transient data within the system. Distribution refers to the active retrieval or the active transmission of information. The infrastructure has to provide communication support and ought to contain mechanisms that facilitate effective delivery or*

increase availability of content (such as caching, replication, prefetching)".

Content Distribution Networks (CDN) are scalable and utilize the already existing bandwidth and resources of the Internet to meet multimedia streaming demands. It is an architecture of Internet structures to make delivery of content as fast as possible. Without this architecture the requests go directly to the server, or sometimes server farms, and they are often situated far from the client creating latency. CDNs however, strive to spread the content across the network with the use of *replicas* of each hosted item to make sure the requested content is close to the user. However, when a user requests data, it is not enough to use information such as *geographic locations* and *network connectivity* to choose the best replica to fetch. This can lead to a client sending a request to an overloaded server. Instead a CDN must gather *dynamic information* such as network load, load on the different replicas and other dynamically changing facts. One can look at a CDN as a network of a widely dispersed network of caches, with some small differences. The content residing on the routers are often not determined by user requests but by other algorithms, and the caches are governed by a rule that sends user requests to the best possible cache to retrieve content from, meaning the cache with the lowest retrieval time.

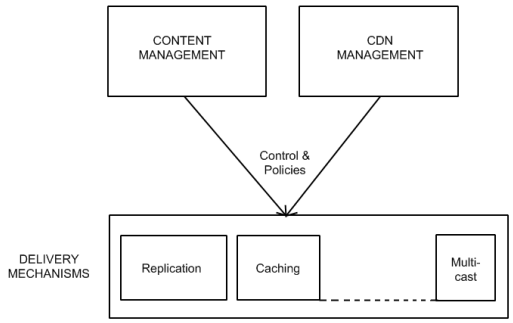
The whole idea behind CDNs is to distribute content in the network based on the metadata of the content and the load on the network nodes[5]. This can be done manually by doing system analysis. Based on this information, network administrators distribute content with the use of proxies and caches. This makes the scalability costly and inaccurate. The ongoing research today on the future CDNs aim to automate the CDN management especially with concern on network conditions [25]. If the applications which create and modify content also automatically distribute the content in the network in an efficient way based on the content's metadata, and these operations are also supported by the infrastructure, we call it a *Content Network* (CN) instead of a CDN. This is to differentiate the ongoing research from the systems focusing on distribution of content only.

The architecture of a CN is closely related to the kind of operations they support and the way content is handled. However, we generally differentiate three classes of operations, i.e., *content management operations, CDN management operations* and *delivery mechanisms*. Content management and CDN managements control the delivery mechanisms. For example, they can make a decision on where to deploy a new replica, based on meta-data from the content and the user patterns.

CDNs provide an architecture where factors such as *throughput, transit delay, jitter and error rate* can be controlled by network administrators. With the use of caches, replication and prefetching, CDNs offer a high reliability and quality. However, this

[5] A network node refers to caches, servers and other deployed infrastructure.

Figure 2.3 High Level Factors [25].



infrastructure implies a high cost. This leads to the next subsection, describing P2P networks.

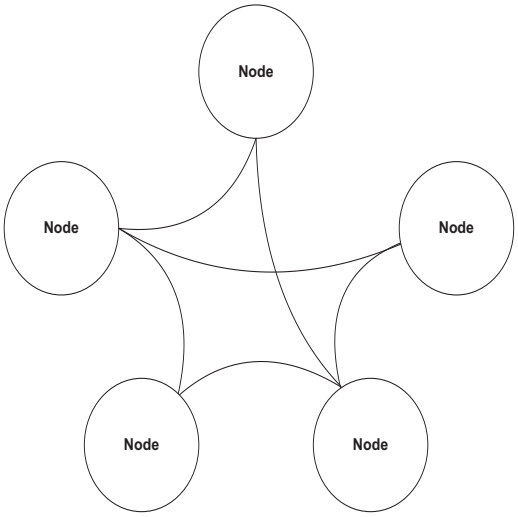
2.3.2 Peer-to-Peer

In a *peer-to-peer* (P2P) network, the resources of the end users are utilized. This means that every user of the network contributes, ideally. A P2P network does not differentiate between servers and clients, instead all users serve as both clients and servers and are referred to as *nodes*. However, in existing P2P networks today like Napster [31], certain methods such as searching, are using a Client-Server model and are prone to failure. However, completely *decentrialized* P2P networks exist as shown in Figure 2.4, where the nodes do not rely on any infrastructure. One such example is Gnutella [15].

P2P networks are very different from *Client-Server* networks. In a Client-Server network, a multimedia stream is served sequentially start to end from one location as show in Figure 2.5. In a P2P network however, a multimedia stream is served from multiple sources as shown in Figure 2.6. This implies that P2P nodes have to use some sort of buffer in order to reduce the *error rate*. Because a node is receiving the multimedia stream from multiple sources, there is no guarantee that the blocks are arriving in the correct order. The sorting has to be done by the application before the playback is started.

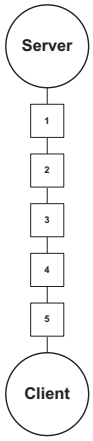
The interesting part of P2P networks is that all nodes provide resources, including bandwidth, storage space, and CPU resources. In a Client-Server model, the clients

Figure 2.4 Decentrialized P2P achitecture.



have to share a limited BW. While in a P2P network, the more connected nodes, the higher aggregate available BW we have. This is because each node is contributing with its uplink. While P2P is very scalable, it is also dependent on a large user population. If there are only a few nodes participating, the aggregated BW is too small to serve multimedia data with a decent quality. This means that P2P architectures can not guarantee a minimal *throughput*.

Figure 2.5 Client-Server request pattern.

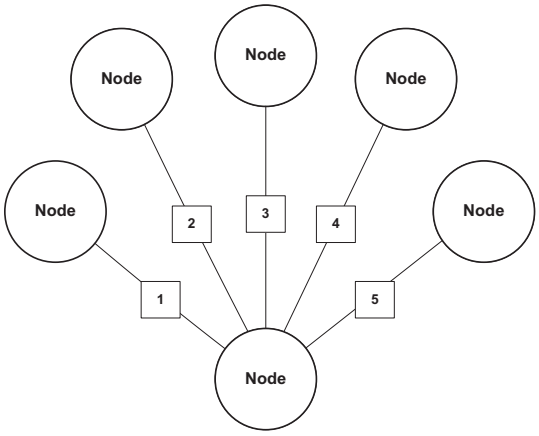


Because the nodes function as intermediate network nodes, they impose a *transit delay* and can cause *jitter*. We have seen from Section 2.1.2 that if the content is

served from disk, this is one million times slower than serving the same content from main memory. This means that in order to reduce *network delay*, P2P nodes have to keep the relevant content in main memory. With *relevant content*, we mean the content other nodes are asking for. Because of the possibility of serving content both from disk and from main memory, this fluctuation in serving time can result in *jitter*.

In a *Client-Server architecture*, a user almost always has a guarantee that the content he/she is looking for is available. The content providers have full control of the distribution, and want to keep end users happy by keeping content available. But when a *P2P architecture* is used, a lot of responsibility is handed to the users. In a purely *decentralized* P2P architecture, the client with all the blocks of the multimedia stream could suddenly disappear, making the content unavailable.

Figure 2.6 P2P request pattern.



Next, we present an example of how P2P works in practice. We chose to present this, as we later in this thesis are evaluating P2P caching algorithms. We use *BitTorrent* [27] for this example, and the basis for the example is a lecture given at the University of Oslo. In a BitTorrent session[6], we have several elements, i.e., a *web server, a static metadata file, a tracker, an original downloader*, and finally a *web browser* with BitTorrent support. The web server hosts a *torrent* file which contains the *IP address* of the tracker. The tracker, as the name implies, tracks all nodes, and needs to know at least one node with the complete file. A node who wants to download the file, accesses the web server with its web browser, and gets the IP address of the tracker. When the node gets contact with the tracker, the tracker provides the node with a list of active nodes called the peer set. This list is usually composed

[6] A session equals the distribution of a single or a set of files.

of 40 leechers and seeds[7]. All nodes regularly report their state[8] back to the tracker.

The initial file is broken into chunks, or *blocks*. To ensure the integrity of each block, the torrent file contains a SHA1 hash for each piece. Each node participating in the P2P session sends reports regularly to the tracker. This report contains information such as an unique node ID, IP, port, quantity of data uploaded and downloaded, status[9]. Nodes connect with each other using full duplex TCP. When connecting, the nodes exchange their list of blocks. Each time a peer has downloaded a block and checked its integrity, it advertises that it has a new block to its *peer set*. Two nodes communicating have two states, i.e., 'Interested' and 'Chocked'. If a node is *'Interested'*, it tells the connected node that it has a block it wants. If a node is *'Chocked'*, it simply tells the other node that it can not send data at the time.

When a node selects which block to request next, it has multiple strategies to chose between. The simplest strategy is to use a *random selection*. With this strategy, a node simply selects a block randomly among the available blocks in the node set. BitTorrent uses another strategy, called *Rarest-first*. With this strategy, the node choses the least represented missing block in the node set. This maximizes the minimum number of copies of any given block in each node set.

BitTorrent serves only five nodes in parallel to ensure efficiency. The five nodes are selected from mainly two criterions, i.e., which nodes also serves us, and which nodes offer the best download rates. BitTorrent also tries to optimize the node set, by randomly unchoking a node to see if this node offers a better service.

While P2P offers a high scalability, it offers no guarantees for *throughput, network delay, jitter or error rate*. These factors have to be satisfactory in order for the playback to have the wanted quality. However, the BW requirements of multimedia streaming mandates a P2P approach. A single server cannot serve thousands of high quality streams. It is costly to deploy network nodes throughout a network in order to provide the needed BW. In the next chapter, we will look at a hybrid P2P solution called SPP which combines both infrastructures like in a CDN, and P2P cooperation.

[7] A seed is a node with the entire file, and a leecher is a node which is still downloading the file.
[8] Percentage of download.
[9] Started, completed, stopped.

2.3.3 The SPP Architecture

As CDN networks are designed today, we are seeing a high cost at the server side, and a high network load. With a P2P architecture the load is distributed among all the nodes, thus resulting at a lower cost spread throughout the network. However, P2P networks do not have the reliability offered by CDNs. At the University of Oslo, Skevik has developed an architecture which uses both P2P and the best features of CDNs. This architecture is called *SPP* [32].

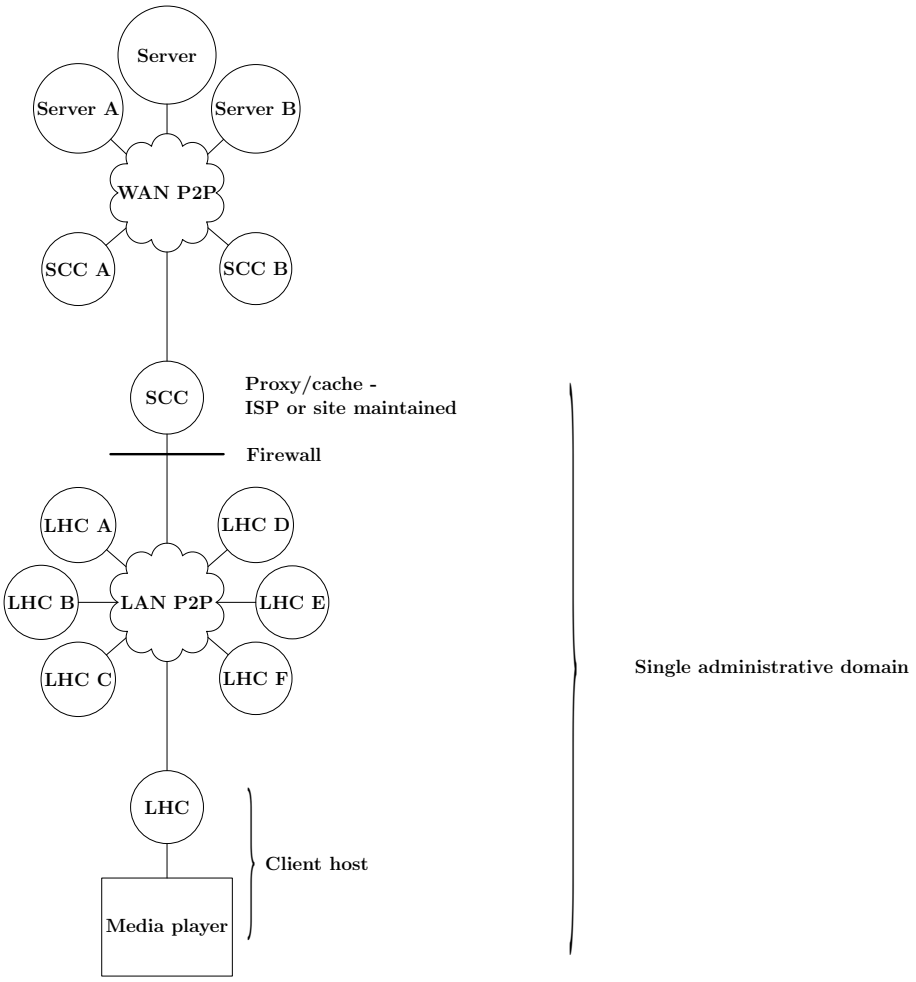
SPP makes use of caches when they are available. The caches are integrated in the architecture to provide the same services as caches have in CDNs. The architecture is a *hybrid* system that shares both some properties from WWW and BitTorrent. As we have seen earlier with the BitTorrent example, the end users have to access a web server in order to request content. However, instead of being redirected to a tracker, the content is stored locally on the server, with connections optionally made through a *proxy cache* that caches content for an entire site or ISP. SPP does not have any search function or indexing, and is only concerned with transmitting data.

The *SPP architecture* is shown in Figure 2.7. We see two P2P parts; one at either side of a firewall. The lower consists of *local host caches* (LHCs). These LHCs are located at end user computers, and can be associated to nodes in the P2P structure. The SCC, or the *Site Content Cache*, functions as a dedicated cache, and can be installed to reduce network traffic. The SCC node is permanent, adding reliability to the architecture, and it also creates clusters of close nodes. We see from the figure that the SCC is the outgoing interface from a LAN, reducing the firewall problems P2P architectures often face. The SCCs and the server also communicate in a P2P fashion.

The SPP architecture is *hierarchical*. The server and all SCC nodes maintain information about blocks located at child nodes. A typical SPP operation would be: A node that wants some content sends a request to the parent, whether it is a server, a SCC or a LHC. If we had only a P2P structure, the node with the relevant content could disappear, resulting in a content starvation. However, with static infrastructure like SCCs and servers, this is avoided. The parent replies with a set of peers, having the requested content. This is similar to a tracker in the BitTorrent architecture. If the parent is a SCC, and the SCC does not have the requested content, it retransmits the request transparently to the server, and the content is retrieved.

SPP combines the scalability of P2P networks, and the reliability and trustworthiness of CDNs. Through the use of infrastructure and BW of participating nodes, it can guarantee the needed BW with both few and many participating nodes. However, issues with *network delay, jitter and error rate* still have to be solved on the

Figure 2.7 SPP architecture [32].



nodes in order to make them a qualified contributor to the network. We see from Section 2.1.2 that the bottleneck and the fluctuating factor that can cause network delay and jitter, is the disk I/O of nodes. The nodes have to keep the relevant data in the memory in order to give the highest and most stable outgoing BW as possible. This brings us to the next section, which is concerned with caching.

2.4 Caching

A cache can be looked at as a temporary storage space and is used for many different purposes. In this master thesis, we differentiate between *Web caching* and *Memory caching*. The idea of Web caching is to replicate and store content closer

to the users, which reduces BW usage, server load and network latency. A Web cache is often dedicated to storing copies of content passing through. However, to further increase the BW and reduce network latency, disk I/O has to be avoided in these Web caches. *Memory caching* keeps relevant content in main memory while storing irrelevant content on disk. If a network architecture like P2P is used, each node also has to function as a Web cache. To increase the aggregate BW and decrease the network latency, the nodes have to perform efficient memory caching. We start this section by looking at caching techniques used by end users to improve their multimedia streaming experience. Then, we look at memory caching and the implications of P2P access patterns.

2.4.1 Caching Techniques for Streaming Media

While the nodes are functioning as Web caches, they also have to present a decent playback to an end user. We will describe three techniques used to make multimedia streaming seem more fluent. All these techniques require some form for caching either at the client or at the server. These techniques are not cache replacement techniques, however they mandate what, when and where to cache. The techniques are *audio/video smoothing, fast prefix transfer*, and finally *dynamic caching* [18].

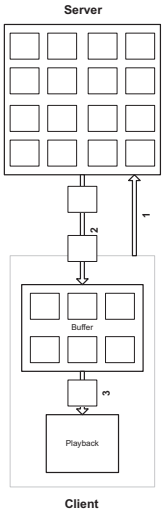
- **Audio/Video Smoothing:** *Audio/Video Smoothing* uses the cache to store frames locally before passing them to the application. This method is used because media content is built up by frames, and frame size differs in number of bits, and the frames need to be displayed at a constant rate. To avoid being influenced by network delay and other network related difficulties, an elastic storage space called a *buffer* is used to store frames Smooth playback is accomplished by implementing a small delay before playing, allowing the buffer to fill.
- **Fast Prefix Transfer:** With audio/video smoothing the user suffers a delay both from filling the buffer, and a small connection delay. To avoid this, the *prefix* of the media stream is distributed to caches close to the user. While the user streams the prefix of the file, the cache can ask the server to start sending the rest. This does not deal with the buffer delay however. *Fast prefix transfer* is a method to fill the client buffer faster than the frames are being played, resulting in a shorter delay.
- **Dynamic Caching:** When two clients want the same media, they can both use one stream. Let us say user A starts playback of the stream at time t, and a second user starts playback at time t+4s. Then, if a cache shared by both users can hold t+(t+4)s of the shared stream, both users can share it, thus avoiding the need for two different streams. The second user will have to

patch his stream, meaning he has to retrieve the data that already has passed either directly from the server, or from a stored prefix residing in the cache.

2.4.2 Memory Caching

We need efficient caching strategies to avoid disk access. We have an efficient caching strategy as long as most requests are served from main memory. In order to have relevant data in main memory, the caching algorithm has to predict the interaction. It has been shown that the popularity of multimedia objects are Zipf distributed [16]. This can be used to predict which multimedia objects are most relevant at a current time, and should be kept in the cache. While the popularity of multimedia objects is the same in both *Client-Server* and *P2P* architectures, the access patterns differ.

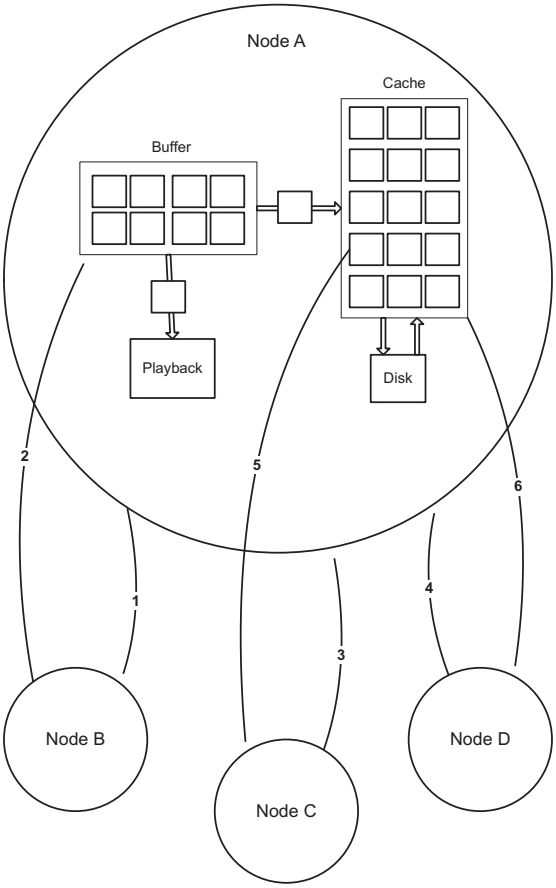
Figure 2.8 Client-Server interaction.



In a Client-Server architecture, the interaction is shown in Figure 2.8. A client would send a request to a server, as shown in step one. Then, the server would start serving the multimedia stream in a sequence of blocks. The client fills the buffer, and after the block no longer is relevant, the block is discarded. The block is no longer relevant the moment the data has been presented to the end user. The only memory caching in this scenario, is in the server. Once the server receives a request for a multimedia object, it knows that the requesting client wants a sequential stream of blocks.

In a P2P architecture however, caching has to be done at each node. Figure 2.9 shows the interaction between four nodes. In step one, Node A requests a block

Figure 2.9 P2P interaction.



from Node B, and Node B sends the requested block back to Node A (step two). This is equal to the *Client-Server* interaction as long as Node A only requests blocks from Node B. However, in a normal *P2P* interaction, Node A would request blocks from multiple nodes. This implies that Node A does not get the blocks sequentially, and would probably need to sort the blocks in a buffer before playback. While clients in a Client-Server architecture discard irrelevant blocks, nodes in a P2P architecture have to store them in order to serve other nodes. In step three and four, Node C and D are requesting one block each. This being the case, both blocks were cached at Node A (step five and six), resulting in faster retrieval time. However, Node A can not predict the access pattern from either Node C or D from this request alone. For example, in a Client-Server architecture, a server knows that after serving a client block one, it will serve block two. In a P2P architecture however, a node can be asked for any block it has previously downloaded. This makes it much harder to predict which blocks to keep in the cache, and which to store on disk.

2.5 Summary

In this chapter, we look at different aspects of multimedia content distribution. As multimedia streaming increases, it implies new requirements on content providers and network architectures. With the high BW consumption of multimedia data, and poor scalability of Client-Server architectures, CDN and P2P hybrids have been proposed. However, to meet streaming requirements such as throughput, disk I/O in P2P nodes has to be avoided. This is done with caching. We need caching algorithms that make caching as effective as possible. With effective we mean a behavior that is compliant with the access pattern in such a way that we minimize the number of data transfers from disk to main memory. Caching in P2P architectures infers new challenges as the access pattern on the serving nodes is very unpredictable. To better understand the implications of a different access pattern, we will in the next chapter present several Client-Server based caching strategies, in order to later implement and evaluate a selection in Chapter 4.

Chapter 3

Existing Caching Algorithms

There exist several caching algorithms developed for Client-Server access patterns. In this chapter, we present a selection of the most used algorithms, and some especially designed for streaming. We choose to divide the caching algorithms into two categories, i.e. *Block-Level caching* and *Stream-Dependant caching*.

3.1 Block-level Caching

Block-Level caching considers a possibly unrelated set of blocks, where each block is treated as an independent item. These are traditional algorithms that are still often used today. This is due to their simplicity, and because these algorithms are effective with traditional Client-Server based networks. In this section, we will discuss three traditional Block-Level Caching algorithms, and one that is especially designed for multimedia files. We also look at the algorithm complexity for *LRU* and *LFU* (with the use of O(N) notation). These two algorithms are chosen, because new P2P caching algorithms are often compared with LRU and LFU [30], and for this reason they are implemented and analyzed later in this thesis.

3.1.1 First-In First-Out Cache Replacement

First-In First-Out (FIFO) is a simple cache replacement algorithm, and it is applied in many areas. In this cache replacement algorithm, all objects are stored in a queue in main memory. When the cache is full and an object needs to be replaced, it removes the first object, and inserts the new object at the end of the queue.

FIFO is both simple and consumes little resources [9], which makes it a popular choice for simple caches. The algorithm does not account for either popularity or

access patterns, which makes it a very static algorithm. For these reasons, this algorithm suits P2P networking poorly. In a heterogeneous environment like P2P, the caching algorithm has to be able to adapt to changes in access patterns and popularity.

3.1.2 Least Frequently Used Cache Replacement

The *Least Frequently Used* (LFU) algorithm is based on the assumption that in order to get the highest cache hit ratio, we keep the most often referenced objects in the cache, while we evict the least referenced ones whenever an object needs to be replaced. In order to identify the most referenced objects, each object has a counter that is incremented each time it is requested. This means that the object whitch is least frequently requested, is always replaced. Although the LFU algorithm accounts for object popularity, it does not account for the access pattern.

LFU has two drawbacks that create problems. A new object needs some time to accumulate enough hits to avoid being replaced, and due to accumulated requests, old objects have a tendency to stay in the cache longer than they are needed. To remedy this problem, *LFU-Aging* and *LFU-DA* [26] is often used. They basically decrease the request counter if certain conditions such as a timeout or a maximum number of requests.

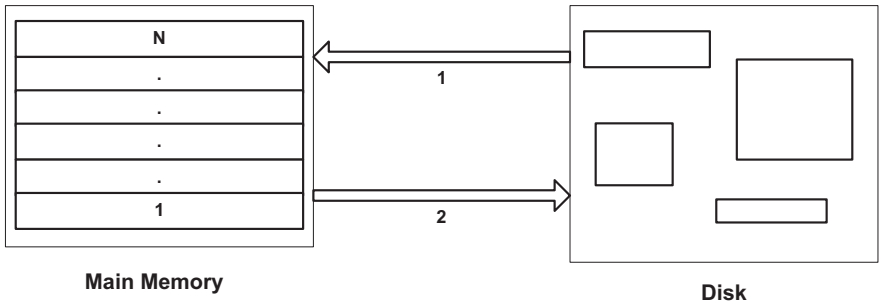
You need to have a simple data structure keeping a mapping between the objects and each reference counter. Every time a request for an object arrives, you have a worst case lookup cost of O(N). If it already is cached, we only have to increment the reference counter and we are already positioned at the correct object, so this has the cost of O(1). If the object is not currently in the cache however, we need to iterate through the cache again, and imposes a worst case cost of O(N), and evict the object which has the lowest reference counter. Finally we add the new object to the cache which has the cost of O(1). So in the worst case scenario we have a cost of O(2 * N + 1) if the object is not in the cache, and O(N + 1) best case.

3.1.3 Least Recently Used Cache Replacement

The idea behind the *Least Recently Used* (LRU) [13] algorithm is to look at locality of reference seen in request streams. It basically replaces the object which has been requested last of the objects in the cache. The basic LRU algorithm uses a *stack*. Every time a request for an object that is not currently in the cache arrives, the object is placed at the top of the stack. If the cache is full, the object at the bottom of the stack is evicted. If the object that is requested is currently in the cache, it

gets moved to the top of the stack. Every object between the top and the previous position of the requested object, are placed one position lower in the stack. The remaining objects are unaffected by this rearrangement.

Figure 3.1 LRU cache replacement.



This algorithm works best when the requested objects exhibit a high degree of temporal redundancy [9], i.e., the object is requested frequently in a specific period of time. This behavior is compliant with serving multiple multimedia streams. For example, if a client serves five other clients who are at the same point in their playback, they are likely to request the same object simultaneously.

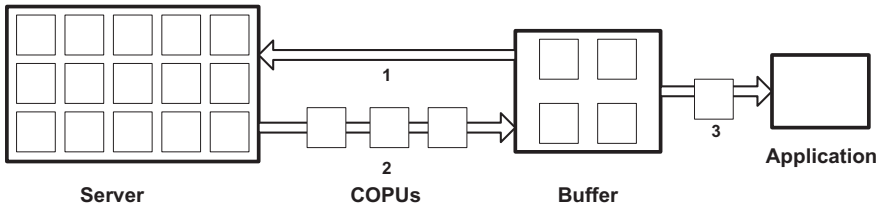
The data structure needed is one stack. When a request for an object arrives, there is a O(N) lookup cost. If the object is cached we have one operation in order to move the object, and N - 1 operations in worst case to rearrange the stack. If the object is not cached, we simply evict the object at the bottom, which is one operation. Then we place the new object at the top, which also is one operation. The worst case operational cost when the object is in the cache, is O (2N - 1), while if the object is not in the cache, it is O(N + 2).

3.1.4 Least/Most Relevant for Presentation Cache Replacement

Least/Most Relevant for Presentation (L/MRP) [23], is designed to handle one single continuous data stream. It is intended for multimedia data and is based on the fact that applications using multimedia data need a continuous time dependent supply of blocks, called *Continuous Presentation Units* (COPU). These *COPUs* have to be loaded into the cache before they are accessed by the application in order to provide the needed *Quality of Service* (QoS).

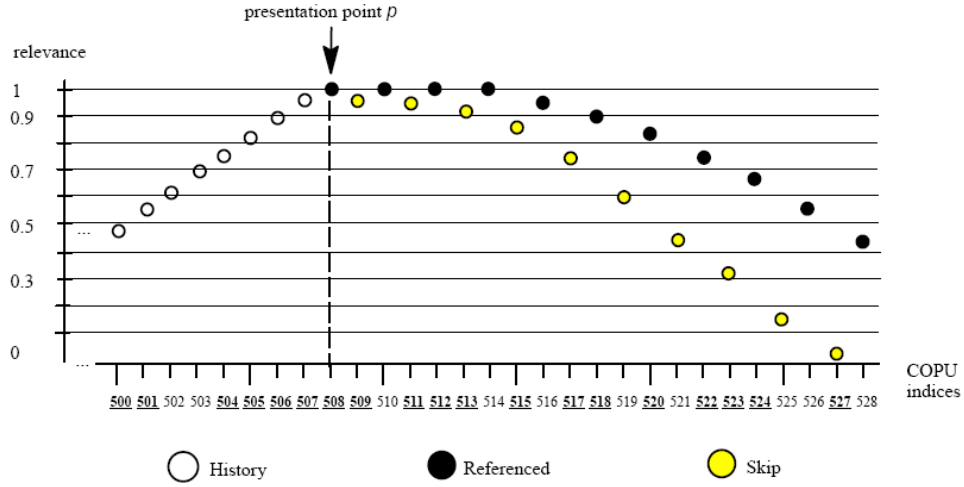
Contrary to the previous caching algorithms which are based on heuristics, this algorithm takes into account the presentation metadata. It considers so-called *inter-*

Figure 3.2 L/MRP Client - Server interaction.



action sets, which are a collection of COPUs. The COPUs are classified as different types, i.e., *rewind, referenced* or *skipped*. Each COPU is given a set of values defining which is replaced or prefetched. Thus, the algorithm aims to cache the most relevant COPUs and replace the least relevant. Because the relevance value also accounts for the number of the COPUs currently being presented, L/MRP evicts the least relevant COPU, and prefetches the most relevant COPU to the current presentation.

Figure 3.3 L/MRP



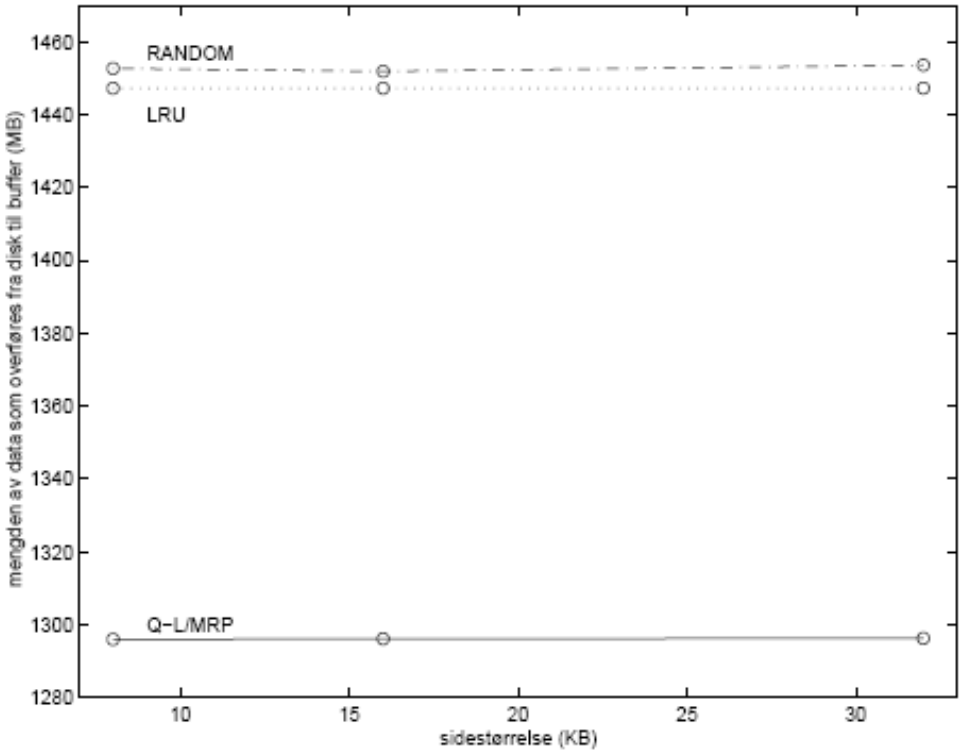
In Figure 3.3, we see one interval of COPUs. Each COPU which is marked as bold underlined, is currently cached. At this present time COPU 527, 525, 523, 528, 520 etc have the lowest relevance values. During each round[1], L/MRP chooses the COPU with the lowest relevance value and evicts it. Then it fetches the COPU with the highest reference value that is not currently cached. This guarantees that

[1] Every time one COPU passes the presentation point, i.e., is consumed.

the COPU that is to be consumed next always resides in the cache.

This algorithm is very costly with regards to CPU requirements. For each round, it has to locate the COPU with the lowest and the highest reference value. It also has to calculate new reference values for each COPU in the interval, which is costly. Another drawback is that it is designed for one single stream only. However, this algorithm gives very few disk accesses, and supports prefetching and interactivity.

Figure 3.4 Results from simulations with Q-L/MRP [17].



Q-L/MRP [17] is designed to solve this problem. This extension to L/MRP adds support for multiple data streams and QoS. We chose to present the results the authors of Q-L/MRP got from their simulations, because it is interesting to see the effect of prefetching and relevance values. The author ran simulations with one server and three clients, which had a cache size of 64 MB. Figure 3.4 shows the results. It shows the amount of data that is transferred from disk to main memory along the Y-axis, and the page size along the X-axis. The results show that with the Q-L/MRP caching strategy, we get a decrease in disk access of 10.3 % compared to the LRU algorithm. This illustrates how important it is to make use of the ac-

cess patterns and the inherent properties of multimedia streams. While this strategy is developed for a regular *Client-Server* based architecture, it still proposes ideas like *prefetching* and *relevance values*, which we take into consideration when we design a caching algorithm for P2P based streaming in Chapter 5.

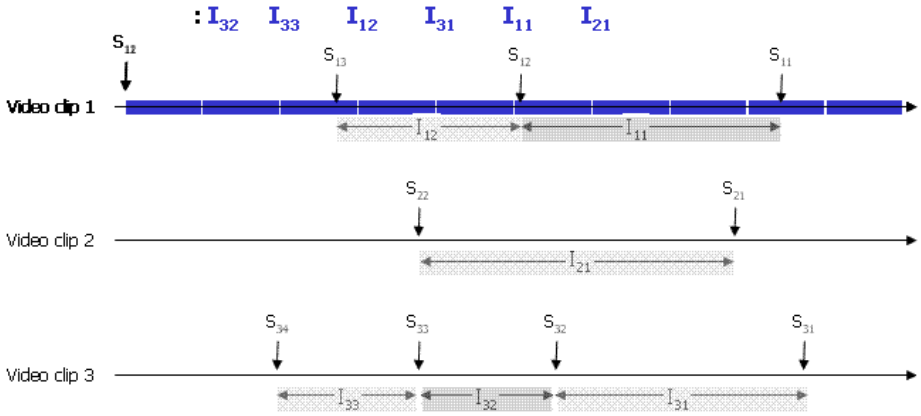
3.2 Stream-Dependant Caching

With the change of dominant content on the Internet from static web documents, towards other content types such as multimedia, we have also seen a change in caching algorithms. *Stream-Dependent caching* considers a stream object as a whole, meaning it treats all blocks in a stream in exactly the same manner.

3.2.1 Interval Caching/Generalized Interval Caching

Interval caching [37] caches entire *intervals* between requests for the same stream. This means that every request for the same stream gets served by the preceding stream. The cache size requirement is proportional with the interval length between the requests. Each interval is measured and in order to get most cache hits, the replacement policy is to cache the shortest intervals while replacing the largest.

Figure 3.5 Interval Caching



In Figure 3.5 we have three different streams with respectively three, two and four consecutive requests. In the example above, the cached entries are in the order 32, 33, 12, 31, 11, 21. If a new request for Video clip 1 arrives, creating a shorter interval than e.g., 21, and the cache is full, the largest interval is swapped with the new,

smaller one.

In order to implement Interval Caching, it is necessary to keep track of the different intervals. When a new request arrives and a new interval is created, the algorithm makes a check to see if there is enough space in the cache for the new interval. If there is enough space, it just keeps the blocks in the cache that is already being served to the client preceding it.

A problem with interval caching is when the clients request only short streams. Even if the access frequency is high on a media item, nothing is cached because the first request is finished before the next one arrives. To solve this, *Generalized Interval Caching* was proposed. Generalized Interval Caching behaves in the same manner as Interval Caching for long streams. For short streams, it keeps track of a finished stream for some time after its termination. Then, it defines the interval for the short stream as the length between the new stream and the position of the old stream if it had been a longer video object.

3.2.2 The BASIC Cache Replacement Algorithm

The *BASIC* cache replacement algorithm [7], is also a *stream-dependent algorithm*. Each round all blocks that are not being accessed are sorted in increasing order of their *futures*. Each block has a future value, which is defined to be the time when the block would be accessed next, if each client maintained its state at the beginning of the *service cycle*. For each block, one needs to calculate which client would access which block next, and this is done with the help of the clients. Each client needs to provide information about which block it needs next. When a block needs to be replaced, BASIC selects the block that would not be accessed for the longest period of time by any of the progressing clients, if each client consumed data at the specified rate from that moment on. If we have several blocks that would never be accessed by any of the current clients, BASIC selects the block with the highest offset ratio. The offset ratio is calculated by $b * i/r$, where b is the block size, i is the index, and r is the video rate.

Like L/MRP, which also looks at information concerning the presentation, BASIC generates a lot of overhead. In [7], the authors have measured BASIC to have 37 ms overhead each round compared to LRU, which uses 1-2 ms. However, BASIC reduces cache misses with approximately 30 percent compared to LRU. BASIC has the advantage of knowing exactly which blocks are accessed next. BASIC further emphasizes the importance of knowing the access pattern when choosing which blocks to cache.

3.3 Summary

This chapter examines multiple caching algorithms. We start by presenting *block-level* caching algorithms, that consider individual blocks. Then we present two *stream-dependant* caching algorithms, which consider the entire stream when making cache replacement decisions. Through results from *Q-L/MRP*, we see that to consider the behavior of the stream is beneficial towards cache hits. The *BASIC* algorithm show us the benefits of knowing the access pattern when cache replacement decisions are to be made. We have this in mind when we design our caching algorithm in Chapter 5.

However, *Block-level* caching algorithms like LRU and LFU do not take the access patterns into concern. In the next chapter, we implement these in order to see how this effects the caching efficiency.

Chapter 4

Performance Analysis and Evaluation of Existing Caching Algorithms

In this section, we describe our results of the performance of the Random Eviction, LFU and LRU algorithms. We are implementing the *Random Eviction* algorithm because it gives us a performance basis for comparing the results from simulations with LFU and LRU. There is no point in implementing and using more complicated and resource intensive algorithms, if they give a poorer efficiency. The LFU algorithm is implemented because we want to emphasize the difference of looking at a media object as a whole, and dividing it into smaller blocks. Finally, we chose to evaluate the LRU algorithm because it is well suited for multimedia streaming (shown in Chapter 3. For both LRU and LFU we want to evaluate the effect of having a P2P access pattern instead of a Client-Server access pattern. We start by describing implementation decisions. Then, we define the evaluation method, metrics, factors and workload. Finally, we analyze the results from the different simulations.

4.1 Simulation Description

In order to perform our evaluation, we start by identifying implementation decisions that affect the simulations. Then, we argue why we chose simulations for our evaluation method. Next, we introduce the metrics we use for evaluation, and the factors we use. Finally, we present the different workloads we use as traces in our trace driver simulations.

4.1.1 Implementation

When implementing the algorithms, we tried to make them as basic as possible. The implementations follow the fundamental ideas without any of the improvements proposed to them. The simulations make no use of static storage like a DBMS. We keep all gathered data in main memory at all times, which demands a high resource consumption when running these simulations. Most of the data is kept in hash tables, for a one-to-one lookup cost. However, because we keep all the data in main memory while we run our simulations, we limit the size of the workloads in terms of *active nodes* and *files*. The entire simulation environment is written in *Java*. More details about the implementations are described in Appendix C.

4.1.2 Evaluation Method

In order to evaluate the different algorithms, we use *trace driven simulations*, which are often used to tune resource management algorithms [19]. A *trace* is a time-ordered record of events on a real system [19], and these traces should be independent of the algorithms we are studying. Several advantages pointed out by Jain [19], which led to the decision of using this evaluation method, are listed below.

- **System not available:** First and maybe most important, is the fact that SPP is still not fully implemented (It is still under development).
- **Credibility:** Because the traces are generated from a P2P architecture, the results are more plausible than if we just have a random generated distribution.
- **Detailed trade-offs:** Due to a high detail level in the workload (trace), it is easy to see effects of changing factors.
- **Less randomness:** When running trace driven simulations, we get no randomness as long as we do not have other random inputs to our model. This is time conserving because we do not have to repeat a certain test several times to get the desired confidence in the results. Ideally we only have to run the test once, however to rule out abnormalities created by hardware or other outside factors, we chose to run each test at least three times.
- **Fair comparison:** Another advantage of using trace driven simulations is that we get a fair comparison of the different caching algorithms. All tests of the different algorithms use the same workload when we compare the effectiveness.

- **Similarity to the actual implementation:** Last we mention that simulation is very similar to the actual system (if not, the results are not valid) because the simulation environment is created as simular to the actual system as possible, thus when implementing, we get a fairly good feeling about how complex the different algorithms actually are, and how difficult it is to implement them on a real system.

4.1.3 Metrics

Metrics are identified by the services offered by the system, and are a set criteria to measure the performance [19]. In our simulations we are using only two metrics:

- **Block Number Closeness:** For the access pattern analysis we are looking at something we chose to call *Block Number Closeness*, referred to as BNC from here on. If we have two requests on the same file, and the block number for the two requests are close, we have a high BNC. For example, if we have two request patterns on one file. In *Pattern A*, we have requests for Block 1, 3, 5, 7, and 9, while in *Pattern B* there is requests for Block 1, 5, 9, 13 and 17. Pattern A then has a higher BNC than Pattern B.
- **Cache Hit Ratio:** For the analysis of the effectiveness of the different caching algorithms, we are using a metric we call *Cache Hit Ratio* (CHR). When we have request for a block, we either have a cache hit or a cache miss. The more percent of the total requests the cache hits constitute, the higher CHR we have.

4.1.4 Factors

We have only one factor in these simulations, the *cache size*. In a P2P network architecture we have a highly heterogeneous environment consisting of clients with different hardware. So when we run our simulations with the different workloads, we have two different node setups for each workload. When talking about different setups, we only differentiate by the amount of installed main memory because this is the only relevant hardware for our simulations.

- User PC - This setup will have a main memory size of 512 MB.
- High End - This setup will have a main memory size of 1024 MB.

We recognize the fact that we can not utilize 100 % of the main memory for caching. The caching has to coexist with a media player for playback, and other running applications as well. We ran some experiments with a multimedia streaming application. We used Windows Media Player [5] and a PC with Windows XP and 1 GB

of main memory. We streamed multiple multimedia files from [1], and measured the main memory usage with *Windows Task Manager*. Our results showed a main memory utilization of approximately 5 %. From this number, we think it is viable to use another 10 % of the total main memory for caching. By keeping the main memory usage of the caching to this percentage, a user can still run multiple applications while participating in P2P multimedia streaming. This means we will run simulations with cache sizes of 102 and 204 blocks, where each block is of the size 512 KB. This is approximately 10 % of the *User PC* and *High End* setup. We measure cache sizes in blocks, and not in MB throughout the rest of this master thesis. The cache sizes vary between 102 and 204 blocks, which respectively equal 50 and 100 MB.

4.1.5 Workloads

We have tested the implementation of the different algorithms with two different traces, called *Workload 1* and *Workload 2* hence forth. Both workloads are generated by SPP, however with Workload 1, the network topology and the access patterns are created by *PlanetLab* [10] and *MediSyn* [35]. While for Workload 2, this is done with *MediSyn* and *Inet* [36]. MediSyn generates a generalized *Zipf distribution* of requests on a video server, while Inet creates a node topology. PlanetLab is a global overlay network, which serves as the node topology for Workload 1. The video server has a selection of 1000 different media files, which ensures heterogeneity. Most nodes stream the video start to end, however some perform some degree of interactivity, like pausing playback, jumping forward or backwards in the file, or stopping the playback before watching the entire stream [21]. The peer selection is also different. In Workload 1, SPP is deployed on PlanetLab, and the peer selection is realistic. However, for Workload 2 the peer selection is random.

Workload 1 has an access pattern where all the clients are asking for one file at approximately the same time. This scenario is useful in order to evaluate performance when we have highly anticipated and popular media content. Such content creates large peaks, and we may have many simultaneous requests for the same content. In this scenario we have 35 different clients, and one file consisting of 515 blocks. Below, in Figure 4.1 we show a portion of the trace file for Workload 1.

Each line corresponds to one entry, and each item is separated with a white space. The first item is a timestamp of the form HH:MM:SS, hours - minutes - seconds. The next item is an IP address uniquely identifying the client performing the request. These two items are equal for all entries. After the IP address, we have two options:

1. Get file.mpg block N from IPADDRESS

Figure 4.1 A section of the trace file for Workload 1.

```
06:09:10 193.10.133.128: GET file.mpg block 4 from 129.240.67.75
06:09:10 198.32.154.195: GET file.mpg block 1 from 193.10.133.128
06:09:10 198.32.154.195: GET file.mpg block 2 from 129.240.67.75
06:09:11 129.240.228.138: block 0 complete, storing to disk
06:09:12 129.240.228.138: GET file.mpg block 1 from 193.10.133.128
06:09:12 129.240.228.138: block 1 complete, storing to disk
06:09:12 193.10.133.128: GET file.mpg block 5 from 129.240.67.75
06:09:12 193.10.133.128: block 3 complete, storing to disk
06:09:12 193.10.133.128: block 4 complete, storing to disk
06:09:13 129.240.228.138: GET file.mpg block 2 from 129.240.67.75
06:09:13 129.240.228.138: block 2 complete, storing to disk
06:09:13 193.10.133.128: GET file.mpg block 6 from 129.240.67.75
06:09:13 193.10.133.128: GET file.mpg block 7 from 129.240.67.75
06:09:13 193.10.133.128: block 5 complete, storing to disk
06:09:13 193.10.133.128: block 6 complete, storing to disk
06:09:14 129.240.228.138: GET file.mpg block 3 from 129.240.67.75
06:09:14 129.240.228.138: block 3 complete, storing to disk
```

2. block N complete, storing to disk

The first line is a request for a block identified by the *GET* keyword. After the GET keyword the requested file is listed, in our case file.mpg. Next, *'block N'* identifies which block is requested, and finally the *'from IPADDRESS'* identifies from which client we are requesting the block. Line two is recorded when a block has finished downloading a block. It starts with *'block N complete'*, and identifies which block that is complete, while *'storing to disk'* is a key phrase stating that the client is storing the block. In this scenario, only the server has the entire file at startup.

Workload 2 imposes a completely different scenario. Here, we have multiple files and 475 nodes. While Workload 1 gives us a scenario with one file and many concurrent requests for the same file, this scenario gives us an impression of the users access pattern as a whole. Meaning, we get the access pattern we get in a *VoD* environment. In Figure 4.2, we show a section of the trace file for Workload 2.

In this trace file, one line is also one entry, and each item is separated with a white space. The first item is a time in seconds because the start of the experiment. The first entry in the example is almost similar to the *GET* entries described previously. The difference is the timestamp and instead of IP addresses we have *DNS* names. The second entry is also almost equal to the *store entries* described previously, except for IP addresses exchanged with DNS names, and in addition, we specify which file the downloaded block belongs to. We also have some slightly different entries. In Figure 4.3, we have a scenario where none of the nodes has the block so the request is transparently routed through the SCC to the server.

The two entries in Figure 4.3 are the result when a block is retrieved through a SCC node. The SCC node *isp11902.domain* starts receiving the block, and the data are

Figure 4.2 A section of the log file for Workload 2.

```
7561 host0.isp7510.domain.: GET file1 block 325 from isp6040.domain.
7561 host0.isp15735.domain.: file57 block 305 complete, storing to disk
7561 host0.isp15735.domain.: GET file57 block 306 from 10.0.0.1
7561 isp20065.domain.: file1 block 239 complete, storing to disk
7561 host149.isp20065.domain.: file1 block 239 complete, storing to disk
7561 host149.isp20065.domain.: GET file1 block 240 from isp20065.domain.
7561 isp11902.domain.: file1 block 163 complete, storing to disk
7561 host53.isp11902.domain.: file1 block 163 complete, storing to disk
7561 host53.isp11902.domain.: GET file1 block 164 from 5545
7561 isp79.domain.: file5 block 113 complete, storing to disk
7561 host62.isp79.domain.: file5 block 113 complete, storing to disk
7561 isp79.domain.: GET file5 block 114 from 1384
7561 host62.isp79.domain.: GET file5 block 114 from isp79.domain.
7561 host4.isp15547.domain.: file27 block 60 complete, storing to disk
7561 host4.isp15547.domain.: GET file27 block 61 from 10.0.0.1
7561 isp3205.domain.: file1 block 47 complete, storing to disk
7561 host11.isp3205.domain.: file1 block 47 complete, storing to disk
7561 host11.isp3205.domain.: GET file1 block 48 from 1784
7562 host0.isp18725.domain.: file1 block 427 complete, storing to disk
7562 host0.isp18725.domain.: GET file1 block 428 from isp20530.domain.
```

Figure 4.3 Showing how the client gets a block through a SCC.

```
7557 isp11902.domain.: GET file1 block 1272 from isp6040.domain.
7557 host26.isp11902.domain.: GET file1 block 1272 from isp11902.domain.
```

being transmitted through the SCC node and to the client *host26.isp11902.domain*. All the files are initially stored at the server *10.0.0.1*.

Even though the trace file includes loggings from SCCs and the server, we emphasize on the performance of the *Local Host Caches* (LHCs). We chose this because our goal is to develop an algorithm for P2P architectures. This implies that we are not measuring performance on the Site Content Caches (SCCs), or the Server.

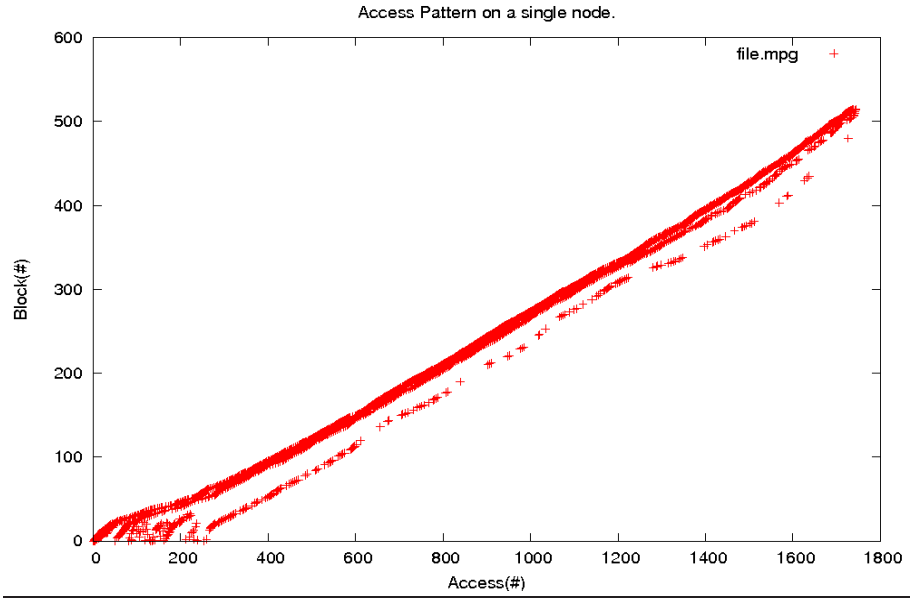
4.2 Simulations

This section is structured as follows: First, we start with identifying the access pattern from the two workloads. Next, we show the results from simulations with the *Random Eviction algorithm, the LFU algorithm* and finally *the LRU algorithm*. All the simulations are run three times to rule out abnormalities. This is enough to get a confidence in the results, because we have no random elements.

4.2.1 Access Pattern

In order to better understand the analysis of simulations with Workload 1 and Workload 2, we illustrate in Figure 4.4 and Figure 4.5, the access pattern of the clients participating. These patterns are retrieved from only one single client, however these patterns are representative for all clients.

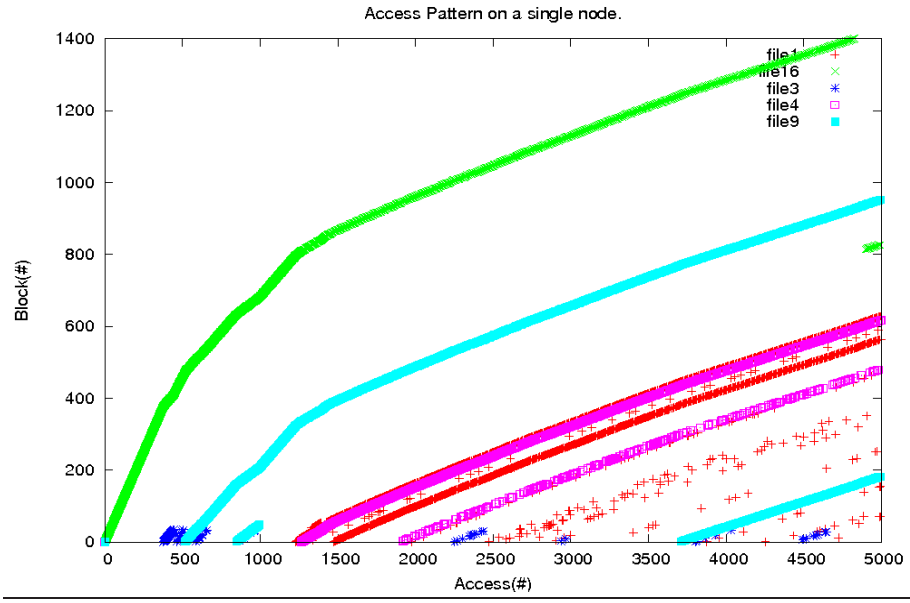
Figure 4.4 Showing an access pattern on one client from Workload 1.



From the pattern shown in Figure 4.4, we can see that we have a fairly small range of blocks that are requested by other clients at a given time. We witness in this graph, a maximum difference between requested block numbers of approximately 40. This is because with Workload 1, all clients are approximately at the same point in the playback of the video stream. For example, when *Client A* requests Block 200, *Client B* requests Block 220. This difference will always be approximately the same as long as no users stops or pauses the video playback. From the graph we are able to deduce that the client currently serves three different peers. After approximately 1400 requests, we have three different lines, representing sequential requests from three different peers.

The pattern in Figure 4.5 is very different from the one in Figure 4.4. According to the description of Workload 2, we no longer have only one file and the nodes are not becoming active at the same instant in time. This creates a different access pattern of the nodes. Figure 4.5 shows the 5000 first accesses to a single node, and as we see, the node is serving six different files to its connected peers. It shows a very low temporal redundancy. For most of the files we are able to determine that

Figure 4.5 Showing an access pattern on one client from Workload 2.



the client is only serving one peer per file. However, for *File 1*, the client is serving blocks with a wide range of block numbers, which results in a very random behavior.

We also see signs of interactivity in the graph. For example, after 900 seconds the requests suddenly stop, which indicate that either the requesting peer stopped the playback, or decided to chose another node to download from.

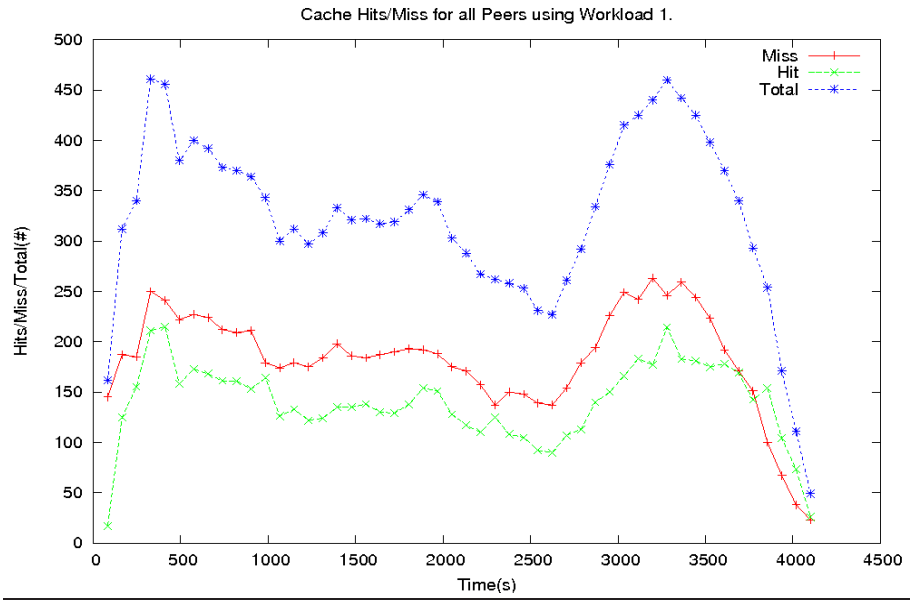
In Chapter 2, we argued that the request pattern is different in a Client-Server architecture compared to a P2P architecture. If the graph had shown the access pattern on a server in a Client-Server architecture, the access patterns on the different files would be more sequential. Meaning, the requests would be for Block N, N+1, N+2 and so forth for each file. In this graph, we see that for most of the files, the access pattern is stippled, indicating non-sequential access. However, we argue that peer requests on a client are predictable. This is because we know from the graph that after a request for one block, we never get a request for a block with a lower block number.

4.2.2 Analysis of The Random Eviction Algorithm

In this section, we analyze the Random Eviction algorithm. We start by analyzing the results from simulations with Workload 1, followed by an explaination of the

results with Workload 2.

Figure 4.6 Workload 1, cache size=102, Random Eviction.



In Figure 4.6, we see the cache hits and misses for all nodes. At startup, we have slightly more cache hits, however after some time the two curves follow each other pretty accurately. With this setup, we have only space for approximately one fifth of the file in the cache. The first LHC requesting blocks from another LHC, is always generating cache misses. For example, if *LHC B* started to request blocks from *LHC A*, none of the blocks *LHC B* asks for will be in *LHC As* the cache. However, if *LHC C* sends a request right after *LHC B* for the same blocks as *LHC B* to *LHC A*, they will still reside in *LHC As* cache. In fact, if *LHC A* had enough space to cache the entire file, *LHC C* would get served exclusively from the cache. In this scenario, we would get 50 % CHR. However, in our setup the nodes have not enough space for an entire file, but we are still getting close to 50 % CHR. We see from the access pattern for Workload 1, that a node has only a few requesting peers, and they are requesting most of the same blocks. Because the nodes in the scenario started the playback at the same time, the block requests have a high BNC. Due to this high BNC, a requesting peer is served by its preceding peer. Meaning, one peer is requesting the same blocks, only a little earlier, and it is enough space in the cache to store the number of blocks between the requests.

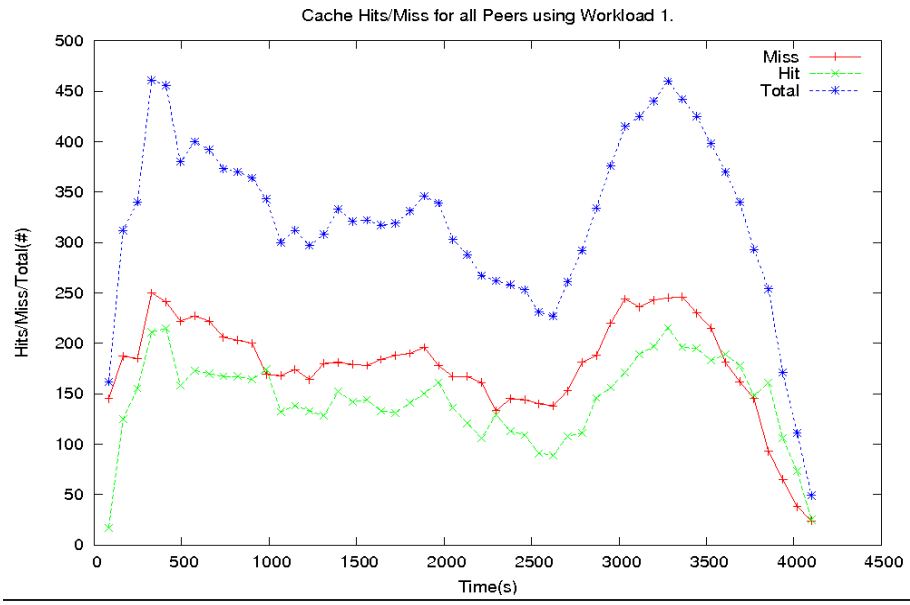
However, the Random Eviction algorithm are evicting a random block from the cache whenever there is need for a replacement. This means that there is a chance that a relevant block is replaced. This chance is expressed in Formula 4.1. The *Block Number Interval*, is the interval between the highest and lowest requested

block number. For example, a node has three requesting peers, i.e., *A, B and C*. The node has a cache size of 10 blocks, and Block 1 to 10 is already cached. A then requests Block 11, while B requests Block 7, and C requests Block 6. Due to not enough space for Block 11 in the cache, the Random Eviction algorithm evicts one of the ten blocks at random. At this point, replacing Block 6, 7, 8 or 10, results in one cache miss. From Formula 4.1 we get a 40 % chance of evicting one of these.

Based on these arguments, we see that the nodes in this scenario, serve few requesting peers. This is also shown in Figure 4.4. With more requesting peers, and if they request the same blocks, we get a higher CHR. However, because of the P2P access pattern, it is not implicit that preceeding requests are serving successive requests. The preceeding request could be for Block 2, 4, 6 and 8, while the succeeding request is for Block 1, 3, 5 and 7. For this scenario the requests are for mostly the same blocks, illustrated by the small number of requesting peers shown in Figure 4.4, . We also get a very small probability to replace a relevant block, because the *block number interval* is small compared to the cache size.

$$\frac{\text{Block Number Interval}}{\text{Cache Size}} = \text{Chance of evicting a relevant block} \qquad (4.1)$$

Figure 4.7 Workload 1, cache size=204, Random Eviction.

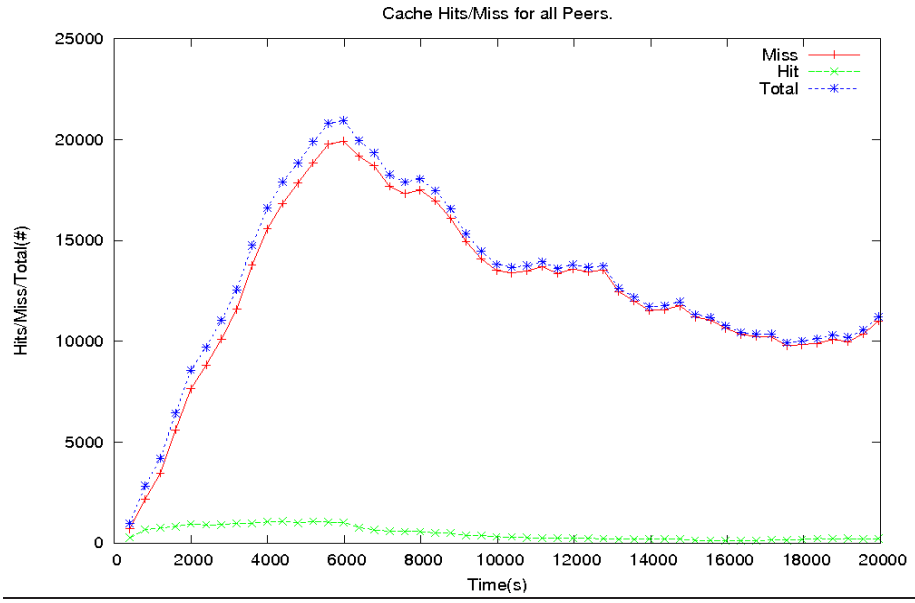


In Figure 4.7, we have a slight increase in cache size and we have enough space for 204 blocks instead of 102 blocks. Comparing the graphs, we see a slight increase in the CHR as we increase the cache size. As we increase the cache size, we also

increase the chance of keeping the relevant blocks in the cache.

Next, we evaluate the analysis from simulations with *Workload 2* using the Random eviction algorithm. From Figure 4.8 we clearly see that the CHR is very low. There are some cache hits among the first couple of thousand requests, however the curve is falling steadily towards zero. Because we chose to operate with fairly small but realistic cache sizes in our specification of the simulations, we get this effect with these kinds of access patterns. We get an effect where none of the blocks that are loaded into the cache are able to serve later requests. In our implementation of the Random Eviction algorithm we always insert a block that is requested into the cache when we have a cache miss. From Figure 4.5, we see tendencies towards that the peers are requesting different files. Actually, the clients are serving one file to one peer, in most of the cases. The impact of this is that blocks loaded into the cache are of no use for later requests for the same file, because in the meantime, the cache is filled with irrelevant blocks.

Figure 4.8 Workload 2, cache size=102, Random Eviction.



We see a clear connection between Figure 4.8 and Figure 4.5. Figure 4.8 shows us a small increase in CHR for the first 6000 requests, and Figure 4.5 is showing more peers requesting the same file in the same period.

The Random Eviction algorithm is indifferent to the access pattern. We get a high CHR for Workload 1 because the requesting peers are requesting almost the same blocks. Because of this, one peer can serve another peer. For Workload 2, we see

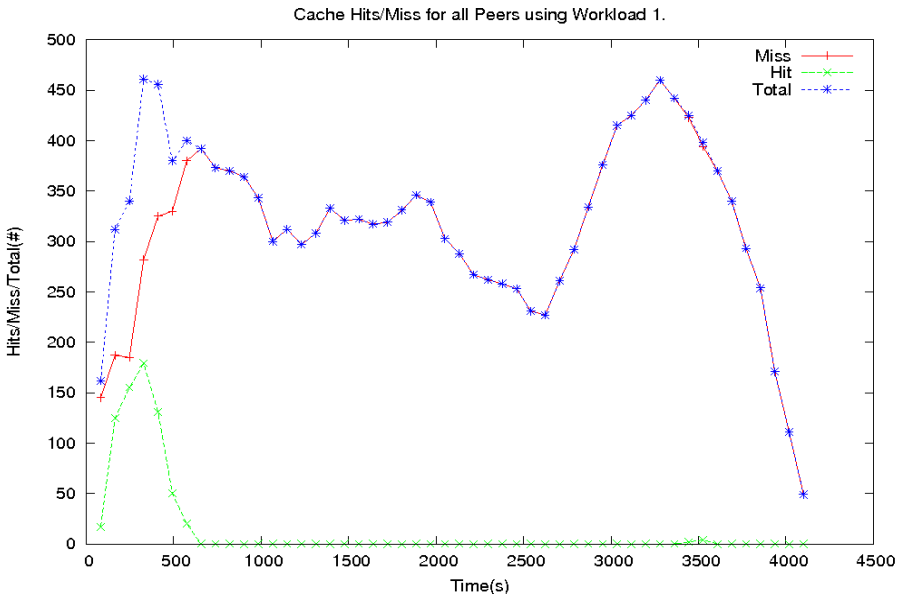
what happens when the peers are requesting different blocks. We get a poor CHR. However, we argue that the Random Eviction algorithm will get a higher CHR, the more peers are requesting the same file.

4.2.3 Analysis of the LFU algorithm

In this section, we analyze the results of our simulations using the *LFU* algorithm. We see from Figure 4.9 and Figure 4.10 that LFU does not improve the CHR run with respect to the Random eviction algorithm.

In multimedia streaming, we request a stream of small blocks, and this is very different from requesting a file as a whole. The access patterns for the files are the same, however the access pattern on the blocks constituting the file, is very different. The problem is the nature of P2P streaming. A peer requesting a file is interested in a long sequence of blocks, and each block is equally important for the peer, however the peer will not download all blocks from one provider. The peer will request blocks from a provider with a certain pattern, for example it requests Block 2, 4, 6, 8 and so forth. Two other peers may request Block 3, 5, 7 and 9. This makes these blocks more relevant for caching than the others.

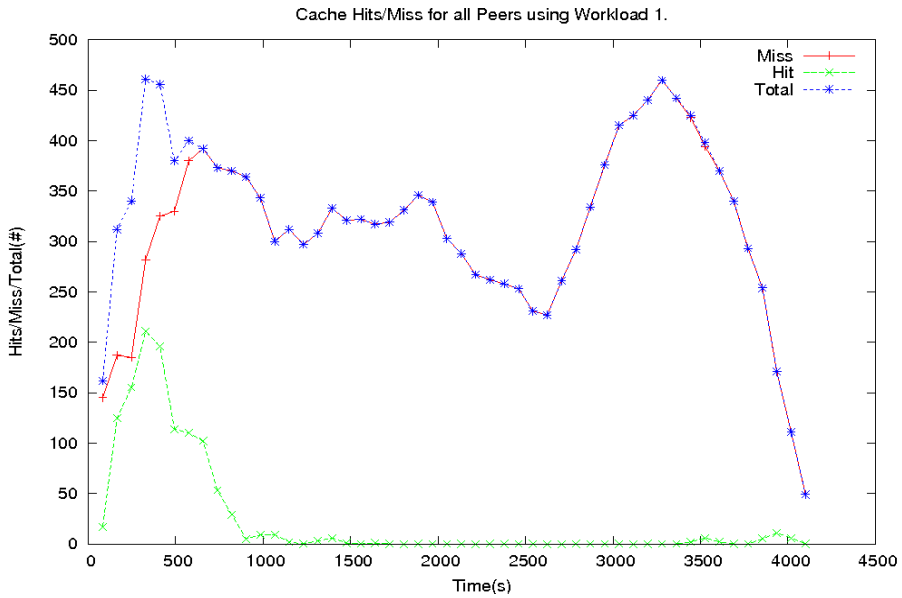
Figure 4.9 Workload 1, cache size=102, LFU.



If we look at Figure 4.9, the CHR are similar to the Random Eviction Algorithm from 0 to 400 seconds, then after 400 seconds, the CHR drops to a minimum. As

we have argumented earlier in Chapter 3, the LFU algorithm is based on the assumption that the most referenced objects at a current time also will be the most referenced objects in the future. This assumption does make sense if we look at a multimedia file from start to end. However, in this scenario we apply this assumption to each block, which basically makes parts of the multimedia file more relevant than others. In the case where a user downloads a multimedia stream from start to end, the LFU algorithm make the first N blocks most relevant (N is the cache size). These blocks remain in the cache, while the rest of the blocks will never be able to get a high enough reference value to replace them. When a new peer sends a request for the same multimedia file, it is served the N first blocks from the cache, while the remaining requests result in cache misses.

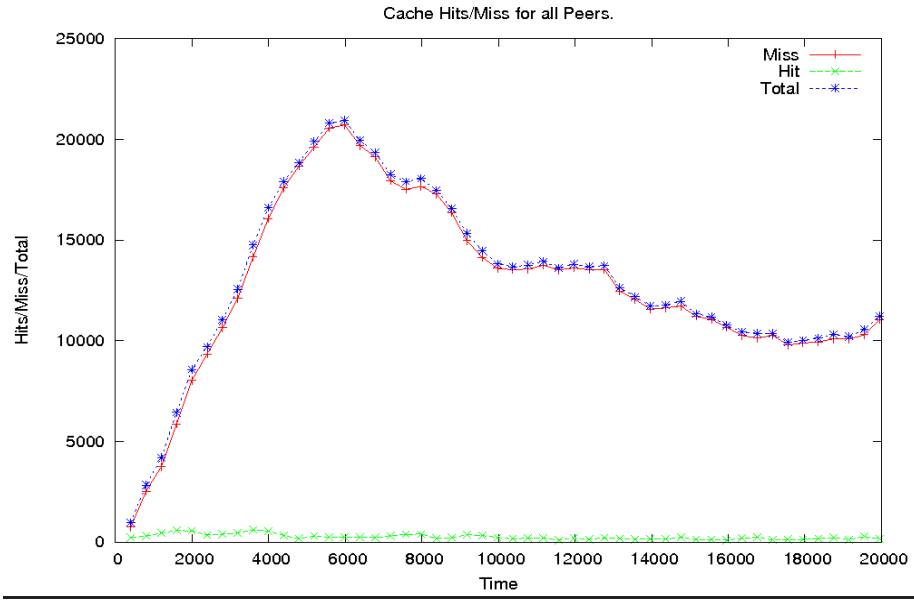
Figure 4.10 Workload 1, cache size=204, LFU.



At startup, when only a few of the nodes have entire files, these nodes get a high number of requests. These requested blocks accumulate a large amount of hits, making it impossible for future blocks to get a high enough reference counter to evict one of these. For example, if we have a cache size of 100 blocks, and a peer is requesting 300 blocks from the client. The 100 first blocks will get cached, while none of the 200 remaining blocks will enter the cache. When we increase the cache size as shown in Figure 4.10, the overall CHR is also increased. The increase is not large enough however, to make the algorithm efficient.

Next, we present the analysis from simulations with Workload 2. The difference from Workload 1, is that we have multiple files, and the nodes are now starting playback at different times. In Figure 4.11, we see a very low CHR. The argument

Figure 4.11 Workload 2, cache size=102, LFU.



that applied to Workload 1 also applies to Workload 2. LFU simply does not replace irrelevant blocks.

The LFU algorithm is designed for requests of multimedia files from start to end, and not multimedia files partitioned into smaller blocks. If a node serves one file only to each requesting peer, and blocks are swapped if they have the same relevance values, then the performance gets equal to the LRU algorithm. For example, if we have a node A and a peer P. P requests a multimedia file from A. The first N blocks are inserted into As cache. Then, when the request for (N + 1)nth block arrives at A, the first block in the cache are evicted. This block is also the Least Recently Used block.

LFU is not well suited for multimedia streaming. It does not handle the partitioning of multimedia files which is done in P2P networks. We see clearly the implications of the access pattern created by doing this. The results would be the same if the requests would be either sequential, like in a Client-Server pattern, or devided, like in a P2P pattern.

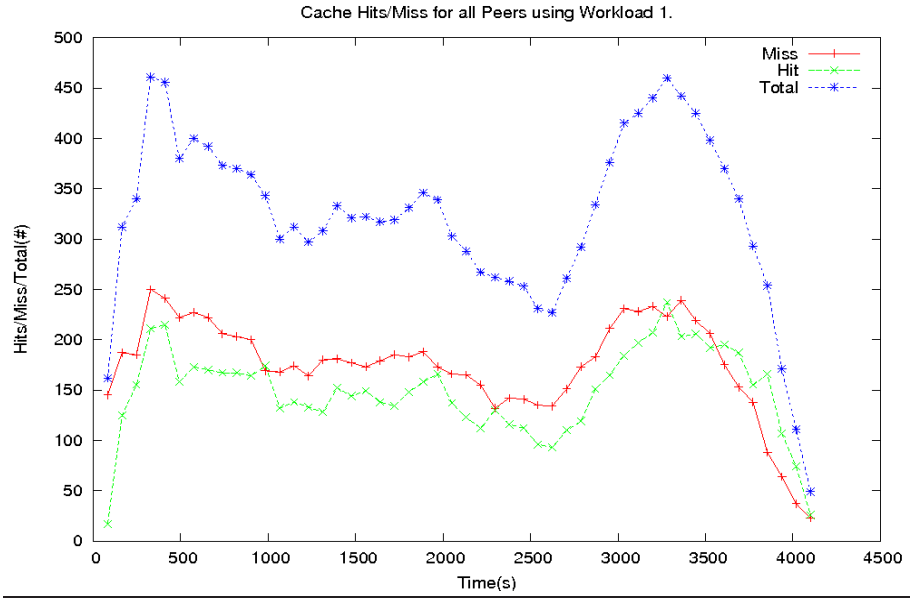
4.2.4 Analysis of the LRU algorithm

In this section, we analyze the *LRU* algorithm. Both Figure 4.12 and Figure 4.13 shows a CHR approximately close to the results from the Random Eviction al-

gorithm.

For LFU to be efficient with respect to P2P multimedia streaming, there has to be a high BNC. For example, we have a Node A with a cache size of 102 blocks, and two connected peers, P1 and P2. P1 and P2 are both requesting the same file. After P1 has requested Block 1 to 100, P2 sends a request for Block 1. This block is still in the cache, resulting in a cache hit. If we were to have four requesting peers, the BNC has to be higher. If the BNC is the same with four requesting peers, the second peer are served by the requests from the first peer, while the two last peers get cache misses.

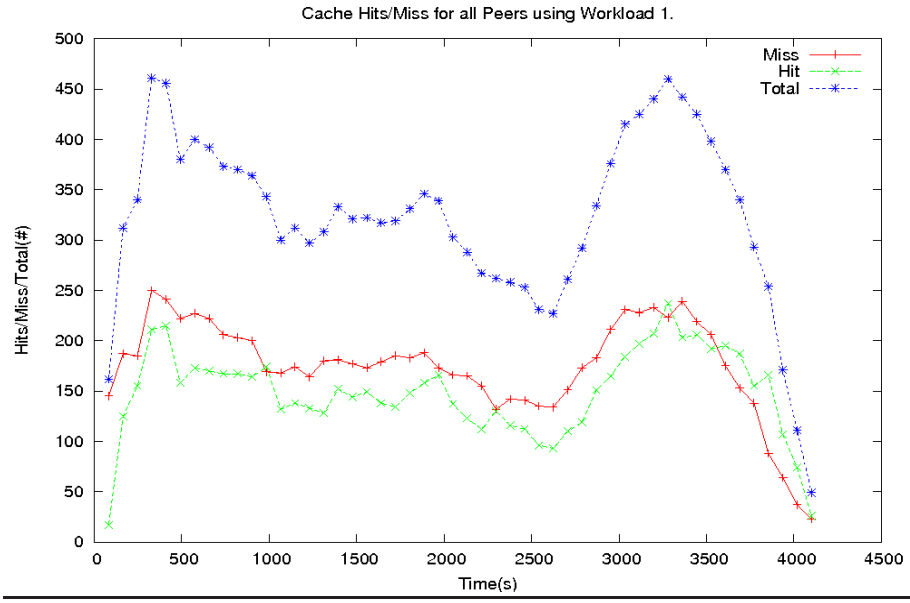
Figure 4.12 Workload 1, cache size=102, LRU.



In Figure 4.4, we see a node with only three connected peers. This graph is representative for all nodes in the SPP architecture with Workload 1. Because we have a list consisting of approximately three requesting peers for all nodes, we do not get as large a benefit from the LRU algorithm as we get with larger peer lists and the access pattern we have with Workload 1. The peer which is ahead of the other peers in the stream only gets to serve one or two other peers. In a scenario where a node has two connected peers, and the BNC is high, the first peer[1] gets only cache misses, while the second peer only gets cache hits benefiting from the fact that the first peer fills the cache with relevant blocks. This gives us a 50 % CHR, which is the case in Figure 4.12.

[1] the peer which is furthest in the multimedia stream

Figure 4.13 Workload 1, cache size=204, LRU.

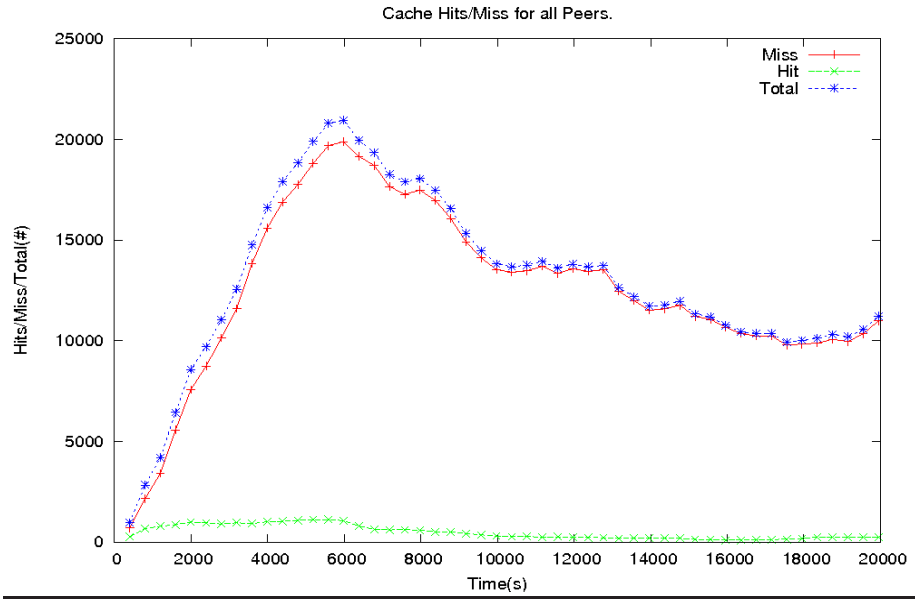


When we are increasing the cache size to 204 blocks, we are actually increasing the size with 100 %. Comparing Figure 4.13 and Figure 4.12, we do not see any difference. As we previously have stated, the clients have very few peers in their peer lists, and from the access pattern of Workload 1, we see that we have a high BNC percentage. We already have a large enough cache size in the simulation to cache all the relevant blocks with a cache size of 102 blocks, and increasing it has no effect. The LRU algorithm is actually equal to the Random Eviction algorithm, except for the probability of replacing a relevant block as shown in Formula 4.1.

Next, we provide the analysis from simulations with Workload 2. Figure 4.14 shows us a very low CHR for the first 6000 requests, then it drops towards zero. This is also similar to the results from the Random Eviction algorithm. The LRU algorithm performs better the more requesting peers a node has, and if the requests have a high BNC. From the analysis of the access pattern of Workload 2, we see that a client in most cases serves one file to one peer. This means that blocks which are cached will never serve any subsequent requests.

The LRU algorithm is very effective when a client has multiple peers wanting the same file. We are able to extend this algorithm to get almost 100 % CHR. This requires that all requests are for the same file, and that once one peer has requested a block, all subsequent requests are for blocks with the same or a lower block number. All we have to do is to prefetch a window of N blocks each time we have a request for a block number higher than any in the window, where N is the dif-

Figure 4.14 Workload 2, cache size=102, LRU.



ference between the block number requested from the peer which is furthest in the stream, and the block number requested from the peer which is backmost in the stream.

We argue that the LRU algorithm is suited for streaming, as long as we have a high BNC for each file, and many peers request the same file. The LRU algorithm also suffers because of the P2P access pattern. If the access pattern had been sequential, like in a *Client-Server* architecture, the CHR for a single file on a node could be expressed as shown in Formula 4.2. However, because of the P2P access pattern, there is no guarantee that a preceeding peer would serve a successive peer.

$$\frac{\text{Number of requesting peers} - 1}{\text{Number of requesting peers}} \tag{4.2}$$

4.3 Summary

In this chapter, we examine three different caching algorithms. First we look at the access pattern on the nodes participating in the simulation. Then we examine the three algorithms more thoroughly.

From analysis of the three implemented algorithms, we are able to identify three characteristics of P2P multimedia streaming which the algorithms do not handle:

- P2P access pattern.
- Object segmentation.
- Low temporal redundancy.

We show that the *P2P access pattern* causes problems for the implemented algorithms. Although, Random Eviction and the LRU algorithm do work with Workload 1. However, because of the P2P access pattern, the CHR is not optimal. Further, all three algorithms rely on a high degree of *temporal redundancy*. We see that with Workload 2, where the temporal redundancy is very low, the algorithms perform poorly. The LFU algorithm does not work with multimedia streaming because of the *object segmentation*. However, object segmentation is also performed in *Client-Server* architectures [18].

None of these algorithms are really taking advantage of the characteristics that multimedia streaming are offering. Characteristics like sequential access make it possible to predict which blocks are going to be requested in the future. In the next chapter, we will design a new caching algorithm which handle the P2P access pattern, object segmentation and sequential access.

Chapter 5

Design of a P2P Multimedia Streaming Caching Algorithm.

Chapter 4 shows that standard cache replacement algorithms are not well suited for use in P2P multimedia streaming. Although some of the algorithms are simple to implement and may be improved with only minor adjustments, we will in this chapter design an algorithm that greatly improves the CHR while not being overly expensive. The algorithm is designed for a SPP environment, however we design it to be as general as possible for use with P2P streaming. We think that no heuristics can be made when dealing with a highly heterogeneous and random environment such as a P2P architecture, so we focus on using probabilities. We propose an algorithm called *Relevance Based Caching* (RBC). This chapter starts by presenting an illustration of the algorithm, and gives a short example. Then we show the algorithm in more detail, and provide a cost prediction. Finally, we summarize the chapter.

5.1 Design Objectives

From observations made in Chapter 2 and 4, we identified three difficulties associated with P2P multimedia streaming, i.e., *low temporal redundancy, object segmentation and P2P access patterns*. The main objectives of the RBC algorithm are to solve these.

Low temporal redundancy implies that sessions where only one peer is requesting a file exists. Hence, we can not solely rely on situations where previous requests are serving[1] successive requests. For example, if a peer is requesting a sequence

[1] A peer is *serving* another peer if the blocks it requests are cached, and the other peer is requesting the same sequence of blocks.

of blocks and no other peers have requested the same sequence earlier, none of the requested blocks are cached. We know that multimedia streaming is sequential, and we therefore prefetch a window of succeeding blocks at each request. By doing this, we allow one peer to serve itself and we avoid problems related to *low temporal redundancy*.

The main memory size set aside for caching on the nodes is often limited compared to the size of the multimedia files being streamed. This entails that there is only enough space for a percentage of the blocks constituting the file. We therefore give each prefetched block a relevance value which consists of the popularity of the block, the P2P access pattern, and the popularity of the file as a whole. These relevance values constitute a total relevance value, which decides which blocks in the window are replacing blocks in the cache. RBC is then accounting for *object segmentation*.

We have shown in Chapter 2 and Chapter 4 that the access pattern on multimedia files are not completely sequential in P2P architectures, which it is in *Client-Server* architectures. A peer may request block N, $N + 2$, $N + 4$, instead of block N, $N + 1$, $N + 2$ etc. Consequently, there is no point in prefetching every block in the window because they are not equally relevant. In considering the *P2P access pattern*, the RBC algorithm is able to use the cache as efficient as possible, as it caches only the most relevant blocks.

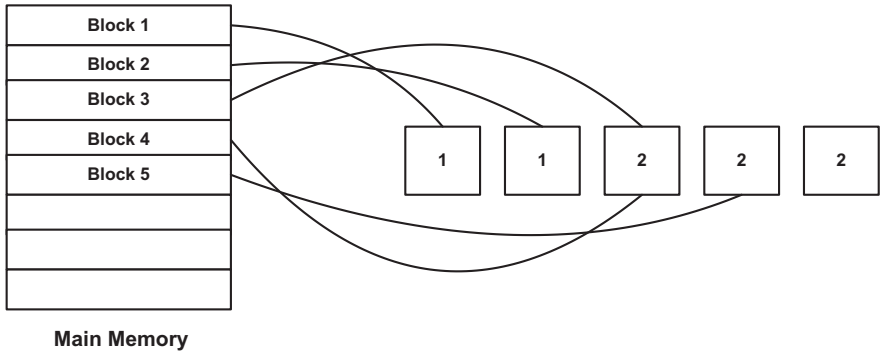
5.2 General Design

In this section, we give the description of the RBC algorithm. It covers the basic ideas, while the next chapter gives a more detailed presentation.

Each time a request arrives, we prefetch a window with W blocks where W is the *window size*. At the first request, W is set to a default value. Each time we have a request for a block that is not in the window, we increase the value W by one. For example, assuming a worst case scenario, where a *Client A* has five peers in its peer list, and all offer the same download speeds. When *Client A* starts watching the stream, it requests Block 1 to 5 from respectively Peer 1 to 5. Peer 1 then has to serve every 5th block from the media stream, creating the largest window size possible for the client.

Each block in the window gets a relevance value R indicating the block popularity. When we have excessive space in the cache, all the blocks in the window are prefetched and the relevance value R is increased with one for each block. In Fig-

Figure 5.1 Relevance values connected to blocks in the cache.



ure 5.1, we see a section of the cache and a section of the data structure keeping the relevance values for the different blocks. In this example, we have an initial value *W* of five. The first request is for *Block 0*, and Blocks 1 to 5 is cached and given a relevance value *R* of one. Subsequently, we get a new request from another node which wants *block two*, and we prefetch the window and update their corresponding relevance values. Blocks 6 and 7 are prefetched because they are not in the cache, and gets a relevance value *R* of one since they are only referenced once. The relevance value *R* of Blocks 3, 4 and 5 is now increased by one since they now are referenced twice. With twice we mean from two different sessions. We chose to differentiate by sessions and not by nodes in order to leave the option to watch more than one video stream open.

If we do not have enough space in the cache for the entire window, we will compare the relevance value *R* of the blocks in the window with the relevance value *R* of the blocks in the cache. For each block in the window, we compare the relevance value *R* with the lowest *R* of all the blocks in the cache. If a block in the window has a larger relevance value *R*, we replace the least relevant block in the cache with that block. For this to work, we need to keep the state of the blocks that are being swapped to disk. If we do not keep state, we will get an effect where we keep irrelevant blocks in the cache. For example, if the blocks in the cache are referenced twice giving them a relevance value *R* of two, then we never would replace any of these blocks, because the blocks getting prefetched never get a relevance value *R* larger than one. This is not entirely true since we have a timeout value associated with each block. However, it may cause an effect where the cache is filled with blocks, and no new blocks will get prefetched until the old blocks in the cache start to time out.

The relevance value *R* will be calculated as stated above, however to further optimize for P2P streaming, we expand the notion of the relevance value. The total

relevance value is calculated as shown in Formula 5.1. It shows the relevance value R_r which is discussed above, plus a global relevance value R_g, and a relevance value R_a.

$$R_{total} = (R_r + R_g + R_a) \tag{5.1}$$

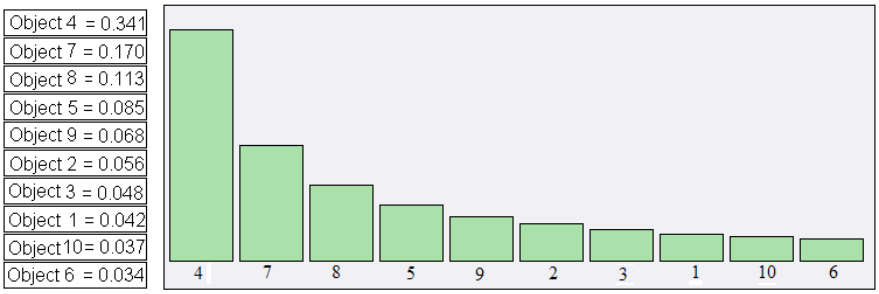
Because streaming media is time dependent, we also propose to add a *time aspect* T to the relevance value. We can assume that after a given time, a block is not valid for presentation any more, or the user has paused the presentation. This results in the final relevance Formula 5.2.

$$R_{total} = (R_r + R_g + R_a) * T \tag{5.2}$$

R_g, is a value based upon popularity of the media object as a whole, and not the individual blocks. The popularity of media objects in *Video on Demand* (VoD) servers is *Zipf distributed* [16], and we make use of this formula to create a precedence for blocks of popular media objects over blocks which are less popular. We show the Zipf formula below:

$$f(k;s,N) = \frac{1/k^s}{\Sigma_{n=1}^{N} 1/n^s} \tag{5.3}$$

Figure 5.2 A sorted list with the relevance values R_g

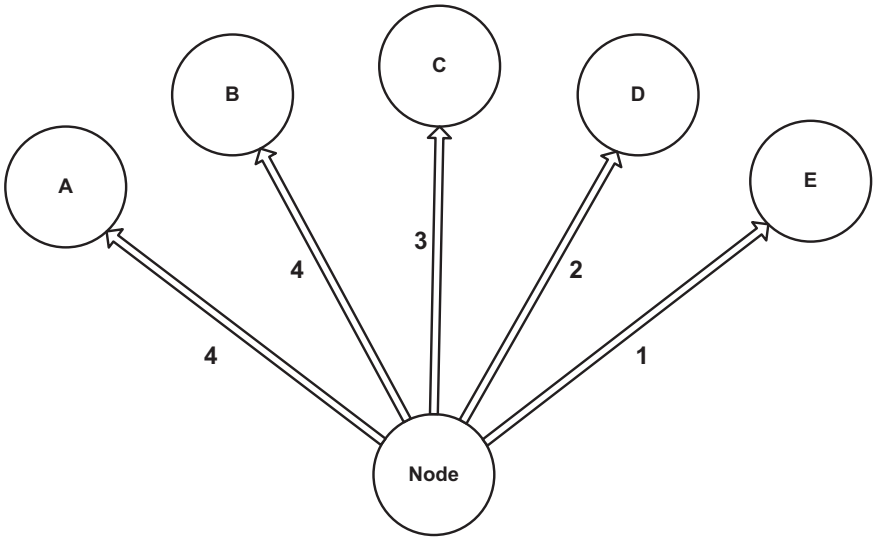


N is the number of elements, k is their rank, and s is the exponent characterizing the distribution. We chose to use one as the exponent to characterize a *standard Zipf distribution*. R_g is a value defining a relevance value for the media object as a whole. We chose to always store the media objects in a list sorted by the R_g value. This list will be small, because a client is able to serve a limited amount of media objects while having a decent playback quality. The rank will be calculated in a *LFU fashion* with the object with the most hits getting the highest *rank*. The ranks will be from one to N where N is the number of media objects and one is the highest rank. In order to differentiate between media objects with an equal number

of hits, we sort by time. This means that the object which is most recently accessed has a higher rank than other objects with an equal number of hits. In Figure 5.2, we see a list sorted by rank and containing the Zipf value or the R_g as we will use it. Our algorithm updates the Zipf values each time there is a change in the order of the sorted list.

R_a is a relevance value based upon the access pattern of the requesting peers. In Figure 5.3, we see a node with five different peers in its peer list. The numbers indicate the amount of time in seconds it takes to download one block from a peer.

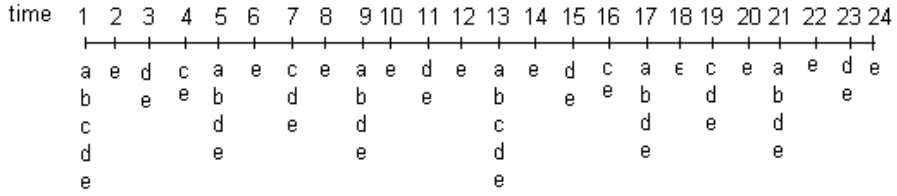
Figure 5.3 Five different peers with different connection speeds.



When downloading a media object we would ask Peer 1 for Block 1, Peer 2 for Block 2 and so forth. The nodes request patterns would look like the ones in Figure 5.4. We recognize the fact that these connection speeds are highly unstable, however we are able to use this information to create a higher precedence with some blocks in the prefetched window. From the *request pattern* in Figure 5.4 we see that Peer D would never serve two consecutive blocks. In fact, in this scenario with constant download rates, Peer D would, after having served one block, always serve the third, fourth, fifth or sixth block after the previously served one.

Our proposal is that at startup, when the first block is requested from the client, we *prefetch* a *window* where all blocks are given a relevance value R_a which is equal for all blocks. The aggregated value R_a for all the blocks in the prefetched window will always be one. When the first request arrives at the node, we store the window. Then we increase the relevance value R_a for the index of the next block that

Figure 5.4 Request pattern from five peers.



gets requested. Additionaly to increasing that value, we equally decrease all other relevance values for the rest of the indexes in the window. The effect of this is that we get a probability value for which block in the window would be accessed next.

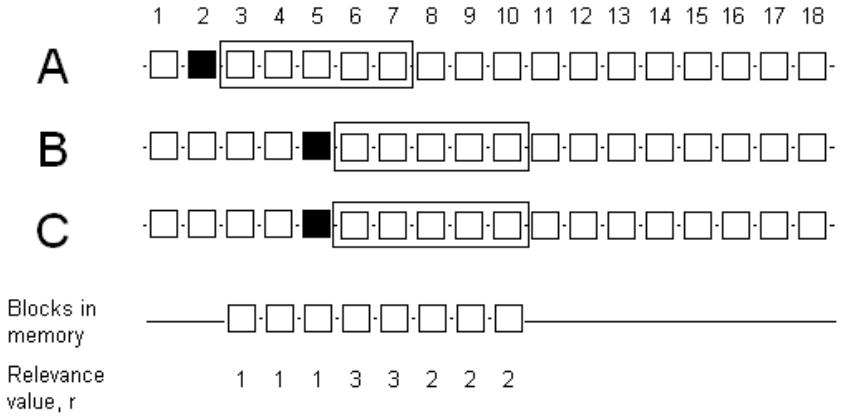
Finally we have a value T, which is a *timeout value*. After a long period T between requests from a peer, there is a high probability that a node requesting blocks has replaced a peer for some other peer with a higher upload speed. By defining a timeout value between zero and one, where one is the value for a live stream and close to zero is a value for dead streams, it is possible to give no longer relevant blocks lower relevance values.

EXAMPLE: We present an example describing the steps of the algorithm. The example focuses on one media object. In Figure 5.5, we have three different nodes; *A, B* and *C*. *A* has currently requested Block 2, while *B* and *C* has recently requested Block 5.

When a request for a block arrives, we immediately try to fetch W blocks into the cache, where W is the window size. The request from *Node A* arrives first so the relevance values for these blocks are calculated first. All the blocks get a relevance value R_r of one, since they are referenced one time each. Then the request from *Node B* is calculated. Blocks 6 and 7 get a relevance value R_r of two, while Blocks 8, 9 and 10 get a relevance value R_r of one. Finally, the request from *Node C* is calculated. Blocks 6 and 7 get a relevance value R_r of three, while blocks 8, 9 and 10 get a R_r of two. In addition, the relevance values R_g and R_a, are added to R_r for each block. The difference between relevance values R_g and R_a, is that the relevance value R_a accounts for each session, while R_g accounts for the entire media object. A block then gets the relevance value shown in Formula 5.4.

$$R_{total} = (R_r + R_g + R_a) * T \tag{5.4}$$

Figure 5.5 Cache replacement in RBC.



5.3 Cost Prediction

In this section, we describe the RBC algorithm in more detail, and give a thorough cost prediction. In order to make this algorithm a proper replacement for already existing algorithms, it has to be usable on a regular user PC. We analyze each step in the algorithm, and calculate the cost in terms of O(N). Finally, we compare the cost from RBC with the cost from LFU and LRU. We calculate the costs of worst case scenarios, as we also did in Chapter 3.

So how about the costs related to CPU resources? Are there too many steps to be made for the processor to handle in a timely fashion? We must remember that the algorithm does not have exclusive access to the CPU. Next, we analyze each step in the algorithm shown in Figure 5.6 and Figure 5.7, and assign an $O(N)$ cost.

We start with *Step 3*, which occurs when we have a *new session*. A new session in this context is whenever a client is asking for a file it has not previously asked for. A client can already have an ongoing session, and if it is asking for another file, this is a different session. So we have to create a new window structure which is a list of numbers containing the relevance value R_a and insert it into the $SessionList$. This is two steps, with the cost of $O(2)$.

Next is *Step 4 and 5*. If the new session is for a file that no other peers are accessing, we have to create a new entry in the $FileList$. We also have to sort the $FileList$ in order to calculate the R_g (Zipf) value. The cost of inserting a new item into the right index into an already sorted list is at worst case $O(N)$, where N is the num-

Figure 5.6 Detailed description of the RBC algorithm.

```
New request                                                                    1
If new session                                                                 2
    Create window structure                                                    3
    If request for not previously asked for file                               4
        Insert file sorted into FileList                                       5
    Else if the file exists in FileList                                        6
        Increment the reference counter for the file                           7
        Sort the list                                                          8
Prefetch window of size W                                                      9
    For each block_a to be fetched                                             10
        Calculate relevance R_total                                            11
        If free space in the cache                                             12
            Insert into Cache                                                  13
        Else                                                                   14
            Find the block_b in the Cache with lowest R_total                  15
            If R_total of block_b is smaller than R_total of block_a           16
                Swap blocks                                                    17
            Else                                                               18
                Do nothing                                                     19
```

Figure 5.7 Steps to calculate R_{total}.

```
Increase R_a for the block in the window structure                             20
Add R_r, R_a and R_g                                                           21
Increment R_r                                                                  22
```

ber of items in the list. The worst case scenario occurs whenever all the files in the $FileList$ are being accessed by different clients. This gives the cost $O(N+2)$, where 2 is the cost of creating an entry and setting the reference value to one.

When the file already exists (*Step 6, 7 and 8*), we have to increment the reference counter for the correct entry, which is one step and has the cost of $O(1)$. In addition, we have to place the updated entry at the right index to keep the list sorted. This also has a worst case cost of $O(N)$ in the case where the updated entry where referenced one time less than all the other entries, making the algorithm iterate through each entry before finding the correct index. This results in $O(N+1)$.

Now, we move on to the part of the algorithm that always happens when we have a request. In *Step 9*, we create a list of the blocks to be fetched, this has a cost of $O(1)$. *Step 10* has the cost of $O(W)$, where W is the size of the window. *Step 11* is a bit more complicated and is compound of *Step 20, 21* and *22*. *Step 20* has the cost of $O(W)$, since each time we increase the R_a value for one block, we have to decrease R_a for all the other blocks in the window. *Step 21* and *22* has the cost of $O(1)$. This sums up to a cost of $O(2W+2)$ for calculating R_{total} for all the blocks.

Step 12 and *13* will have the cost of $O(2)$, since there is one step to check for free space in the cache and one for inserting the block into the $MainMemoryList$.

The six final steps (14 to 19) cover the case where the cache is full, and we need to do a swap. *Step 15* where we find the block with the lowest total relevance value has the cost of $O(N)$, where N is the total number cached blocks. *Step 16* decides whether we should do a swap or not and cost $O(1)$, while the swap also costs $O(2)$. This constitutes a total cost of $O(N+2)$.

When adding the costs we only calculate the worst case costs. This means that when we have an *If/Else* we chose the cost that is highest. This results in the final cost as shown in Formula 5.5:

$$\begin{aligned} & O(2) + O(N_f + 2) + O(2W + 2) + O(N_m + 2) \\ & = \\ & O(N_m + N_f + 2W + 8) \end{aligned} \tag{5.5}$$

N_m is the number of entries in the $MainMemoryList$ while N_f is the number of entries in the $FileList$. Below, in Table 5.1, we see a cost comparison of the four implemented algorithms. We see that RBC, is more complex and resource consuming than the others. However, we argue that with the CPU resources we have today,

the cost for RBC is acceptable.

Algorithms	O(N)
RANDOM	O(2)
LFU	O(2 * N + 1)
LRU	O(2N - 1)
RBC	O(N_m + N_f + 2W + 8)

Table 5.1: Cost Analysis of RANDOM, LFU, LRU and RBC.

5.4 Summary

In this section, we propose an algorithm called *Relevance Based Caching* (RBC). It uses prefetching to benefit from the sequential access imposed from multimedia streaming, and identifies the P2P access pattern to cope with the partitioning of the request sequence. At each request, it prefetches a window of N blocks. When prefetching this window, RBC gives each block a relevance value composed of the block popularity, the access pattern and the file popularity. When it has to replace a block in the cache, is replaces the block with the lowest total relevance value. This ensures that we always cache the blocks that have the highest probability of being requested. We also look at the CPU resource cost, and we show that the RBC algorithm is feasible with regards to CPU resource consumption.

In the next chapter, we will implement and evaluate the RBC algorithm. We compare it to the previously evaluated algorithms in order to see how RBC handles P2P access patterns, low temporal redundancy and object segmentation.

Chapter 6

Performance Analysis and Evaluation of the RBC Algorithm.

In this section, we present the performance evaluation of the results obtained from the simulations using the RBC algorithm. We start by illustrating implementation decisions. Then, we proceed with defining a goal for the performance, metrics, factors and the workloads. Subsequently, we explain the simulations, and finally give a summary.

6.1 Simulation Description

In Chapter 4, we argumented why we chose to use simulations for our performance analysis. We use simulations also for this evaluation for the same reasons as listed in Chapter 4.

6.1.1 Implementation

Like with the implementation of existing algorithms, we also here keep all data in main memory while running the simulations. This limits the workload size somewhat. We have followed the description of the RBC algorithm from Chapter 5, with two exceptions. We have not implemented the *timeout value T*, and the *Window Size* is static throughout the simulation run time. These choices were made due to time constraints.

6.1.2 Metrics

We use the same metrics for this evaluation as for the evaluation of the existing algorithms:

- **BNC:** For the access pattern analysis we look at something we chose to call *Block Number Closeness*, referred to as BNC from here on. If we have two requests on the same file, and the block number for the two requests are close, we have a high BNC. For example, we have two request patterns on one file. In *Pattern A*, we have requests for blocks 1, 3, 5, 7, and 9, while in *Pattern B* there is requests for blocks 1, 5, 9, 13 and 17. Pattern A then has a higher BNC than Pattern B.
- **CHR:** For the analysis of the effectiveness of the different caching algorithms, we use the *Cache Hit Ratio* metric. When we have a request for a block, we either have a cache hit or a cache miss. The larger the percentage of the total requests the cache hits constitute, the higher CHR we have.

6.1.3 Factors

Before we start explaining the evaluation, we give an overview of the parameters that affect the performance. Parameters are divided into two categories: those that will be varied during the evaluation and those that will not. The parameters to be varied are called *factors* and their values are called *levels* [19]. We vary all our parameters except *Relevance G*, to measure which parameter has the most effect with regards to CHR. The Relevance G value reflects the popularity of the multimedia file as a whole, not individual blocks. The factor is based on the *Zipf formula*, and is therefore restricted to give a maximum value of one. The factors we will use are the following:

- **Cache Size:** The *Cache Size* factor is the same as the one we used in the simulation of the existing algorithms. It defines the Cache Size in amount of blocks. One block is in all our simulations defined as 512 KB. In this evaluation we vary the Cache Size between 102 and 204 blocks, which equals approimately 50 and 100 MB respectively.
- **Relevance R:** From Chapter 5, we recall the description of Relevance R. It holds a value indicating in how many windows the block is referenced in. The default value of this factor is one. We are increasing the value to two in some simulations. When we use default settings and a block has the Relevance R value of two, another block with a Relevance R value of one are still able to evict the first block with the combination of Relevance A and G. When increasing the Relevance R value, we actually dwindle the effect

of Relevance A and G to only differentiate between blocks with equal Relevance R. For example, we have a block that is referenced in four windows. We recall Formula 5.1 from Section 5:

$$R_{total} = (R_r + R_g + R_a)$$

R_r, R_g and R_a are *Relevance R, G and A* respectively. With a default setup, the highest *Relevance G* value we are going to get is one. This, we get from the Zipf formula:

$$f(k; s, N) = \frac{1/k^s}{\Sigma_{n=1}^{N} 1/n^s}$$

To get a *Relevance G* value of one, we have only one file in the file list. The highest possible value for *Relevance A* is also one with the default setup. The sum of R_g and R_a then never exceeds two. A block that is referenced four times will get a total relevance value of $R_g * 4 + R_a + R_g$, which equals $2 * 4 + 1 + 1 = 10$. A block which is referenced five times, will always get a total relevance value higher than ten, thus it will always be cached if the choice was between this block and the former.

- **Relevance A:** *Relevance A* is the relevance value describing the access pattern. We run our simulations with *Relevance A* values of 1 and 20. When we say a *Relevance A* value of one, each block in a *Window* gets the *Relevance A* value of 1 devided by Window Size at startup. The reason for running simulations with the value one is related to the explanation of *Relevance R*. We do not want this relevance value to be too important. However, we are running simulations where the *Relevance A* value is 20 and a *Window Size* of 20, where each block gets the *Relevance A* value of one initially. This is to measure the importance of the *Relevance A* value.

- **Relevance A Increase:** This factor defines at which rate the *Relevance A* value increases. We run our simulations with *Relevance A Increase* values of 0,05 and 0,10. With these two values we hope to measure whether a quick increase or a slow increase is the most effective with regards to CHR.

- **Window Size:** The *Window Size* defines how many successive blocks we try to retrieve once we have a request. We run our simulations with Window Sizes of 10, 20 and 40 blocks. We start with 20, then try to increase the number to 40, and finally decrease to 10. This will give us an indication of which Window Sizes are the most effective with regards to CHR.

6.1.4 Workloads

We use both *Workload 1* and *Workload 2* which are described in Chapter 4. This is a choice made to make the analysis as fair as possible. In addition, we use a third workload, *Workload 3*. This workload has a maximum number of active nodes of

40, and operates over approximately 7 days. It has a total of 1143 nodes. We use this workload to verify observations made from Workload 1 and 2. Workload 3 is generated in the same fashion as Workload 2.

6.1.5 Performance Goal

The main goal is to show that this algorithm is suited for P2P multimedia streaming. We want to show that RBC handles the *P2P access pattern, object segmentation* and *low temporal redundancy*. Workload 1 is easy to predict because all nodes are requesting the same file, and start playback at the same time. The prefetching done by RBC should handle this scenario very well, and we expect high CHR values. For Workload 2 and 3, which have a much more random access pattern and thus is harder to predict, we think all results better than the existing algorithms are acceptable.

6.2 Simulations

In this section, we present the results from our simulations. First, we are defining a default setup. This default setup is not optimized. The values are set before we start the simulations and have any results. However, they reflect our assumptions about which values are giving the best performance with regards to CHR. In the default setup we have set the factors to:

- **Cache Size:** The *Cache Size* factor is set to 102 blocks. In Chapter 4 we define that each block is of the size 512 KB, and 102 blocks equals approximately 50 MB.
- **Relevance R:** *Relevance R* is set to one in the default setup. This means we increase the Relevance R value of a block by one, each time it is referenced in a Window. We chose this value because Relevance R is the main part of RBC, and is designed to function independently of *Relevance A and G*. If we only were to have Relevance R, and not Relevance A or G, it would be natural to use one as the default value.
- **Relevance A:** *Relevance A* is designed to differentiate between two blocks with an equal *Relevance R* value. The Relevance A value for a block is dependent on the *Window size*, because it is divided with this value. For example, if we have a Window Size of 20, each block gets the Relevance A value of 1 / 20, which equals 0,05. This is the startup value for each block. We chose to set the Relevance A value to one to prevent the situation shown in Formula 6.1.

$$R_r + 1 < R_r + R_a \tag{6.1}$$

We are aware that *Relevance A* has a theoretical chance of becoming one for a block, thus making the left and right side of the equation equal.

- **Relevance A Increase:** We chose to set the *Relevance A Increase* factor to 0,05. This number is low enough to make a medium paced increase in the Relevance A values.
- **Window Size:** The *Window Size* is set to 20 in the default setup. This makes up approximately 20 % of the default *Cache Size*. From the access patterns we analyzed in Section 4, we saw a tendency towards nodes getting requests from few peers. With the *Window Size* set to 20, and a *Cache Size* of 102, there is enough space to effectively retrieve the relevant blocks in a Window.

6.2.1 Results from Default Setup

In this section, we analyze the results from our simulations with the default setup which was described earlier. We start by introducing the results from Workload 1, then we are showing the results from Workload 2.

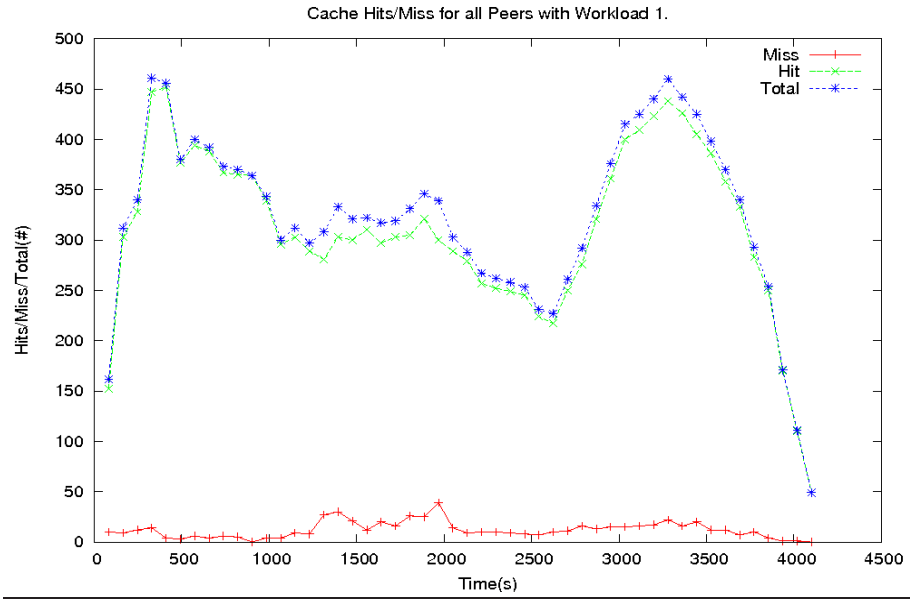
Figure 6.1 shows us a graph with time in seconds along the *X-axis*, and the cache hits/miss/total along the *Y-axis*. We have verified the workload by comparing the curve for the total hits with the curves from the previous results from simulations with the existing algorithms.

The graph shows us a very good CHR. We have close to 100 % CHR, and this shows us how well suited the RBC algorithm is for this kind of workload. The peers always request the same file, and they are requesting blocks which have block numbers close to each other. For example, two peers are requesting the same file. The first peer requests *Block 1*, and *Block 1 to N*, where N is the *Window Size*, gets swapped into the cache. Then the peer requests *Block 10*, and *Block 10 to 10+N* are retrieved. The next peer then requests *Block 1*, and because all *1 to N* blocks are already in the cache, we do not need to prefetch any of these. The request will be a cache hit, because the first peer has previously requested the block in question. In fact, we will always have cache hits for requests as long as the cache size devided by the number of requesting peers equals a sum greater than the *Window Size*, as shown in Formula 6.2.

$$CacheSize/RequestingPeers > WindowSize \tag{6.2}$$

For example, with our default setup with a *Cache Size* of 102 blocks, we can have five peers requesting blocks without getting cache misses (102 / **5** > 20). We have

Figure 6.1 Workload 1, default setup, RBC.



previously analyzed Workload 1 in Chapter 4, and we argumented that most nodes had aproximately three requesting peers. Albeit, we still get cache misses. The cache misses come from nodes with more than five requesting peers, and where the peers requests blocks with block numbers far apart. We are also getting cache misses if the peers request block numbers with an interval larger than the *Window Size* as shown in Formula 6.3. For example, a peer requests Block 1, and Block 1 to N is prefetched. Then if next the peer requests Block N+1, it results in a cache miss.

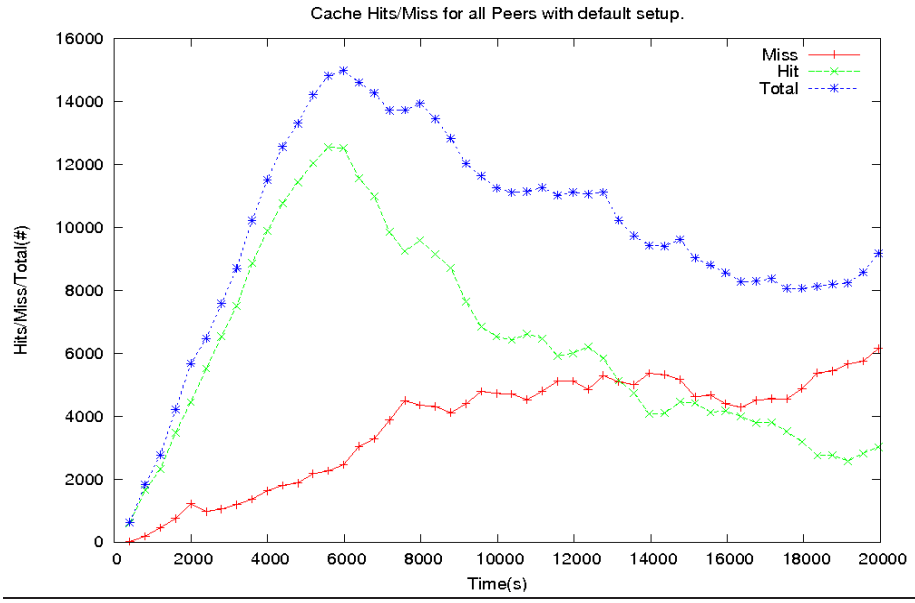
$$BlockNumberInterval > WindowSize \tag{6.3}$$

Next, we analyze the results we got from simulations with Workload 2. The graph from Figure 6.2 shows us a very good CHR the first 6000 seconds, then the CHR drops, for then to stabilize at approximately 1000 seconds. We have already in this section established two formulas describing when we can have a cache miss. We show that after 6000 seconds, the nodes have more than five requesting peers, resulting in a situation where the cache no longer can hold all the blocks in the windows to be prefetched, as shown in Formula 6.4.

$$CacheSize/RequestingPeers < WindowSize \tag{6.4}$$

An effect of having a larger peer list, is that the *block number interval* also gets larger. For example, if a node streams multimedia data from five different peers

Figure 6.2 Workload 2, default setup, RBC.



with the same download rate, it would request blocks with a block number interval of five. If the node downloads from more than 20 peers, the interval is larger than the *Window Size* which is set to 20, and this results in a lower CHR.

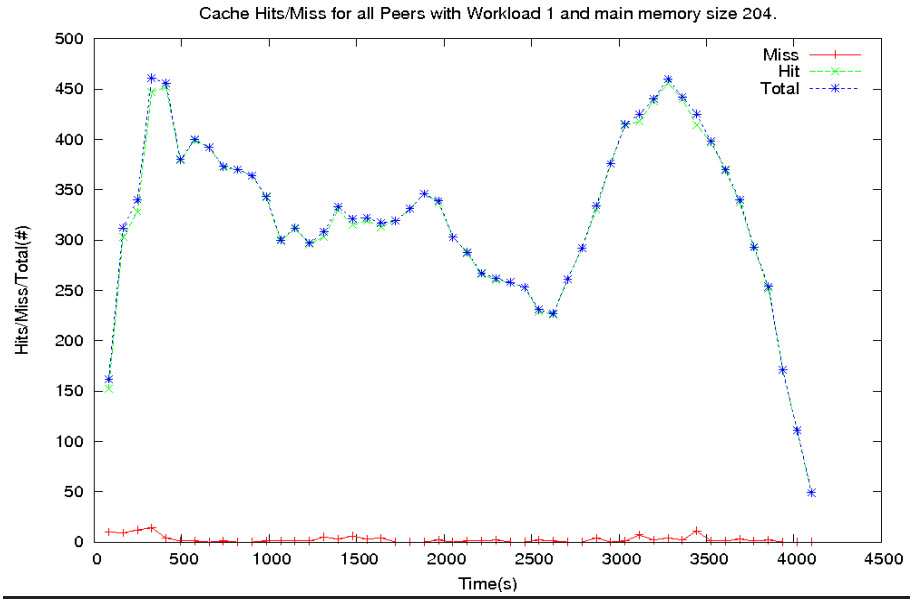
The results in this section are an indication that the RBC algorithm does perform well with P2P multimedia streaming. Even with the low temporal redundancy in Workload 2, it performs far greater than the previously examined algorithms. From the analysis we establish two formulas defining when we can have a cache miss shown in Formula 6.3 and Formula 6.4. We see that for Workload 1, we get close to 0 % CHR, while for Workload 2 we get a varied CHR which stabilizes at approximately 35 %. The assumption made that the nodes have small peer lists in Workload 2 is wrong. From the CHR we see in Figure 6.2, we see that the nodes have a higher number of requesting peers than assumed.

6.2.2 Experiments with Cache Sizes

In this section, we set the *Cache Size* factor to 204 blocks. The rest of the factors still have the default values introduced earlier. We also here start with explaining the results from Workload 1, and then presenting the results from Workload 2.

In Figure 6.3 we see the results from simulations with Workload 1. The results are very similar to the results we got with the default setup as shown in Figure 6.1. The

Figure 6.3 Workload 1, Cache Size=204, RBC.

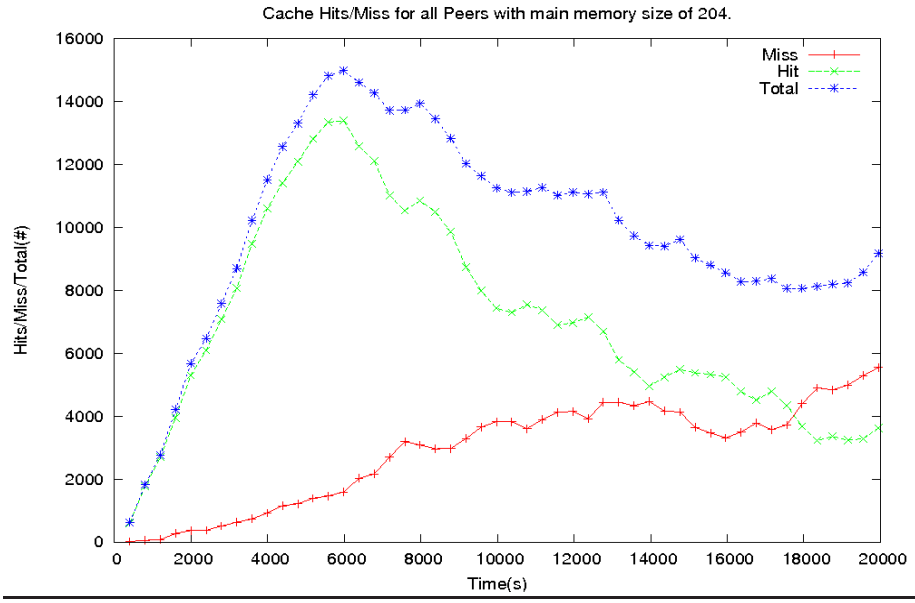


difference is that this time we got an even higher CHR. With the defaut setup, we have approximately 10 cache misses every second, while in when we increase the Cache Size factor to 204 blocks, we get almost zero. We remember Formula 6.4 describing when we can have cache misses. In this simulation we increase the *Cache Size* to 204, increasing the amount of requesting peers a node can handle to $(204/X < 20|X = 10)$ approximately 10 peers. It seems that increasing the number of peers the nodes can have almost eliminates the cache misses, resulting in 100 % CHR. This substantiates the Formula 6.4.

Next, we present the results from simulations with Workload 2. In Figure 6.4 we show the results from simulations were we increase the Cache Size to 204 blocks, and we get similar results as with the default setup shoen in Figure 6.2. The difference is that the CHR decreases faster per second with the default setup, compared to the results where we increase the *Cache Size* factor. We get this effect, because like with Workload 1, the nodes can have more peers in their peer lists before we get cache misses. We also see that the CHR stabilizes after approximately 10000 seconds. This indicates that the nodes peer lists increase in number until 10000 seconds have passed in Workload 2, and the number stabilizes at a number greater than 10. We can at this point argue that the higher the result of $CacheSize$ divided on the number of $RequestingPeers$ is in relation to the $WindowSize$, the lower CHR we get. We see this tendency as we increase the Cache size.

In this section, we set the *Cache Size* factor to 204 blocks. We see that with increas-

Figure 6.4 Workload 2, Cache Size=204, RBC.



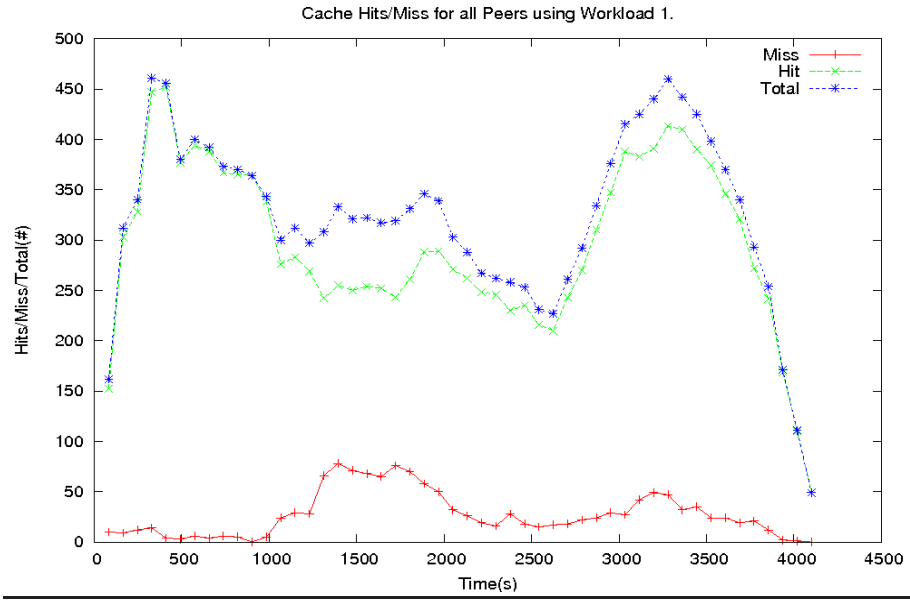
ing the Cache Size factor to 204 blocks, we get an effect where we achieve a higher CHR. For Workload 1 we actually nearly eliminate the cache misses, resulting in almost 100 % CHR. We further allege that the higher the sum of the $CacheSize$ divided on the number of $RequestingPeers$, is in relation to the $WindowSize$, the lower CHR we get.

6.2.3 Experiments with Relevance R

In this section, we introduce results from simulations where we vary the *Relevance R* value. We start by explaining the results from Workload 1, then showing the results from Workload 2. As we have argued earlier, this makes *Relevance A* and *Relevance G* less influential.

Figure 6.5 shows us a lower overall CHR than we had with our default setup in Figure 6.2. After 1000 seconds we have a decrease in the CHR in both graphs, however we have a larger decrease in Figure 6.5. With a higher Relevance R value, we get more dependent on caching the entire *Window* to be sure we cache the relevant blocks. Because *Relevance A* and *G* are less influential, we are no longer able to differentiate between individual blocks with as high precision as before. For example, in Figure 6.1 we get a very high CHR even though we know that several nodes has more than five peers in their peer lists. Relevance A and G does not serve any purpose before $CacheSize/RequestingPeers > WindowSize$. After

Figure 6.5 Workload 1, Relevance R=2, RBC.



1000 seconds the peer lists start to grow larger than five peers, and we are no longer able to chose individual blocks inside a window with as good precision as we do with lower *Relevance R* values. At this point we can conclude that the Relevance R value has to be balanced with the Relevance A and G values to get a greater CHR even when $CacheSize/RequestingPeers > WindowSize$.

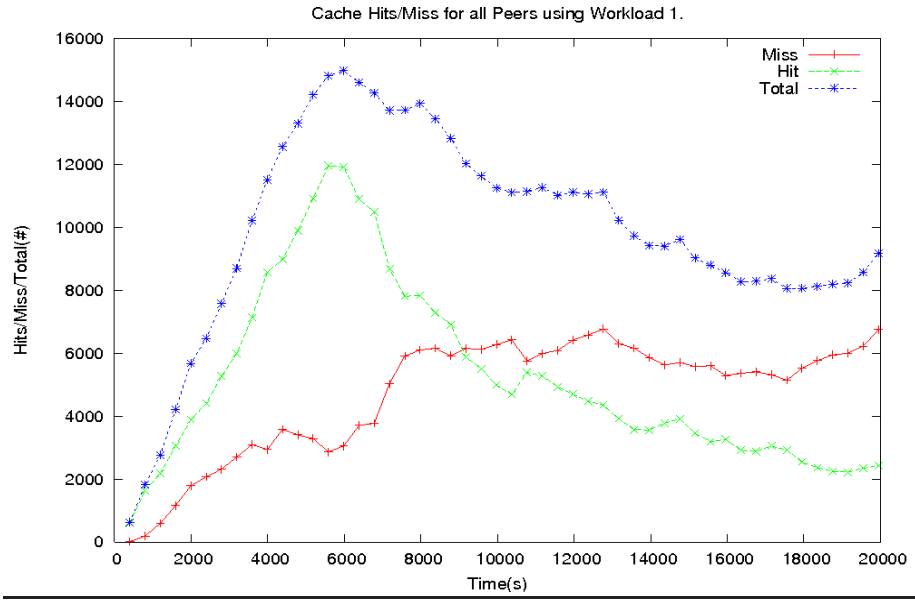
Next, we present the results from simulations with Workload 2. When comparing this result with the results with the default setup, we see that the CHR drops to 50 %[1] after 9000 seconds (seen in Figure 6.6) with an increase Relevance R value, and after 13000 seconds with the default setup as shown in Figure 6.2. We also see that the CHR stabilizes at a lower CHR percentage with a higher *Relevance R* value, than with the default setup.

These results agree with the previous reasoning for why a higher *Relevance R* value gives us a lower CHR. After 1000 seconds, the nodes start to get peer lists larger than five. However, this time the Relevance A and G values are not helping the nodes to chose which blocks in a window are the most important, resulting in caching of less relevant blocks. We are also able to conclude that the lower $CacheSize$ divided by the number of $RequestingPeers$ are in relation to the $WindowSize$, the less relevant *Relevance A* and *G* becomes.

In this section, we increase the *Relevance R* value from one to two. We see that

[1] This is indicated by the point where the cache hit curve and the cache miss curve meet.

Figure 6.6 Workload 2, Relevance R=2, RBC.



this gives us a lower CHR due to the diminishing effect this has on the impact of *Relevance A* and *G*. We also conclude that *Relevance A* and *G* are less influential the lower $CacheSize$ divided by the number of $RequestingPeers$ are in relation to the $WindowSize$.

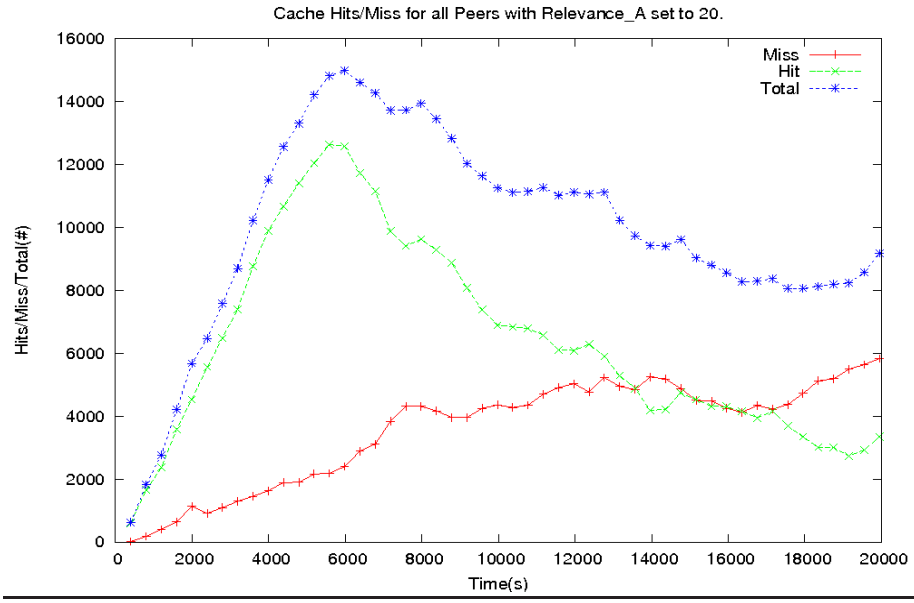
6.2.4 Experiments with Relevance A

In this section, we evaluate results from simulations where we have changed the behavior of *Relevance A*. We chose not to introduce results from Workload 1 in this section, as we have seen earlier that Workload 2 reflects the results from Workload 1. We start by showing results from increasing the initial value of *Relevance A*, and then we to present results from changing the rate at which *Relevance A* increases.

We can see in Figure 6.7 that until 13000 seconds into the simulation, the graph is equal to the graph from the default setup. Then, from 13000 seconds and forth, the CHR is a few percent higher than with the default setup. In other words, the increase in the initial *Relevance A* value does not have an effect until after 13000 seconds.

The effect of increasing the initial value for *Relevance A*, is that it is more influential with respect to which block is cached. We can see from the graph in Figure 6.7 that the results do not stabilize until 10000 seconds have passed. The access

Figure 6.7 Workload 2, Maximum Relevance A=20, RBC.

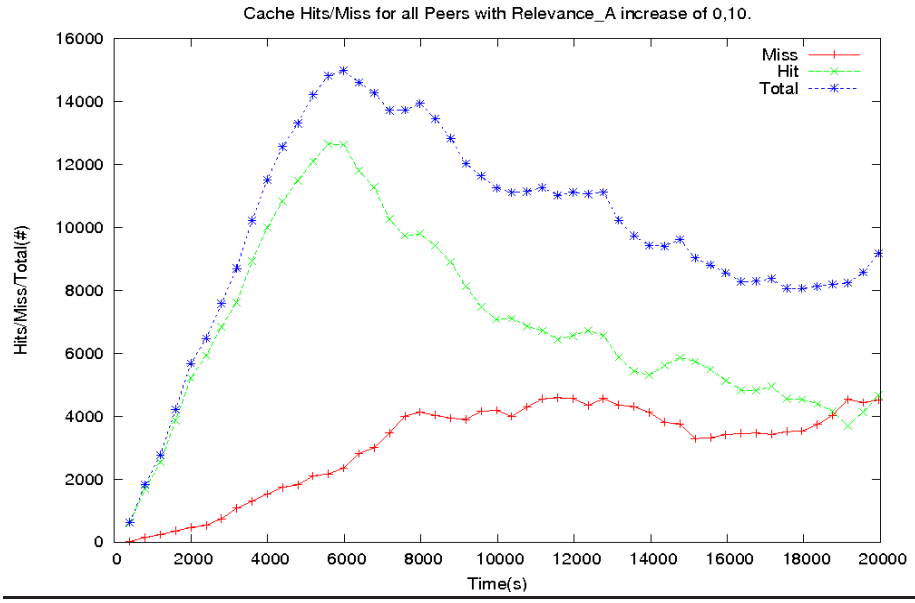


pattern is changing in the period from 1 to 10000 seconds, when the nodes peer lists are being populated. This makes the access pattern less useful because we are not able to make out a clear pattern in this phase. After 13000 seconds we see that *Relevance A* starts to make a difference. This is due to the peer lists being more stable. The late influence of the Relevance A value, is an indication on that the access pattern is identified too slow. From these arguments, we can say that if the nodes peer lists are changing often, we are not able to identify an access pattern quickly enough with the current *Relevance A Increase* factor, for the *Relevance A* value to influence the CHR.

Next, we increase the rate at which the *Relevance A* increase to 0,10. This creates an effect where the access pattern is identified more quickly, and this remedies the CHR somewhat during the unstable start up phase. Figure 6.8 shows us a higher CHR than what we get from increasing the initial *Relevance A* value. We actually do not reach a 50 % CHR until after approximately 19000 seconds, and we end at approximately 50 % CHR.

In this simulation we have doubled the rate at which the *Relevance A* values increase, and we can see from the graph that this has an effect where the access pattern is identified quickly enough to make a difference during unstable periods. By unstable periods, we mean when the peer lists are often changing. This concludes that the more quickly we increase the *Relevance A* values, the more use we can make of the access pattern as this is identified faster.

Figure 6.8 Workload 2, Relevance A Increase=0,10, RBC.



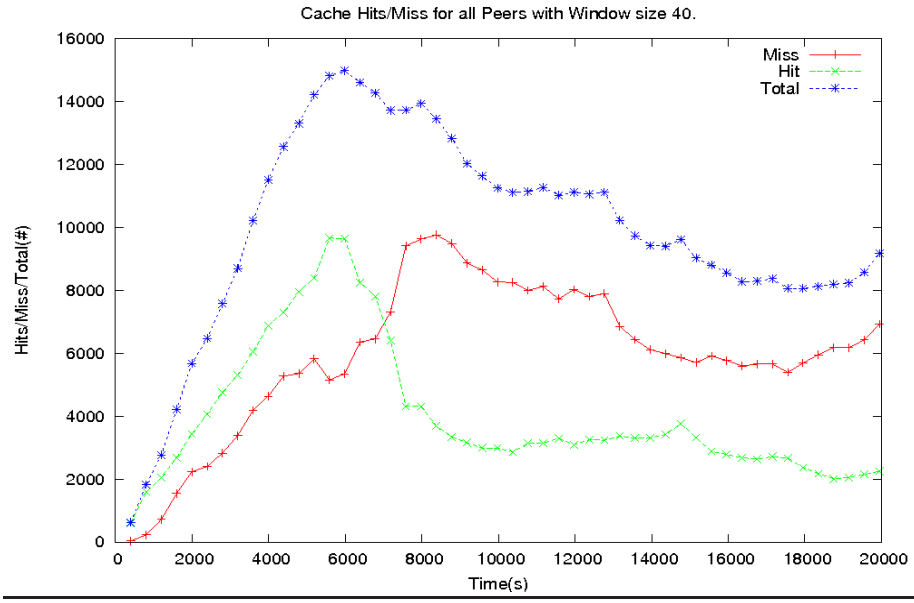
In this section, we experiment with increasing the maximum *Relevance A* value, and increasing the rate at which it is increased. We conclude that increasing the maximum *Relevance A* value does not make a difference on the performance before we have identified an access pattern. However, we are able to identify this access pattern earlier if we increase the rate at which *Relevance A* increases.

6.2.5 Experiments with Window Sizes

In this section, we show the results from simulations where we vary the *Window Sizes*. We chose also here to not present results from simulations with Workload 1. We start by increasing the Window Size to 40, for subsequently to decrease it to 10. By doing this we will illustrate Formula 6.2. With a larger Window Size, we see from the formula that we can have less requesting peers before getting cache misses.

The effect of this is easy to locate in Figure 6.9. The CHR drops to 50 % after approximately 7000 seconds, and it stays lower compared to results from earlier simulations. However, we see that the CHR after 20000 seconds is not much lower than with the default setup. With a *Window Size* of 40 blocks, and the other factors still at default values, the nodes can only have $(102/X > 40; X = 2)$ two peers in their peer lists before we start to get cache misses. For example, a node has

Figure 6.9 Workload 2, Window Size=40, RBC.

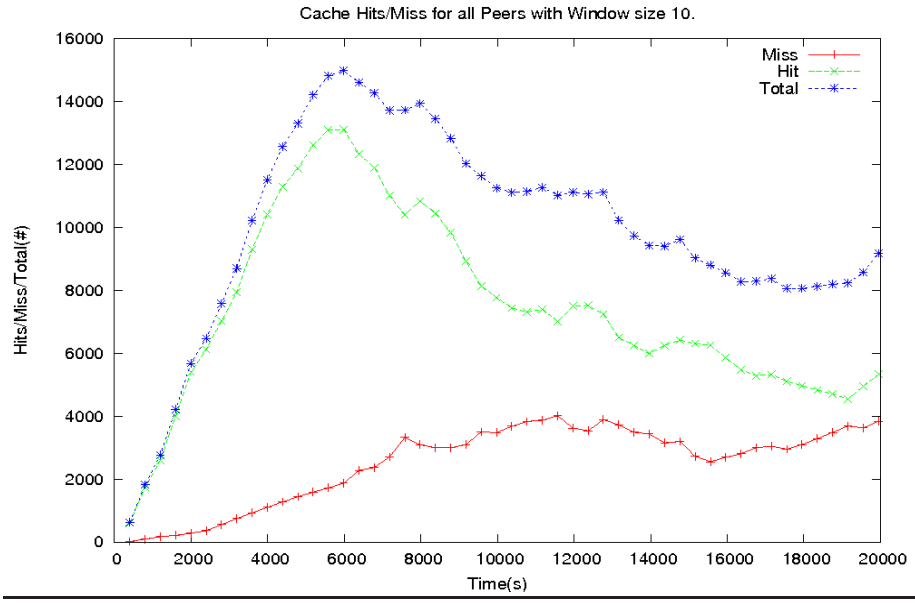


four peers requesting blocks. The first peer requests *Block 1*, and *Blocks 1 to 40* is cached. The next peer requests *Block 40*, and *Blocks 40 to 80* is cached. Then when the next peer requests *Block 80*, we only have enough space for *22 blocks*, and we have to begin swapping blocks to and from the cache. This results in a probability of swapping out relevant blocks, and can lead to cache misses.

Next, we evaluate the results from simulations where we set the *Window Size* to 10 blocks. Comparing results from simulation with a Window Size of 40 (shown in Figure 6.9), and results from simulations with a Window Size of 10 (shown in Figure 6.10), we see a large increase in the CHR when we decrease the Window Size. The CHR actually never drops below 50 %, and this is the best result we have gotten so far with only changing one factor. When comparing these results with the results we got with the default setup shown in Figure 6.2, we also see a tendency towards the cache miss curve stabilizes at an earlier stage with a smaller *Window Size*, than it did with the default setup. The CHR also ends up at approximately 60 %, which is an improvement of 25 % (the default setup ended up at 35 %). When we decrease the *Window Size*, we also increase the number of peers that can send requests to a node before we have cache misses. However, at the same time we increase the chance of $BlockNumberInterval > WindowSize$. However the *Window Size* of 10 seems to be larger than the *block number intervals* for this workload.

In this section, we increase and decrease the *Window Size*. The results further strengthen Formula 6.4. Formula 6.3 is still not proven and we do not have any

Figure 6.10 Workload 2, Window Size=10, RBC.



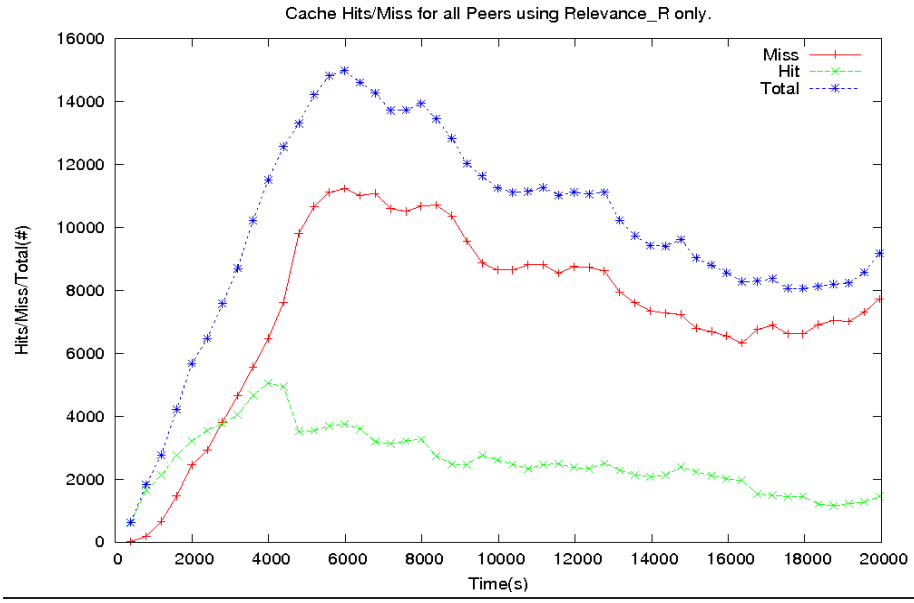
means of varying the *Block Number Interval* to test the feasibility of the formula.

6.2.6 Experiments with Combinations of Relevance Values

In this section, we explain the results from simulations where we test which relevance value is the most important. We chose to use *Relevance R* in all the simulations, because this is the basis of our algorithm. First, we present results from simulations with *Relevance R* only. Then we show results from the combinations; *Relevance R and Relevance G* and *Relevance R and Relevance A*.

Figure 6.11 shows the results where we only use the *Relevance R* value. We see that the CHR reach 50 % after approximately 3000 seconds, which is 10000 seconds earlier than with the default setup. In addition, we end up at approximately 16 % CHR, which is 19 % lower than with the default setup. We also see a tendency towards that the CHR stabilizes as early as after 3000 seconds. When we remove *Relevance G* and *Relevance A*, we look at every block in a *Window* as equally relevant. If the Formula 6.4 is correct, we can say with certainty that a client gets requests from more than five different peers already after 1000 seconds. This is shown where we start to get cache misses. We got a good CHR percentage after this point in the previous simulations because we could then separate the most relevant blocks in a *Window* from the rest with the use of Relevance A and G. This shows that without *Relevance G* and *Relevance A*, we are more dependent on ful-

Figure 6.11 Workload 2, Relevance R only, RBC.

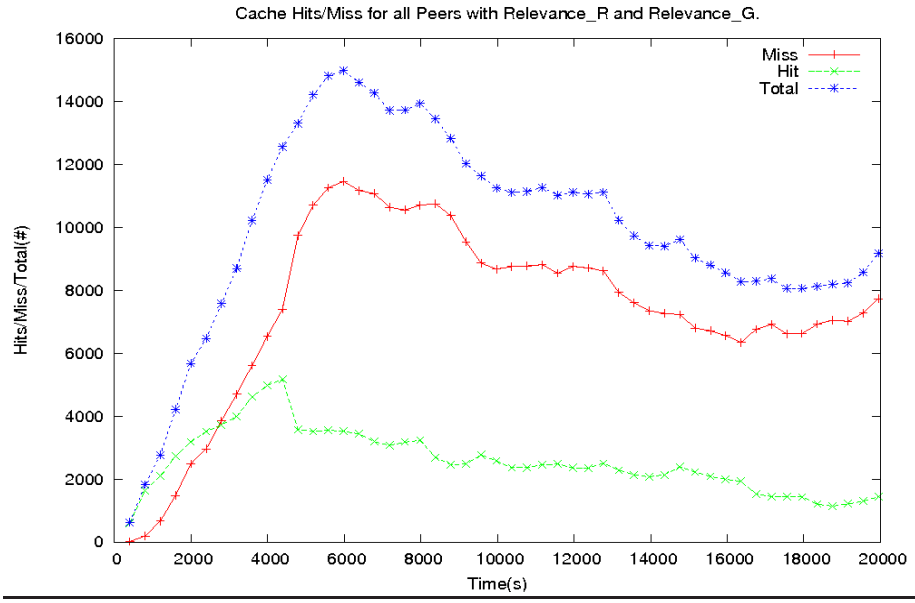


filling Formula 6.4.

Next, we introduce the results from simulations where we only use *Relevance R* and *G*. Relevance G is, as we recall, a relevance value based on how popular a file is as a whole. Figure 6.12 showing the results with only Relevance R and G, is almost identical to Figure 6.11 showing the results with only Relevance R. An observant reader will see small differences, however they are too small to be significant. This is an indication that the *Relevance G* value does not have a major influence on the CHR. An explanation for this can be found in the access patterns in Workload 2. As we recall from the analyzis of the access patterns, we saw that the requesting peers on a node more often than not were requesting different files. The *Zipf distribution* gives us a value distribution where the majority of the files are given almost equal *Relevance G* values, and only the most popular files get a higher value. Because all the files are of equal relevance in this case, differentiating between the files does not have a desired effect. They are of equal relevance because Workload 2 is too short to properly incorporate a Zipf distribution. In other words, because the popularity of the files in Workload 2 is not Zipf distributed, the *Relevance G* value has no effect.

Finally, we show the results from simulations with *Relevance R* and *A*. Relevance A is a relevance value based on the access pattern the requesting peers have on a node. Figure 6.13 illustrate the results, and we see a graph similar to the graph we got from the default setup. From the results of simulations with only *Relevance R*

Figure 6.12 Workload 2, Relevance R and G, RBC.



and *G*, this is as expected. Relevance G had no effect on the CHR, so to get the results we got with the default setup, *Relevance A* had to make the difference. We clearly see the importance of the access pattern in this simulation. With Relevance A, we are able to differentiate between the blocks in a *Window* in order to make the most relevant blocks more likely to be cached. And this is very important when we do not have enough space to cache the entire Window of every request.

In this section, we are trying different combinations of relevance values. We see that because the media file popularity is not Zipf distributed, *Relevance G* has minimal effect. However, *Relevance A* has a large impact on the CHR. This is due to the fact that we do not have enough space to cache the entire Window for each request, and Relevance A is able to distinguish the most relevant blocks in the Window.

6.2.7 Verification of Results using Workload 3

In this section, we infer a new workload called*Workload 3*. We will first show results from a simulation with the default setup, then we will decrease the *Window Size* and increase the *Relevance A Increase* value. By doing this we will verify the assumptions we have made earlier in this chapter about Formula 6.4.

Comparing results from simulations with the default setup as shown in Figure 6.14, and the results from simulations were we have decreased the Window Size and in-

Figure 6.13 Workload 2, Relevance R and A, RBC.

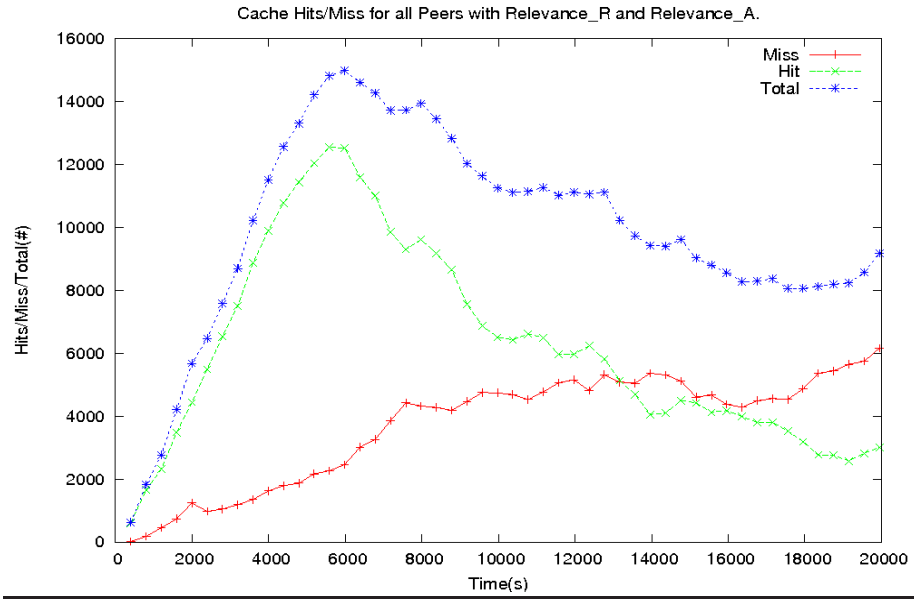
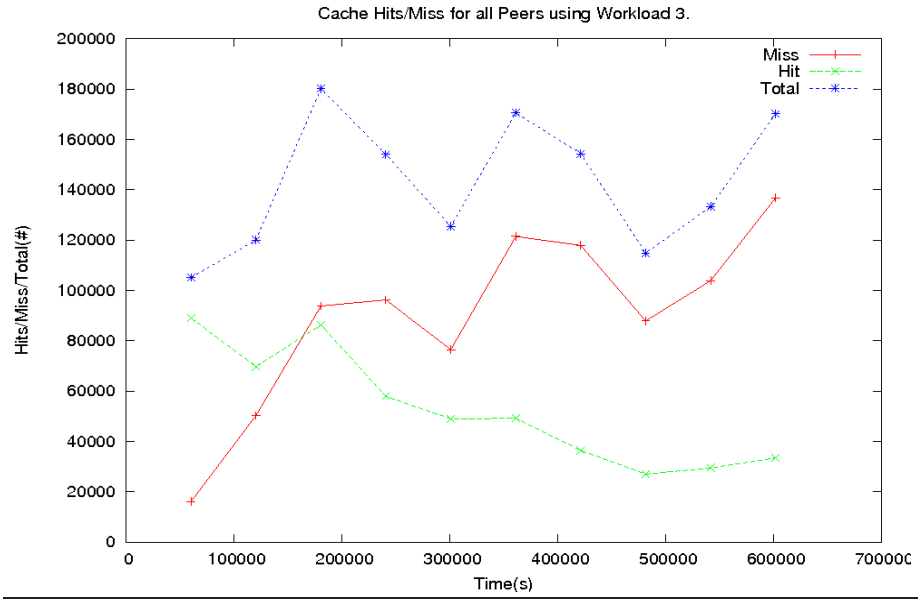


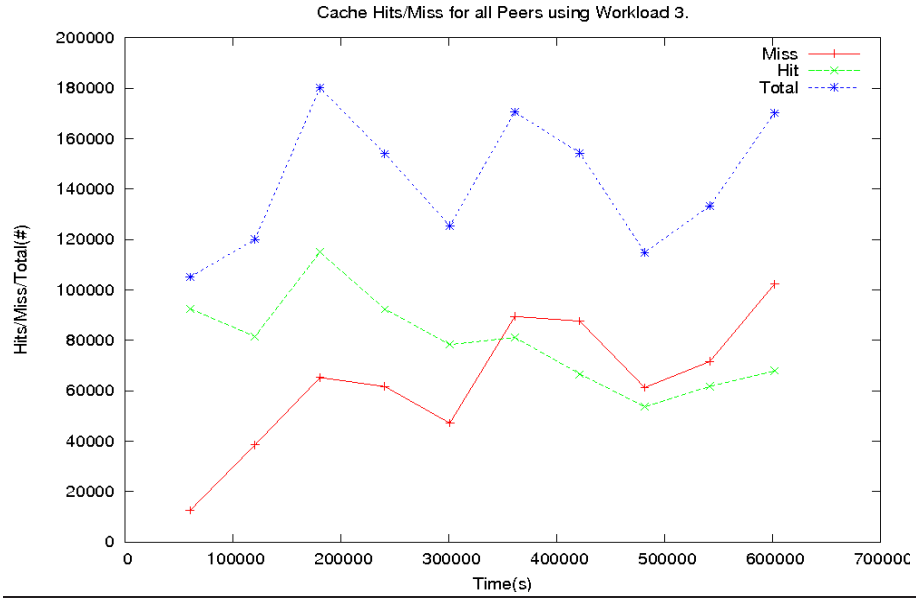
Figure 6.14 Workload 3, default setup, RBC



creased the Relevance A Increase value as shown in Figure 6.15, we see a great improvement of CHR in the latter. All we had to to was to decrease the *Window Size*, and increase the *Relavance A Increase value*. By doing this, we increase the number of requesting peers a node can serve before is start to get cache misses.

We are getting a lower CHR with this workload than with Workload 2. However, the important thing to notice is that by tuning important factors like *Window Size* and *Relevance A Increase*, we are able to improve the performance. Therefore, by analyzing the behavior of the nodes in a specific P2P environment, we are able to tune this algorithm to give better performance for that specific purpose.

Figure 6.15 Workload 3, Window Size=10, Relevance A Increase=0.10, RBC.



6.3 Summary

In this Chapter, we show with by using the default setup, that the RBC algorithm performs very well with all workloads. RBC successfully handles low temporal redundancy, object segmentation and P2P access patterns. With Workload 1 we get close to 100 % CHR, while we get 35 % for Workload 2 and 23 % for Workload 3. Furthermore, we are able to establish two formulas describing when we have cache misses, shown in Table 6.1.

	Formula
1:	$CacheSize/RequestingPeers > WindowSize$
2:	$BlockNumberInterval > WindowSize$

Table 6.1: Formulas describing when we have cache misses with RBC.

By experimenting with the different factors, we are showing that the *Cache Size* is influential towards the CHR. However, the Cache Size has to be compliant to the number of requesting peers in order to get the desired increase in CHR. If we have 20 requesting peers, and only slightly increase the Cache Size, this has a minimal effect. Next, we show that the relevance values have to be balanced or else they render each other useless. When we increase the *Relevance R* value we see that we do not get a positive effect from *Relevance A* and *G*, like we do with the default value.

We further show that the access patterns from the requesting peers are very relevant when choosing which blocks to cache. As we increase the rate at which *Relevance A* increases, thus clarifying the access pattern more quickly, we get an increase in the overall CHR. We also see that by increasing the initial Relevance A value, we get an increase in CHR. This shows us how important it is to handle the P2P access pattern correctly. Finally, we further strengthen the formulas shown in Table 6.1 by varying the *Window Size*. From the experiments with different factors, we claim that the ratio shown in Formula 6.5 is closely related to the CHR. The smaller the ratio is, the lower CHR we get.

$$CacheSize/RequestingPeers : WindowSize \tag{6.5}$$

Chapter 7

Conclusions

In this thesis, we have designed, implemented and evaluated a new caching algorithm called *Relevance Based Caching* (RBC). It is designed for P2P multimedia streaming and is based on existing knowledge as well as new ideas. We start this chapter by summarizing our contributions in Section 7.1. Subsequently, we list what could have been done differently in Section 7.2. Finally, we present ideas for future work in Section 7.3.

7.1 Contributions

In this section, we summarize our contributions. The first subsection summarizes the contributions from the design and implementation. In the following subsection, we summarize the contributions from the evaluation of the RBC algorithm.

7.1.1 Design and Implementation

We chosed to implement the simulation environment in Java, which is very flexible and easy to work with. There exists a great amount of documentation, and the active user community is large. The language proved to be more than able to express the complexity of the different algorithms. However, Java has some performance issues compared to languages such as C. Nonetheless, with the relatively small size of our simulations, this posed no problems. Another drawback with Java is that we have no control of the garbage collection, which we would have preferred.

We chose not to use a DBMS for holding data, mainly due to time constraints. There is a limited amount of data a PC can hold in main memory at a time, which

we experienced. This limited the size of the workloads somewhat, in terms of active nodes.

7.1.2 Evaluation

By implementing and evaluating several simulations, we have shown the limitations of existing caching algorithms when used together with multimedia streaming and P2P access patterns. Yet, LRU can be used for multimedia streaming as long as we have a high BNC for each file, and many peers requesting the same file.

We successfully designed and implemented a new caching algorithm for P2P multimedia streaming, called *RBC*. We have shown through multiple simulations, that with this algorithm we significantly increase the CHR compared to existing algorithms. RBC handles the P2P access pattern, object segmentation, and low temporal redundancy.

Furthermore, we have shown how important the access pattern of requesting nodes are in a P2P multimedia streaming environment. We showed that the faster the access patterns are identified, the better CHR we get. Without the access pattern, the RBC algorithm is dependent on caching the entire *Window* in order to get a high CHR. This is because the algorithm would look at all the blocks in the *Window* as equally important, and would not be able to distinguish the most relevant blocks. If the algorithm has to cache the entire *Window*, it emphasize a higher demand on the amount of available Cache Size.

$$(CacheSize/RequestingPeers) : WindowSize \tag{7.1}$$

We have shown that the factors used in the simulations can be optimized to greatly increase the CHR. None of the factors where tuned before we ran the simulations. The three most influential factors are the ones illustrated in Formula 7.1, and is closely related to the CHR. As long as we have a 1:1, we will not get cache misses, and the CHR should be close to 100 %. In addition, if the ratio is higher than 1:1, we have enough space to cache the entire Window, making Relevance A and G unnecessary. On the other hand, if the ratio is below 1:1, these relevance values become equally important. The ratio can be used to optimize an application when a user knows network characteristics such as access patterns. We could also use this ratio to increase the performance of clients. A client could disable Relevance A and G until a certain ratio threshold is reached. This would be helpful because a client often has to do other resource intensive operations like playback and data encoding.

7.2 Critical Assessment

The theoretical foundation of P2P caching algorithms, is still very fresh. There are no accepted P2P caching algorithms today, thus we chose not to compare our results with an algorithm designed for P2P traffic. However, we have chosen to examine existing and well established algorithms, to identify the shortcomings they expose with P2P network traffic patterns. By doing this, we hope to contribute to the knowledge platform that we see is developing in the P2P caching community.

We have chosen to use few factors in our simulations. This somewhat limits the posibilities of the simulations. The number of factors is closely related to the fact that the workloads are static and time consuming to generate. To have the possibility of varying a factor such as the number of requesting peers, even more information about the efficiency of our algorithm could have been obtained. However, with the selected factors, we are still able to test and evaluate the RBC algorithm thoroughly.

The workloads could have been larger and more complex. With a larger workload, the confidence of the results is increased. To handle larger workloads than the ones we use in our simulations, a DBMS has to be incorporated in the simulation environment. However, this would have complicated the implementation and consumed precious time. A larger workload may also imply a more complicated analysis, which may result in less concrete conclusions.

We use multiple relevance factors, i.e. *Relevance R, Relevance A* and *Relevance G*. Algorithms like LFU and LRU are successful, not only because they work, but also because they are simple. Few variables are often beneficial. However, we have shown that with three variables, the algorithm is not too resource demanding, while being very effective.

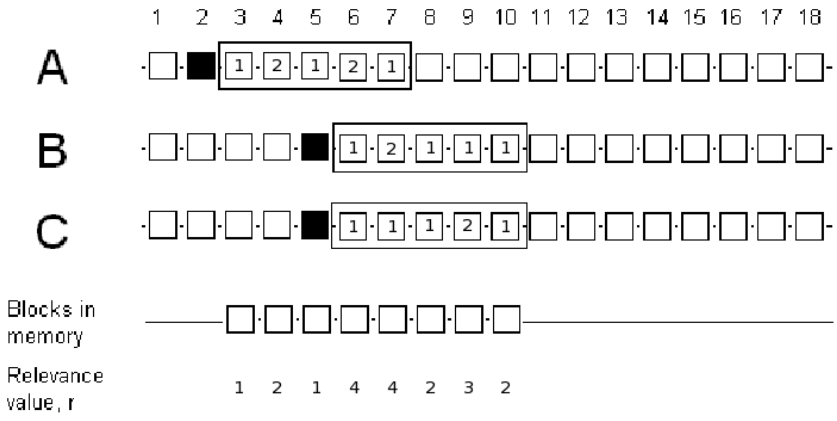
7.3 Future Work

P2P multimedia streaming is a fairly new field of research. In this section, we present some directions research in this field may take, based on the research done in this thesis.

Fewer Relevance Values In Section 7.2, we say that fewer variables are often beneficial. Our results show that the relevance value from the access pattern is very important. A future research field could be to use the main principals of RBC, however only use the *Relevance A* value. We propose to still to use prefetching

of windows. However, instead of adding *Relevance R* values, the algorithm can add values associated with only the access pattern as shown in Figure 7.1. The figure shows three active sessions, where each index in the windows has gotten a relevance value. This value is associated with the number of times an index is accessed. An approach like this would be a natural extension and can apply much to what is learned in this thesis.

Figure 7.1 Cache replacement proposal.



Zipf Distribution: The workloads are Zipf distributed, however Workload 1 and 2 are not long enough to express the distribution properly. So our Relevance G value had minimal effect on the performance. In order to test this relevance value, future evaluations of the algorithm could use a workload where the popularity of the files are properly Zipf distributed. We still think this value can increase the CHR, however future work can make use of a multiplier value to increase the importance of the Relevance G value.

Real Data: All of our workloads are generated by applications. It would be interesting to use traces gathered from live P2P multimedia streaming with real users.

CPU Utilization: The algorithm has not been tested on individual user PCs. It would be interesting to see the performance when deployed in an environment such as PlanetLab [10]. In such an environment we could have measured the CPU resource consumption.

Bibliography

[1] Age of conan. *http://community.ageofconan.com*.

[2] Gmail. *http://www.gmail.com*.

[3] Hotmail. *http://www.hotmail.com*.

[4] Moving picture experts group. *http://www.chiariglione.org/mpeg/*.

[5] Windows media player. *http://www.microsoft.com/windows/windowsmedia/player/10/*.

[6] Youtube. *http://www.youtube.com*.

[7] Buffer replacement algorithms for multimedia storage systems. In *ICMCS '96: Proceedings of the 1996 International Conference on Multimedia Computing and Systems (ICMCS '96)*, page 0172, Washington, DC, USA, 1996. IEEE Computer Society.

[8] S.-H. Gary Chan and S.-H. Ivan Yeung. Broadcasting video with the knowledge of user delay preference. *IEEE Transaction on Broadcasting*, 49(2):150–161, 2003.

[9] Umesh Chejara, Heung-Keung Chai, and Hyunjoon Cho. *Performance comparison of different cache-replacement policies for video distributio in CDN*. Springer Berlin / Heidelberg, 2004.

[10] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[11] Bram Cohen. Incentives build robustness in bittorrent. *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[12] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, 2005.

[13] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. *Joint International Conference on Measurement and Modeling of Computer Systems*, pages 143–152, 1990.

[14] Jayanata K. Dey-Sircar, James D. Salehi, James F. Kurose, and Don Towsley. Providing vcr capabilities in large-scale video servers. *International Multimedia Conference*, 1994.

[15] André Dufour and Ljiljana Trajkovic. Improving gnutella network performance using synthetic coordinates. In *QShine '06: Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks*, page 31, New York, NY, USA, 2006. ACM Press.

[16] Carsten Griwodz, Michael Bar, and Lars C. Wolf. Long-term movie popularity models in video-on-demand systems: Or the life of an on-demand movie. Technical report, 1997.

[17] Pål Halvorsen. Bufferhåndtering i multimedia datahåndteringssystemer. *Elektronikk, årg. 32*, (nr. 9):64–67, 1998.

[18] Markus Hofmann and Leland R. Beaumont. *Content Networking - Architecture, Protocols and Practice*. Morgan Kaufmann Publishers, 2005.

[19] Rai Jain. *The art of computer systems performance analysis*. John Wiley & Sons, Inc., 1991.

[20] Xuxian Jiang, Yu Dong, Dongyan Xu, and B. Bhargava. Gnustream: a p2p media streaming system prototype. In *ICME '03: Proceedings of the 2003 International Conference on Multimedia and Expo*, pages 325–328, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Taeseok Kim, Hyokyung Bahn, and Kern Koh. Considering user behavior and multiple qos supports in multimedia streaming caching. *J. VLSI Signal Process. Syst.*, 46(2-3):113–122, 2007.

[22] Col MacCárthaigh. *Joost Network Architecture*. Joost N.V, April 2007.

[23] Frank Moser, Achim Kraiss, and Wolfgang Klas. L/mrp: A buffer management strategy for interactive continuous data flows in a multimedia dbms. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 275–286, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[24] Andrew M. Odlyzko. Internet traffic growth: Sources and implications, 2003.

[25] Thomas Plageman, Vera Goebel, Andreas Mauthe, Laurent Mathy, Thierry Turletti, and Guillaume Urvoy-Keller. From content distribution networks to content networks - issues and challenges. *Computer Communications*, 29(5), March 2006.

[26] Stefan podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys(CSUR)*, Volume 35, Issue 4:374–398, December, 2003.

[27] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *4th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, Feb 2005.

[28] Sylvia Ratnasamy, Andrey Ermolinskiy, and Scott Shenker. Revisiting ip multicast. *SIGCOMM Comput. Commun. Rev.*, 36(4):15–26, 2006.

[29] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. *First International Conference on Peer-to-Peer Computing (P2P'01)*, 2001.

[30] Osama Saleh and Mohamed Hefeeda. Modeling and caching of peer-to-peet traffic. *Network Protocols. Proceedings of the 2006 14th IEEE*, pages 249–258, Nov 2006.

[31] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, pages 170–184, February 19, 2004.

[32] Karl Andre Skevik. The spp architecture, a system for interactive vod streaming. *Faculty of Mathematics and Natural Sciences, Univesity of Oslo*, 2007.

[33] Andrei Sukhov, Prasad Calyam, Warren Daly, and Alexander Iliin. Network requirements for high-speed real-time multimedia data streams. *III IPv6 Global Summit(Internet. New Generation -IPv6), Moscow*, pages 28–33, November 2004.

[34] Andrew S. Tanenbaum. *Modern Operating Systems*. Alan Apt, 2001.

[35] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. Medisyn: a synthetic streaming media service workload generator. *In NOSSDAV '03:Proceeding of the 13th international workshop on Network and operating systems support for digital audio an video, pp. 12-21. ACM Press*, 2003.

[36] Jared Winick and Sugih Jamin. Inet-3.0: Internet topology generator, 2002.

[37] Lin Wujuan, Law Sie Yong, and Yong Khai Leong. A client-assisted interval caching strategy for video-on-demand systems. *Comput. Commun.*, 29(18):3780–3788, 2006.

Appendix A

List of Abbreviations

BNC	Block Number Closeness
BNI	Block Number Interval
BW	Bandwidth
CDN	Content Distribution Network
CHR	Cache Hit Ratio
CN	Content Network
CPU	Central Processing Unit
DBMS	Data Base Management System
FIFO	First-In First-Out
FPS	Frames Per Second
I/O	Input/Output
JVM	Java Virtual Machine
LFU	Least Frequently Used
LHC	Local Host Cache
L/MRP	Least/Most Relevant for Presentation
LRU	Least Recently Used
RBC	Relevance Based Caching
RTCP	Real Time Control Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SCC	Site Content Cache
SMIL	Synchronized Multimedia Integration Language
P2P	Peer-to-Peer
WWW	World Wide Web

Appendix B

Running The Simulations

We ran all our simulations on a PC with Linux and Java 1.5 SDK. The simulation factors are all contained in the Constants.java class, and have to be changed prior to running the simulation. In order to run a simulation, the program takes four arguments:

- **WORKLOAD:** This defines the path to the trace-file.
- **ALGORITHM:** This parameter definesg which algorithm to run. We have four different options; 0, 1, 2 and 3. 0 for LFU, 1 for LRU, 2 for RANDOM and 3 for RBC.
- **LOG_RESOLUTION:** This parameter sets the granularity of the graph.
- **CACHE_SIZE:** This decides how many blocks the cache can hold.

To run a simulation, type the following command in the program folder:

- java cachingmain/Main WORKLOAD AlGORITHM LOG_RESOLUTION CACHE_SIZE

Every 100th second a string is printed on the form:*'timestamp active_nodes*'. The *'timestamp'* is the timestamp from an entry in the tracefile, identifying how far in the simulations you have gone.

When the program finishes, the program creates a new file called 'out.dat', as output. This file contain four columns of numbers. The first column is the time-interval, the second is the number of cache misses, the third is the number of cache hits, and the last is the total number of requests. This file is created for use with *Gnuplot*, which is a program for creating graphs. *Gnuplot* takes a list of commands as input. An example of such a list is shown in Figure **??**:

Figure B.1 An example of a Gnuplot script.

```
set terminal postscript color
set out "out.ps"
set data style linespoints
set title "Some title."
set xlabel "Time(s)"
set ylabel "Hits/Miss/Total(#)"
plot "out.dat" using 1:2 title 'Miss', \
"out.dat" using 1:3 title 'Hit', \
"out.dat" using 1:4 title 'Total'
```

In order to record the access pattern on a node, the RANDOM cache replacement algorithm has to be used. Before starting the simulation, the NODE_TO_LOG member in Constants.java has to be set to the node which should be logged. In addtition, LOGG_START and LOGG_END has to be set to desired numbers. The interval can not be too large or else the JVM will run out of heap space. The NODE_TO_LOG has to be manualy chosen from the trace file. When the simulation finishes, a log file named outAccess.datFILENAME, where FILENAME is the name of each file that has been accessed, is created. All these files then has to be combined with the use of a Gnuplot script, for then to use Gnuplot to create a graph.

Appendix C

Source Code

In this appendix, we present a selection of the source code developed for this master thesis. We chose to only present the code associated especially with the RBC algorithm. If the reader would like to see how the other algorithms are implemented, all the source code is enclosed on the CD. We start by presenting the main *RBC class*, for subsequently to present some of the most important functions incorporated in the *Node class*.

```
/*
 * RBC.java
 *
 * Created on 15. mars 2007, 18:36
 *
 */

package cachingmain;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Vector;

/**
 *
 * @author Administrator
 */
public class RBC {

    private Utilities.LoggStruct logg;
    private Utilities util;
    private int mainMemorySize;
    private Hashtable<String, Node> nodes;
    private BufferedWriter out;

    private int value = 100;
    private int increment = 100;

    /**
     * Creates a new instance of RBC
     *
     * @param intervals - The resolution of the log file.
     * @param first - The first timestamp.
     * @param last - The last timestamp.
     * @param util - An utility object.
     * @param cacheSize - The memory size in number of blocks.
     */
    public RBC(int intervals, int first, int last, Utilities util, int cacheSize)
    {
        this.util = util;
        this.nodes = new Hashtable<String, Node>(89, 0.80f);
```

```
        // Getting a logg object for the logging
        logg = util.CreateLoggStruct(intervals, first, last);
        this.mainMemorySize = cacheSize;
        // Create the first node holding all the blocks.
        Node server = new Node("10.0.0.1" , Constants.SERVER_CACHE_SIZE, logg);
        nodes.put(server.GetNodeId(), server);

    }

    /**
     * Parses a string accordingly to plans
     * @param str - The string to parse.
     * @param old - Defining of what type the string is.
     */
    public void Parse(String str, boolean old)
    {
        boolean output = true;

        String[] words = str.split(" ");
        String client = words[1].substring(0,words[1].length()-1);

        // Write output each 100 second.
        int progress = Integer.parseInt(words[0]);
        if(progress / value >= 1)
        {
            System.out.println(progress + " " + nodes.size());
            value = value + increment;
        }

        // Discard storing messages since we allways assume they are present on disk.
        if(words.length > 3)
        {
            if( words[6].equals("storing") || words[5].equals("storing") )
                return;
        }

        // We do not care if the node connects.
        if(words[2].equals("CONNECTION"))
            return;

        // If it disconnects, we remove the node to free space.
        if(words[2].equals("LOSTCONNECTION"))
        {
            nodes.remove(client);
            return;
        }

        // If the filter is set to 1, we only log the LHCs.
        if(Constants.FILTER == 1)
        {
            String start = client.substring(0, 4);
            if(!old && !start.equals("host"))
                return;
         }

        // If the node is not created, we create it.
        if((nodes.get(client)) == null)
        {
            Node node = new Node( client, mainMemorySize, this.logg);
            nodes.put( client, node );
        }

        // Get the node which the block is requested from.
        Node accessedNode = nodes.get(words[7]);

        // If the accessed Node is not created, we create a new.
        if(accessedNode == null)
        {
            accessedNode = new Node( words[7], mainMemorySize, this.logg );
            nodes.put( words[7], accessedNode);
        }

        // Create a session if its not yet created.
        accessedNode.CreateSession(client , words[3]);

        // Handle the file and calculate Zipf values.
        accessedNode.HandleFile(words[3], client);

        // Prefetch window of size WINDOW_SIZE.
        accessedNode.PrefetchWindow(Integer.parseInt(words[5]), client, words[3], progress);
    }

    public void WriteLog()
    {
```

```
            this.logg.WriteToFile("out.dat");
        }
}
```

In the included code above, the important method is the *Parse* method. For each log entry, this method is called. The method generates output in order to track the progress, and apply different filters. It also handles the creation of nodes. However, three important method calls made in this method are, *accessedNode.CreateSession(..), accessedNode.HandleFile(..) and accessedNode.PrefetchWindow(..)*. The *CreateSession* method creates a session, which keeps track of the *Window* and the access pattern in that *Window*. Next, the *HandleFile* method is keeping a list of requested files in a node, sorted by the number of induvidual requests. With individual requests, we mean from different nodes. This list is used to calculate the Zipf values. Finally, the *PrefecthWindow* method tries to prefetch a *Window*. This method handles the comparison of relevance values between blocks in the cache, and blocks to be prefetched. All these methods are public methods in the Node class, which brings us to the next inserted code.

```
/*
 * Node.java
 *
 * Created on 15. mars 2007, 17:50
 *
 */

package cachingmain;

import java.security.acl.Owner;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Vector;

/**
 *
 * @author Administrator
 */
public class Node {

    private String nodeId;
    private int cacheSize;
    private Hashtable<String, File> fileLookupList;
    private Hashtable<String, Session> sessions;
    private Utilities.LoggStruct logg;

    public String GetNodeId() { return nodeId; }
    public LinkedList<Block> Cache;
    public Hashtable<String, Block> Disk;
    public LinkedList<File> SortedFilelist;
    public Utilities.LoggStruct LoggAggregate;

    /**
     * Constructor
     *
     * @param nodeId – String. Uniquely identifying the node.
     * @param cacheLength – int. The size of the memory in number of blocks.
     * @param logg – Utilities.LoggStruct. The logg for registering cache miss/hit/total.
     */
    public Node(String nodeId, int cacheLength, Utilities.LoggStruct logg)
    {
        this.nodeId = nodeId;
        Cache = new LinkedList<Block>();
        cacheSize = cacheLength;
        // Set the initial capacity to a prime number for increased performance.
        Disk = new Hashtable<String, Block>(8209, 0.80f);
        SortedFilelist = new LinkedList<File>();
        fileLookupList = new Hashtable<String, File>();
        sessions = new Hashtable<String, Session>();
        LoggAggregate = logg;
    }
```

```
/**
 * Creates a session.
 * @param owner – String. The name of the node starting the session.
 * @param fileName – String. The filename.
 */
public void CreateSession(String owner, String fileName)
{
    if(sessions.get(owner) == null)
    {
        Session session = new Session(owner, fileName);
        sessions.put(owner, session);
    }
}


/**
 * Method for keeping a sorted list of files.
 * @param fileName – String. The name of the file.
 * @param client – String. The name of the client requesting the file.
 */
public void HandleFile(String fileName, String client)
{
    boolean fileListChanged = false;

    // Get the file object.
    File file = fileLookupList.get(fileName);

    // If the file does not exist, create it, set the relevance value and register the client.
    if( file == null )
    {
        file = new File(fileName);
        file.SetRelevanceValue(file.GetRelevanceValue() + 1);
        // insert the file into the lookup table.
        fileLookupList.put(fileName, file);
        file.RegisterClient(client);
    }
    else
    {
        // If the client is registered with this file previously, do nothing.
        if(file.IsRegistered(client))
            return;

        // Update the relevance value
        file.SetRelevanceValue(file.GetRelevanceValue() + 1);
        // Remove the file from the sorted list before inserting it again.
        int i = 0;
        for (File elem : SortedFilelist)
        {
            if(elem.GetFileName().equals(file.fileName))
            {
                SortedFilelist.remove(i);
                break;
            }
            i++;
        }
    }

    // Add the file to the sorted list.
    if(SortedFilelist.size() != 0)
    {
        int listSize = SortedFilelist.size();
        boolean inserted = false;
        for(int i = 0; i < listSize; i++)
        {
            if(SortedFilelist.get(i).GetRelevanceValue() <= file.GetRelevanceValue())
            {
                SortedFilelist.add(i, file);
                inserted = true;
                break;
            }
        }

        if(inserted == false)
            SortedFilelist.addLast( file );
    }
    else
        SortedFilelist.add( file );
}

/**
 * Method to get the associated Zipf value with a file.
 * @param filename – String. The file.
 * @return – double. The Zipf value.
 */
```

```
private double GetZipfValue(String filename)
{
    double value = 0;

    for(double i = 1.0; i <= SortedFilelist.size(); i++)
        value = value + ( 1.0 / i );

    // Find the rank
    double rank = 1.0;
    for(File file : SortedFilelist)
    {
        if(file.GetFileName().equals(filename))
            break;
        rank++;
    }

    value = ( 1 / rank) / value;
    return value;
}

/**
 * Prefetches a Window of blocks.
 * @param blocknr – int. The block that is requested.
 * @param client – String. The client requesting the block.
 * @param filename – String. The file the client requests.
 * @param timestamp – int. The timestamp of the request.
 */
public void PrefetchWindow(int blocknr, String client, String filename, int timestamp)
{
    Block block;

    // Get the session the request belongs to.
    Session session = sessions.get(client);
    session.RegisterBlockNumber(blocknr);
    session.StartIndex = blocknr;
    session.EndIndex = blocknr + Constants.WINDOW_SIZE – 1;

    // Check if the blocks are created, and if not create them and insert into secondary memory.
    // Keys in Disk are on the form: filename:blocknr.
    for(int i = blocknr; i < (blocknr + Constants.WINDOW_SIZE); i++)
    {
        block = Disk.get(String.format("%s:%s", filename, i));
        if( block == null )
        {
            block = new Block( i, filename );
            Disk.put( String.format( "%s:%s", filename, i ), block );
        }
        // Set the relevance R value.
        block.SetRelevance_R(block.GetRelevance_R() + Constants.RELEVANCE_R);
        // Add the blocks to the session window if they arent there.
        if(session.GetBlock(block.GetBlockNr()) == null)
            session.Window.add(block) ;
    }

    // Unreference previous blocks.
    session.UnReference(blocknr);
    block = Disk.get(String.format("%s:%s", filename, blocknr));

    if(block.InMainMemory)
        LoggAggregate.InsertIntoColumn(1, timestamp);
    else
        LoggAggregate.InsertIntoColumn(0, timestamp);

    // Now we are certain all the blocks are in secondary memory.
    // Next step is to Calculate R_total for each block to be fetched.
    for(int y = 0; y < session.Window.size(); y++)
    {
        Block tmp = session.Window.get(y);

        if(tmp.InMainMemory)
        {
            // if the block allready is in main memory, do nothing.
        }
        else
        {
            // If the memory is empty, just insert the block.
            if(Cache.size() == 0)
            {
                Cache.add(tmp);
                tmp.InMainMemory = true;
                continue;
            }

            double TotalRelevanceNew = 0.0;
```

```
                // Which relevance values are we using?
                if(Constants.RELEVANCE_R_ENABLED)
                    TotalRelevanceNew = TotalRelevanceNew + (double)tmp.GetRelevance_R();

                if(Constants.RELEVANCE_A_ENABLED)
                    TotalRelevanceNew = TotalRelevanceNew + session.accessPattern[y];

                if(Constants.RELEVANCE_G_ENABLED)
                    TotalRelevanceNew = TotalRelevanceNew + GetZipfValue(tmp.GetFileName());

                // For each block in the memory, compare with tmp.
                for(int i = 0; i < cacheSize; i++)
                {
                    double TotalRelevanceOld = 0.0;

                    if(Constants.RELEVANCE_R_ENABLED)
                        TotalRelevanceOld = TotalRelevanceOld + (double)Cache.get(i).GetRelevance_R();

                    if(Constants.RELEVANCE_G_ENABLED)
                        TotalRelevanceOld = TotalRelevanceOld + GetZipfValue(Cache.get(i).GetFileName());

                    // If the memory is not full, insert it.
                    if(Cache.size() < cacheSize)
                    {
                        tmp.InMainMemory = true;

                        if(TotalRelevanceNew >= TotalRelevanceOld)
                        {
                            Cache.add(i, tmp);
                            break;
                        }
                    }
                    else
                    {
                        // If the tmps relevance is larger than any in the memory,
                        // replace.
                        if( TotalRelevanceNew >= TotalRelevanceOld)
                        {
                            tmp.InMainMemory = true;
                            tmp = InsertIntoArray(i, tmp);
                            tmp.InMainMemory = false;
                            break;
                        }
                    }
                    // If theres free space in the memory, insert last.
                    if(tmp.InMainMemory)
                    {
                        Cache.add(tmp);
                        break;
                    }
                }
            }
        }
    }

    /**
     * Method to add a block to the memory.
     * @param index - int. The index where the block are inserted.
     * @param block - Block. The block to be inserted.
     * @return - Block. The block that is replaced.
     */
    private Block InsertIntoArray(int index, Block block)
    {
        Block out = Cache.getLast();
        Cache.removeLast();
        Cache.add(index, block);

        return out;
    }

    /**
     * A class for holding session relevant data.
     */
    private class Session
    {
        private String sessionOwner;
        private String fileName;
        private int firstInWindow;

        public int StartIndex;
        public int EndIndex;
        public LinkedList<Block> Window;
        public double[] accessPattern;
```

```
    /**
     * Constructor
     * @param owner − String. The client that started the session.
     * @param fileName − String. The file being accessed.
     */
    public Session(String owner, String fileName)
    {
        this.sessionOwner = owner;
        this.fileName = fileName;
        this.Window = new LinkedList<Block>();
        accessPattern = new double[Constants.WINDOW_SIZE];
        double value = Constants.RELEVANCE_A / Constants.WINDOW_SIZE;
        // Initialize the accessPattern array.
        for(int i = 0; i < accessPattern.length; i++)
            accessPattern[i] = value;
    }

    /**
     * Method to handle relevance A values.
     * @param blocknr − int. The blocknr being requested.
     */
    public void RegisterBlockNumber(int blocknr)
    {
        int index = blocknr − StartIndex;
        double value = Constants.RELAVANCE_A_INCREASE / (Constants.WINDOW_SIZE − 1);
        for(int i = 0; i < accessPattern.length; i++)
        {
            if(i == index)
                accessPattern[i] = accessPattern[i] + Constants.RELAVANCE_A_INCREASE;
            else
                accessPattern[i] = accessPattern[i] − value;
        }
    }

    /**
     * Method to get a block from the Window.
     * @param blocknr − int. The blocknr of the block to get.
     * @return − Block.
     */
    public Block GetBlock(int blocknr)
    {
        for(Block bl:Window)
        {
            if(bl.GetBlockNr() == blocknr)
                return bl;
        }
        return null;
    }

    /**
     * Unreferenced blocks with a blocknr lower than the newhead.
     * @param newhead − int. The blocknr of the requested block.
     */
    public void UnReference(int newhead)
    {
        int it = 0;

        // If the newhead is < than the first block in the list, we remove the WINDOW_SIZE first items.
        if(Window.get(it).GetBlockNr() > newhead)
        {
            // Remove blocks from the window
            for(int i = 0; i < Constants.WINDOW_SIZE; i++)
                Window.removeFirst();
         }

        while(Window.get(it).GetBlockNr() < newhead)
        {
            Block tmp = Window.get(it);
            tmp.SetRelevance_R(tmp.GetRelevance_R() − Constants.RELEVANCE_R);
            it++;
        }
        // Remove blocks from the window
        for(int i = 0; i < it; i++)
            Window.removeFirst();
    }

    // Internal class
    private class SessionBlock
    {
        private int blocknr;
        private String fileName;
    }
}
```

```
    // A class holding properties associated with files.
    private class File
    {
        private String fileName;

        private Hashtable connectedClients;
        private double relevanceValue;
        public double GetRelevanceValue() { return relevanceValue; }
        public void SetRelevanceValue(double value) { relevanceValue = value; }
        public int filelistIndex;
        public void SetFilelistIndex(int value) { filelistIndex = value; }
        public int GetFilelistIndex() { return filelistIndex; }
        public String GetFileName() { return fileName; }

        /**
          * Constructor
          * @param filename – String. The file name.
          */
        public File(String filename)
        {
            this.fileName = filename;
            this.relevanceValue = 0.0;
            this.connectedClients = new Hashtable();
        }

        /**
          * Method to check if a node is registered for this file.
          * @param name – String. The name of the node.
          * @return – boolean. True if the node is registered.
          */
        public boolean IsRegistered(String name)
        {
            if(connectedClients.get(name) == null)
            {
                return false;
            }
            return true;
        }

        /**
          * Method to register a node.
          * @param name – String. The name of the node.
          */
        public void RegisterClient(String name)
        {
            connectedClients.put(name, "temp");
        }
    }
}
```

In this code, the most important method is the *PrefecthWindow* method which start on line 163. This is the core of the implementation of the algorithm. The method start by retrieving the session for the requesting peer, and registering the access pattern in line 168. Then it proceeds with creating the blocks if they do not exist, and establish a *Window*. While establishing the *Window*, it also adds *Relevance R* values to each block currently in the *Window*. Next, in line 191, the blocks that previously was in the *Window*, are now unreferenced. Finally, we compare all the blocks in the *Window* with the blocks in the cache. If the total relevance value of a block in the *Window* is larger than the total relevance value of a block in the cache, we evict the block with the lowest relevance value from the cache.

Appendix D

The CD

Enclosed to the master thesis is a cd containing the source code, a copy of the master thesis, and the workloads used in the simulations. The workloads are compressed with gzip. All the classes are contained in the package 'cachingmain'. Following is a short description of what the different classes contains:

- **Block.java:** The Block class is a class for holding data concerning a block.
- **Constants.java:** This class contains different constants defining the different factors used in the simulations.
- **LFU.java:** LFU.java is an implementation of the Least Frequently Used cache replacement algorithm.
- **LRU.java:** This class contains the code for the Least Recently Used cache replacement algorithm.
- **Main.java:** The Main class is the entry point for the run time environment. It handles the parameters.
- **Node.java:** Node represent a node in the simulation and contains most of the methods associated with a node.
- **Parser.java:** After the parameters have been handled, the Parser class creates a cache replacement algorithm object of the correct type, and starts to iterate through the trace file.
- **RANDOM.java:** RANDOM.java is the implementation of the RANDOM cache replacement algorithm.
- **RBC.java:** This is the implementation of the Relevance Based Caching algorithm.

- **Utilities.java:** The Utilities.java class contains different methods used by all the cache replacement algorithms.

All the code is thouroughly commented, and a reader is encouraged to look at the files for a better understanding.