Architectural Principles for Large Scale Web sites: a Case Study

Tristan Linnenbank

Thesis for a Master of Science degree in Computer Science from the University of Twente, Enschede, The Netherlands.

Date	December 6, 2006
Assignment Organization	Hyves
	Herengracht 252, 1016 BV, Amsterdam, The Netherlands
Graduation Organization	University of Twente
	Faculty of Electric Engineering, Mathematics and Computer Science
	Department of Computer Science
	Architecture and Services of Network Applications Chair
	Drienerlolaan 5, 7522 NB, Enschede, The Netherlands
Graduation Committee	dr. L. Ferreira Pires (University of Twente)
	dr.ir. D. A. C. Quartel (University of Twente)
	dr.ir. R. van de Meent (University of Twente)
	ir. K. Kam (Hyves)
	R.J. van Alteren (Hyves)
Available to public	1st of March 2007

Abstract

Nowadays, almost everyone has access to the Internet. On the Internet, a large amount of Web sites exists. In practice, most Web sites start as small Web sites with a small scale architecture and might experience an enormous unexpected and/or unanticipated growth in users, page views and features. Because of this growth, the Web site's architecture of today has limited applicability for tomorrow's growth. This increased use and adoption of the Web site's services causes a demand for a higher level of scalability and redundancy, when compared to the original requirements of the current architecture. Google, MySpace and MSN are examples of such large Web sites that probably had to handle with these problems. We do not know for sure, since except for Google, little is known about the architectures of these Web sites.

In this Master thesis, we gain insight into the architectures of large scale dynamic Web sites like Google and Myspace by using the Web site of the Hyves service as a case study. The Hyves Web site currently experiences an enormous growth in users and page views and the current architecture is expected not to be scalable for this growth. Within the case study, we identified architectural principles for designing and implementing such large scale Web sites. The results from a case study are not necessarily applicable for designing and implementing any large scale Web site, therefore the identified principles are not general applicable.

We restricted our research to the media service part of the architecture of the Hyves Web site, and divided the research up to three smaller researches that identified architectural principles for the resource management layer, the application logic layer and the presentation layer. Part of the architectural principles were identified by studying and analyzing the current architecture first. Secondly, we analyzed the current experienced problems with the Hyves Web site. The analyzed problems and the solutions we designed to solve these problems were used to identify the other part of the architectural principles.

By analyzing the current architecture we identified the following architectural principles:

- Supporting services within the large scale Web site should be prevented from being single points of failures. During our research we experienced that supporting services that were not regarded as critical were implemented as single points of failures. During the growth of the large scale Web site, these services became more critical and the design decision to implement these services as single points of failures became invalid. Redundancy, improved performance and portability are mentioned as possible solutions;

- One should prevent tight coupling as much as possible. This concerns hard coding parameters in the application and using limited resources like storage for more than one part of the service. During our research we experienced that both concerns limited the growth of the case study's Web site;

- Implement application-level monitoring from the start when designing a part of a large scale Web site. System-level monitoring the applications and machines that are used to implement the service is not sufficient. The performance and functioning of in-house written applications should also be monitored and also the aggregate performance and functioning of the applications and machines that are used to the implement the service.

By analyzing the currently experienced problems and the solutions we designed to solve these problems we identified the following architectural principles:

- The load of resources should be balanced using a scheme in which the chance that an item is stored on an arbitrary storage node is equal for all storage nodes and all stored items. In our case study we have seen that current performance problems can be solved when introducing such a scheme.
- Every design should incorporate that current available resources such as processing capacity and storage capacity are not sufficient when used as a stand-alone solution. This principle overcomes the problem of a growth of a large scale Web site that outperforms the growth of the capacity of available resources. As a solution, the use of pools and/or clusters of resources are mentioned.
- The use of pools and/or clusters of resources also create redundancy and scalability. The current problems in our case study showed that some parts were not scalable, since it was not possible to add additional capacity to increase the performance of that part of the service. The same problems also showed that in case of failure, that part of the service was stopped because there was no redundancy to overcome a failure.
- For the use of any large scale Web site, research the request pattern. When designing a large scale Web site incorporate the monitoring of the requests. Current performance problems in our case study were solved by analyzing the requests served by the large scale Web site and adjusting the architecture handle such requests more efficiently.

Preface

This Master's Thesis is the final product of my Computer Science education at the department of Computer Science, at the Faculty of Electrical Engineering, Mathematics and Computer Science of the University of Twente, The Netherlands.

I would like to thank a number of people who have given me the opportunity to complete this thesis:

From the University of Twente, I would like to thank Luís Ferreira Pires and Remco van de Meent for their valuable input while discussing my progress and approach. Luís and Remco, thank you for keeping me on track and for the guidance during this research.

From Hyves, I would like to thank Koen Kam for the opportunity to do my research at such a large scale Web site and Ramon van Alteren for being a nice co-worker and discussing several implementation details. I also would like to thank Reinoud Elhorst for philosophizing about certain problems and solutions on the media service during my research. I would also like to thank the rest of my co-workers at Hyves for the pleasant time at Hyves Headquarters.

Finally, I would like to thank Cynthia, my girlfriend, for her support, for her patience, for reading the several versions of the thesis and hearing me talking way too much about my graduation project ☺.

Lelystad & Amsterdam, The Netherlands, December 6, 2006.

Tristan Linnenbank

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overall introduction to the work reported in this Master's thesis. It first presents the motivation behind the reported work, followed by the objectives to be achieved. Subsequently, this chapter illustrates the approach applied in accomplishing these objectives. This chapter ends with a global overview of the structure of this report.

1.1 Motivation

Nowadays, almost everyone has access to the Internet. On the Internet, a large amount of Web sites[1] exists. Web sites are either static or dynamic. In the first case, the content consists of a set of HTML-pages which are placed on a Web server by an administrator. The contents of the Web site only changes when a different set of HTML-pages is placed on the Web server. In the second case, the content of a Web site is generated either periodically (e.g., every three hours), or instantaneously (e.g., on every request). The context of a generation (e.g., the user, the time of the day, the location of a user, the information in a database) can influence the generated contents at some time and in the future.

The more Web sites are used, the more requests they receive. When a single Web server cannot cope with the number of requests, for example, when the maximum number of requests per server is CPU-bounded, multiple load-balanced servers using content replication techniques can be used to increase the service's performance. Content-replication is a technique that uses redundant resources. In the case of dynamic Web sites, the content is often stored in a database. Content-replication of databases is called database replication. In such a setup, multiple databases are used (replication slaves) that requests for updates from a single database (replication master).

For our example, a solution like content replication is theoretically sufficient to increase the service's performance, but in practice, when the number of slave requests increases and/or the number of information changes increases above a certain threshold, the replication master is not fast enough to serve all its slaves. Furthermore, it is not designed to have many large content repositories and to keep them synchronized. This issue on content replication is a typical example of the use of a standard technique for increasing performance that appears to be insufficient in practice when applied at large scale dynamic Web sites.

Within this thesis, we define large scale dynamic Web sites as Web sites that demand for non-standard techniques for improving performance in order to enable a higher level of scalability and redundancy for storing, processing or serving (or any combination of these three options) large amounts of requests. A higher level of scalability reduces overhead and/or extends the maximum number of requests that a system stores, processes or serves. The non-standard techniques are embodied in scalable architectures for large scale Web sites. A higher level of redundancy increases the maximum number of failures that have to occur to disable the entire service.

[1] We use *Web site* according to the Associated Press guidelines [WIKIW], and not *web site* or *website* as sometimes seen.

In practice, most Web sites start as small scale Web sites with a small scale architecture and might experience an enormous unexpected and/or unanticipated growth in users, page views and features. Because of this growth, the Web site's architecture of today has limited applicability for tomorrow's growth. This increased use and adoption of the Web site's service causes a demand for a higher level of scalability and redundancy, when compared to the original requirement of the current architecture. The increased use and adoption of the service also limits the possibilities of migrating to an improved scalable architecture because of two reasons. First of all, it is not possible to shut down the Web site's service for a long period of time and replace it with an improved one, because the down-time leads to the loss of page views, orders, advertisements, users and eventually, money. Secondly, user data (e.g., photos, entered information, etc.) in the old architecture should be accessible in the new architecture because we do not want users to experience data loss caused by a migration.

Google [GOOGLE], MySpace [MYSPACE] and MSN [MSN] are examples of such large Web sites that probably had to handle with these problems. We do not know for sure, since except for Google [GLABS], little is known about the architectures of these Web sites. Probably the companies behind those Web sites are not willing to share their intellectual properties. However, Web sites like those mentioned, are the proof that it is possible to create and maintain such large and complex dynamic Web sites and to overcome the standard performance problems that arise from such scales.

In this Master thesis, we gain insight into the architectures of large scale dynamic Web sites like Google and MySpace and we identify principles for creating architectures for such large scale dynamic Web sites. By taking into account these principles when creating a Web site, from the start a more scalable and redundant architecture is created and problems caused by poor scalability can be prevented. These principles can also be applied on existing large scale Web sites when developing a new architecture to enable a higher level of scalability and redundancy.

We use the Web site of Hyves [HYVES], a Dutch online community, as a case study of a large scale dynamic Web site. The Hyves Web site currently experiences an enormous growth in users and page views and the current architecture is expected not to be scalable for this growth.

In the next sections we provide an overview of our case study and its problems and challenges.

1.2 Case Study

Hyves is an online community, which is a group of people communicating and interacting with each other by means of information technologies, typically the Internet, rather than face to face [WIKIOC]. "Hyves" is derived from the English word "hives", which, according to [AHD00] is "a place swarming with activity" or "to live with many others in close association".

Hyves's members can create or extend their own hyves of friends by inviting and adding each other to their hyves. For its members, the Web site offers sophisticated user profiles (see Figure 1.1) and photo albums. In addition to hyves based on friendships, one can also join hyves based on locations, events or interests.

The service of Hyves is based on in-house written software in PHP [PHP] and it offered via a Web site (`http://www.hyves.nl`) using tools like MySQL [MYSQL], Apache [APACHE], Squid [SQUID], Nagios [NAGIOS], IPVS [IPVS] and Gentoo Linux [GENTOO] running on about 200 servers.

The Hyves service consists of these services:

- the Web site (user profiles, Web blogs, user messages);
- a chat service;
- a media service (storing, processing and serving of media items like photos and videos).

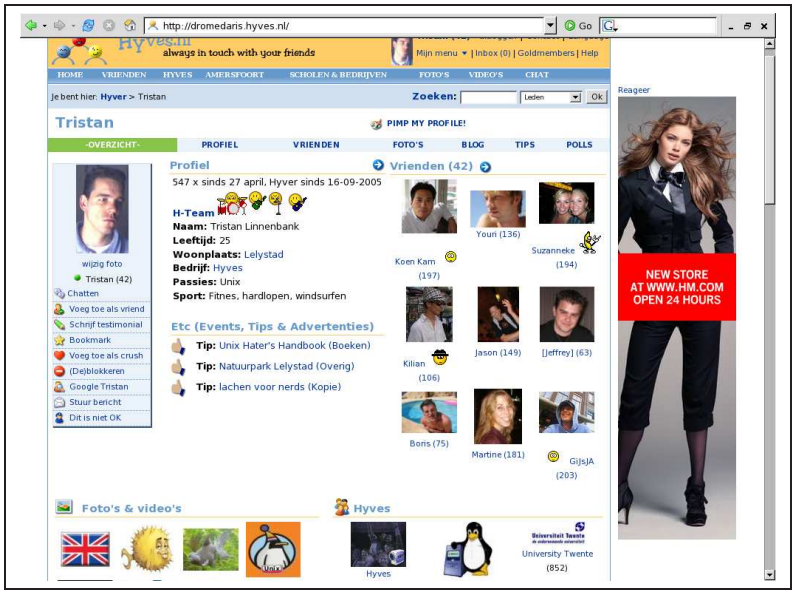

Figure 1.1: **A Hyves profile**

Figure 1.2: **The Hyves Web site**

Examples of scalability problems with the Hyves service at the start of this research were:

- Database scaling: 32-bits MySQL can address 2GB of RAM-memory, but the indexes in the database eventually will be larger than 2GB. Performance decreases dramatically in case the indexes cannot be stored entirely in RAM because of disk access caused by paging.
- Chat: how to offer a chat service to more than two million users?
- Media growth: Currently more and more photos and other media are uploaded. A good storage and serving architecture is necessary to offer millions of media items to millions of users. It is expected that the current architecture does not scale out well. How can we change the current architecture to a more scalable solution?
- Maintenance: how to maintain, configure, backup or install 200 or even more servers?

Because of the complexity and size of the Hyves service, we had to select one specific problem to address in this thesis. We have chosen the media growth problem to be addressed in this thesis. The main argument for choosing the media growth problem is that the media service includes all layers of a typical distributed application [ALONSO] and, therefore, when analyzing it, we might identify architectural principles for all the three layers.

What Hyves distinguishes from other so called “online communities” is that Hyves does not limit its members by some sort of quota on the number of photos that can be uploaded or on the aggregate disk usage as a result of these uploads. Not limiting its members with respect to the photo uploading is a strategic choice. This makes Hyves a popular place to share photo’s (see in Figure 1.2 the “Photos” and “Video” sections) as can be derived from the number of photo downloads.

In addition to photo sharing, Hyves recently released video sharing in which Hyves continues its strategy to allow its members to share unlimited amounts of videos. Currently just small videos are supported (i.e., from a mobile phone camera) but future use might allow larger and longer videos. Hyves is considering offering file sharing in the nearby future (i.e., Powerpoint presentations, Word reports). All these items that can be uploaded are referenced with the term media.

Within the Hyves media service, three functions exists:

- Media items are stored;
- Media items are processed;
- Media items are served.

These functions are affected by the increasing number of page views and members.

1.3 Problem and Challenges

The number of Hyves members and sub sites (e.g., Classifieds, Hyvertising, Photo’s, etc.) is still growing. Next to the growing number of members (see Figure 1.3), the number of page views per day is also growing (see Figure 1.4). This implies that more server capacity is needed to service all users and sub sites.

1.3.1 Media Growth Storage Challenges

The Hyves service experiences a large number of file uploads by its users (see Figure 1.5). The number of uploaded items is expected to increase because of the following reasons:

- the number of uploads per period of time is larger than the number of deletions per period of time. Actually, deleted items are not deleted from storage, but only their entry in the database is deleted;

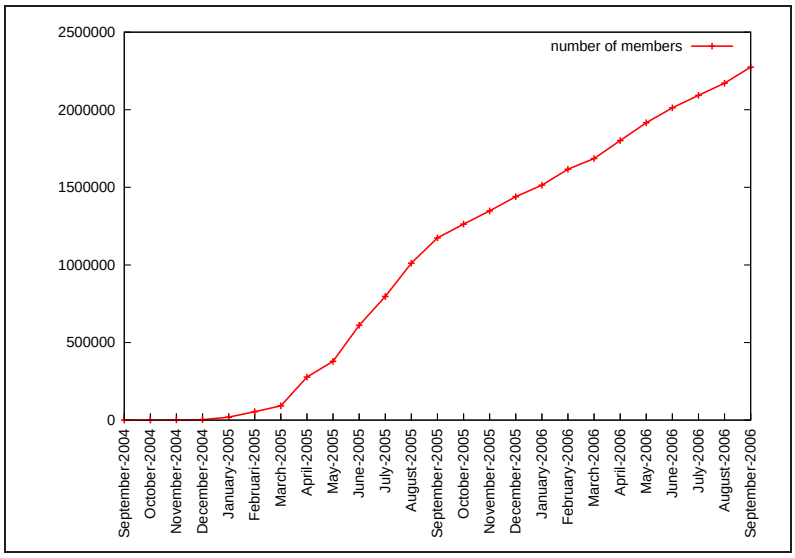

Figure 1.3: **Graph of the number of members as known on the first of each month, displaying the period of October 2004 - September 2006.**

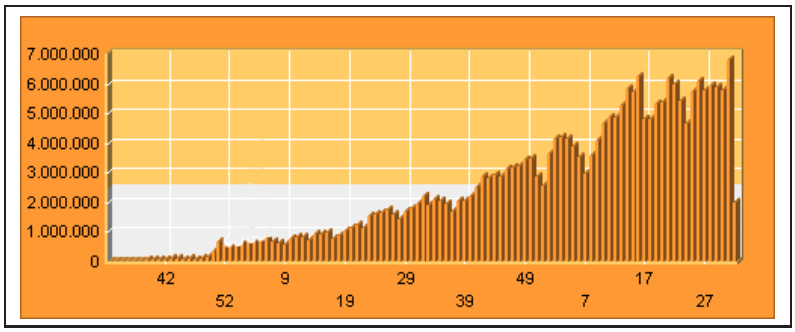

Figure 1.4: **Page view statistics for the last 24 months (generated at 9th of August 2006), as measured by OneStat [ONESTAT]. The numbers in the graph should be multiplied by 10, because OneStat, an external page-view statistics producer, does not support large amounts of page-views. Every bar corresponds to the total number of page views for a week.**

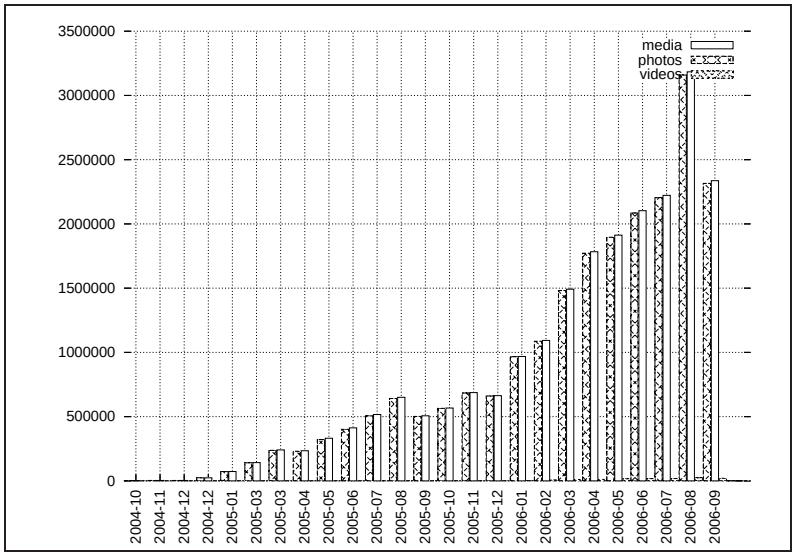

Figure 1.5: **Number of uploads measured approx. per month as measured on the 15th of September 2006. Media: total number of uploads (video and photos). Video: number of video uploads. Photos: number of photo uploads. Note that the results for the last month are not definite, because September had not ended yet at the creation of this graph.**

- the number of users increases and therefore the number of potential uploads increases too;
- the number of media types is expected to increase, therefore increasing the likelihood that a user may upload even more items.

Each uploaded and stored media item takes space in the storage of the Hyves media service. Items are also processed. The processed media items are also stored in the storage of the Hyves media service.

The increase of the number of uploads demands for a scalable storage architecture. A first version of such a storage architecture is currently in use, but it is not expected to be scalable enough.

1.3.2 Media Growth Processing Challenges

In the Hyves service, every media item is processed. For example:

- photos are rendered to different geometries into the JPEG-format to fit in nicely into the layout of the Web site;
- videos are downsized and rendered to the FLV-format so that they are accessible in the Web site. Also a JPEG-thumbnail is rendered from the video for the Web site.

Processing the uploads takes time and processing capacity. At the beginning of this research, items not yet processed (those are queued to a render daemon), they were rendered instantly on a Web server. The rendering process takes time, so the user experienced a delay between the request and the result. The rendering process also takes CPU-power of a Web server, that otherwise could process more requests. To improve user experience, the time between the upload of the item and the availability on the Web site should be as low as possible.

When the number of uploaded items increases, the processing capacity demands also increase. This demands for a scalable processing service. The current processing service is not expected to be scalable enough for the future growth.

1.3.3 Media Growth Serving Challenges

Visitors of the Web site (members and non-members) can download media items. Each newly uploaded item has the potential of being downloaded multiple times. Because of the increase of the number of uploads the number of downloads is also likely to increase. To handle the increased number of downloads, the capacity has to be extended. This demands for a scalable download service.

1.4 Objectives

We formulate our overall research question based on the motivation and the context of this thesis as follows:

> *Which architectural principles should be applied in the design of a large scale Web site?*

In practice, large scale Web sites' architectures conforms to the architecture of a typical distributed information system [ALONSO], which consists of three layers: a resource management layer, an application logic layer and a presentation layer. Furthermore, the challenges in Section 1.3.1, 1.3.2 and 1.3.3 fit into this division. Therefore to answer our research question, first the following sub-questions need to be answered:

> *What architectural principles should be applied in the design of a scalable and redundant storage architecture for a large scale Web site?*
>
> *What architectural principles should be applied in the design of a scalable and redundant processing architecture for a large scale Web site?*
>
> *What architectural principles should be applied in the design of a scalable and redundant static content serving architecture for a large scale Web site?*

To answer these sub-questions, we set ourselves the following objectives:

1. Make the high-level architecture of Hyves service explicit in terms of functions and components. Give a more in-depth view of the Hyves media service that reflects the initial situation at the beginning of our research and which allows analysis and evaluations;
2. Identify the architectural principles and decisions that have been applied in the design of the large scale Web site of our case study;
3. Identify current and future bottlenecks of the Hyves media service;
4. Design an improved Hyves media service architecture;
5. Identify the architectural principles and decisions that have been applied in the improved design of the media service architecture.

1.5 Approach

The first and the second objective are achieved by studying the initial architecture and implementation as found at the beginning of our research. In this study, we perform these two steps:

- Define the Hyves service architecture in terms of its parts and their relations;
- Define the Hyves service architecture in terms of components and their relations.

- Identify the architectural principles that were applied at the design and the implementation of the current architecture and its implementation.

The third objective is achieved by gathering log data from the machines and by processing this data using utilities specially written for this purpose. We also wrote utilities to generate graphs from this data. Subtasks of the third objective are:

- Identify the information flow in the Hyves media service;
- Identify current performance problems by using measurements;
- Identify future performance problems using measurements.

The fourth objective is achieved by researching alternatives for the current architecture and to provide a design ready to implement. Subtasks of this objective therefore are:

- Analyze the characteristics of the identified performance problems;
- Create requirements for the possible solutions, using the characteristics of the performance problem;
- Propose possible solutions and select the best one or design a solution;
- Describe how the solution should be implemented;
- Implement the solution;
- Evaluate the solution with respect to the problem.

The fifth objective is performed by studying the identified problems and the designed solutions for these problems. Subtask of this objective therefore are:

- Identify the architectural principles by using the identified performance problems found in the current Media service architecture and its implementation;
- Identify the architectural principles that have been applied in the design of the solutions for the identified performance problems.

1.6 Restrictions

Before starting the research, we would like to identify six restrictions which should be taken into account while reading this thesis.

While using a case study we gain deep insights in large scale Web sites. However, case studies in general have some restrictions, therefore the identified architectural principles in this report may not hold for a generic large scale Web site but at least for the Hyves Web site. We encourage readers to implement the architectural principles gained in this thesis according to their own situation.

Our case study research has to take place in a dynamic environment: the current architecture of our case study is still work in progress. Any architectural improvement done by us should be done with respect to a minimum of downtime and a minimum of impact for the other parts of the service. This might influence possible measurements.

At the start of the research, the media service of the Hyves service supported photos only, but currently it supports both photos and videos. Therefore the research focuses only on the architectural principles of a design of a photo-based media service. A video and photo based media service compared to a photo-only based media service requires a different setup, but nonetheless, it is expected that most architectural principles that are derived from our research are also applicable for a video and photo-based service. We

expect this, because the current work flow (storing, processing, serving) for videos is comparable to the work flow for images. Further, at the moment the videos are rather small and the file sizes are comparable to some of the images, therefore storing and serving the videos is somewhat similar to serving and storing images. Of course this has to be reconsidered when larger videos are allowed.

Further, because of the performance issues, any identified and solved bottleneck during our research are implemented as soon as possible to allow a further growth and adoption of the service. As a consequence, we will do our research in an incremental way.

Our design of an improved architecture should reflect the architectural principles found in our research.

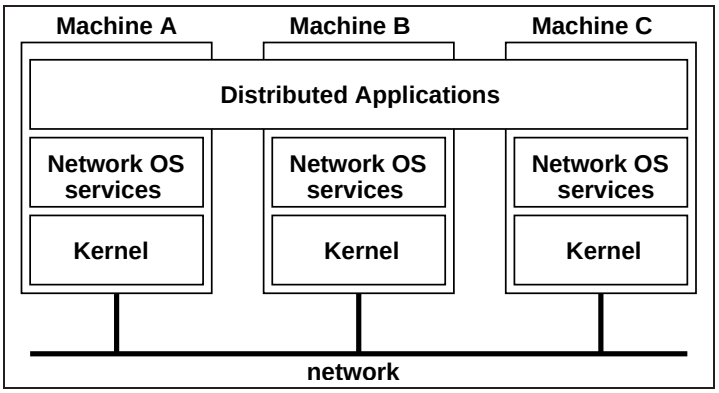

Figure 1.6: **Different levels to apply scalability and redundancy. We focus on the distributed application. This image was taken from [DSYS].**

In the introduction, we defined scalability and redundancy. In our research questions, we state that we want to identify principles for the design of a scalable and redundant architecture. Scalability and redundancy are widely used understandings and both can be applied extremely and at different levels (kernel, network, application, see Figure 1.6). We focus on the distributed application the large scale Web site provides and assume that supporting subsystems like the network and the operating system are a solid base for the distributed application.

1.7 Structure of this Document

This thesis is further structured as follows:

Chapter 2 gives an overview of the Hyves service architecture and its implementation. In this chapter we provide a high-level overview of the Hyves architecture explicit in terms of functions and components. Furthermore, we provide an in-depth view of the Hyves media architecture that allows analysis and evaluation to identify the architectural principles and decisions that have been applied in the design of the Hyves Web site.

In chapter 3 we present the implementation details for the Media Service architecture. Furthermore, we present the interactions with the service, the internals of the service and the implementation details of the service. These service implementation details are the starting point for our analysis.

Analysis and design in a typical research are executed in order and therefore described in single chapters. In our research, we worked using an incremental approach: research leads to a design, and this design needs to be improved, so more research is necessary and so on. Further, we have three problem areas: media storage, media serving and media processing. In our research we experienced that media storage and media serving are two problem areas that are closely related. Therefore we combine these two problem areas in one chapter. In chapter 4 we present our analysis and design activities for Media Serving and Media Storage. Chapter 5 presents our analysis and design activities for Media Processing.

We conclude our thesis by answering our research questions and presenting an overview of all the architectural principles identified during our research in Chapter 6. Further we present some recommendations that can be useful for Hyves.

Next to the thesis, an appendix that shows the utilities that we have written to aid our research is attached to this document.

Chapter 2

Architecture

In this chapter we provide a high-level overview of the Hyves architecture explicit in terms of functions and components. Furthermore, we provide an in-depth view of the Hyves media architecture that allows analysis and evaluation to identify the architectural principles and decisions that have been applied in the design of the Hyves Web site. Therefore this chapter discusses the situation as found at the beginning of our research.

We start the high-level overview of the Hyves service by decomposing the architecture in layers and discuss each of them. Thereafter, we give an overview of the implementation of the Hyves service. After this, we provide a more in-depth overview in terms of functions and components of the Hyves media service. This chapter concludes with the architectural principles that were applied when the discussed architecture was designed and implemented.

2.1 Initial Architecture

Like typical distributed information systems, our case study's architecture conforms to a three-tier architecture [ALONSO] with a resource management tier that provides access to resources like storage, databases and interfaces to other systems, an application logic tier that combines the resources, and a presentation tier that enables interaction between the users and the application. Figure 2.1 displays this layered architecture of the Hyves service. In the following sub sections we present the parts of each tier.

2.1.1 Resource Tier

In the resource management tier, several resources can be found (see Figure 2.1). These resources are:

- The management database resource that holds configuration data for multiple parts of the application;
- The rendered media resource, which is a repository of files containing the rendered media items;
- The media resource, which is a repository of files containing the original media items;
- The member database that holds all data for the members of Hyves. For example, the information put into user's profile is stored in this database;
- The contact database that holds the address books of the Hyves members (imported from Hotmail, Gmail, etc.);
- The friend invitation database that holds all unconfirmed Hyves invitations to possible new members of the Hyves service;
- The Search database resource that stores all Search information. This database is used for implementing the search function in the application.

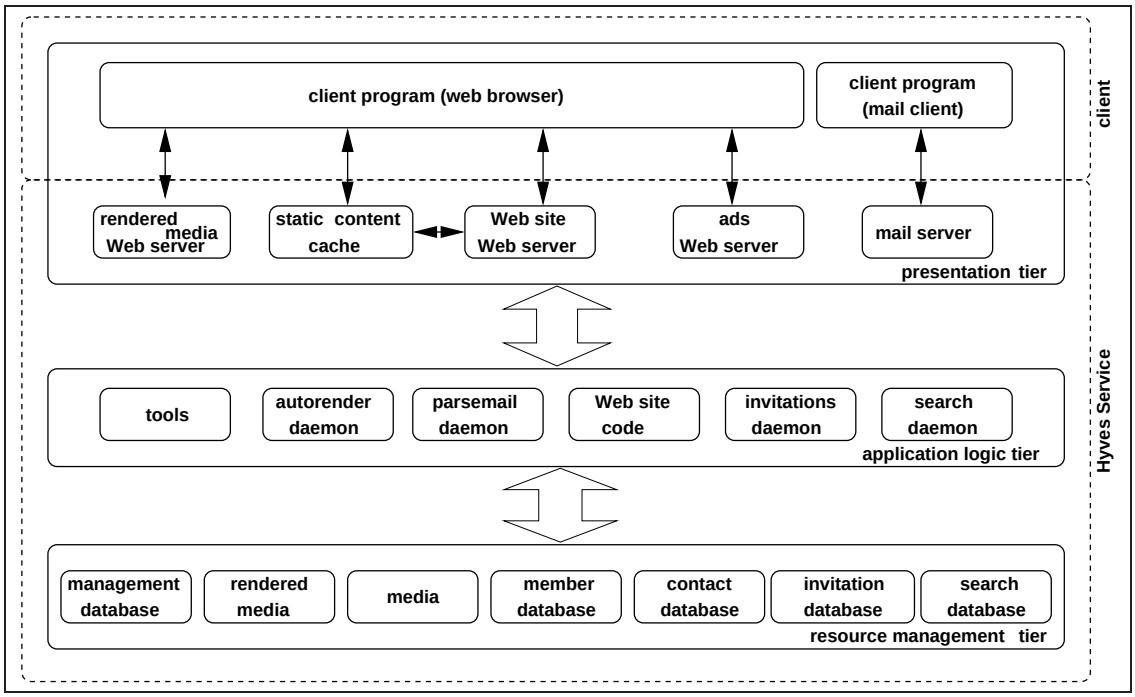

Figure 2.1: **Three-tier architecture of the Hyves service**

2.1.2 Application Logic Tier

The application logic tier contains all application logic. It uses the resource management tier, and serves the presentation tier (see Figure 2.1). The components in the application logic tier are:

- Tools, which is a collection of scripts and programs that are used to maintain the application;
- Autorender daemon, which is a process that renders all uploaded media items. It uses the media resource and stores the rendered images on the rendered media resource. It uses the member database resource to obtain data on the uploaded media items;
- Parse-mail daemon, which is a process that parses all the e-mails sent to the Web site. Users can send e-mails to the Web site for (mobile) blogging. The daemon reads the mails and insert the blogs into the database. If attachments are included, they are stored at the media resource;
- Web site code is a collection of mostly PHP-scripts that renders the Web site. These scripts use all databases and interact via Web servers with the members of the service;
- Invitation daemon, which is a process that sends an invitation message to a possible new member;
- Search daemon, which is a process that updates the Search databases periodically so the Search databases contains actual information. It uses the member database.

2.1.3 Presentation Tier

The presentation tier contains all services that interact with the client programs like Web browsers and e-mail clients. (see Figure 2.1). On the server side of the presentation tier, the following parts exists:

- Rendered media Web server, which serves all the rendered media to the members;

- Static content Web cache server, which serves all static content to the members. Examples are static images such as Smiley's and Java-Script sources;
- Advertisements Web server, which serves all the advertisements to the members;
- Web site Web server, which serves all dynamic content to the members. Examples are the Web site's home page but also the pages containing the member profiles;
- Mail server, which handles e-mail interaction. For example, mobile blogging and member invitations use e-mail as a transport medium.

For the client side of the presentation tier, Hyves assumes that users use Web browsers and e-mail clients to access the service. The Web browsers must allow the use of cookies [COOKIE], JavaScript [WIKIJS] and Flash [FLASH].

2.2 Implementation

This section discusses the implementation of the components that we presented in the previous section. First, we discuss the implementation of the components of the three tiers, and then additional implementation details.

2.2.1 Resource Tier

The Management database uses MySQL [MYSQL] and runs on one host.

The rendered images usually have a smaller file size than the original images, so the storage capacity of the Rendered Media repository is smaller than the storage capacity of the Media repository. The Rendered Media repository is stored on two volumes. The first volume contains all rendered media items with $0 \leq id \leq intervalboundary$, where `id` is a unique number assigned by the application. This first volume is called "interval 0" or "first interval". The second volume contains all rendered media items with $intervalboundary < id$. This volume is called "interval 1" or "second interval". Both volumes are redundantly stored on two disks and each disk is placed in one server each, for performance and redundancy. These volumes are not exported, because the applications serving the media items are located on the same servers. When there is demand for more storage, either another interval is introduced to store more media items or the volumes are moved to disks with a larger capacity.

For the Media repository, network attached storage (NAS) [WIKINAS] is currently used. The NAS exports its file system to all Web servers in the cluster. Previously, when more storage capacity was needed, the NAS was replaced by another NAS with a larger capacity. Currently, when more storage capacity is needed, media items with a low media `id` are moved to disks on other machines. The moved media items remain accessible via the NAS by means of NFS [WIKINFS] shares and symbolic links. This process of expanding the storage capacity is displayed in Figure 2.2. Figure 2.2 (a) displays the initial single NAS. This NAS is replaced with one with a larger capacity (see Figure 2.2 (b)). Figure 2.2 (c) displays the results of moving media items to an external disk. These media items remain accessible via the NAS using a NFS link between the NAS and the server with the external disk. Figure 2.2 (d) displays the next iteration of moving of media items.

The Member Database is implemented using a master-slave setup. The master handles all database write transactions and the slaves handles all the data requests. When more performance is needed, the number of slaves is increased. When more storage capacity is needed, the disks in the database servers are replaced with disks with a larger capacity or the database is divided to a setup with more masters and a set of slaves for each master. The Contact and the Friendinvitation Databases are implemented without a master-slave setup. If there is a demand for extra capacity for these databases, a master-slave setup is will be introduced. The Search Database is implemented using three redundant machines without a master-slave setup.

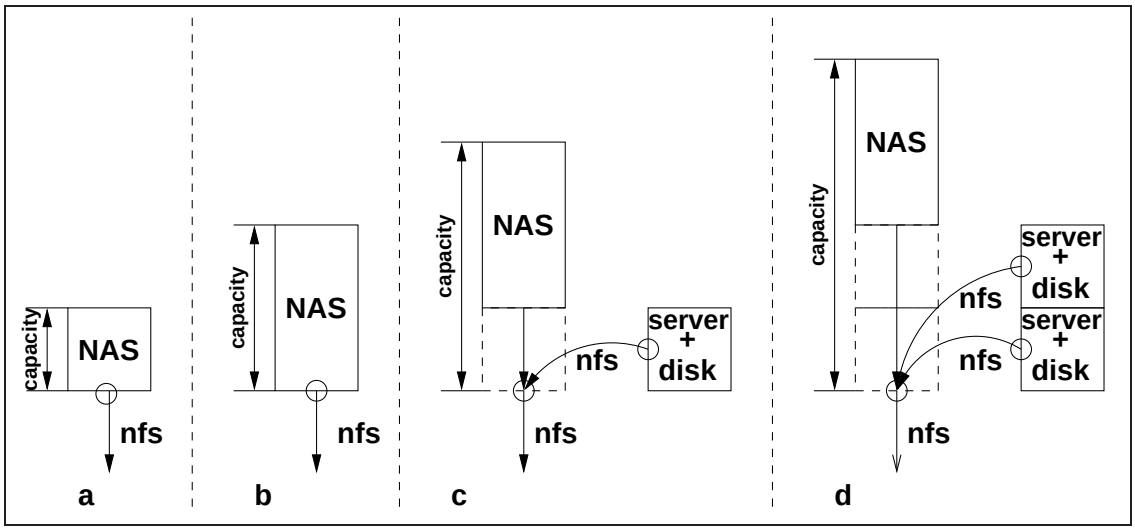

Figure 2.2: **Current storage capacity expansion: (a) original NAS; (b) larger NAS; (c) and (d) NAS and disk(s) with moved media items.**

2.2.2 Application Logic Tier

The "Tools" collection of scripts is written in Perl [PERL], Bash [BASH] and Awk [AWK]. They are executed either periodically (using Cron [CRON], e.g., cleaning up log files, making back-ups) or instantaneously (at the command line, e.g., when deploying the code).

The Autorender daemon [DAEMON] is written in PHP [PHP] and runs on one of the Rendered Media Web servers. It peeks into the Member Database to discover newly uploaded media items. These items are then rendered using the `convert` command from the ImageMagick software suite [IMGMGK]. Newly rendered media items are then stored in the Rendered Media repository.

The Parse-mail daemon is written in PHP [PHP] and runs on a machine dedicated to daemon processes. It receives e-mails from Hyves members, parses them, and converts them into content, and stores the content in the Member Database. If the parsed e-mail contains attachments, then these are stored in the Media repository and the Member Database is altered so the Autorender daemon can render these items.

The Web site code is written in PHP [PHP] and runs on Web site Web servers. It uses all the databases to generate the dynamic content for the Web site.

The Invitations Daemon, which is written in PHP, runs on the machine dedicated to daemon processes. From the information stored in the Invitation Databases, it generates messages to (non-) Hyves members. The messages are forwarded to the Mail server.

The Search daemon uses the Member database to generate information for the Search databases. Therefore it disconnects one Search database server from the Web site, fills it with new search information and connects it again with the Web site. Every three hours one Search database server is updated.

2.2.3 Presentation Tier

The Web server for the rendered media is implemented using Apache [APACHE]. This Web server runs on 4 redundant hosts (two per interval). When there is a demand for more performance, the number of Web servers per interval is increased.

Static content cache is a caching reverse proxy implemented using Squid [SQUID], which caches the static content. This caching reverse proxy runs redundantly on two hosts. When there is a demand for more performance, the number of Web servers is increased.

The Web server for the advertisements is implemented using Apache [APACHE]. This Web server runs on a single host. When there is a demand for more performance, the number of Web servers is increased.

The Web server for the Web site's dynamic content is implemented using Apache [APACHE]. This Web server also serves static content to the Static content cache. This Web server runs on about 40 redundant hosts. The number of hosts is regularly increased to increase the maximum number of requests.

The Mail server is implemented using Postfix [POSTFIX]. This runs on a single host. Currently, there is no perspective on how to scale this service when more performance and/or storage capacity is needed.

2.2.4 Vertical Sub-systems

The overview of the architecture parts of Figure 2.1 given so far, is not complete, because additional techniques are used to implement the architecture. These additional techniques are presented in this section.

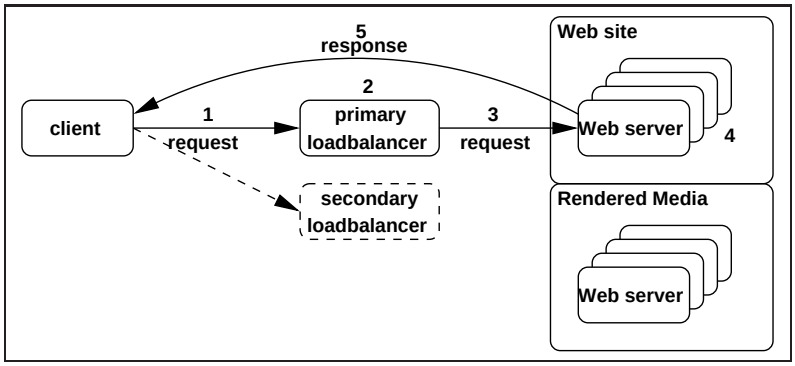

Figure 2.3: **Load balancer setup. (1) client sends a request to a Web server cluster. (2) load balancer intercepts the requests, determines cluster from request and chooses a cluster node. (3) requests is forwarded to chosen cluster node. (4). Cluster node Web server handles requests. (5) the response is send directly to the client.**

All the Web servers are load balanced using IPVS [IPVS]. The load balancer software runs on two hosts: a primary and a secondary load balancer. The secondary load balancer takes over when the primary load balancer fails. These load balancers direct requests designated at different clusters to the correct pool of servers. The response returns directly to the client. The load balancer maintains a list of connections between clients and Web servers. When a client makes consecutive requests and the time intervals between any consecutive requests are not larger than a certain period of time, these consecutive requests are forwarded to the same Web server. If the time interval between two consecutive requests exceeds a certain period of time, the load balancer again elects a server to forward the request to. This is displayed in Figure 2.3. The clusters currently defined and load-balanced are summarized in Table 2.1.

cluster's DNS entry	cluster function	load balanced IP address
*.hyves.nl	dynamic Web site	213.193.242.126
interval0.rendered.startpda.net	rendered media interval 0 Web servers	213.193.242.128
interval1.rendered.startpda.net	rendered media interval 1 Web servers	213.193.242.130
cache.hyves.nl	static content Web cache	213.193.242.123
clarice.startpda.net	advertisements Web server	213.193.242.127

Table 2.1: **Load balanced clusters**

The Web site Web servers are connected to the pool of Member databases slaves for requesting information from the Member database and to the Member database master for storing information into the Member database.

The NAS is not only used for storing the media items, but also for the database backups and the code (Web site as well as Tools).

System-level monitoring is available to determine statistics (e.g., disk utilization, database on host A is running, Web server on host B is running, etc.). System-level monitoring currently runs on one host. On the same host, also application-level monitoring is running to gather statistics about running processes like MySQL, Apache and Squid.

2.3 Media Service

The Media Service part of the Hyves architecture consists of Web servers that serve the rendered media items, file servers that store the rendered and the original media items and the Autorender daemon that processes media items. These components are depicted in Figure 2.4.

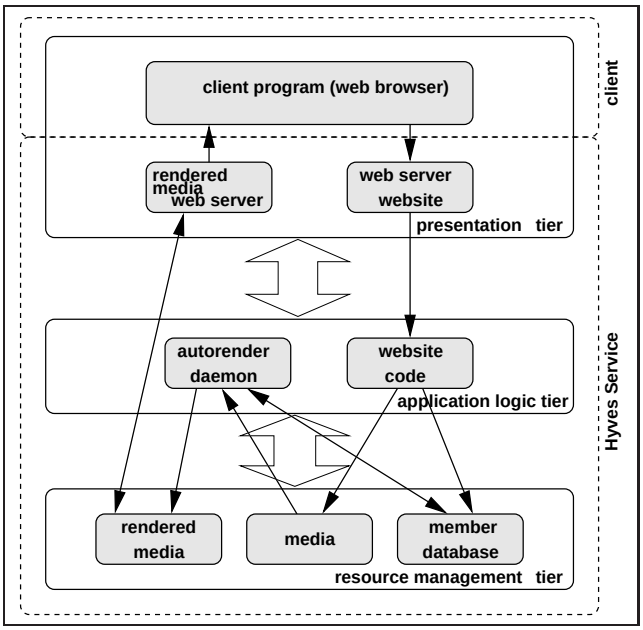

Figure 2.4: **Media Service in the architectural view**

The media service performs the following functions:

- uploading of media items (Web site code, Member database);
- storing of media items in the Media repository (Media repository);
- processing of media items into rendered media items (Autorender, Member database);
- storing of rendered media items in the Rendered Media repository (Rendered Media repository);
- serving the rendered media item requests (Rendered Media repository, Rendered Media Web servers, Web site code, Member database).

The flow of the functions of the Media service is depicted in Figure 2.5.

The next sections give an overview of the functions mentioned above and how the components of the Media Service (see Figure 2.4) interact to perform these functions.

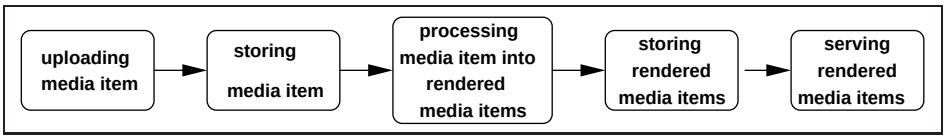

Figure 2.5: **Media Service flow. A media item is uploaded. The media item is stored in the media repository. The Autorender Daemon renders the media item into the rendered media items. These are stored in the Rendered Media repository, and thereafter, they can be served to clients.**

2.3.1 Media Uploading and Rendering

Uploading of media items can happen on each Web server in the cluster. If the uploaded items file type is recognized and accepted, the media item is assigned a media `id` (which is stored in a database) and the item is copied to the Media Repository using the media `id` as file name. Because of the interaction the Web site offers the uploader of the file, the file is directly rendered into 75x75 format. This is achieved by pushing the Web browser of the uploader to refresh its page for one in which the result of the upload is showed, using this rendered image. A sequence diagram of this scenario is given in Figure 2.6.

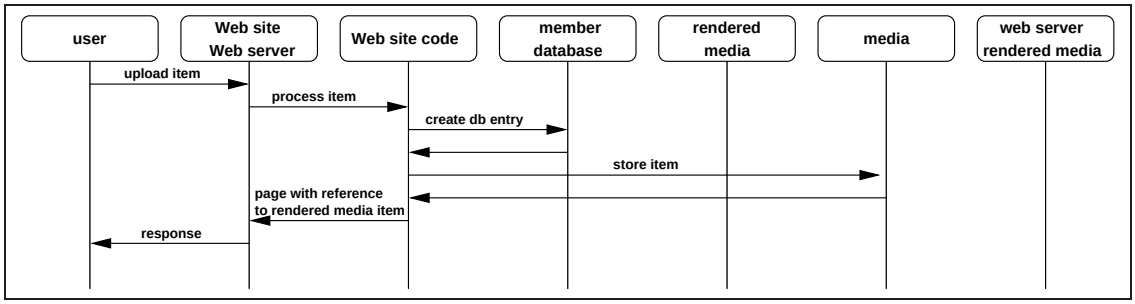

Figure 2.6: **Sequence diagram of the scenario in which a media item is uploaded**

The Web browser therefore requests a Web server to serve the media item. If the media item is not yet rendered (if its media `id` is above the `coldmaxmediaid` `id` and if it cannot be found in the Rendered Media Repository) the Web server renders the image only for the requested resolution. Therefore it loads the media item from the Media Repository, renders the item, stores the item in the Rendered Media Repository and serves the item to the requester. This process is displayed in the sequence diagram in Figure 2.7.

When a media item is uploaded, is it appended to the “hot” part of the Rendered Media repository. The “hot” part is defined as the part of the repository that the Autorender daemon is to process. The “cold” part is defined as the part of the repository that the Autorender daemon has processed already.

Figure 2.8 displays this “hot and cold” setup.

If a media item was processed by the render daemon, its media `id` (a by the application assigned unique number) it is below the `coldmaxmediaid` pointer. When a user requests such a media item, the following process occurs. The Web site’s code checks if the media id is below the `coldmaxmediaid`. If so, then the Web site’s code generates the URL for the media item. For this, it first checks the media id of the requested media item. If the media id is below the interval boundary, it generates an URL for the media item designated at the first interval. If the media `id` is above the interval boundary, an URL for the media item designated at the second interval is generated. We display this process in the sequence diagram of Figure 2.9.

If a media item is not processed by the render daemon, but already by the site, the following process occurs. The user requests a Web page. The Web site’s code checks the media `id` of the media item. If the media `id` is above the `coldmaxmediaid` pointer, the Web site’s code checks if the rendered media item exists in the rendered media repository. If so, the Web site’s code generates an URL. The client requests the URL and the rendered media item is collected from the Web site directly. This process is displayed in Figure 2.10.

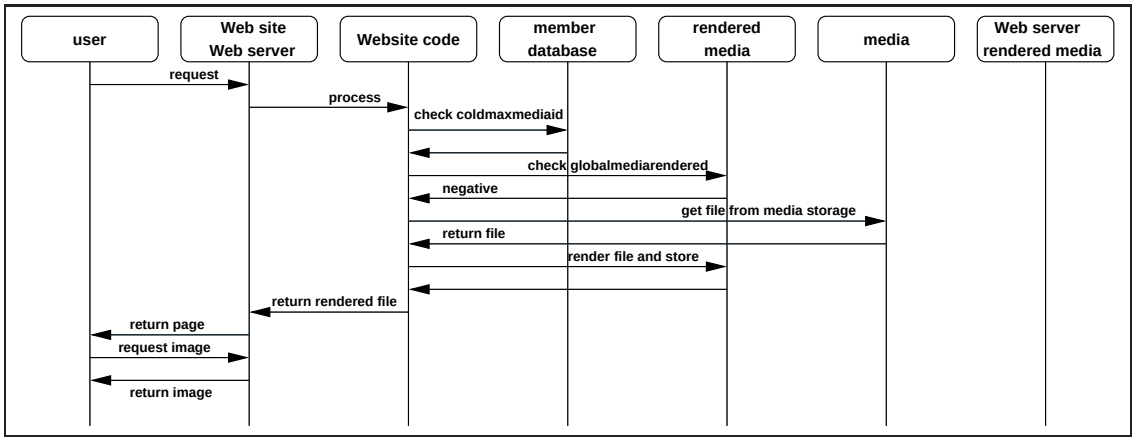

Figure 2.7: **Sequence diagram of the scenario when a request is made for a media item that is in the hot part of the Rendered Media Repository and is not rendered yet.**

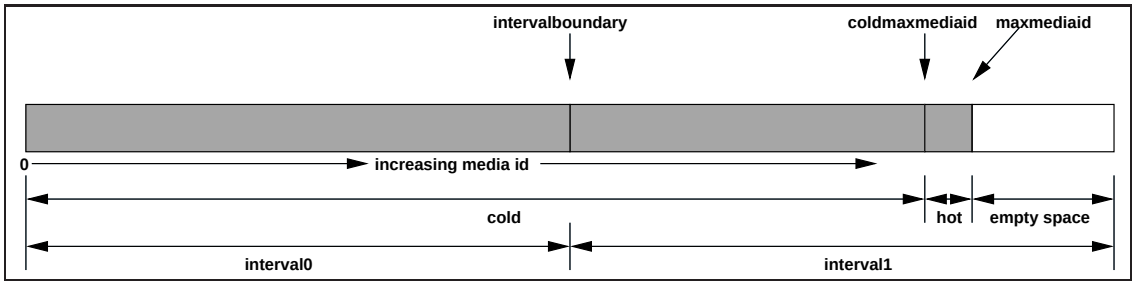

Figure 2.8: **Hot and Cold rendered media repository structure**

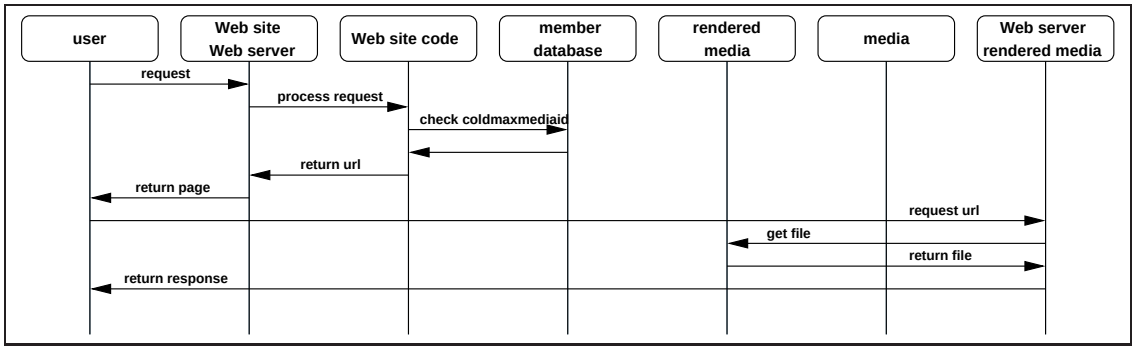

Figure 2.9: **Sequence diagram of the scenario when a request is made for a media item that is in the cold part of the Rendered Media Repository.**

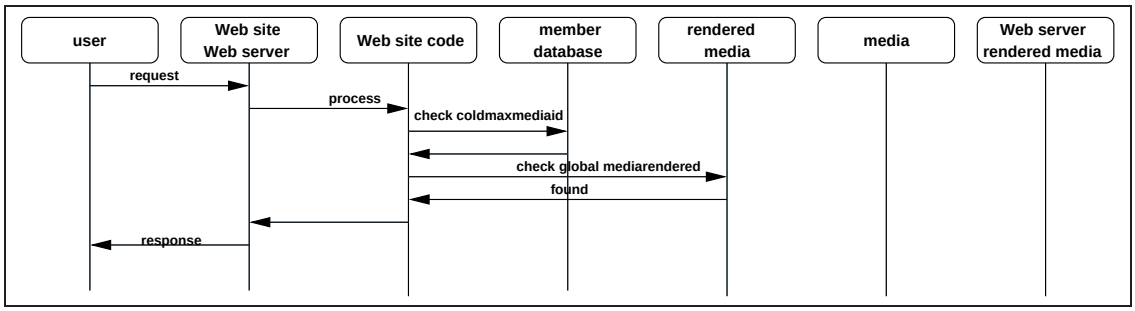

Figure 2.10: **Sequence diagram of the scenario when a request is made for a media item that is in the hot part of the Rendered Media Repository and is already rendered.**

All these scenarios are summarized in Figure 2.11

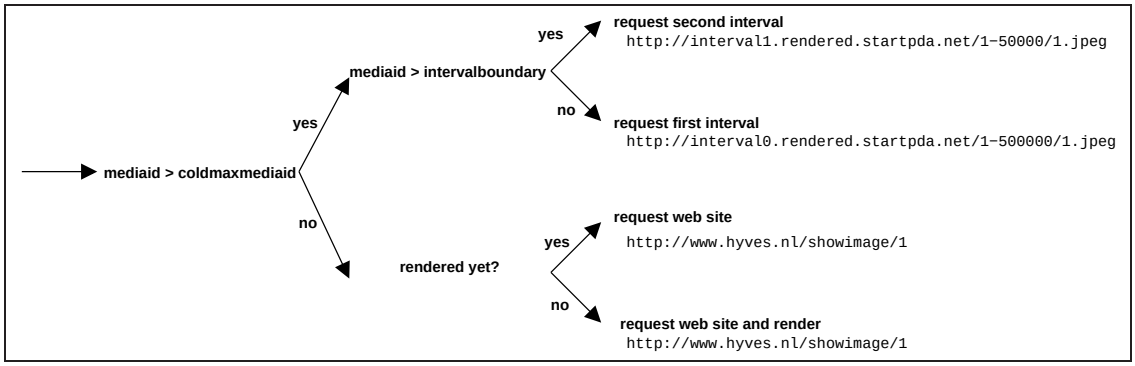

Figure 2.11: **Summary and URL generation scheme.**

Simultaneously the auto-render daemon is running. This daemon checks for newly uploaded images by comparing the value of `coldmaxmediaid` and `maxmediaid` and renders them into certain formats depending on their file type. This process is displayed in the sequence diagram of Figure 2.12. An overview of the file types is given in Table 2.2. After rendering, the rendered media items are stored in the cold part of the Rendered Media Repository and the so called cold pointer (`coldmaxmediaid`) is increased. The auto-render daemon renders the next image. The rendering process is done sequentially by media `id`.

2.3.2 Media Storage

In Section 2.2 we argued that there are two repositories for media, namely a Rendered Media repository and a repository for the original media items.

The original media repository is stored on a NAS and exported via NFS [WIKINFS]. Media items are sequentially stored in directories per 50.000 media `id`'s. In the case of Rendered Media, this implies that about 300.000 media items are stored per directory; in the case of the Media repository, 50.000 media items are stored per directory.

The interval boundary is always defined per directory of 50.000 items.

The rendered media repository is stored (as stated) on two volumes, and those volumes are stored on two machines per volume.

We have analyzed that on average, the aggregate file size of the rendered media items is about one sixth of the original media file size.

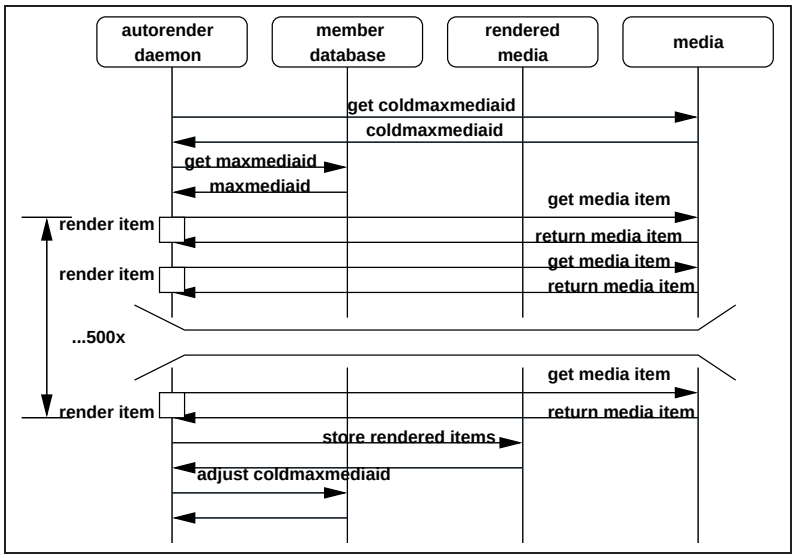

Figure 2.12: **Sequence diagram of the Autorender daemon.**

file type	render format
image	JPEG 50x50 JPEG 70x70 pixels JPEG 120x120 pixels JPEG 200x200 pixels JPEG 500x500 pixels JPEG 700x700 pixels
video	FLASH movie JPEG 50x50 pixels
PDF	JPEG 50x50 pixels

Table 2.2: **Render formats per uploaded file type**

2.3.3 Media Serving

Because of the amount of requests and media items, media serving is done by two clusters of two Apache Web servers, load-balanced by using IPVS.

The first cluster of Web servers serves the domain `interval0.rendered.startpda.net`. This cluster serves only the media items that are located in the first interval. The second cluster of Web servers serves the domain `interval1.rendered.startpda.net` and serves only the media items that are located in the second interval.

Each Web server has its own copy of the media items of the interval it serves. The media items are served with an HTTP expiration date of one year after the request to allow the users Web browser to cache the media items.

The Web site PHP code generates the URLs (see Figure 2.11) for the media items. If a media item's `id` is above the `coldmaxmediaid`, the media item is requested from a dynamic content Web server because the media item is in the hot part of the Rendered Media Repository. This Web server checks if the media item is already rendered. If not, the Web server renders the media item, stores the media item in the Rendered Media Repository and serves the request to the client (Figure 2.7). If perhaps another Web server has already rendered the media item, the media item is server by the Web server (Figure 2.10).

If a media item's `id` is above below the `coldmaxmediaid`, the generated URL provides that the Web browser requests the media item via the media-rendered Web servers (Figure 2.9).

2.4 Architectural Principles

This section presents our observations of the Hyves Web site architecture and based on these observations, the architectural principles for large scale Web sites are defined.

2.4.1 Prevent Single points of Failures

We have observed several single points of failures in the Hyves Web site. Examples are the NAS, the Mail Server, the advertisement Web server and a single daemon server. If the NAS fails, there is no access to backups, the original media items and the Tools repository for maintaining the Web site. If the Mail server fails, the Hyves Web site is unable to receive e-mails from clients or to send e-mails to clients. If the advertisements Web server fails, no advertisements can be showed. If the daemon server fails, several processes such as the Friendinvitation daemon stall.

Because not all these services that are provided by these parts have to be accessible at every moment in time, these services are not considered as critical. A single point of failure is only critical if the service is essential for the large scale Web site.

At the beginning of our research, the Autorender Daemon was a single point of failure, but not very critical. The Autorender Daemon was capable of processing a backlog caused by downtime within half a day and at the beginning of our research, there were lesser page views and therefore the impact of the Web site rendering media items on demand was less when compared to nowadays. When the number of users, the number of page views and the number of uploads increased, the impact of the Web site rendering media items was increased and the Autorender daemon was no longer able to process a backlog within half a day. The Autorender Daemon service became critical.

The Mail server, the advertisements Web server and the daemon server are not very critical at the moment. It is expected that these services become critical as well, when performance demands increase because of a growth in users or page views, because all these services provide a sort of processing service, similar to the Autorender Daemon.

To prevent this, there are three options. The first option is to increase the performance of the service. In case of failure, this reduces the time to process the backlog caused by the service stopping from running.

Increasing the performance also reduces the chance of the service becoming a bottleneck. Note that this solution only minimizes the effects of a failure and not prevents the service for failure.

The second option is to require that for each service, there are at least two instances, running on different machines. In case of a failure on one machine, the instance on the other machine continues to run the service. This prevents or reduces the length of a possible backlog. As a consequence, the service should allow multiple instances simultaneously and thus adds complexity to the design to prevent concurrency problems. Another benefit from this option is that the performance of the service is the aggregate performance of the multiple instances. If there is demand for more capacity, more instances can be started. Note that it is possible to run multiple instances from different services on one machine.

A third option is to design the service more portable. In case of failure, the service can be installed/started on another machine and the service remains accessible.

As an architectural principle for designing large scale Web sites, we conclude that supporting services within the large scale Web site should be prevented from being single points of failures, even when these services are not critical at the moment. Three possible solutions were discussed: increasing performance to minimize effects of a failure, design redundant instances for the service and design the service with portability in mind.

2.4.2 Loose Coupling

Although the Hyves Web site can be placed into the typical three-tier architecture, some parts are tightly-coupled in the implementation.

The interval-boundary that determines which media `id`s are in the first and second interval is hard coded programmed in the application. This implicates that when the value of the intervalboundary has to be adapted (e.g., because rendered media items should be moved from the second to the first interval), the source code of the Web site's code has to be adapted and redeployed to all Web servers.

Another example is the NAS. It serves as the storage for original media items, but also as a storage for database back-ups and as a file server for the Tools repository. We have experienced that backup-scripts need to be adapted to store their data on other hosts, because the Media repository contained so many media items, that there was no space for storing back-ups. We also experienced that back-ups were removed to create space for the Media Repository.

We think that dependencies between different sub-parts of the Hyves Web site need to be removed. The value of the interval-boundary should be editable without redeploying the code afterward and back-up files should not interfere with storing media items.

Small Web sites often start with one server that performs all the services. the requests are served with a web server which also runs a database for the dynamic content and that stores static content. Backups are made on the same machine etc.. When the Web site evolves in a large Scale Web site, dependencies between parts of the Web site need to be removed as soon as possible.

We consider loose coupling as an architectural principle for large scale Web sites. The examples of tight coupling discussed above limits the growth of parts of the large Scale Web site and therefore should not be implemented.

2.4.3 Application-level Monitoring

We have observed that the Hyves Web site performs fully system-level monitoring and for some applications application-level monitoring. Examples of system-level monitoring are "how much space of the disk is free at the current moment?" and "what is the state of the first network interface of machine X?". Examples of application-level monitoring are "how many queries does the MySQL database on host X currently process?" and "how many requests does the Apache Web server on host X currently serve?". Current application-level monitoring of the Hyves Web site only entails statistics of applications that are not written by Hyves. Examples of applications that were not written by Hyves but are used in the Hyves Web site and are monitored

at application-level are MySQL, Apache and Squid. Examples of applications that were written by Hyves but are not monitored at application-level are the daemons (Autorender, Parsemail) and Web site statistics (total number of media items, total number of members). Within the existing application-level monitoring, there are no aggregate views (e.g., the total number of processed queries by all MySQL servers). We can conclude that only parts of the service are monitored, but not the service as a whole.

A current problem is that at some times, more media items are uploaded per period of time than the Autorender Daemon can render during that period of time. Because of the growth of the number of users, one also expects a growth in the number of uploaded media items. Thus, when introducing a processing service like the Autorender Daemon, one could expect that at a certain moment in time, the number of media items uploaded per period of time overtakes the number of media items processed per period of time.

The problem discussed in the text above illustrates the desire of application level monitoring of applications written by Hyves. If the number of uploaded media items per period of time was monitored, and the number of rendered media items per period of time also, then with these data one is enabled to determine the moment that the maximum capacity of the Autorender daemon will be reached. If this moment could be determined in advance (e.g., when extrapolating the results from the monitoring), one could anticipate and improve the Autorender daemon before the expected problem starts to take effect. This enables proactive maintenance.

Because of a continuous growth of users and sub sites of the Hyves service, one should expect a growth in page views and an increasing demand of capacity. In the above text we have argued that currently there exists no monitoring for the self-written applications used in the Hyves Web site and we have given an example of a problem that could be dealt with pro-actively, if such monitoring had existed. We also argued that because of the growth of the number of users and sub sites, this issue might occur again in the future.

Another recent problem illustrates the desire for monitoring the application as a whole. Earlier in this chapter we mentioned that the rendered media is stored on two intervals. We also mentioned that these intervals are two volumes. With system-level monitoring only, we receive warnings about the disk usage (volume). As a response to these warnings, we moved data from the second interval to the first interval. The second interval disk usage is decreased and we can continue the service a little bit longer. This process is repeated every now and then when the second interval reaches 100% disk usage. Because of the number of uploads doubling every month (which we did not know at that time), the movement of media items followed each other each time sooner. Suddenly it appeared that the first interval was also filling up. If the first interval is full, and the second one also, we cannot move media items from the second to the first interval.

If application monitoring had existed, that monitored the aggregate disk usage for the rendered media and we also used extrapolation, the moment we would reach 100% disk usage could be calculated in advance.

Next to enable detection of future problems, application-level monitoring also allows verification of the applied measures. For example, suppose that as a measure the capacity of a certain part of the architecture is expanded, does it really enable more requests of might there be another problem?

From this, we conclude that a good architectural principle would be to *implement application-level monitoring from the start for every storing, processing and serving part of the service when designing a large scale Web site*.

Note that monitoring the application by a monitoring service is not sufficient. The monitored values should be looked at periodically and the values should be interpreted correctly. Monitoring only takes into account the values of the current moment and in the past, prediction and extrapolation are necessary to increase the value of monitoring.

There exist good solutions for automatic monitoring of pre-defined services [NAGIOS], [CACTI], which are already in use at Hyves. Therefore there is no need to write an entire monitoring solution, but the services one has to write for a large scale Web site should allow monitoring.

Chapter 3

Media Service

3.1 Introduction

In this chapter we present the implementation details for the Media Service architecture. Furthermore, we present the interactions with the service, the internals of the service and the implementation details of the service. These service implementation details are the starting point for our analysis, which is presented in Chapter 4 and 5.

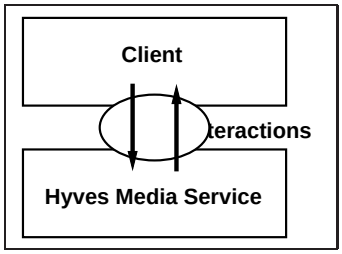

Figure 3.1: **Global View Information Flow**

The service provisioning is constituted by three parts (see Figure 3.1):

- The client that interacts with the service. The client is a Web browser that interacts with the service using the HTTP protocol or the client is a e-mail program that interacts with the service using the SMTP protocol. We do not discuss the client in this chapter;
- The interactions between client and service, which are encapsulated in HTTP requests and responses. We discuss the interactions in Section 3.2;
- The service which replies the interactions from the client using a HTTP Web server. We discuss the service in Section 3.3.

3.2 Interactions

By analyzing the Hyves Web site architecture and its implementation details, we identified the working and the purpose of the system:

- The service has to process requests for media items and as a response it has to serve the requested media items;

- The service has to process uploaded media items. After processing, the processed media items are made available for requests and responses.

From these two functions we can derive three interactions between the client and the service:

- Media Requests (requests for the media items);
- Media Responses (responses to the request, the media items);
- Media Uploads (to transfer user media items to the service).

These three interactions are discussed hereafter.

3.2.1 Request Media Interaction

Description

When a user wants to request a rendered media item, the client interacts a Media Request to the system.

Attributes

The requested rendered media item has a unique `id`, which is a number that identifies the media item. Because a rendered media item is rendered into multiple formats, the request also has to contain the requested format.

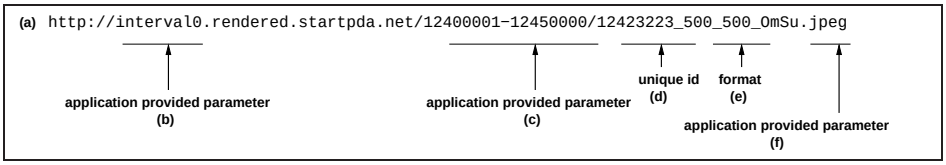

Figure 3.2: **Request**

Next to these attributes, a request also has implicit attributes, such as originator (address of the client that did the request) and the time of the request.

This request is executed as a HTTP request. The HTTP request contains the requested URL (see Figure 3.2(a)). This URL contains all explicit attributes, such as the `id` (Figure 3.2(d)) and the format (Figure 3.2(e)). The URL also contains some implicit attributes, such as the service access point to which the request is directed (Figure 3.2 (b)) and the directory in which the rendered media item can be found (Figure 3.2(c)).

The Web server that receives the HTTP-request logs the HTTP-request in its log files. Therefore we can say that the Media service logs the media requests.

3.2.2 Response Media Interaction

Description

In answer to the media request of the user, the Media Service replies with the use of the Media Response interaction.

Attributes

A Media Response is formed by a media item with the requested `id` and `format`.

Next to these attributes, a Media Response also has implicit attributes: size and time. The response is sent to the client in response to the request that the client has sent before. This response is executed as a HTTP response.

In the case that the Media Service cannot find the requested media item, a HTTP 404 response is sent to the client in response to the request.

3.2.3 Upload Media Interaction

Description

When a client uploads a media item, it uses the Media Upload interaction.

Attributes

Attributes of the Upload Media Interaction are the `type` and the `size` of the Upload.

Next to this intrinsic characteristic of an Upload, there is also an extrinsic characteristic of the number of Uploads in time, because the Upload Media Interaction is executed in a HTTP Post interaction. There are implicit attributes: type (file type), size (file size), time and originator (address of the client that requested the Upload).

We do not consider Upload Media interactions using SMTP or HTTP as separate interactions. We do not focus on the “upload” of the media item, we focus on the effect of the interaction; a media item must be processed and made available for requests and responses.

3.3 Hyves Media Service

In this section, we present the internals of the Hyves Media Service. The components and their relations to other components are explained below. The internal components are displayed in Figure 3.3.

3.3.1 Rendered Media Web server Interval

The Rendered Media Web server receives a Request from the Client. If the requested media item is in cache, the Web server returns a Response immediately. If the requested media item is not in cache, the Web server fetches the media item from the Rendered Media Storage (see Figure 3.4).

The Rendered Media Web server for the first and the second interval have the same internal functioning, but a different storage has been defined for each interval.

3.3.2 Storage

This section entails Rendered Media Storage Interval 0, Rendered Media Storage Interval 1, Media Storage and Rendered Media Global Storage.

The Rendered Media Storage receives requests from the Media-rendered Web-server and returns Responses. It receives Uploads from the Auto-render daemon and stores files (see Figure 3.5).

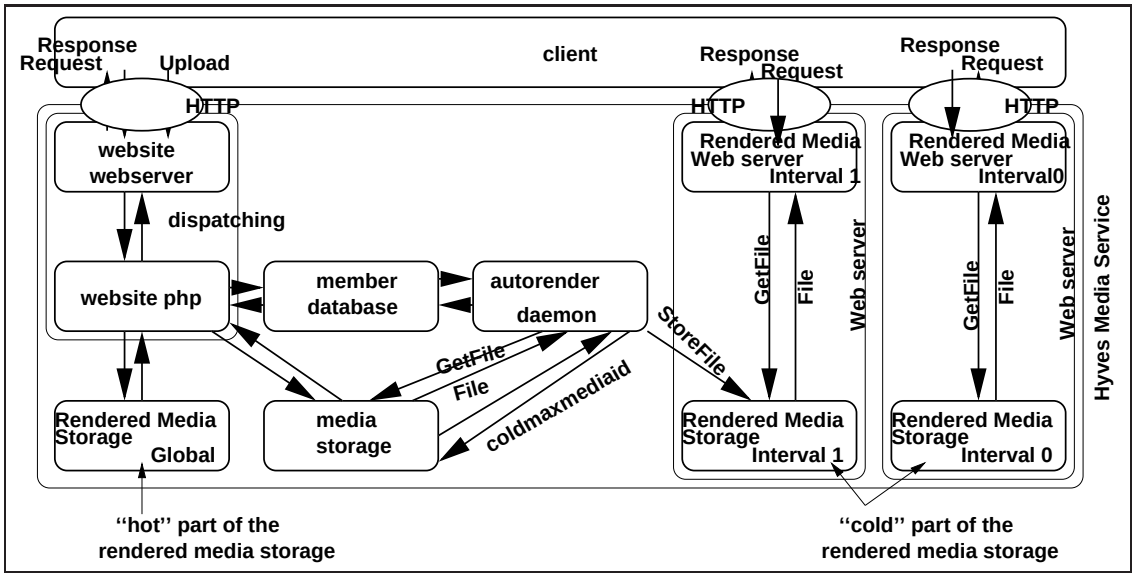

Figure 3.3: **Hyves Media Service internal components.**

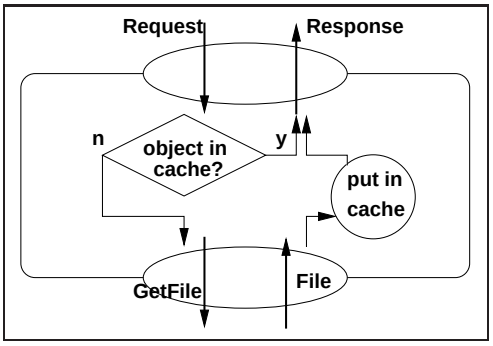

Figure 3.4: **Flow Diagram of a Rendered Media Web Server.**

The Rendered Media Storage internal functioning for both the intervals are equal, but only the second interval receives requests to store media items. There is no essential difference between Media Storage and Rendered Media Storage.

The Rendered Media Global Storage receives requests for media items from the Web site PHP code and returns the media items. The Rendered Media Global Storage also stores the media items rendered by the Web site PHP code.

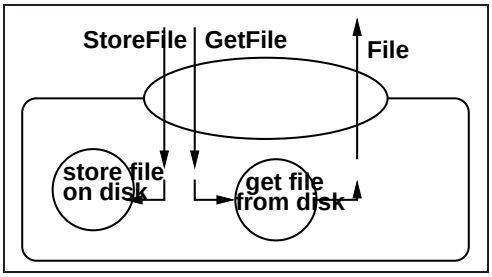

Figure 3.5: Flow diagram of a local storage.

3.3.3 Autorender Daemon

The Autorender Daemon processes uploads when `coldmaxmediaid` $<$ `maxmediaid`. The Autorender requests uploads to the Media-rendered Storage, asks the database for the values of `coldmaxmediaid` and `maxmediaid`, does writes updates to the database, requests Media Storage and receives Responses from the Media Storage (see Figure 3.6).

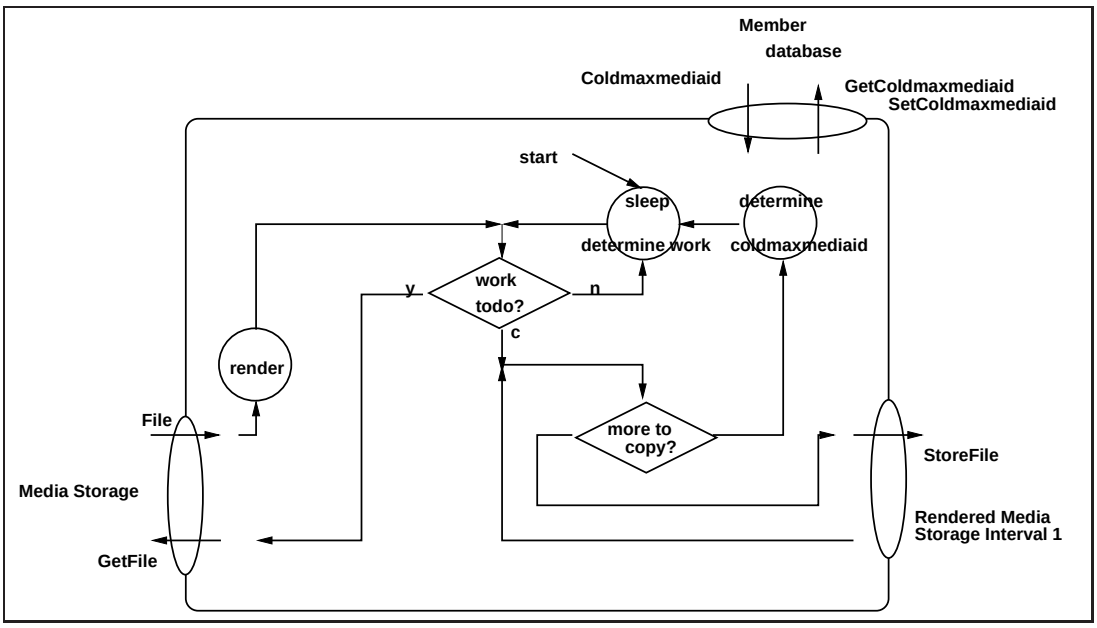

Figure 3.6: **Flow diagram of Render Daemon.**

The Autorender process starts at “start” in Figure 3.6. It asks the database the values of `coldmaxmediaid` and `maxmediaid`. If these values are not equal, the Autorender has to process media items. Otherwise, it sleeps for a certain period of time and the process starts over again.

When the Autorender has to process media items, it requests the Media Storage resource for the media item and renders the media item. If the number of media items to process is more than 500, 500 media items are rendered. If the number of media items to process is less than 500, only that amount of media items is processed.

The “work to do?” process switches to the process of copying the media items. In a loop, each rendered media item is copied to the Rendered Media Storage. When this is finished, the new value for `coldmaxmediaid` is calculated and written to the database. The Daemon then sleeps for a certain period of time and this process starts over again.

3.3.4 Member Database

The Member Database receives requests for queries to calculate `maxmediaid` (which is the `id` of the media item that was most recently uploaded) and `coldmaxmediaid` (which is the `id` of the media item that was most recently rendered by the Autorender daemon). After an Autorender iteration, the Member database also receives requests for queries to update `coldmaxmediaid` (see Figure 3.7).

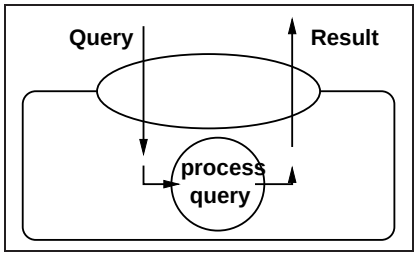

Figure 3.7: **Flow diagram of the Member Database.**

3.3.5 Web site PHP code

When the Web site PHP code receives a Media Request, first it determines if the requested media item is already rendered. If the media item is already rendered, the Web site PHP code returns the media item if it is in cache. Otherwise, the Web site PHP code will request the Rendered Media Global Storage for the media item. Then the media item is put into cache and return as a Response to the Request. If the requested media item is not yet rendered, the Web site PHP code requests the Media Storage for the original media item and renders the media item. This rendered media item is then stored in the Rendered Media Global Storage and then, after putting the media item into cache, served as a Response (see Figure 3.8).

3.3.6 Web site Web server

The Web site Web server dispatches requests and uploads from clients to the Web site PHP engines.

3.3.7 Remarks

In the above service decomposition, there are three access points that use the same service primitives “Request” and “Response”. In a typical service, there would be just one access point for the same primitives.

The three service access points are there because of historical reasons. The access point at the Web site Web server was established for enabling not rendered but requested media items to be rendered in the application. The service access point at Rendered Media Web server Interval 1 was established as a resolution to the problem that one Rendered Media Storage was not sufficient.

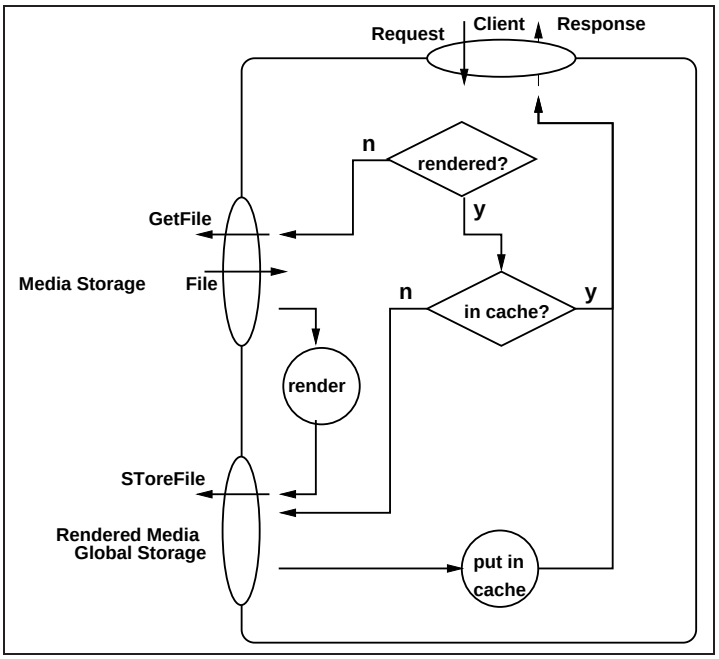

Figure 3.8: **Flow diagram of the Web site PHP code.**

These three service access points implicate that there is additional routing that has been pre-programmed by the application when pages are built by the Web site PHP code. Based on the `id` the request is routed to one of the three service access points (see also Figure 2.11).

3.4 Restrictions

In this service decomposition we do not take the “Delete” service primitive into account.

Chapter 4

Media Serving and Storage

4.1 Introduction

In this chapter we provide an overview of our research and design activities for Media Serving and Storage. We combine Media Serving and Media Storage in this chapter because during our research we experienced that the problems in these two areas are closely related.

In this chapter we present the research on Media Serving and Media Storage scalability problems that are currently experienced within the Hyves Web site. The reasons to research these problems is that some of these scalability problems cause a decrease of the performance of the Hyves Web site to unacceptable low levels and that the current implementation of the Media Storage and Media Serving is expected not to scale well (e.g., inefficient use of resources). Researching these scalability problems might identify architectural principles for designing large scale Web sites.

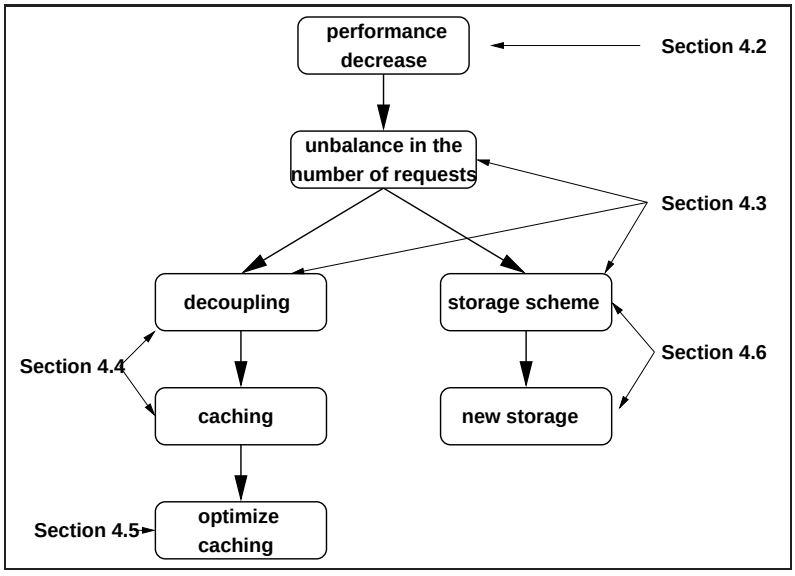

Figure 4.1: **Flow of researches in this chapter.**

In our initial research for Media Serving and Media Storage we focus on a performance problem with Media Serving: the time between requesting and receiving media items was unacceptable long for certain rendered media items. In the first focus (see Section 4.2), this performance problem is researched and solved. The

implementation of the solution to the performance problem raised a demand for further research, because it appeared that the performance problem is just temporarily solved because it is part of a larger problem area: the scheme to store media items on multiple storage nodes. In the second research (see Section 4.3), we focus on this scheme to store media items on multiple storage nodes and the characteristics of the problem were identified and two solutions were proposed: decoupling media serving and media storage and a different storage scheme. Both solutions demanded for more research on how to implement them. These researches are presented in the third (see Section 4.4) and fourth focus (see Section 4.5). This flow of researches is displayed in Figure 4.1. This flow and the next sections in this chapter are presented in a logical order that does not correspond with the order in which these results have been obtained.

For every research in this chapter, we present a problem description, a analysis, requirements for possible solutions, possible solutions and a discussion to choose the best solution. For all discussed problems we propose an implementation of the solution and sometimes we implement the solution. In that case, we also present an evaluation of the implemented solution. This is summarized in Figure 4.2.

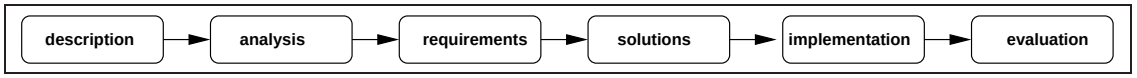

Figure 4.2: **Structure of each discussed research.**

4.2 Performance decrease

4.2.1 Problem description

The symptoms of the performance decrease for the Hyves Web site are that when requesting a page from the Hyves Web site, the Web browser is waiting an unacceptable long time for a response from a Rendered Media Web server.

4.2.2 Analysis

From Chapter 3 we know that Rendered Media Web server is implemented using two clusters of Web servers, each serving a different volume with media items. When requesting a page from the Hyves Web site, we saw in the status bar of the Web browser that it was waiting for responses from the second interval.

The first and the second interval are implemented identically: identical machines, identical software, connected to the same network switch, etc. (see also Chapter 3), and investigation with system utilities showed that none of the machines experienced problems (e.g., disk corruption) that could cause this decrease in performance. We noticed a difference in the percentages of CPU-time spent waiting for input and output: servers serving the second interval spend more time waiting for input and output (I/O) when compared to servers serving the first interval. The only I/O these servers perform is with their network interface and their disks. System utilities showed that network bandwidth used is far below the maximum so we concluded that the server's disks are overloaded.

Our best guess was that the performance problems are caused by a difference in load (e.g., the number of requests, the size of the requests). More requests lead to more disk access and when disk access increases too much, a server has to wait a long time before the requested file is delivered by the disk. This could be a possible explanation of the symptoms of the performance problem.

Our research goal has been to identify the cause of the performance problems. Our hypothesis was that the performance decrease is caused by a difference in load. Our research method aimed at determining the number of requests per interval. Therefore we collected the log files of the Web servers serving the downloads for both intervals and processed them with tools we wrote especially for this purpose. The tools counted the number of requests per interval.

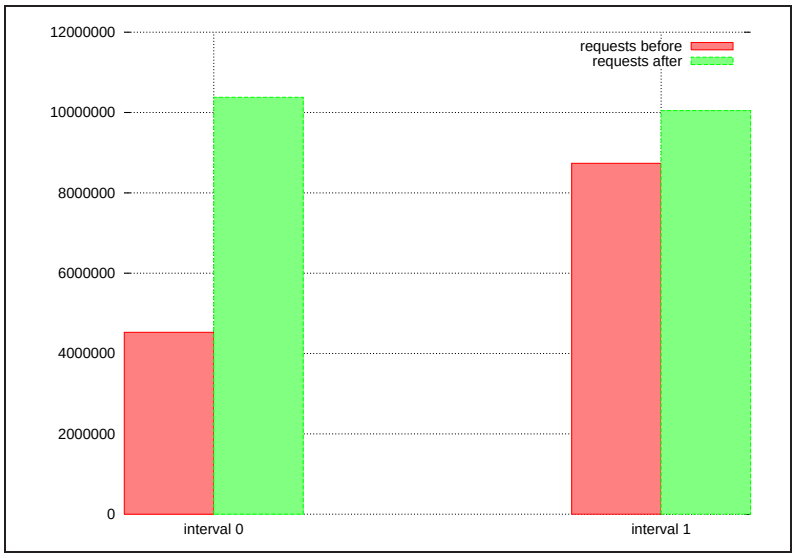

Figure 4.3: **Number of requests on one day, per interval. Dark/red: requests per interval in the initial situation. Light/green: requests per interval after moving media items from the first to the second interval.**

In Figure 4.3 the number of requests per interval is displayed: the number of requests is displayed in the y-axis of the graph and on the x-axis of the graph denotes both intervals. For this analysis, only the bars showing 'requests before' are relevant. The usage of the 'requests after' bars becomes clear further in this report.

From this graph, we can conclude that:

1. a difference in load for both intervals exists. The second interval receives about twice as much requests than the first interval.

2. since the second interval experiences performance problems and the first interval does not, and Web servers for both intervals are exactly the same, we can conclude that not all processing capacity of the first interval is used.

From the analysis, we concluded that the performance problem for the Web servers serving the second interval are caused by significant more requests to serve, and this amount of requests approaches the maximum number of requests a disk can serve.

4.2.3 Requirements

Currently, we use two redundant servers for two intervals. A solution may not break the current available redundancy. Redundant copies is a requirement, since:

- in case of a disk failure, the rendered media items remain available to be served;

- in case of recovering from a failure (e.g., disk corruptions or a failed server) the rendered media items can be copied instead of rendering again. Rendering the rendered media items costs more time than copying.

4.2.4 Solutions

The following solutions fix this performance problem and are displayed in Figure 4.4:

1. Increase processing capacity for the second interval (i.e., add another Web server and disk to the second interval). This solution can be implemented by using more Web servers serving requests for the second interval (see Figure 4.4(b)).

2. Decrease load for the second interval by moving load from the second interval to the first interval. This solution can be implemented by copying media items from the second interval to the first interval. After the copy is complete, the copied media items can be deleted from the second interval (see Figure 4.4(c)).

3. Decrease load for the second interval by moving load from the second interval to a third interval, which should be introduced. This solution can be implemented by using additional machines to create the third interval (see Figure 4.4 (d)).

4. Any combinations of the solutions above.

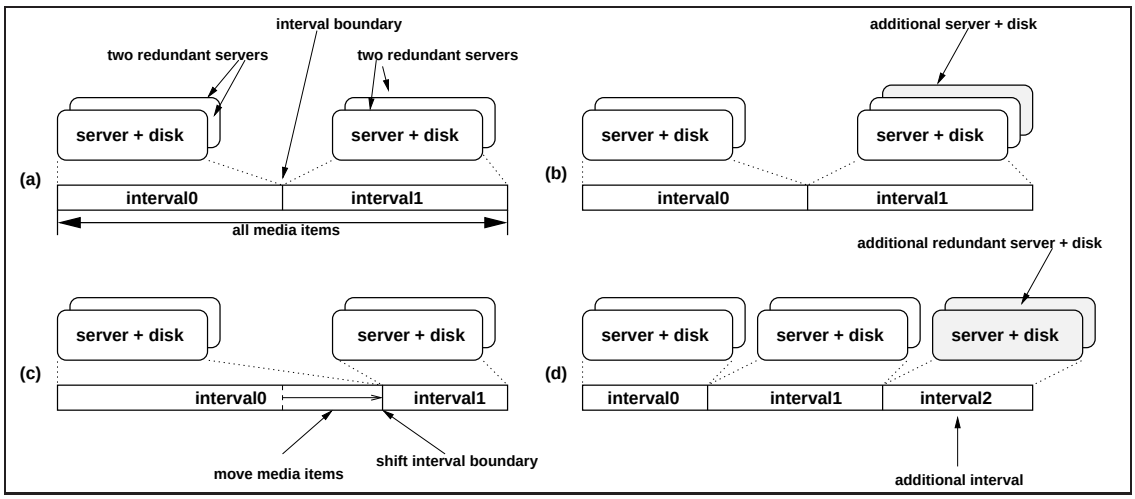

Figure 4.4: **Possible improvements for increasing performance. (a) In the original situation, each interval is served by two redundant Web servers. (b) A possible solution is to add an extra server to the second interval. (c) Another possible solution is moving load from the second interval to the first interval. (d) Another possible solution is to introduce an additional interval.**

These possible solutions can be characterized according to:

- how long the Web site suffers from even more performance decrease as a consequence of the copying of media items. Making copies is time and resource consuming: the performance problems of the Web servers of the second interval are caused by disks that can not cope with the large amounts of requests. Copying data from the second interval to another disk (e.g., to an additional disk in the case of the first solution, to the first interval in the case of the second solution or to the third interval in case of the third solution) only increases the number of the requests that the disk has to cope, and therefore performance decreases even more temporarily.

- The time at which we can take benefit from our solution. The solutions presented above take time for preparation, finishing and executing actions. With some solutions we can take benefit (i.e., experience an improved performance) before the actions of the solution are completely executed.

- The number of machines needed for the solution. If additional machines are necessary for the solution, we have to take into account the time needed for purchasing and preparing the machines and disks. Purchasing machines is a long administrative process. Alternatively, we can take machines from other parts of the Hyves Web site. That would lead to a possible performance decrease of these other parts of the Hyves Web site.

The first solution (increasing processing capacity) would result in more redundant copies than necessary (one copy is the minimum). This is inefficient use of resources, but on the other hand, it leads to a performance increase. Copying the entire volume of the second interval would take a long time, and during that time, the second interval would experience a performance decrease. We can take benefit from this solution only until after the last rendered media item is copied. Further, we need additional machines and disks for this solution. This extends the time before we can take benefit from our action. We have to say that this solution really fixes he problem: additional performance is needed, and the solution provides additional performance.

For the second solution (decrease load for the second interval by moving load to the first interval) not all rendered media items need to be copied to the first interval, therefore the time needed for this solution is less than with the first solution. Further, if we copy the items sequentially by `id`, we can execute this solution in small portions. After copying each portion of rendered media items, we can move the interval boundary which effects the actual moving of the load. Because of the change of the interval boundary actually effects the moving of the load, we can schedule the deletion of rendered media items in the second interval to off-hours. For this solution, no additional machines are needed, so we can save time when compared to the first and third solution. We have to say that this solution does not increase the performance of the second interval, it only utilizes the current machines more. Although it leads to a better balance with respect to the number of requests per interval, it also leads to a unbalance with respect to the number of stored rendered media items per rendered interval.

The third solution (introducing an additional interval) demands for more machines. This increases the time before we can take benefit from our solution. If we decide not to move any rendered media items to the third interval but just wait for it to start filling, the load remains on the second interval. When we do decide to move rendered media items to the new interval, we can take benefit of this new interval, but after we have copied the rendered media items, which will take time.

Because the performance of the Web site is unacceptable, there is no time to order and prepare new machines or take them from other parts of the Web site. As a solution to the performance problem of our initial situation, we decided for the second solution as a short term improvement. For this solution, no additional machines are required, and therefore we can save time by choosing this solution. Another time saving aspect of this solution is that only a part of the rendered media items need to be copied.

Because it is expected that this solution is only temporarily (when the number of media items increases it is expected that again a performance problem is experienced at the second interval), there is also a demand for a longer term improvement. This longer term improvement should increase processing capacity, for the reason that it is expected that the number of requests increases in the future.

4.2.5 Implementation

To implement the second solution, we have to determine the number of media items we are moving from the second to the first interval. Therefore we have to take into account the number of rendered media items per interval in the initial situation and the number of requests per interval. From this, we can calculate the desired number of rendered media items per interval and we can derive the number of rendered media items we have to move.

In the initial situation, the media service of Hyves contained 7.5 million media items. Of those 7.5 million items, 4.85 million media items are stored in the first interval and the rest of the items (2.65 million) are stored in the second interval. About $\frac{2}{3}$ of the requests are experienced at the second interval and about $\frac{1}{3}$ of the requests are experienced at the first interval. This is summarized in Table 4.1.

In the desired situation, the number of requests is equally distributed over the two intervals. Therefore we have to 'move' $\frac{1}{4}$ of the requests for the second interval to the first interval. Therefore we have to move $\frac{1}{4}$ of the rendered media items at the second interval to the first interval. $\frac{1}{4} \cdot 2.65M$=$0.66M$. This means that in the desired situation, the first interval has $4.85 + 0.66M$ rendered media items and the second interval has $2.65 - 0.66M$ rendered media items. Because we expected that in a period of time we again have to move rendered media items, we decided not to move $0.66M$ rendered media items, but 1 million.

In this new situation, the first interval contained 6 million media items and the second interval contained the rest (about 1.5 million media items). We decided for 6 million because $\frac{2}{3}$ of the requests were at the second interval. $\frac{2}{3} \cdot 7.5M$=$5M$. So at least we wanted 5 million media items in the first interval. To be ready for the future, we added another 1 million media items.

Number of ...	Interval 0	Interval 1
... media items	4.85 million	2.65 million
... requests	4.2 million	8.4 million

=

Number of ...	Interval 0	Interval 1
... media items	$\frac{2}{3}$	$\frac{1}{3}$
... requests	$\frac{1}{3}$	$\frac{2}{3}$

Table 4.1: **Calculation on the number of rendered media items to move - initial situation.**

Number of ...	Interval 0	Interval 1
... media items	$\frac{4}{6}$+($\frac{1}{4} \cdot \frac{2}{6}$) = $\frac{3}{4}$	$\frac{1}{4}$
... requests	$\frac{1}{2}$	$\frac{1}{2}$

=

Number of ...	Interval 0	Interval 1
... media items	5.51 million	2 million
... requests	6.3 million	6.3 million

Table 4.2: **Calculation on the number of rendered media items to move - desired situation.**

Now we calculated how many rendered media items to move, we can move the rendered media items. This process of moving media items is displayed in Figure 4.5. Figure 4.5 (a) displays the initial situation. In the initial situation there is free space on both volumes. To move media items from the second to the first interval, those media items are copied from the second to the first interval (see Figure 4.5 (b)). After the copying, the interval boundary is moved to the new value (see Figure 4.5 (c)). After this, the media items that were copied to the first interval are deleted from the second interval (see Figure 4.5 (d)). After this, the goal situation is reached.

4.2.6 Evaluation

After media items were moved, users no longer reported that their browsers had to wait long for a response of the servers serving the second interval of media items. We also noticed a small decrease in the percentage of CPU-time spent waiting for I/O at the servers serving the second interval and we noticed an increase in the percentage of CPU-time spent waiting for I/O at the servers serving the first interval.

In order to verify that the solution fixed the problem, we measured the number of requests again. Our hypothesis is that if the solution fixed the problem, the total number of processed requests is increased and that the difference in load is decreased. The total number of requests should be increased because of the more efficient use of the first interval and the difference in load is should be decreased because that was the goal of the solution.

The results of the measurements are displayed in Figure 4.3 using the 'requests after' bars. In this graph we see that the number of processed requests at the first interval is approximately 10.5 million requests and the number of processed requests at the second interval is about 10 million requests. The total number of requests processed (20.5 million) is larger than in the initial situation (13.5 million). We can also see in this graph that the number of requests per interval is more balanced when compared to the initial situation.

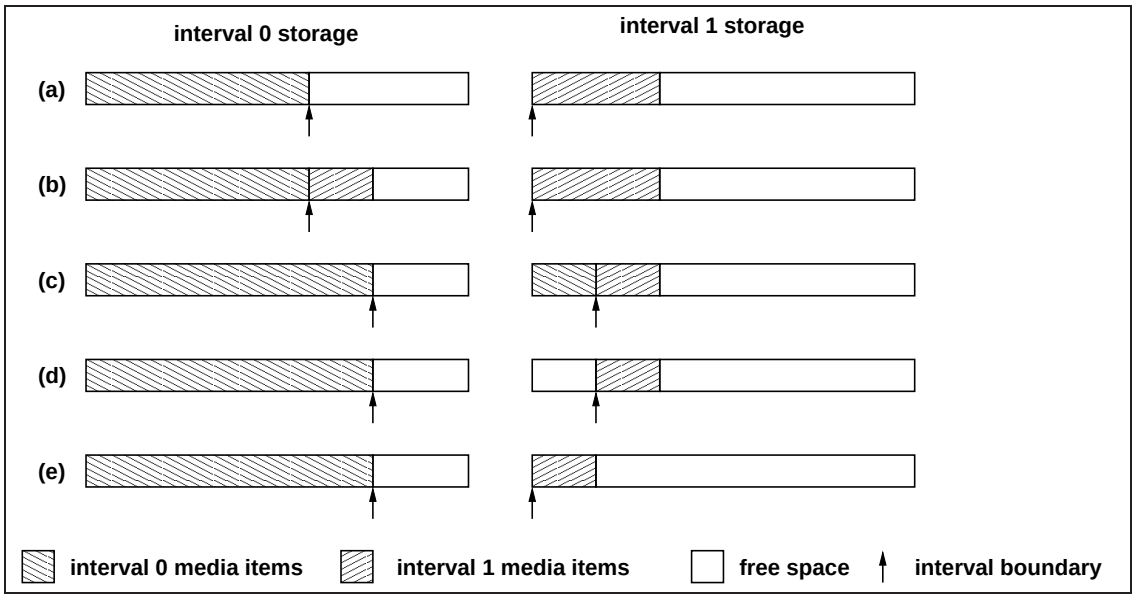

Figure 4.5: **Moving media items from the second interval to the first interval. (a) Initial situation (b) Copying media items from the second to the first interval (c) Move the interval boundary to its new value (d) Delete media items from the second interval that are also available via the first interval (e) Goal situation: media items have been moved.**

We conclude from the measurements that the total number of requests is increased and that the difference in load is decreased. With these numbers and the positive user experiences, we concluded that the problem is fixed temporarily.

Remarkable is that the total number of requests is more than expected: When we consider Figure 4.3 and assume the value of the number of requests for the second interval in the initial situation as a maximum (about 9 million) for a cluster of Rendered Media Web servers, then in the new situation, we expected the total number of requests being twice as much as that value (18 million). Instead of that, the total number of requests is higher (20.5 million).

After this short term improvement, we implement the long term improvement. We freed up 6 machines (three for each interval) from other parts of the Web site and equipped them with larger disks (500GB). Because of these larger disks (previous disks were 280GB), we decided not to introduce an additional interval because two intervals with improved storage capacity should be enough for the time being. We do not perform any measurements to verify the effect of this improvement, because evidently, three redundant servers serving two intervals perform better then two redundant servers serving two intervals.

4.3 Unbalance in the number of requests

4.3.1 Problem description

From the analysis and solutions in Section 4.2, we concluded that although the number of requests per interval is more balanced, the number of stored media items per interval is still unbalanced: the first interval contains three times more media items than the second interval. This does not seem like a problem, but in the future, this might lead to a problem with the current Media Storage implementation.

Considering the expected growth of the number of media items in the future and the fact that the new rendered media items are stored on the second interval, it is expected that over a period of time, the number of requests per interval will be more unbalanced and that results in a performance problem at the second

interval again. Then again we have to move rendered media items from the second interval to the first one. Because of the media growth, the performance problem would keep appearing and we would have to repeatedly moving rendered media items between the first and the second interval. This process is visualized in Figure 4.6.

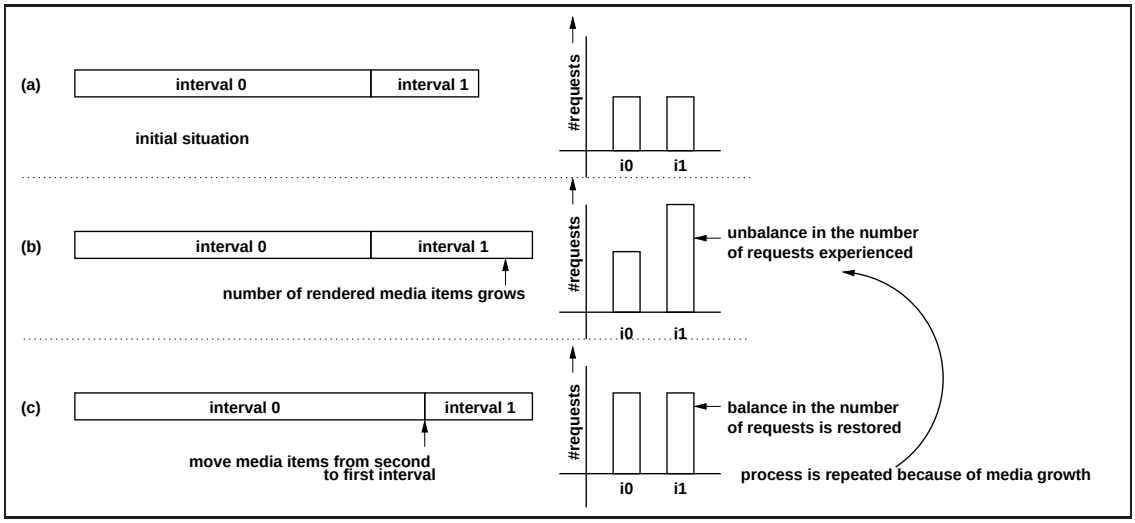

Figure 4.6: **Media growth visualized. (a) Initial situation. Web server clusters for both intervals serve the same amount of requests. (b) Number of rendered media items is increased. The number of requests at the second interval is increased. (c) Rendered media items are moved from the second to the first interval. Both Web server clusters serve the same amount of requests again.**

Currently, the first interval contains three times as much media items than the second interval. Because of the growth in media items, we expect that over a period of time this ratio will increase. As a consequence, we need a very large storage for the first interval.

The current implementation does not use very large storages that might be required in the future and is also not designed for repeatedly moving rendered media items between storage intervals when performance decreases. Therefore it it necessary to analyze the consequences of the current approach used in the Hyves Web site.

4.3.2 Analysis

In the previous research, we did the following observations:

- A balance in the number of requests for both intervals implicates that there is no balance in the number of stored rendered media items per interval. This was a result of the fact that we had to move rendered media items to cause a balance in the number of requests for both intervals;
- A balance in the number of stored rendered media items per interval implicates that there is no balance in the number of requests for both intervals. This observation reflects the initial situation in our previous research when the rendered media items were almost balanced, but the second interval received more requests;
- Addition of new rendered media items to the second interval and items in the second interval being requested more often then items in the first interval implicates that if we want to maintain a balance (either in the number of requests or the number of stored rendered media items per interval) to prevent performance problems we need a (continuous) process to enforce this.

If we combine the first observation and the expected media growth, then it would result in a relatively very large first interval and a relatively very small second interval. This raises the question if it is possible to have just one storage, because if you have a "super storage", why also have a smaller storage? Further, until now, disks are available with capacity large enough to hold all data of the first interval. What if the demanded storage exceeds the capacity of the currently available disks? If there are no such disks, we have to design a solution that uses larger storage systems (e.g., a NAS) instead of a single disk or a solution that combines single disks. For example, we can extend the current setup with another interval. However, instead of dividing rendered media items over two volumes, we have to divide them over three volumes. This would add complexity because for every interval we then would have to decide the number of items to store. Furthermore, when we decide to move rendered media items from the last interval to the last but one, we probably also have to move rendered media items between the other intervals, because the addition of rendered media items would disturb the balances between other intervals also.

The second observation raises some questions. Why were there more requests for the second interval in the initial situation? Is this just occasional because perhaps a popular range of rendered media items is currently stored in the second interval and when the number of media items grows, this popularity fades out? Or is it structural? Another question is if we have to intervals to share the load of requests, why is the load not equally shared?

The third observation demands for a (continuous) process to maintain a balance. Currently, this process is not automated because to change the position of the interval boundary, the Web site's code has to be redistributed to the Web servers. This can not be done at any time, because the code is always work in progress. The code has to be declared "stable" before it can be deployed. Furthermore, because the large amounts of data and loaded servers, moving data is an intensive process: it has to be scheduled to off-hours, if the connection between the servers fails, the process has to be restarted, if the copying is delayed the process might last longer than the scheduled time and continues to run in the peak hours, it interferes with the Autorender Daemon and so on. It also implicates that we have to handle the data twice: once when it is stored in the second interval and once for moving it to the first interval.

We can generalize the questions from the observations discussed above into two main questions:

1. How can the fact that more requests for the rendered media items in the second interval be characterized?;
2. Why is the number of requests not equally distributed over the two intervals, if we have two intervals to share the load?;
3. How about the process to endorse a balance between the two intervals?

In this analysis, we answer these three questions. We start with the first question, because we think that the other two questions are related to the first one.

Our explanation for more requests for the second interval is popularity:

- Users do not know the media item yet so they are more likely to request it. Recently uploaded media items are used in various places in the Hyves Web site (e.g., every user's profile contains an overview of the most recently uploaded media items by that user, the front page of the Hyves Web site shows an overview of the most recently uploaded media items by the users);
- It is more likely for Web browsers to have already requested media items in their cache, but not requested media items cannot be in the cache. The cache of the Web browser stores Web sites contents. When a Web site is visited more than once, a Web browser can request the cache to provide (parts of) the content, instead of requesting the Web site again. This only holds when the served content is cachable (e.g., not changed often or static content);
- Recently uploaded media items are stored and served using the second interval.

To be able to evaluate the popularity of new media items, we have to analyze the requests processed by the Web servers more thoroughly. Our hypothesis was that, if popularity is an issue, then when we consider the `id` (the unique number of a media item, see Chapter 3) of a requested media item, more requests are done for items with a larger `id`. Our method of research has been to collect all the processed requests from the log files of the Web servers and to consider the `id`s of these requests.

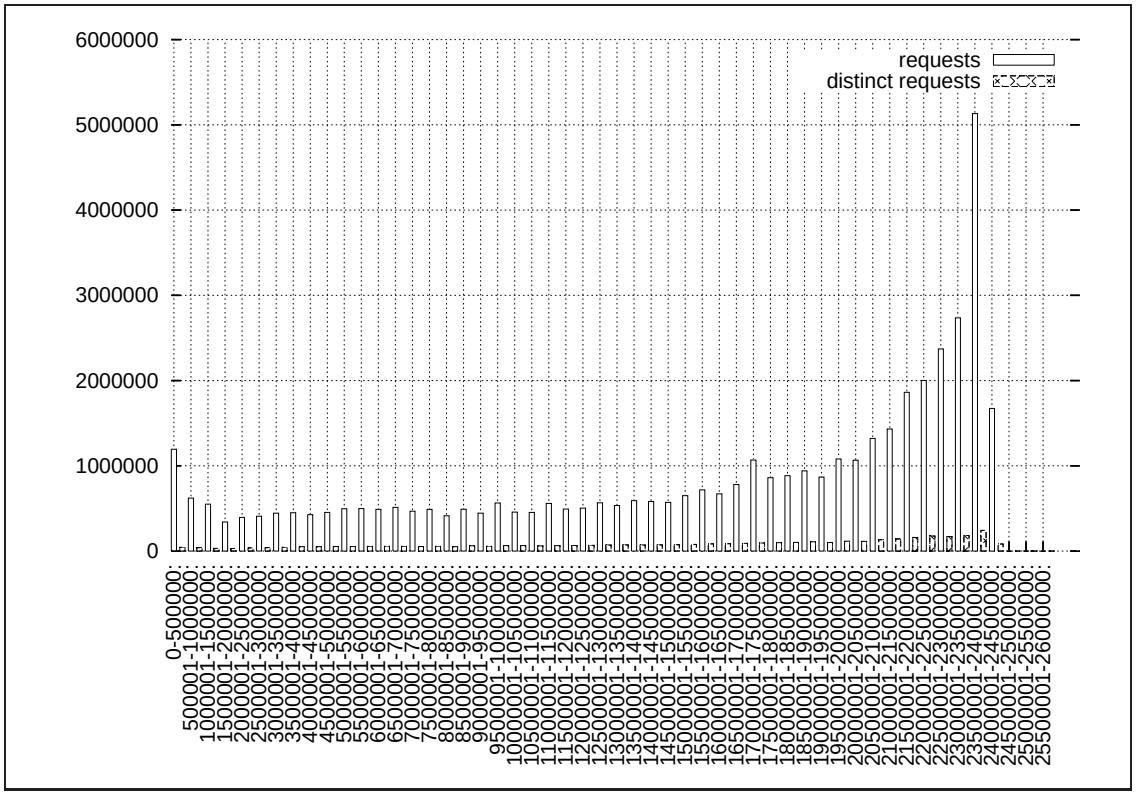

Figure 4.7: **Number of requests per media id on 26th of August, 2006. Requests: Number of requests per media id on 26th of August, 2006, grouped by 0.5 million ids per partition. Unique requests: number of requested ids per media id, grouped by 0.5 million ids per partition.**

These measurements are displayed in Figure 4.7. Because of the large number of possible `id`s, we partitioned the `id` space in partitions of 0.5 million `id`s. On the x-axis, the domain for a partition is displayed (0-500000 indicates that the corresponding bar shows results for all requests with an id between that value). On the y-axis, the number of requests is displayed. For this analysis, only the “request” bars are relevant. We do not take the last partition of 0.5 million `id`s into account, because in that partition not all `id`s are used, and therefore comparing the number of requests for this partition to other partitions would not be correct. The graph was created using the logs of August 1st, 2006. During the research, several graphs were created on other dates, but a curve like the one in Figure 4.7 is seen for all dates.

From this graph we could conclude that our hypothesis is correct. The graph shows that more recently uploaded media items are requested more often than less recently uploaded media items. Furthermore, because during this analysis, we only created graphs with curves like the one in Figure 4.7, we conclude that popularity is an structural issue. This also answers one other question in this analysis: the load is not equally distributed between both intervals because the second interval always receives the most requested rendered media items.

The problem is characterized by two factors:

- The number of items that are to be stored is too large to fit on one storage. The set of items has to be partitioned and stored on multiple storage nodes that are equal in storage and serving capacity.

- The (linear) partitioning/storing scheme and the number of requests per item result in a proportionally unequal number of requests per partition and, therefore, the maximum number of requests becomes limited by one storage node.

4.3.3 Requirements

From our analysis, we can derive the following requirements for possible solutions:

- A new storage setup should consist of multiple nodes that share in the load caused by processing requests and storing rendered media items;
- Create a scheme for storing the rendered media items on these storage nodes that enforces:
  - that the load of requests is equally distributed over the storage nodes. This requirement prevents that the maximum number of requests becomes limited by one storage node that contains all popular rendered media items;
  - that the load of rendered media items is equally distributed over the storage nodes. This requirement prevents one storage node from becoming completely filled and others not, and therefore prevents the need to (manually) move rendered media items between storage nodes.
- The requirements listed above should be maintained when increasing the number of nodes.
- It should decouple serving rendered media items from storing rendered media item, because:
  - This prevents adding disks to increase serving capacity instead of storage capacity, and
  - prevents adding machines to increase storage capacity instead of serving capacity.

4.3.4 Solutions

Because every requested rendered media item is requested at least one time from the disks (otherwise the rendered media item cannot be served), we cannot completely decouple serving and storing of rendered media items. However, we could minimize the amount of requests for the storage nodes using caching. We researched if caching is applicable in our situation and how to implement it in Section 4.4.

The research to a new storage scheme with the requirements listed above is presented in Section 4.6.

Caching, if applicable, as well as a new storage scheme can be combined.

4.4 Caching

4.4.1 Problem description

To decouple the serving of rendered media items from the storing of rendered media items as a solution for the disk access performance problem in Section 4.2, caching was analyzed.

Caching is a technique storing the results from a intensive process for later use so the next time the stored results can be used instead of repeating the process to calculate the same results again. A drawback of caching is the possibility that the original contents is changed and the stored result therefore becomes invalid. In our case the intensive process is the fetching of a rendered media item from disk and the drawback does not hold since the media items are not changed once rendered. Hyves's Rendered Media Web servers only serve static content, so the replies are always cachable.

In this section we present the research if caching is applicable in our situation to decouple the serving of rendered media items from storing rendered media items. It seems logical to cache popular requests, but until now, we only have identified popular partitions of 0.5 million media `id`s. We also have to identify popular requests, before we are certain that caching is a solution.

4.4.2 Analysis

Each partition in Figure 4.7 contains 0.5 million `id`s. From Chapter 2 we know that per media item, there are six rendered media items, one per format. Per partition of 0.5 million `id`s, there are 3 million possible distinct requests. Figure 4.7 indicates that sometimes there are more than 3 million requests (e.g., 5 million, see the 23500000-2400000 partition). Therefore we conclude that some requests are received multiple times per day.

Caching can be a solution to decrease the number of disk accesses per interval. This would decouple the maximum number of requests that can be fetched from disk from the maximum number of requests that can be served. This enables us to store more rendered media items in the second interval, so that the unbalance between the number of stored rendered media items per interval can be relieved.

In Figure 4.7 we can see that there are more requests than the number of possible distinct requests per interval. This holds mainly only for the most recently uploaded media items. From this, we can conclude that caching is applicable for the second interval. To know if caching is also applicable for the first interval, we need to extend our analysis.

Therefore we need to identify from the total number of requests per partition, the number of distinct requests. For example, if the total number of requests for partition A is 5, and the number of distinct requests for that partition is 2, we know that on average every rendered media item that was requested, was requested 2.5 times.

Our hypothesis is that if the less recently uploaded media items are also requested, then there is a large difference between the number of requested items and the number of distinct requested media items.

Our method is to count the number of distinct requests per interval. In this context, 'distinct' means distinct ids (so a request for a media item with a certain `id` and a certain format and another request for a media item with the same `id` but a different format are counted as one distinct request).

These results are also displayed in Figure 4.7 using the 'distinct requests' bars. When we compare the number of requests per partition with the number of distinct requests per partition, we see that for every partition the number of distinct requests is much lower.

We calculated how much more requests were received compared to the number of distinct requests. The results are displayed in Figure 4.8. In this graph we see per partition of 0.5 million media ids the factor of how much more the total number of requests per partition is when compared to the number of distinct requests per partition. For example, from Figure 4.7 we saw that in the first partition there were 1.2 million requests in total and 0.044 million distinct requests, so the factor is 26.

From the results in Figure 4.7 and Figure 4.8, we can conclude that not all the possible requests are made, but just a small selection and that the requests in that selection are made on average at least 7 times and on average at most 26 times. We concluded that our hypothesis is correct and that caching is applicable in our situation.

To be able to implement a caching solution efficiently, we have to investigate the traffic more thoroughly.

Next to an `id`, a requests can indicate a certain rendering of the media item with a certain size (resolution).

We want to analyze if certain resolutions of images are requested more that other resolutions of images. Our hypothesis is that if that is the case, then when counting the requests that indicates a certain resolution, then we should notice a difference in the number of requests per resolution. Our method has been to collect log files from the Web servers and to process these by counting all the requests for all possible resolutions. The results are displayed in Table 4.3 and graphed in Figure 4.9.

In the graph in Figure 4.9, we see that the "75" resolution is requested most, followed by the "50" and "120" resolutions. Remarkable is that the "200" resolution is almost never requested (only 28.386 requests from 43 million requests in total).

From Figure 4.9 and Table 4.3, we concluded that the "75" resolution is requested most and that the different resolutions are not requested equally. We use this result in the implementation of a caching solution.

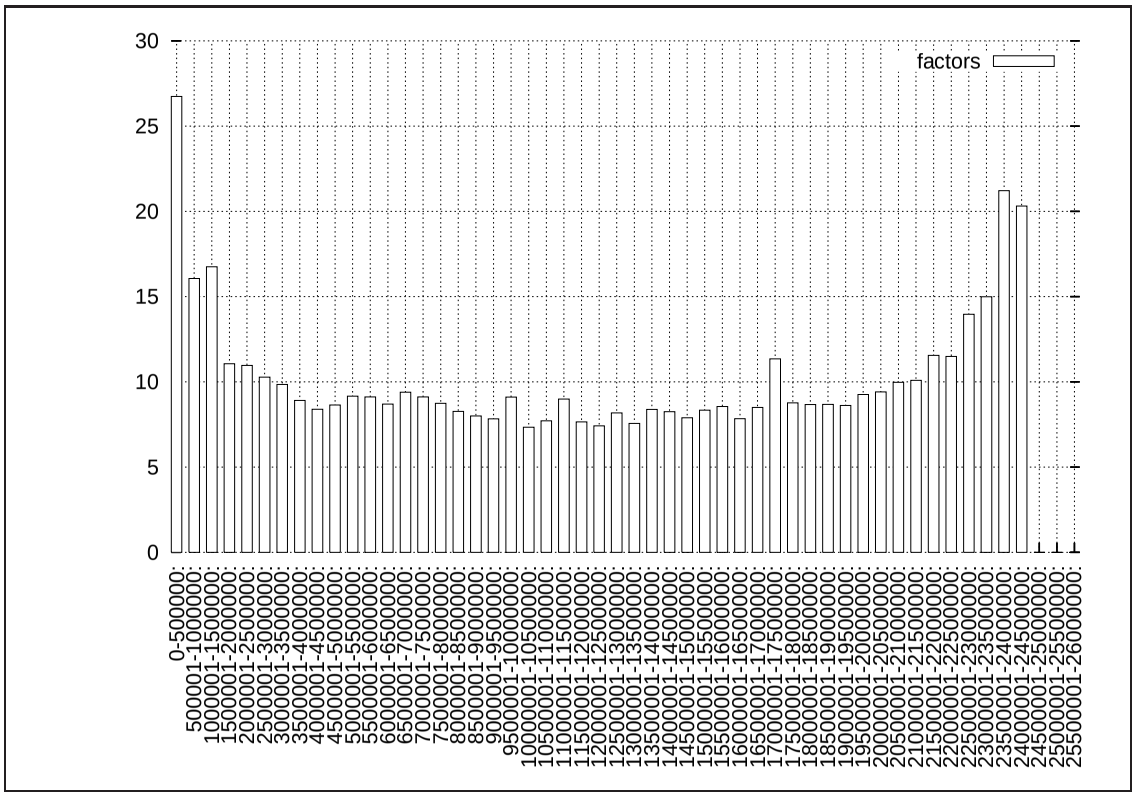

Figure 4.8: **Number of total requests divided by the number of distinct requests, grouped by partition of 0.5 million media ids. Graph created with the results from 1st of August, 2006.**

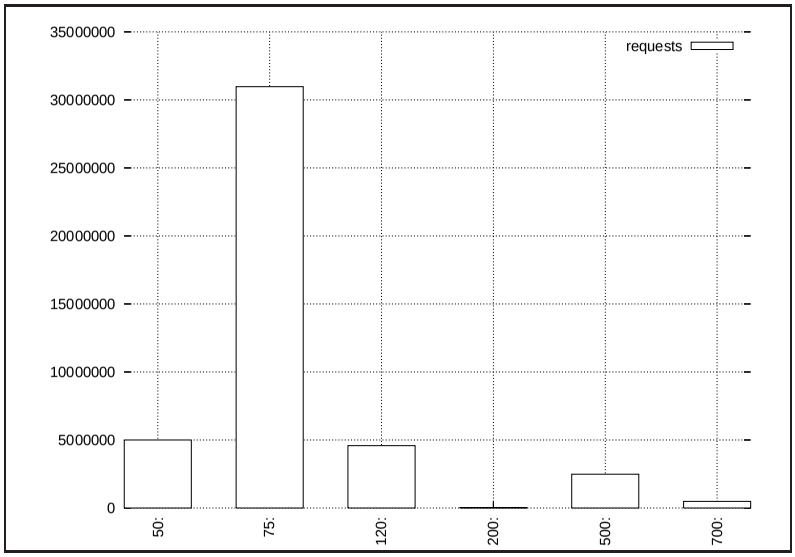

Figure 4.9: **Number of user-requests per resolution. On the y-axis, the number of requests is displayed. On the x-axis, all possible resolutions are displayed.**

Resolution	Amount	Per cent
50	4998784	11.48 %
75	30968683	71.11 %
120	4581470	10.52 %
200	28386	0.07 %
500	2486386	5.71 %
700	486711	1.12 %
Total	43550420	100 %

Table 4.3: **User requested resolutions on August 1st, 2006.**

4.4.3 Requirements

For caching solutions, there are proprietary and open-source solutions. Hyves requires that the solution is not a proprietary solution, because it is Hyves's policy to rather gain or create the knowledge and/or technology, than to purchase a proprietary solution.

A cache memory is limited in size. When the cache memory is entirely filled, a replacement policy decides which item to delete from the memory to make space for a new item.

Operating systems cache file accesses in free memory. For example, when starting the same application twice, the second time the operating does not has to access the file system. According to [CHEN02], an operating system's cache replacement policy is LRU (last recently used, which means that the last recently used cached object is deleted from cache to make space for a new object). This is not suitable in our case, because of the large number of items to cache. In case of large numbers of items to cache, it is possible that with a LRU cache replacement policy, a certain request that should be cached (because is its requested often) is not cached, because the cache memory is not large enough to cache all the distinct requests between two subsequent requests for the same item. Therefore we require that the caching solution uses a different cache replacement policy than LRU.

Furthermore, we need more cache memory than an operating system normally has (the operating system only uses unused memory for caching) because of the large amount of rendered media items. Therefore we require that the cache solution is not limited to system memory only.

4.4.4 Solutions

Caching solutions are either in software (caching proxies) that has to be run on an operating system or in hardware (caching appliances).

Software caching solutions for Web sites are caching Web proxy servers or caching reverse proxy servers [WIKIWP]. Caching Web proxy servers are designed for caching for a group of users, caching reverse proxy servers are designed for caching a (group of) servers. Therefore we use a caching reverse proxy. In the software category, there exists Squid [SQUID], PET [PET], Netscape Proxy Server [NPS] and Microsoft ISA [MSISA].

The caching appliances are routers with additional caching software. Examples are Cisco Cache Engine [CISCOCE] and Juniper's Application Acceleration Platform [JUNIPER].

Hyves required no proprietary solutions, therefore caching appliances, Microsoft ISA and Netscape Proxy Server are no options. Furthermore, these solutions are not available for Linux.

Squid and PET are the remaining solutions because they are free. PET is a spin-off of a university project. It does not have a large community of developers and is not widely-used. Squid has a large community of developers, is widely used and exists much longer than PET. Furthermore, there exists little experience with Squid at Hyves. Further, with Squid we can use cache replacement policies other than LRU. We were not

able to find out which cache replacement policies PET allows. Squid also allows the use of cache disks in addition to system memory. Therefore we have chosen for Squid as a caching solution.

4.4.5 Implementation

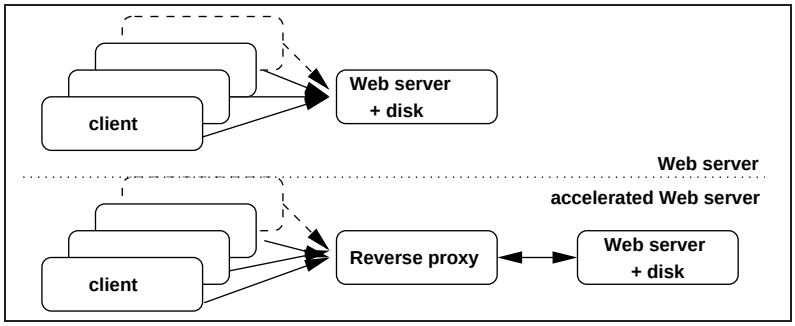

Figure 4.10: **Accelerating a Web Server by use of Reverse Proxy**

Squid has three modes: proxy mode, caching proxy mode and reverse proxy mode. The first two modes are used to accelerate a connection to another network (e.g., the Internet) and the third mode is used to accelerate a Web server. We use Squid in reverse proxy mode to accelerate our Web servers. Accelerating a Web server is a technique in which an additional machine is placed between the Internet and the Web server (see Figure 4.10). This additional machine (a so called reverse proxy) receives the requests that are directed to the Web server. If the reply of the Web server to a certain request is cachable and in cache, the reverse proxy will reply immediately with the cached result instead of forwarding the request to the Web server. By using this mechanism, requests for popular items do not have to be forwarded to the Web server. This is faster, because the cached items are served from memory instead of disk. Even in case the proxy server uses a disk for caching, this prevents forwarding requests to the Web server. A Web server that serves as a source for a reverse proxy is called a “back-end server” or “origin server”.

In section 4.4 we have shown that from the resolutions the Hyves Web site offers, the smaller resolutions (“75”, “50” and “120”) are requested more than the larger resolutions (“200”, “500”, “700”). Evidently, media items with smaller resolutions have smaller file sizes than media items with larger resolutions We tune the Squid configuration to use a cache replacement policy in which caching small files prevails above caching large files, using the heap GDSF algorithm [HPLTR1], [HPLTR2]. This algorithm combines LFU (least frequently used) and replaces large objects in preference to smaller objects. In our situation this results in rather replacing rendered media items with the larger resolutions (“200”, “500”, “700”) than replacing rendered media items with the smaller resolutions (“50”, “75”, “120”).

Squid can be configured with or without caching on disk. Although only in-memory caching is especially attractive because of its speed, in addition we decided to use caching on disk, which is an option Squid offers. Although caching on disk is slower, we can use this to limit the number of disk accesses on the origin server or to allow more distinct disk accesses on the origin server. This optimizes the use of the origin server and decouples the serving of rendered media items from the storing of rendered media items.

First we implemented Squid on one server. When we completed the configuration, we added the reverse proxy server in the load balancer cluster for the second interval Rendered Media Web servers for testing the configuration. When we got more machines for caching purposes, we copied the configuration and added the servers to the clusters for the first and the second interval. One by one, the original Rendered Media Web servers were taken out of the load balancer clusters. This reduced their function to backend-servers for the proxy servers. Using this approach, we did not cause any down-time for the Hyves Web site.

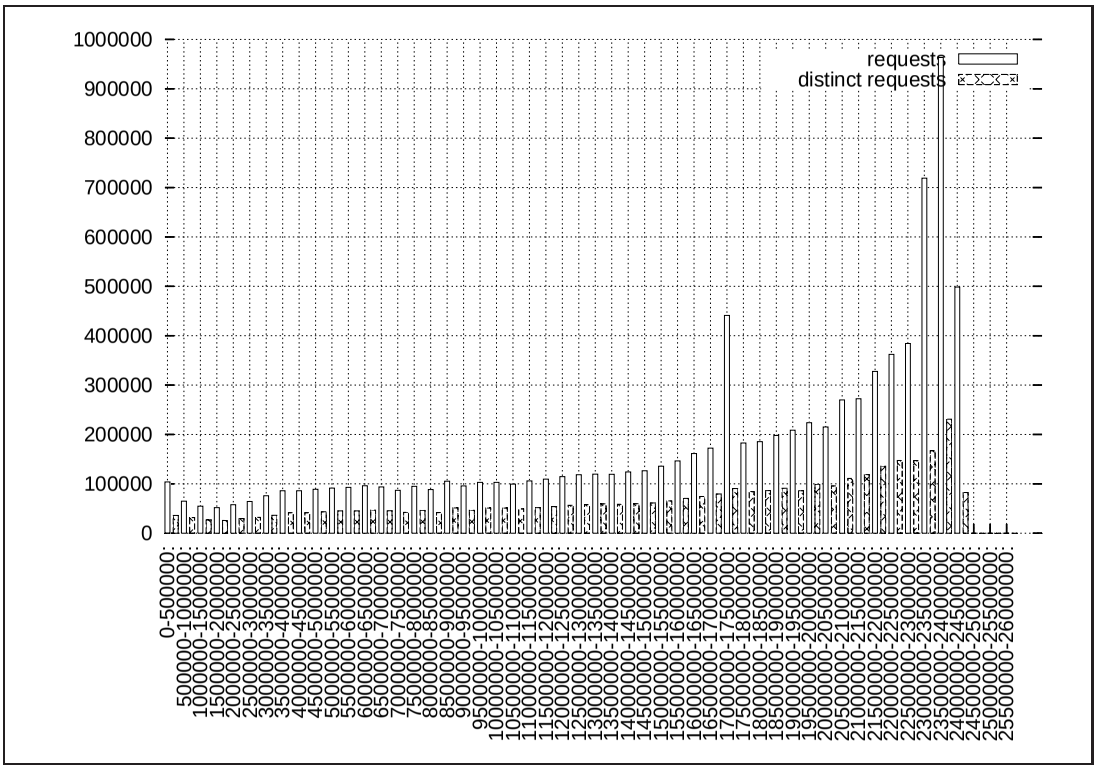

Figure 4.11: **Back-end-requested Media-ids.**

4.4.6 Evaluation

To verify whether number of disk accesses is limited, we measured the number of back-end requests and also the number of distinct back-end requests. The number of backend-requests and the number of distinct backend-requests are displayed in figure 4.11. In this graph the x-axis depicts the partition of 0.5 million media `id`s and the y-axis depicts the number of backend-requests for that partition. In this graph the values for the partition of `id`s 2400000-24500000 should not be considered, since this partition is not fully used.

When compared to the number of requests at the reverse proxies (see Figure 4.7), we can see that on average one fifth of the requests are forwarded to the origin servers. This is a reduction of 80 per cent. Because it is somewhat difficult to see when comparing Figure 4.7 and Figure 4.11, we made this more clear in Figure 4.12. In this graph, per partition of 0.5 million media `id`s, the share of the number of backend-requests to the total number of requests is displayed.

When comparing Figure 4.7 and Figure 4.11, we can also see that the ratio between the number of requests and the number of distinct requests is increased, which indicates a improved use of the origin server.

We conclude that the implementation of the reverse proxies as a solution to minimize requesting the storage nodes and disk access is a success because it reduces the use of the storage nodes with 80 per cent. Therefore it decouples the serving of rendered media items and the storing of rendered media items.

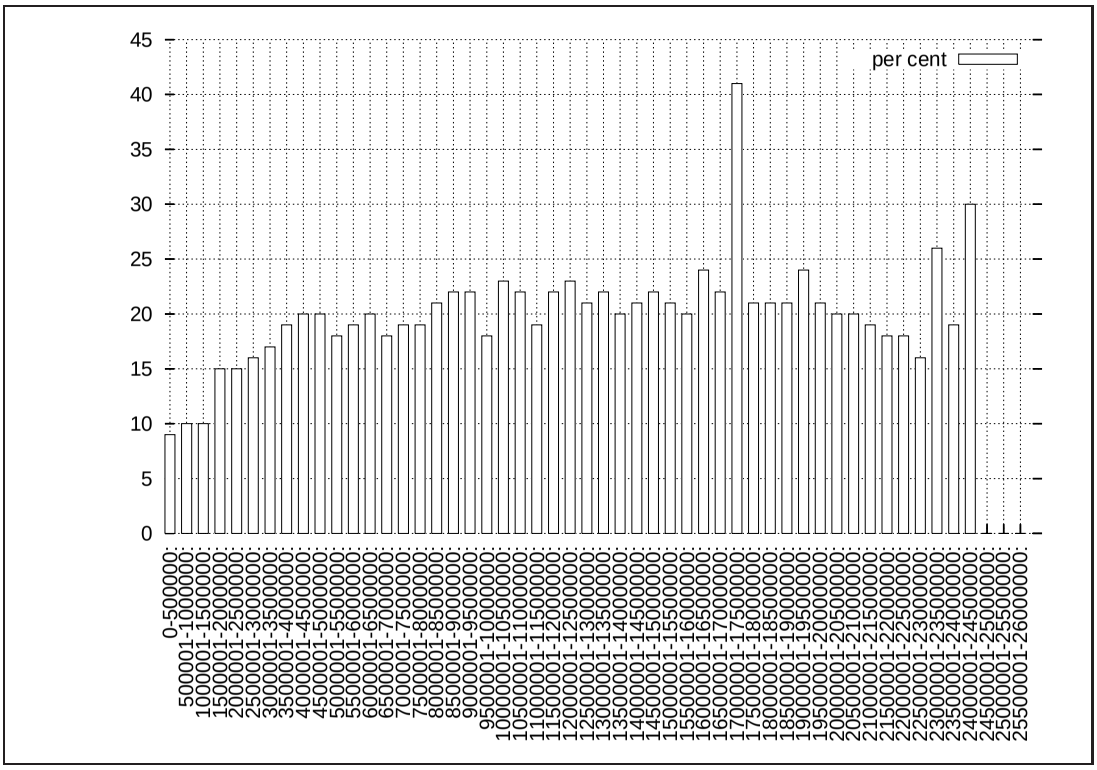

Figure 4.12: **Reduction of the number of request as a result of the use of a caching reverse proxy. On the x-axis the partitions of 0.5 million media ids are depicted. On the y-axis the percentage of the number of requests that were forwarded to the backend.**

4.5 Optimizing caching

4.5.1 Problem description

Caching was one step towards a solution that decouples serving of rendered media items and storing rendered media items.

With the setup with the caching reverse proxies, also the total number of served requests was allowed to grow. However, because of the expected growth in media items and rendered media downloads, it is expected that the number of backend-requests eventually increases, because of the number of back-end-requests is just a share of the total number of requests. Therefore it is expected that reaching the storage node's maximum capacity causes performance problems again.

The problem can be seen in two different perspectives. One perspective is that the performance problem is caused by too much backend-requests at the backend-servers. The other perspective is that the capacity of the backend-servers should be increased.

The second perspective in handled in the next section (see Section 4.6) when we analyze and design a new storage architecture.

The analysis and design of a solution within the first perspective is discussed in this section.

4.5.2 Analysis

In this analysis, we focus on the aspect of decreasing the number of backend-requests at the backend-servers. For improving our caching solution, we have to analyze the current situation.

A cache can contain relatively more cache hits when the set of data to be cached is relatively smaller: when decreasing the number of items that can be cached, the chance of an object being in cache increases. A cache's performance also increases when the requests are less diverse because the chance that a item that already has been cached is requested, increases. Further, if we can increase the cache's size, more objects fit into the cache, and therefore the chance that a cached item is requested, increases.

In Chapter 3 we showed that the Rendered Media Web servers are load balanced. In Section 4.4 we described how we replaced the Rendered Media Web servers with the caching reverse proxy servers. Therefore, the caching reverse proxies are also load balanced.

The load balancer binds a client temporarily to a caching reverse proxy server in order to distribute the clients over the pool of caching reverse proxy's. Therefore it is possible that any caching reverse proxy can receive any request. This results in two possible scenario's:

- Scenario 1: it is possible that a request for a popular rendered media item was received at more than one caching reverse proxy and therefore the item is cached at more than one caching reverse proxy. Instead of using the cache memory for other popular items, it is used to store one popular item multiple times. It decreases cache performance. because the number of distinct items in the cache decreases;
- Scenario 2: it is also possible that a request for a certain rendered media item was received at one caching reverse proxy which does not have the requested item in cache. But there might be other caching reverse proxies that do have the requested item in cache. Instead of using the other caches, the cache fetches the rendered media item from the backend-server. It can also occur that at the same time, the rendered media item is removed from cache, because is it not requested enough when compared to other items in that cache.

Both scenarios lead to inefficient use of the caching solution. Performance would be improved if a rendered media item is stored in just one cache (because that would increase the number of distinct items in cache) and the requests for that rendered media item are only directed to that cache.

When taking the number of requests and the number of distinct requests (see Figure 4.11) into account, this actually might be happening. Another possible explanation might be that the cache memories are too small to contain all popular items, and therefore sometimes a caching reverse proxy server has to request a backend-server multiple times for a rendered media item.

4.5.3 Requirements

We want to improve our caching solution by requiring that rendered media items are not allowed to be stored in more than one cache.

4.5.4 Solutions

When requiring that a rendered media item only can be stored in one caching reverse proxy, we have to

- find a solution to handle requests for the same item at the caching reverse proxies (e.g., what to do when caching reverse proxy A receives a request for a rendered media item that is cached at caching reverse proxy B);
- find a solution to direct requests for a certain rendered media item to a specific caching reverse proxy, instead of directing it to all caching reverse proxies.

So if a cache receives a requests for a reply it does not have in cache, the cache should request other caches if they might have a cached reply for the request. This is called cooperative caching [CHEN02], [DSYS]. Such a setup is displayed in Figure 4.13.

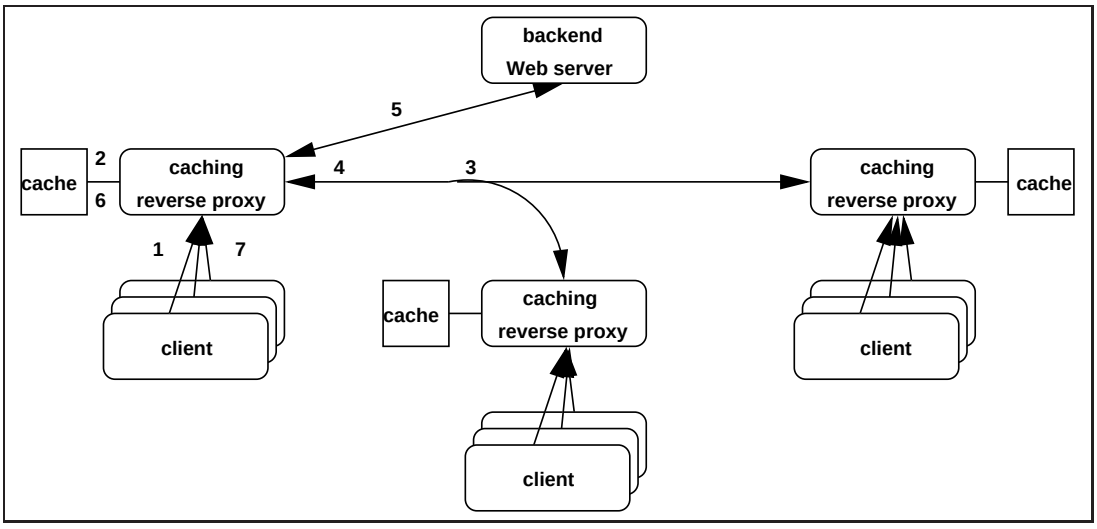

Figure 4.13: **Cooperative Caching Setup. (1) A client requests an item from the caching reverse proxy. (2) Caching reverse proxy checks it cache. If the item is cached, the item is returned to the client. (3) If the item is not cached, the caching reverse proxy contact other caching reverse proxies. (4) A response is sent to the caching reverse proxy. The response is either the requested item or a message that the requested item is not cached. (5) The caching reverse proxy requests the backend Web server and receives the requested item. (6) Possibly the item is stored in cache and (7) sent to the client. This figure is partly copied from [DSYS].**

Another solution is that the client requests all caching reverse proxy servers, and only the server that has the requested rendered media item in its cache, replies to the response. But suppose none of the caching

reverse proxies reply because the requests concerns a rendered media item that is not in the caches (e.g., the rendered media item has just been uploaded)?.

Another solution is that the client is aware of in which caching reverse proxy a certain rendered media item is cached, or should be cached. This can also be applied to the caching reverse proxy: if the caching reverse proxy receives a request for a rendered media item it knows in which other caching reverse proxy it is cached or should be cached. Then we can create partitions, and each caching reverse proxy then has to cache just a small data set of the entire data set. However, we must prevent an unbalanced load for the cache servers, because this would limit our growth, as we have seen the results of other analyses in this research.

We studied the Squid application [SQD04] and found out that caching reverse proxy servers can be placed in a caching hierarchy (with cache peers and cache parents) in which caching reverse proxy servers are able to request each others caches (cache peers) if the requested rendered media item cannot be found in their own cache. If none of the caching reverse proxy servers has cached the requested item, the request is forwarded to the backend-server (parent). Different techniques are available for implementing such a hierarchy when using Squid:

- Inter Cache Protocol (ICP). Using ICP a Squid in a caching hierarchy is able to request (one by one in a sequence or by broadcast) other ICP cache peers if they might have a cached reply to that request;
- Cache Digest (Digest). A cache digest is a summary of the contents of a cache. In a Cache Digest hierarchy, the cache peers exchange Cache Digests so a cache peer can determine in advance which cache peer to ask for the requested item (with a limited certainty). Because the contents of the cache changes over time, the Digest has to refreshed from time to time;
- Cache Array Routing Protocol (CARP). When a cache peer of a CARP hierarchy receives a request, the request is processed by a mathematical procedure which determines to which cache peer the request is forwarded to. For a certain request, the mathematical procedure always produces the same result on any cache peer in the CARP hierarchy. Therefore a certain request always is forwarded to one particular cache peer.

[SQD04] also gives guidelines for the above strategies. ICP is not suitable when the number of cache peers increase because of the broadcasts. If the number of cache peers increase it will result in cache peers being more occupied of processing ICP requests than processing their own client's requests.

Cache Digest is not suitable for large caches, because the calculation of a digest is time and space consuming. When the number of cache peers increases, the more cache peers have to download each others Digests. Because these digests are space consuming, there will be a trade off between the number of items that can be cached and the number of correct forwarded requests.

CARP is the best solution, because the additional calculations are relatively simple and this calculation limits contacting another caches peer to at most one if case the item is not cached at the caching reverse proxy that receives the request from the client. Further enforces the mathematical procedure to select the cache peer to forward the request to, an equal distribution of the requests to the cache peers, and therefore the number of requests is balanced at all cache peers. We failed to implement CARP because of unknown reasons. Contacting support using mailing lists and searching did not solve the problem.

Because of the urgency of the performance issue, we decided to design a comparable solution. The URLs that are used for the requests to our media service are constructed by our application, so it is possible to extend our application with logic to construct the URLs in such a way that certain requests are always devised to certain cache servers.

In this design, we can include the requirements that we identified earlier in Section 4.3. Therefore the requirements for this design are:

1. balanced sharing of the load
2. partitioning of the data set so each cache server serves one partition

We can realize these requirements using a 'modulo' approach: we can use the media items id and a modulo function to create partitions. Because of this modulo function, each partition contains an equal share of 'unpopular' and 'popular' media items and thus the load is shared. Using this modulo approach also realizes that a certain request is always directed to a certain cache server.

In addition to balancing the load and increasing cache performance , the modulo approach also has the benefit of allowing a scalable growth: if more media items are to be served or to increase cache performance (e.g., when the number or rendered media items has increased, and therefore the number of rendered media items in each partition), the number of cache peers can be extended.

The modulo problem has as a drawback that by increasing the number of partitions (e.g., in case of growth) and decreasing the number of partitions (e.g, in case of failure) almost all items have to be remapped. Consistent Hashing [AKAMAI], [WIKICH], [WCCH] can be used to minimize the number of re-mappings. Consistent hashing is explained in 4.6, where we also use it in our storage design.

4.5.5 Implementation

Code for consistent hashing was added to the Web site's code. The consistent hashing algorithm was altered to use a database to detect the number of cache nodes instead of a fixed number in the algorithm. In case of failure of a node, a record in the database is updated and consistent hashing is adjusted to the new number of cache nodes. Without this adjustment, the Web site code should be deployed every time in case of failure and recovery of a cache node. In Chapter 2 we discussed that the approach in which we have to redeploy the Web site's code should be prevented.

We added four new entries to the load balancer configuration: from `modulo0.rendered.startpda.net` to `modulo3.rendered.startpda.net`. The network interfaces on the reverse proxies were adjusted to allow requests for the new load balancer entries. By keeping the old load balancer entries intact, the old interval setup stays available for the users.

The Squid configuration was adjusted to allow requests with the new URL scheme. Squids can be reconfigured without downtime.

The adjusted Web site PHP code was deployed to a single machine in the cluster. This allowed us to test the new code and the new configurations.

After this, the code was deployed to the other Web servers in the Web site cluster.

4.5.6 Evaluation

To verify our results, we measured the number of requests and the number of back-end requests (see Figure 4.14). The total number of requests was 46 million and the total number of back-end requests was 7 million. This means that 15 per cent of the requests were requested at the origin server (see Table 4.4).

Measurement	Interval Setup with Caching	Consistent Hashing Setup
Total requests	43 million	46 million
Back-end requests	8.9 million	7.0 million
Ratio Back-end/Total Requests	20 %	15 %

Table 4.4: **Comparison back-end requests interval setup vs. modulo setup**

In the old situation with caching, but without consistent hashing, we had 43 million requests in total and 8.9 million back-end requests. This means that about 20 per cent of the requests was requested at the origin server (see Table 4.4).

We conclude that the consistent hashing approach leads to a decrease in the number of back-end requests. To decrease the number of back-end requests even more, we suggest increasing the number of cache nodes

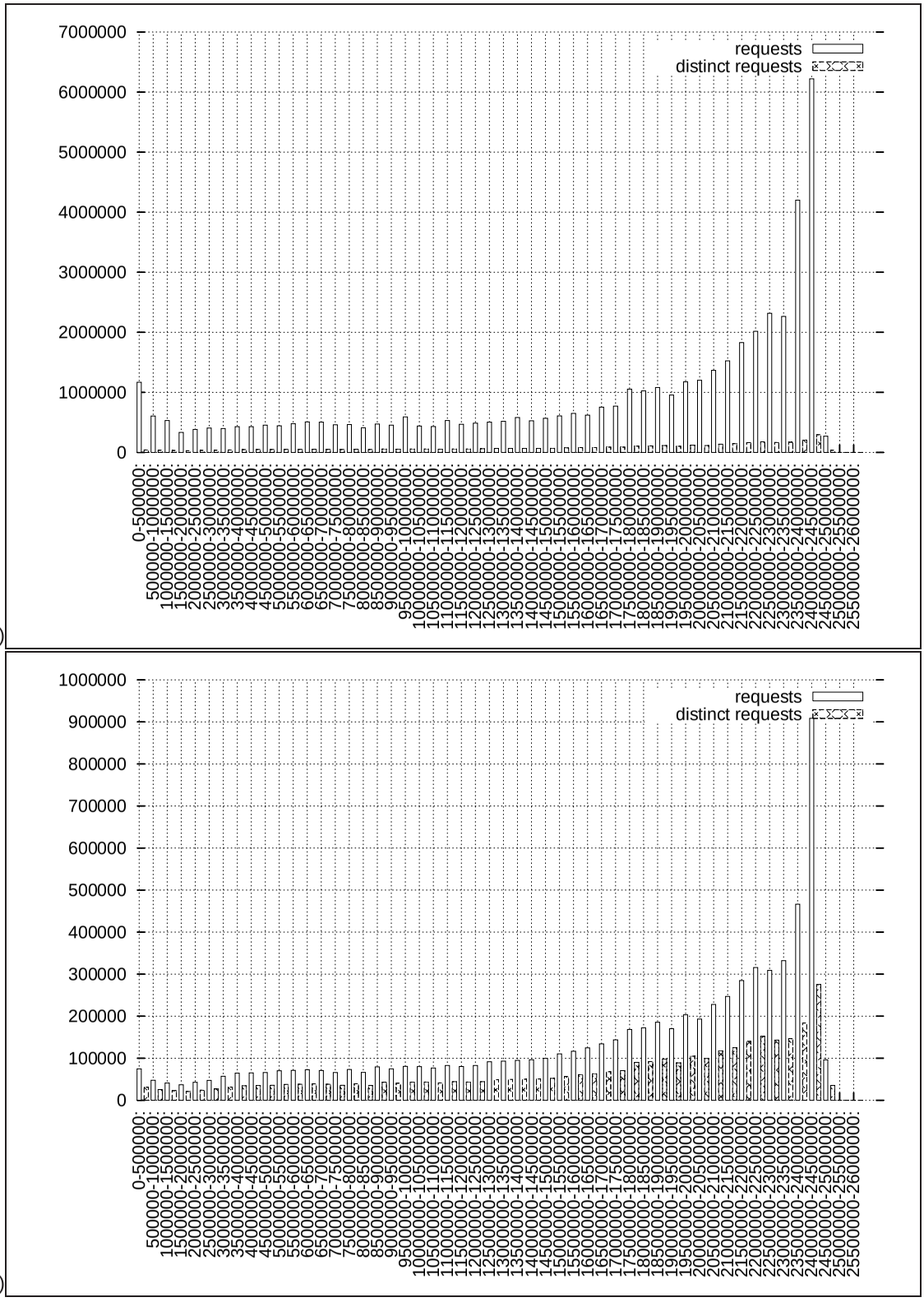

Figure 4.14: **(a) Requests per id with the modulo approach (b) Back-end requests per id with the modulo approach.**

(currently there is about 32GB of memory cache and 32 GB of disk cache for 1.5TB of data). However, it is not possible to decrease the number of back-end requests to 0% because of the continous uploading of new media items.

4.6 Storage

4.6.1 Problem description

In this section we want to design a new storage architecture, that overcomes the current analyzed problems that are stated in the previous sections.

4.6.2 Analysis

In Section 4.2 we analyzed that disk access capacity of the second interval is not sufficient, while the disk access capacity of the first interval is not fully used. In Section 4.3 we analyzed that because of the number of items to store, storing items on more than one storage is necessary, but the current (linear) storage scheme implicates that one storage node limits the maximum number of requests. In Section 4.4 we analyzed a caching solution to decrease the number of disk access and in Section 4.5 we analyzed the caching solution even more to decrease the number of disk accesses even further. In Section 2.2.1 and 2.4 we analyzed the current storage setup for original media items, how it is expanded to increase capacity, that it is a single point of failure and that it should be decoupled from other tasks like storing backups.

Although the original media storage is only discussed in Chapter 2, in the future it is expected that the current setup is not scalable. When the current capacity has to be expanded, media items are copied to disks on other machines and they are made accessible via NFS shares. If we continue to offer more and more media items via a single NFS share via the NAS, it is expected that in time, the maximum capacity of the NAS is reached. We have experienced the same with the Rendered Media Storage, and therefore we should prevent this situation.

If we analyze the model that we discussed in Chapter 3, we know that original media items are only requested by the Autorender Daemon and the incidentally if the Web site has to render the media item. From this chapter, we know that rendered media items are requested very often. From Chapter 2 we also know that on average, the aggregate file size of the rendered media items is about one sixth of the original file size.

If we combine the storage for original media items with the storage for rendered media items into one single storage, we can reduce the number of media ids stored on one interval to about one seventh of the number of media is that is currently stored in a interval. In Section 4.5 we analyzed that if we decreased the diversity in requests, the caching performance improves. As a consequence, we have to increase the number of intervals, to be able to provide sufficient storage capacity for all the data.

4.6.3 Requirements

Redundancy is a requirement because in case of a disk failure, the rendered media items are still available to be served and in case of recovery from a failure, copying the rendered media items takes less time then render them all over again. We require that at least there is one copy of every rendered and original media items. We require that a new storage architecture further decouples serving rendered media items from the storing of rendered media items, because:

- This prevents adding disks to increase serving capacity instead of storage capacity, and
- prevents adding machines to increase storage capacity instead of serving capacity.

Therefore we require that the maximum number of copies of every rendered and original media items equals one.

We require that original media items and rendered media items are stored in the same storage, instead of in two separate storages in order to reduce the number of media ids stored on one interval, to reduce the number of possible requests for that interval and to increase caching performance. We also expect that maintaining one storage instead of two reduces the work involved with this (e.g., deleting media items, which takes actions on both storages in the current setup).

We require that we can extend the number of storage nodes to anticipate to increased storage capacity demand.

We require that the a storage architecture should consist of multiple storage nodes that share the load caused by processing requests and storing rendered and original media items and the new storage architecture uses a scheme for storing the original and rendered media items on these storage nodes that enforces that:

- the load of requests is equally distributed over the storage nodes. This requirement prevents that the maximum number of requests becomes limited by one storage node that contains all popular rendered media items. This requirement should be maintained when the number of storage nodes is increased or decreased;
- that the load of rendered media items is equally distributed over the storage nodes. This requirement prevents one storage node from becoming completely filled and others not, and therefore prevents the need to (manually) move rendered media items between storage nodes. This requirement should be maintained when the number of storage node is increased or decreased.

Hyves prefers a solution that is developed and maintained in-house rather that a proprietary solution in order to stay independent from vendors and consultants.

4.6.4 Solutions

Solutions for the resource layer (storage) can be generalized to two options:

- purchase a total proprietary storage solution;
- constructing a solution using building blocks and creating the logic to glue the building blocks.

Both options are discussed hereafter.

Examples of total storage solutions are Panasas [PANASAS], CoRaid [CORAID], Equallogic [EQUALLOGIC] and EMC's Clariion [CLARiiON]. These solutions provide a network designed to attach computer storage devices such as disk array controllers. A SAN allows a machine to connect to remote targets such as disk on a network for block level I/O.

Example of storage building blocks are CoRaid boxes [CORAID], Equallogic [EQUALLOGIC], a standard NAS or any machine with large disks and file serving software.

Software to incorporate these storage blocks into one large storage are distributed file systems, which is a file system that supports sharing of files and resources in the form of persistent storage over a network [DFS], [DSYS]. Examples are Coda [CODA] and GFS [RHGFS]. An overview of distributed file systems is given in [IBBD]. Traditional distributed file systems are add-ons or implemented in the operating system kernel. To make this setup scalable not only in a storage capacity sense, but also in a storage serving sense, a cluster of servers that can access the distributed file system is necessary.

There are also distributed file systems that are implemented at the application level, although they are mostly referred to as Content Delivery Networks or content distribution network [DSYS]. Examples are

[MOGILE] and [CORAL]. These application level distributed file systems are implemented on top of network file systems and use an add-on or are implemented in the application that uses the file system.

When purchasing a total storage solution, the price per gigabyte of storage is higher, because we also have to pay for backup-software, data-retention software, information life-cycle management software, etc.. We also become dependent of the supplier for support. Next to this, most total storage solutions are more complex and have more functionality than we demand from a storage solution (e.g., back-ups, snap shots).

When choosing to design and create a storage solution, the price per gigabyte of storage is lower because no additional software has to be purchased. In case we build it ourselves, the technology is developed in-house and we are less dependent of a supplier (although this costs time and thus money to develop). A little bit of dependency still remains because we have to buy the storage blocks and in case of failures, we still have to request for support from the supplier. Most of the dependencies can be solved by buying spare parts, so in case of failure, these parts are at hand.

We decided for designing and implementing a solution instead of purchasing a total storage solution because of the lower costs and the consequence that the technology is build in-house.

We started the implementation with building a cluster with a distributed file system. During the implementation of this cluster, we revised our vision on the suitability of a cluster with a distributed file system as a solution to requirements. The main reasons for revising our vision on the suitability of a cluster are that the used cluster technology has a maximum file system size of 8TB and the fact that it is very intensive to create and maintain a cluster. The desired media storage should allow more capacity than 8TB and should not be difficult to maintain. We decided that Hyves does not need a cluster based distributed file system in which we can read, write and manipulate files many times. From Chapter 3 we know that Media items are uploaded once, read once for rendering, rendered media items are written once after rendering and requested many times. We realized that a content delivery network or a application level distributed file system matches our requirements more than a cluster with a distributed file system because we do not desire the additional capabilities of a cluster with a distributed file system. Therefore we decided to design a new implementation based on an application level distributed file system. In the next section, only the implementation of the new storage architecture based on application level distributed file system is discussed, which only contains the design. We did not implement this storage architecture because of a lack of time.

4.6.5 Implementation

The base of our new storage architecture is an application based distributed file system, which is built on top of a document based system using HTTP and FTP.

The clients that would use our new storage architecture are the caching reverse proxy servers (to requests rendered media items) and the Autorender Daemon (to requests the original media item and to store the rendered media items). The caching reverse proxy servers are responsible for almost all the requests to the new storage architecture. These requests are HTTP-requests. The Autorender Daemon currently uses NFS to request the storage architecture, but this can be altered to use HTTP. Storing the rendered media items by the Autorender Daemon can be altered to use HTTP, but we can also use FTP for this. Although FTP and HTTP are more resource consuming than NFS because FTP and HTTP are handled in special applications and NFS is handled in the kernel, we must take into account that almost all the requests to the new storage architecture are done via HTTP.

By using FTP and HTTP as transport protocols, we have a solution for the problems with NFS (see Chapter 5).

To provide storage capacity, we use servers on which we install Web server software, in order to enable access to the files with HTTP. If we have chosen to use FTP for storing files, we also have to install FTP server software. We can use Web servers with large disks (approximately a storage capacity of 1TB per server), Web servers with CoRaid like boxes (approximately a storage capacity of 6TB per server) or NASs (approximately a storage capacity within a range of 1TB to 9TB per NAS). We refer to a server with Web server software and a large disk as storage node. In our new storage architecture we want to allow a

heterogeneous set of storage nodes because it is likely that in a period of time disks, CoRaid boxes and NASs with larger capacity become available. Further, it also allows us to reuse the parts of the current used storage.

In order to distribute the data over these Web servers, we have to partition the set of data (original media items and rendered media items). Each Web server should hold a partition of the data. To partition the data in such a way the items are equally distributed over all the Web servers, we use consistent hashing [AKAMAI], [WIKICH], [WCCH].

Consistent hashing uses two principles, which we use in order to distribute the data over the storage nodes. The first principle is a hashing function, which is necessary to classify items into a limited set of classifications. The second principle is partitioning the set of classifications. When storing a media item, the item is first classified into one classification. Then the media item is stored on the Web server that stores all media items for a certain set of classifications. When requesting a media item, the request is classified to the same classification in which the stored media item is classified. Then the requested media item is requested from the Web server that stores the media items for that particular set of classifications (see Figure 4.15).

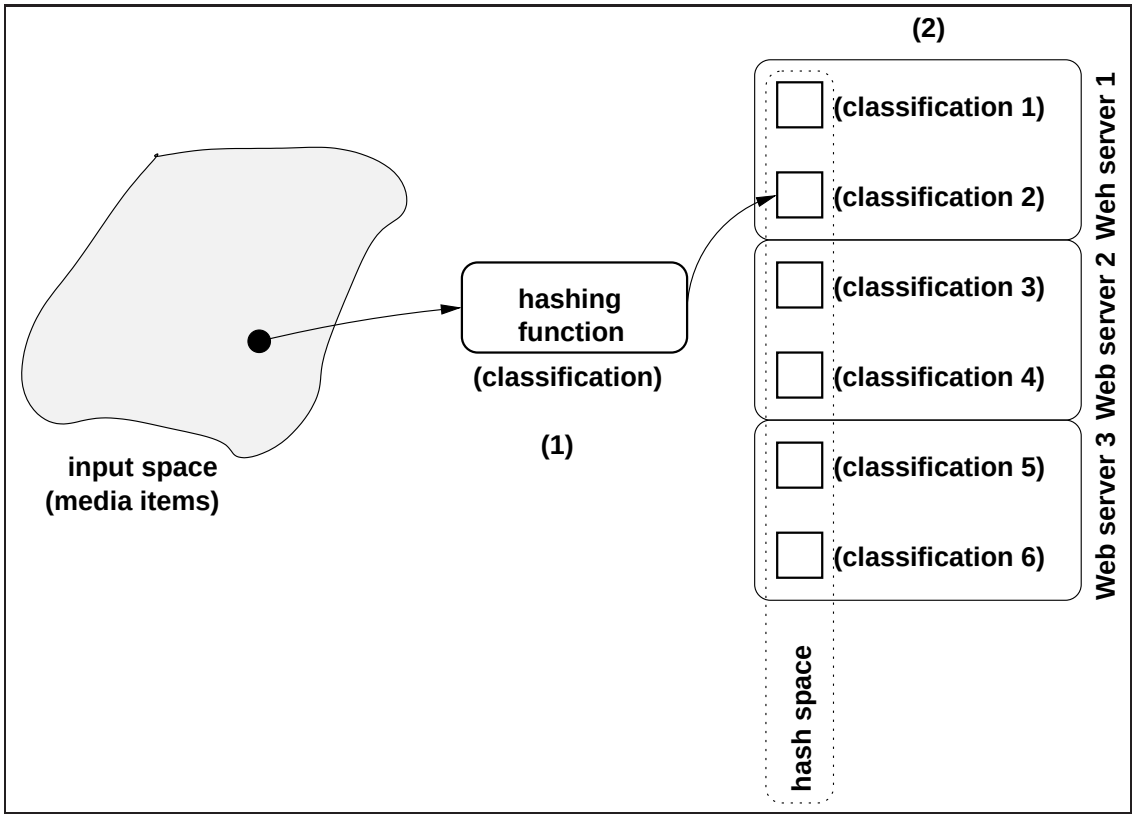

Figure 4.15: **Consistent Hashing. (1) first an input value is classified. (2) the classifications are grouped into partitions.**

When considering the design so far, the design is not different from the current rendered media storage. In the current rendered media storage the classification algorithm is “the first 5 million media `id`s are fit into classification 1, the next 5 million media `id`s fit into classification 2, etc.”, and the partitions are the current intervals. The difference is the use of a hashing function as a classification scheme.

A hashing function (or hash algorithm) is a reproducible method of turning data (usually a message or a file) into a number suitable to be handled by a computer. These functions provide a way of creating a small digital “fingerprint” from any kind of data [HASH]. By creating this fingerprint, we can classify items. Although the term “fingerprint” is often connected to uniqueness, this does not necessarily be the case.

To illustrate the use of a hashing function in our storage design, we use the modulo operation, which finds the remainder of the division of one number by another (see Figure 4.16). Because the outcomes of the modulo operation are cyclic when using increased inputs, equal distribution of the items to the possible outcomes is guaranteed when using a large number of items (i.e., classifying 24 subsequent media items results in 2 media items per classification). Suppose we use the media `id` of a media item, then we can classify the media items into classes with the same remainder after a division by 12.

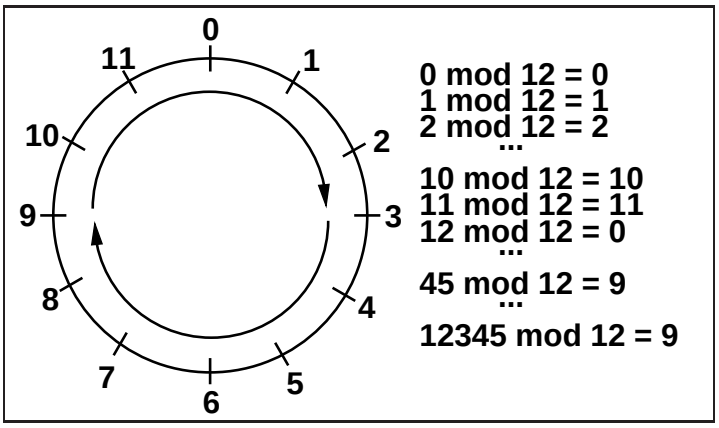

Figure 4.16: **Example of Modulo operation. In this example, we use 12 as the value for the base of the modulo operation. The remainder of the division of 12 by 12 equals 0. The base parameter of the modulo operation limits the number of possible outcomes (12 in this example). Two subsequent values (e.g., 33 and 34) result in two subsequent outcomes (e.g., 9 and 10).**

To complete the example, we have to group the possible classifications in our example and assign these groups to different Web servers. Suppose we have three storage nodes, two with 1TB of storage capacity and one with 2TB of storage capacity. Then we have to create three groups, because the number of groups should equals the number of storage nodes. Two groups entails one fourth of the classification space each (i.e., classifications 0-2 to one group, 3-5 to the other group), and the second group entails one half of the classification space (i.e., classifications 6-11). This also illustrates the use of heterogeneous storage nodes.

The example shows that it is required that the number of classifications should be larger than the number of storage nodes to enable to create partitions that are weighted with the capacity of the storage node. In the future, the number of storage nodes might increase because storage capacity demands might increase. Therefore, the number of classifications should be far more larger than the number of storage nodes.

In our example we used the modulo operation as a hashing function. Because of the cyclic behavior of the outcome of the modulo operation when using increased inputs, when the base is too large, many media items with increasing media `id` are placed on the same storage node. Suppose we use 1200 as base to the modulo operation and we create three groups. Two groups then will entail the classifications 0-299 and 300-599 and the other group will entail 600-1199. This would result in 300 subsequent media items on the same storage. When the number of media items is large enough, this effect is minimized.

We can also use another hash function like MD5 [WIKIMD5] or SHA-1 [WIKISHA]. These functions have the effect that when the input is slightly changed, the output is changed significantly. In cryptography, this is called the Avalanche effect [AVALANCE]. Because of the Avalanche effect that occurs with these two functions, randomization is introduced. Therefore media items with subsequent media `id`s will not have subsequent outcomes.

Now we have showed how we can partition the dataset in order to store it on multiple storage nodes, we have to implement this scheme to enforce that data item reads and writes are directed to the correct storage node. Because we use HTTP, we could decide for a HTTP redirector (see Figure 4.18). Such a redirector is capable of processing the request and to redirect the user (e.g., the Autorender Daemon or caching reverse proxy) to the correct storage node. If we decide to use FTP as transport protocol for writing media and

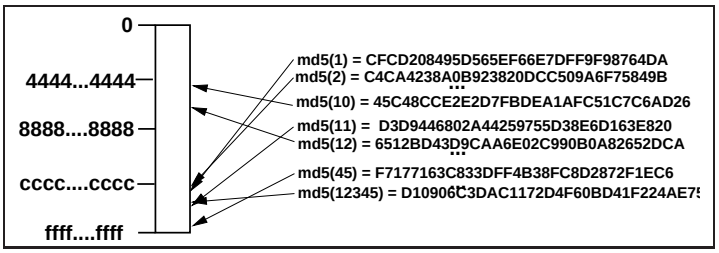

Figure 4.17: **Example of MD5 operation. The outcomes of an MD5 operation vary from 0 to 2^{128} (it is custom to use the hexadecimal notation).**

rendered media items, we could implement a FTP redirector, based on the workings of an HTTP redirector. The HTTP redirector processes the requests, determines the correct storage node and then redirects.

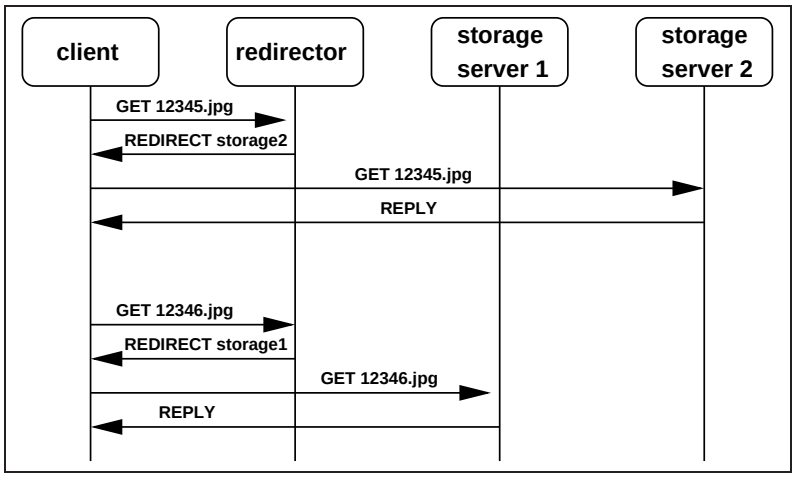

Figure 4.18: **Using a redirector**

We have showed how media items can be classified and that classifications are grouped to be stored on one storage. When the number of storage nodes is increased, we have to create new groups. In our example, we used the groups that hold classifications 0-2, 3-5 and 6-11. We increase the number of storage nodes in our example to 4 storage nodes. The added storage node has a capacity of 2 TB. Then we have to create 4 groups. The first two groups for the storage nodes with a 1 TB capacity entail the classifications 0-1 and 2-3, the second two groups entail classifications 4-7 and 7-11. This results in the first storage node to not store items with classification 2 anymore, the second storage node not to store items with classifications 4 and 5 anymore and third storage node not to store items with classifications 7, 8, 9, 10 and 11 any more. The second storage node has to store items with classification 2 also, the third storage node has to store items with classifications 4 and 5 and the newly added storage node has to store items with classifications 6 to 11.

To not lose any data, media items should be copied to the storage nodes that store them in the new configuration. When all the storage nodes contain the classifications that they suppose to store in both the current and the new configuration, items of classifications that not supposed to be stored in the new configuration can be deleted on the first three storage nodes.

Therefore, reconfiguration of groups of classifications is done in three phases.

In the first phase, a storage node collects the items of the classifications that should be stored in the new configuration on the node from other storage nodes. Also the table or algorithm that matches the classification with one of the storage groups is altered, so that media items are stored in the storage group that

holds a certain classification at current and the media items are stored in the storage group that should hold a certain classification in the new configuration. One could implement rate controlling in order to be able to regulate the load caused by the reconfiguration. We refer to this first phase as “collect phase”. When all the storage nodes have finished collecting, the first phase is over and the second phase is started.

In this second phase, the algorithm or table that matches classifications with storage groups is altered to the new configuration. Then items are only stored and requested from the storage node that stores the item in the new configuration. We refer to this phase as “reconfiguration phase”. After this, the third phase can start.

In the third phase, all the storage nodes delete media items from the classifications that should not be stored anymore on that storage node. One could implement rate controlling to be able to regulate the load caused by deleting the media items. We refer to this phase with “delete phase”. When all storage nodes finished the deletion, the third phase is over.

When the third phase is completed, the reconfiguration is completed. The same procedure can be used to reconfigure to increase or decrease the number of storage nodes.

During reconfiguration, additional space on the storage nodes is required, since for a period of time, some media items can be stored multiple times: on the original group and the group for the new configuration. This is caused by making available the data items for both the current number of storage nodes and the new number of storage nodes. Consequence from this is also that one cannot expand the storage array when the maximum capacity is reached. It is necessary to monitor the storage array and start expanding in time.

Experience with the new storage setup should be gained to answer questions like how long does it take to reconfigure.

Because of the use of heterogeneous capacities, it is possible to extend the storage array with a number of small storage nodes in order to increase serving capacity or a large storage node in order to increase storage capacity or mix. The storage is scalable and can be extended by a storage box and a front-end Web server per expansion. But it is also possible to expand by using a i larger amount of storage boxes and front-ends.

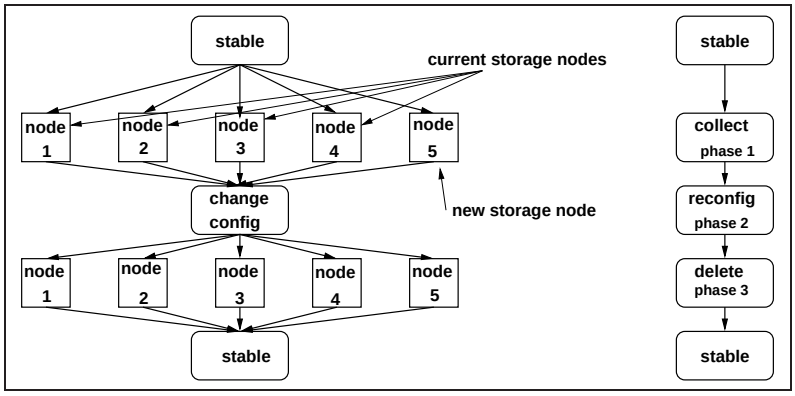

Figure 4.19: **Reconfiguration procedure**

Now we achieved to create a storage that can be extended if storage capacity demands increase and that guarantees distribution of the media items according to the capacity of the storage nodes, we can extend our design to create redundancy to satisfy the redundancy requirement.

To achieve a redundant setup, we have to incorporate redundant storing of the media items in our design. We do this by partitioning the space on a storage node into two halves. On the first halve, data items are placed. The second halve contains a copy of the first halve of another storage node. Because the storage arrays may have different sizes, this is not an option. For example, the backup of a large storage node might not fit on a smaller storage node. This can be solved by shifting the classification space with a phase equal to the largest phase of a group.

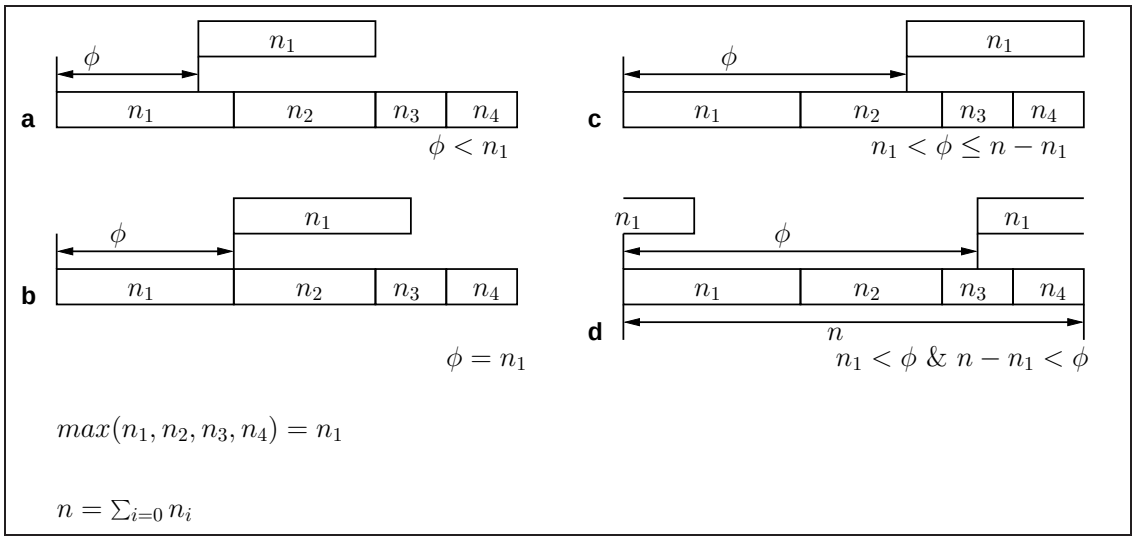

Figure 4.20: **Phase shift explanation**

In Figure 4.20 (a) one can see that when using a phase shift (ϕ) smaller than n_1 ($\phi < n_1$) then the backup of that partition partly is stored on the same storage box. When using a phase shift equal to the phase of the largest partition (Figure 4.20 (b)) then this is not the case. Even a larger phase shift is possible (Figure 4.20 (c)), but it should not exceed $n - n_1$, since then the backup is partly stored on the same storage node (Figure 4.20 (d)). If necessary, redundancy can be expanded by using a larger share of the storage node for backup (e.g., dividing the capacity of a storage node into three equal parts, the first partition for the data and the other partitions for the backup of other media items).

From the same Figure, we can also derive that the capacity for the largest storage box must not exceed half of the capacity of the storage array, because then a shift of the hash space by more than a half of the capacity results in an overlap, and therefore causing that it is not possible to store the backup of the largest storage node on other parts of the storage array. We can also derive that the minimum number of storage boxes in the storage array equals two.

In case of failure of one storage node, we can alter the algorithm or table that matches the classifications with the storage groups. When the storage node is replaced, it can be filled with the data from the backup.

Now we have extended our design with redundancy, we have to alter the reconfiguration procedure and requesting and storing a media item. In the reconfiguration procedure, the collect-phase also has to incorporate collecting backup-media items next to collecting media items. The delete-phase also has to incorporate deleting backup-media items.

When a media item is stored, it also has to be stored on the storage node that handles the backup. When a media item is requested, the request procedure has to be aware of possible failures. In case of failure, the request is retrieved from the backup-location.

The application and its daemons should be modified to use the new storage architecture and to get and put files using FTP or HTTP.

Just one configuration file is the basis for the configuration files per storage node, front-end and tool.

The following steps have to be taken in order to implement the designed solutions: Storage nodes have to be prepared. Web server software should be installed.

Next to hardware preparations, four programs/scripts are to be written:

- a script that can collect the necessary data items after adding or before removing a storage array. The script has to take into account the current number of storage nodes and the number of storage nodes

after wards.

- a script that can delete unnecessary data items after adding or removing a storage array. The script has to take into account the new number of storage nodes.
- a script that stores a data item on the storage nodes. The script has to take into account to also store the data item in the correct storage node that serves the backup. Further, the script should take into account the fact if the array is expanding or shrinking, so the new placed data item is available in both old an new situations (if the expanding of shrinking affects the location of the data item).
- a script that fetches a data item from the correct storage node.

On the servers that act as front-end to the storage array, HTTP and FTP daemons should be configured.

A procedure for extending, removing or restoring a backup has to be written.

We have to find a good solution to security. FTP uses plain-text passwords. This is no problem when using FTP internally, but when the traffic has to traverse the Internet because of multiple co-locations, this is not desired. We have to research how to make this more secure. Secure FTP might be an option, as well as using SSL, SHFS or tunneling the traffic using a VPN.

Now, we can describe the migration process, in which we transfer the current implementation to the new implementation.

For the old design storage cluster, additional CoRaids are ordered. These CoRaids could be used to build the new storage setup.

1. Machines need to be prepared (e.g., formatting, partitioning, creating LVM volumes, making file systems, ftp daemon configuration)
2. Create the scripts that are mentioned above.
3. Create a script that collects current media items from the originals repository and the rendered media repository and that injects the media items into the new storage.
4. Test if this goes OK.
5. Test 'shrinking' and 'expanding' scripts.
6. Inject additional media items.
7. Test application with the new storage.
8. Disable uploads. Finish renderings and photo orders.
9. Deploy code changes
10. Restart uploads, renderings and photo orders.

In case of emergency during the deploy:

1. deploy previous version of the code
2. restart old uploads, restart old renderings and photo orders.

After a successful out-roll of the new storage

1. break down old storage cluster
2. expand new storage with old storage cluster hardware

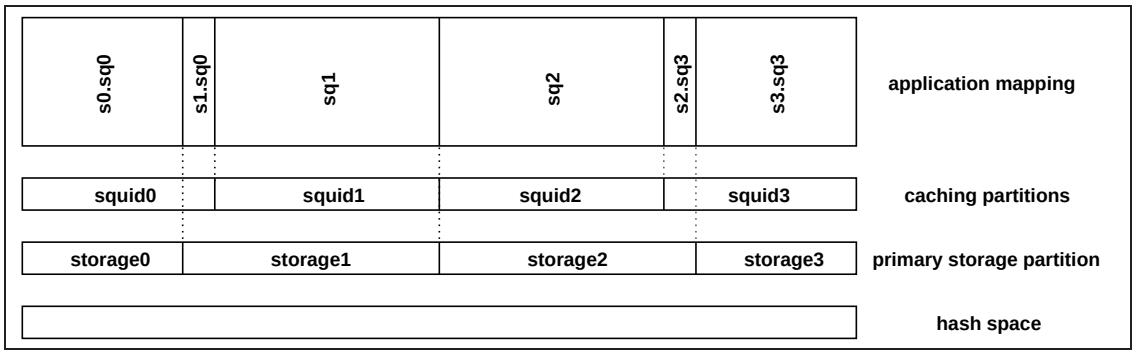

Figure 4.21: **Logical Presentation Layer Design on top of Logical Storage Design. The backup layer is not displayed in this figure.**

4.7 Storage and Serving Review

Now we designed a new storage solution, we also might want to redesign the serving solution.

Instead of creating a new partitioning scheme, we use the same one as used in Section 4.6. In this way, we can take advantage of the same benefits: minimize the number of re-mappings and an equal distribution of requests.

The number of caches or their sizes does not necessarily have to correspond to the number of storages. This implicates that it is needed for a cache to be able to direct requests to multiple storages.

In the current use of the cache, it has a single accelerating host and multiple cache peers (with each the same information). We need a way to tell the cache to which storage node to forward the request.

We do this by introducing extra domain names. Suppose `http://interval0.rendered.startpda.net` has a DNS A record, and `http://s0.interval0.rendered.startpda.net` has not, than the latter directs to the same address as the one with an A record.

Squid is able to notice the difference between a request starting with `s0` or without by means of ACLs (access control list). We can use the ACLs to redirect requests to a certain storage by using the options in Figure 4.22.

```
acl s0url dstdom s0.sq0.rendered.startpda.net
acl s1url dstdom s1.sq0.rendered.startpda.net
cache_peer storage0 no-query 80 allow s0url
cache_peer storage1 no-query 80 allow s1url
accelerating_host interval0.rendered.starptda.net
```

Figure 4.22: **ACLS**

By the use of the accelerating host line, all s0.interval0.rendered.startpda.net are replaced with interval0.rendered.startpda.net.

In this way we can create the scheme in Figure 4.21.

4.8 Conclusion

In this chapter we have discussed our research and design activities for Media Serving and Media Storage. We started with discussing the initial currently experienced performance problem in which the maximum

serving capacity of the second interval was reached. Rendered media items in the second interval were requested more often, so a solution was designed in which rendered media items were moved from the second to the first interval in order to balance the load of requests to both intervals.

Because of the growth of the number of media items, we realized this is just a temporarily solution, and therefore we continued our analysis. We identified that because of the increasing number of rendered media items and the used scheme to store the rendered media items, an unbalance in the number of requests and the number of rendered media items per interval requires (continuous) maintenance to create a balance.

From this, we derived two strategies to solve the problem of the unbalance between the number of stored and requested items per interval: we analyzed a caching solution to decouple serving and storing rendered media items per interval and we designed a new storage solution that enforces a balance in the number of stored and requested items per interval. Both strategies can be used simultaneously and such a solution is designed and presented.

Chapter 5

Media processing

5.1 Introduction

This chapter discusses the analysis and improvements we did on the Autorender daemon, the processing service of the media part of the architecture.

We identified problems with the Autorender Daemon when we analyzed and improved the Media Storage. These problems include:

- unexpected set-backs of the value of `coldmaxmediaid`, causing media items being rendered multiple times;
- not using all processor capacity, while there are items to be processed;
- NFS problems causing the Autorender Daemon to stall;
- a problem with the used render software, in which the render software enters an infinite loop causing the Autorender to stall.

For each problem in this chapter, we present a problem description, an analysis, requirements for possible solutions, possible solutions and a motivation for choosing one of the solutions. From each problem, we derive requirements for a new version of the Autorender Daemon that overcomes the current problem. Therefore we combined all solutions and requirements to create a new design for the Autorender Daemon. This design has been implemented and we evaluate our design and implementation. Therefore the structure of this chapter reflects the diagram in Figure 5.1.

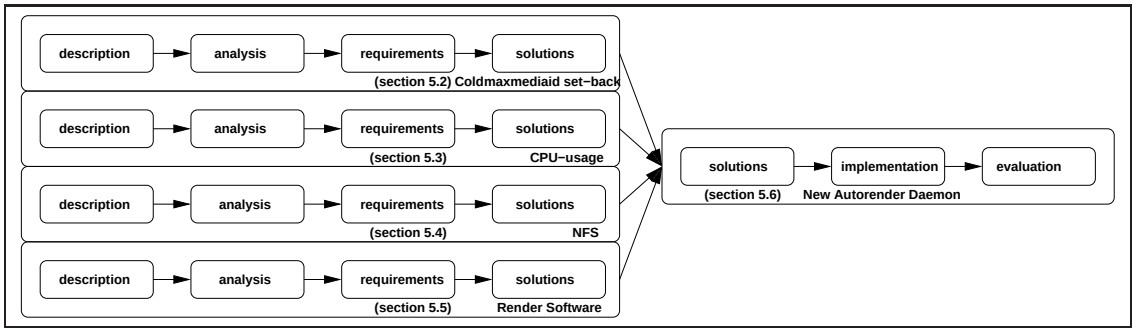

Figure 5.1: **Structure of each discussed research.**

5.2 Coldmaxmediaid set-back

5.2.1 Problem description

Because of the experienced problems, we investigated the Autorender Daemon's log files, in which the Autorender Daemon periodically writes its status (e.g., number of media items to render, timestamp, if the daemon is busy rendering or copying of media items, etc.). We noticed that sometimes the number of items to render suddenly increased with additional 50.000 media items. This behavior was not anticipated, so we consider this behavior to be unexpected and undesirable.

5.2.2 Analysis

The number of items to be processed depends on two factors: the number of media item uploads per period of time and the number of media items that are processed per period of time. We illustrate this with Figure 5.2.

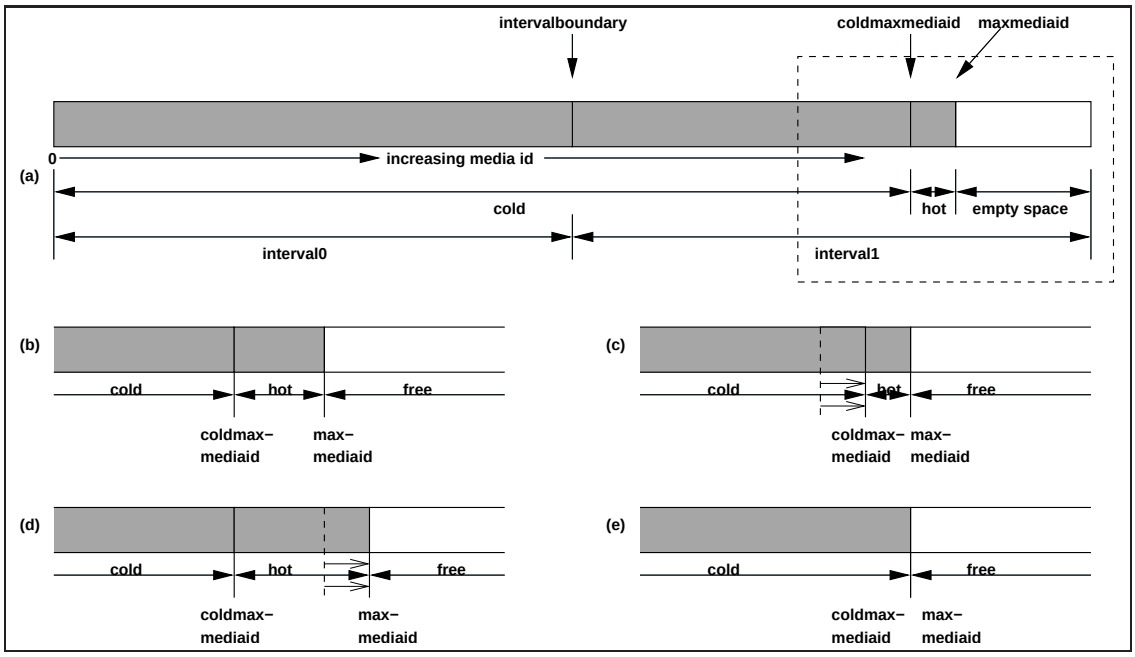

Figure 5.2: **"Hot and cold" setup analyzed: (a) Hot and cold setup as discussed in Chapter 2. In (b), (c), (d) and (e), we focus on the area surrounded with the dashed rectangle. (b) Initial situation. (c) The Autorender Daemon increases coldmaxmediaid id because media items were rendered. (d) The Web site PHP code increases maxmediaid because media items were uploaded. (e) Steady state. The Autorender Daemon has rendered all uploaded media items.**

We focus on the area that is surrounded by the dashed rectangle in Figure 5.2 (a). This area is displayed in 5.2 (b), showing the cold part, which contains media `id`s from rendered media items, the hot part, which contains media `id`'s from media items which are not rendered in all formats, and free space, in which media `id`s from uploaded media items can be placed. Figure 5.2 (c) shows an increased value for `coldmaxmediaid`. This increased value of `coldmaxmediaid` is set by the Autorender Daemon after finishing rendered a number of hot media `id`s. Figure 5.2 (d) shows an increased value for `maxmediaid`, which is set by the Web site PHP code after storing uploaded media items. Figure 5.2 (e) shows the steady state in which the Autorender Daemon has rendered all hot media `id`s. The value of `coldmaxmediaid` and `maxmediaid` are equal in this situation and as a consequence, the hot part contains no media `id`s.

From Figure 5.2 we can derive that the hot part actually works as a queue: items are removed at the head of the queue by the Autorender Daemon and items are inserted at the tail of the queue when they are uploaded. The number of media `id`s in the hot part equals the length of the queue. Because the Autorender Daemon and uploading of media items are executed in parallel, the queue length changes over time. When per period of time more media id's are rendered (i.e., removed from the queue) than uploaded (i.e., inserted in the queue), the queue length decreases.

As explained, the number of items to be processed by the Autorender Daemon depends on the number of items processed per period of time and the number of items that were uploaded in that same period of time. Therefore the cause of the sudden increase of the number of items to be processed as experienced with the Autorender Daemon is limited to two factors: a sudden increase of `maxmediaid` or a sudden decrease of `coldmaxmediaid`. Further investigation of the log file of the Autorender Daemon showed that the value of `coldmaxmediaid` decreases with the amount of 50.000 items.

Because the Autorender Daemon is programmed to render the media items with a media `id` between the values of `maxmediaid` and `coldmaxmediaid` (see Chapter 2), this results in 50.000 media items being rendered and copied to the rendered media storage more than once. However:

- a media item that is processed by the render daemon does not need to be processed again. Processing a media item more than once is a waste of time and CPU power;
- because of the URL generation scheme (see Figure 2.11) and the interaction of the components (see Figure 3.3) large queue length implies that the Web site Web servers spend time and CPU power on the rendering of media items, instead of serving pages.

We want to identify how often and at which times this decrease in the value of `coldmaxmediaid` occurs to identify the impact of the problem. We analyzed this by monitoring the number of media items that need to be rendered. Therefore we created a script that allowed us to measure at any moment in time the number of media items that needed to be processed. We used this script and the Hyves's monitoring tool to create a graph of the number of items to be rendered. This has been a first step towards application monitoring (see the Architectural Principles of Chapter 2).

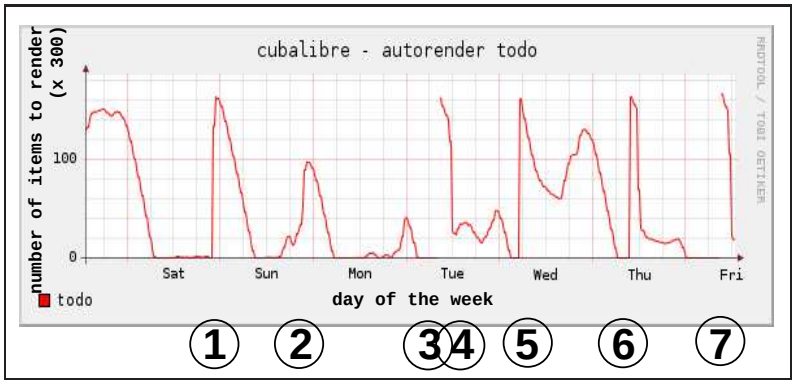

Figure 5.3: **Visualization of the queue length of the Autorender daemon.**

The resulting graph is displayed in Figure 5.3. The x-axis of the graph denotes the time of the day and the y-axis of the graph denotes the number of media items that need to be processed at that moment in time.

① in Figure 5.3 indicates an occurrence of that the `coldmaxmediaid` was set back with 50.000 items. The number of items that need to be processed was around 0 for some time and suddenly the number of items is increased by 50.000.

② in Figure 5.3 indicates a stale NFS mount. When a server experiences a stale NFS mount, the server is not able to read or write to an NFS network share. Because the daemon depends on these shares for

writing the rendered media and reading the original media files, the daemon is not able to process media items. Therefore the curve at ② is more steep. We identified that the server experienced a stale NFS mount by analyzing the server logs.

③ indicates another occurrence of that the `coldmaxmediaid` was set back with 50.000 items. We fixed the error manually by restoring the `coldmaxmediaid`. ④ shows this manual restore of the `coldmaxmediaid`. ⑤, ⑥ and ⑦ show the set-back and the manual restore of the value of `coldmaxmediaid`.

By interviewing the programmer of the Autorender Daemon, we learned that only when the file which holds the value of `coldmaxmediaid` is not readable or does not exists, the value of `coldmaxmediaid` is set to the value of `maxmediaid` minus 50.000 items in order to guarantee the robustness of the daemon.

From Figure 5.3, we can conclude that this happens at least once a day. Since the file containing the value of `coldmaxmediaid` is located on an NFS-share, it is most likely that a stale NFS mount is the cause of this error. The system's log files has confirmed our suspicions. Therefore, we can conclude that the cause of the set back of the `coldmaxmediaid` is that the file containing the `coldmaxmediaid` is on a NFS share, and this share becomes stale.

Because there is just one Autorender Daemon process, in case of failure, the number of items that need to be processed increases immediately. This increase would be less stringent if there were multiple instances of the Autorender process, because it is less likely that all instances would experience the same problems at once.

From the above analysis, we can draw the following conclusions:

- The problem of the value of `coldmaxmediaid` being set-back by 50.000 media items is caused by the variable being located in a configuration file which is placed on a NFS share;
- Stale NFS handles cause the Autorender Daemon to stall;
- Because there is one Autorender Daemon process and there is only one server running the Autorender Daemon, in case of failure the number of items to be processed increases quickly.

We have to research how to prevent or anticipate these stale NFS shares and we have to investigate if it is possible to run multiple Autorender Daemons that can share the load, and in case of a failure of one Daemon, items can remain being processed.

5.2.3 Requirements

Based on our analysis, we define the following requirements for a new architecture for the Autorender Daemon:

- A mechanism different from NFS shares or memory has to be designed for sharing the value of `coldmaxmediaid`, because stale NFS handles caused the loss of this value. Sharing this variable in memory is also not an option, because the value would be lost in case of failure.
- Multiple daemons should enable redundancy to keep the service running.

5.2.4 Solutions

A solution is to share the value of `coldmaxmediaid` is to use a database for this purpose. The `maxmediaid` is also shared using the database, so it is possible to do the same for `coldmaxmediaid`. Because the database uses a master-slave setup (see Chapter 2), the value remains available even in case of failure of a database slave.

As a solution to create redundancy, we can improve the design of the Autorender Daemon to enable the use of multiple instances on different machines.

5.2.5 Implementation

The design and implementation of a solutions that fulfills the requirements above are discussed in Section 5.6.

5.3 CPU usage

5.3.1 Problem description

In Section 5.2 we introduced application level monitoring of the Autorender Daemon. When we apply this in combination with the available system level monitoring (see Figure 5.4) we conclude that although there is a large number of media items that need to be rendered, the machine only uses half of its processing capacity.

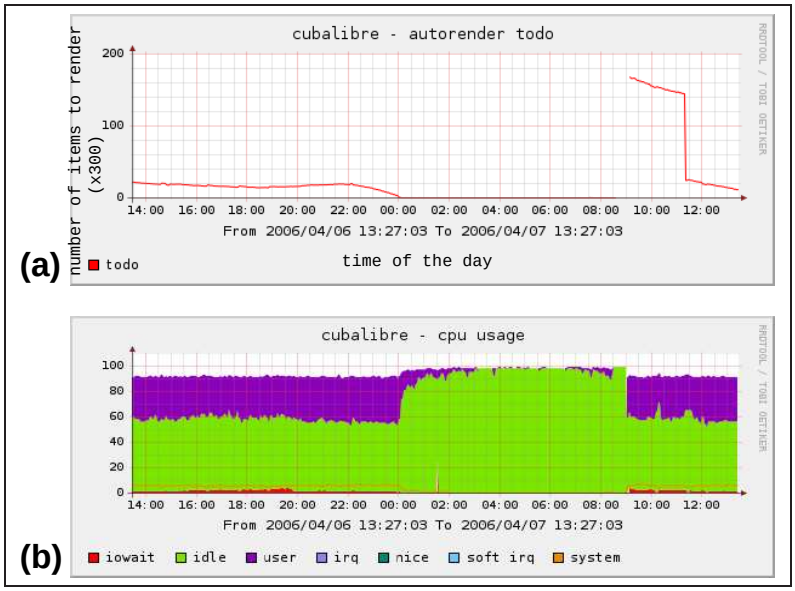

Figure 5.4: **Autorender Daemon Queue and Processor Usage. (a) Length of the Autorender daemon queue in time (b) CPU-usage of the machine that runs the Autorender daemon.**

Because of the Hot and Cold setup (see Chapter 2), not rendered media items are served by the Web site Web servers. Therefore when the number of not rendered media items increases, the number of requests that are processed by the Web site Web servers instead of the Media Service Web servers also increases. The analysis in Section 4.3 showed that recently uploaded media items are requested relatively more then not recently uploaded media items. Therefore when the number of items that need to be rendered increases, the load of rendered media items requests shifts from the Media Service Web servers to the Web site Web servers, resulting in a performance decrease.

Therefore, we have to find a way to minimize the number of media items that need to be rendered.

5.3.2 Analysis

In our analysis, we translated this problem to a queuing theory model. In queuing theory's context, a service station is a group of servers, a server is a process that processes the items in the queue, which are called clients or requests (the media items) [ASIQT], [IQT]. In Figure 5.5. a server machine can be seen as a

service station. The server machine contains multiple processing units on which we run the Autorender Daemon process, which can be seen as a server. The hot media `ids` can be seen as clients in the queue.

In the analyzed situation, there is only one machine running one instance of the Autorender Daemon process.

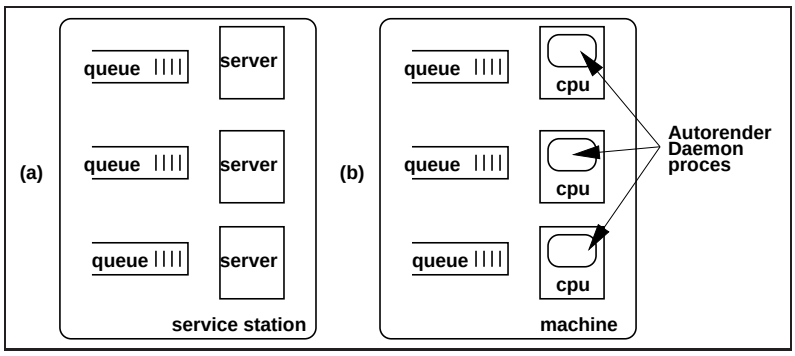

Figure 5.5: **Queuing Theory Definitions and Analogy. (a) Queuing Theory Definitions (b) Analogy to our situation.**

In the queuing theory, there exists three options in order to decrease the queue length:

- increase the number of servers per service station;
- increase the performance of the server (lowering the average service time per client);
- increase the number of service stations.

We can translate these options from queuing theory to real solutions.

Increasing the number of servers would result in running multiple Autorender Daemon processes on a single machine. The aggregate processing capacity of the servers can be used to render the media items. For this option, the machine has to have multiple processing units, otherwise multiple processes have to compete for one processing unit, which does not result in improved performance.

Increasing the performance of the server would result in using faster processing unit per machine. In that case, Autorender Daemon process renders more items in the same time.

Increasing the number of service stations would result in using multiple machines, and an instance of the Autorender Daemon process would run on each machine. The aggregate processing capacity of the service stations can be used to render the media items. Next to increasing performance, this also provides redundancy: in case of failure of one machine, another machine continues to render media items.

We can also combine the three options. For the first and third option, we have to analyze if it is possible to have multiple instances of the Autorender Daemon running and if these instances do not interfere with each other. To perform this analysis, we have to inspect the code of the Autorender Daemon, in order to identify possible concurrency problems.

For the second option, we have to analyze why only half of the processing capacity of the machine is used. Furthermore, we can analyze the Autorender Daemon in order to optimize the process. For this, we also have to inspect the code of the Autorender Daemon.

We present the Autorender daemon in pseudo code in Figure 5.6.

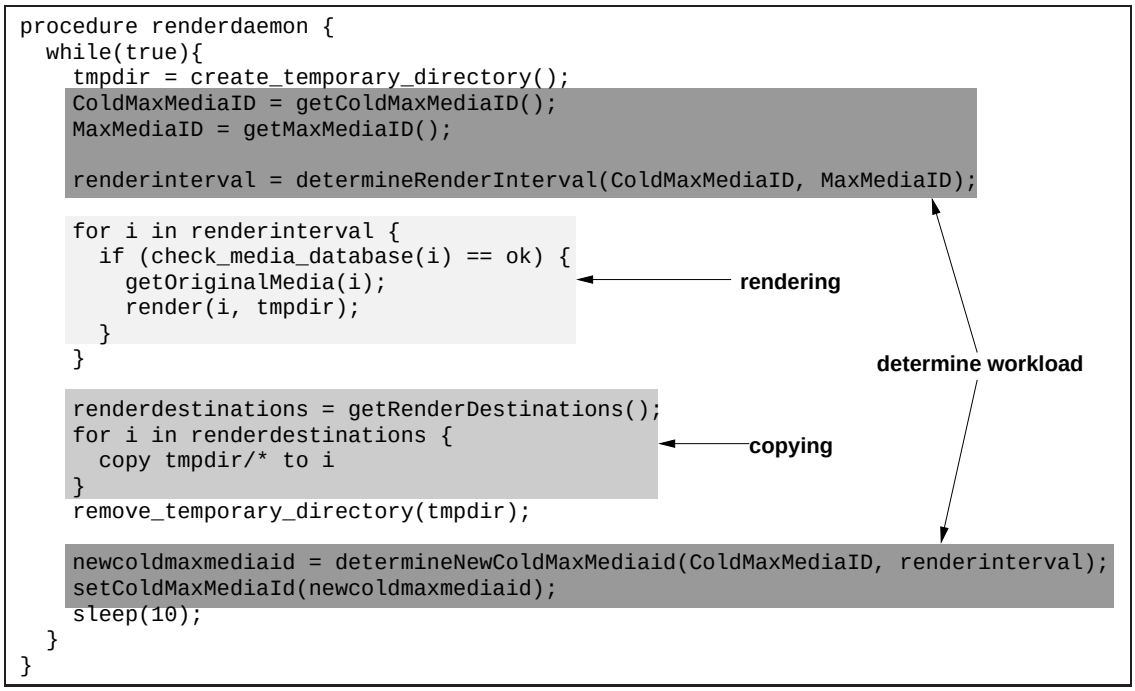

Figure 5.6: **Pseudo code of the Autorender daemon.**

From Figure 5.6 we conclude that the work of the Autorender Daemon consists of three sequential steps:

1. determine which items to render;
2. render the media items;
3. copy the rendered media items to the Rendered Media Storage.

The Autorender Daemon runs on a “dual core single processor” machine. A dual core single processor machine has two cores that can execute instructions. To benefit from this, there should be multiple tasks (at least two) because a task can not run on two cores simultaneously. From the code inspection was learned that the work of the Autorender Daemon is done sequentially. This explains why only a half of the processing capacity of the machine was used.

Because of the above sequence, we can derive that when the daemon is rendering, no items are copied, and when the daemon is copying items to their destinations, no media items are rendered.

From Figure 5.7, which is part of a log file from the Autorender Daemon, we conclude that rendering and copying the media items take a relatively long time, and the getting and setting the values of `coldmaxmediaid` and `maxmediaid` do not. System analysis tools showed that rendering media items utilizes a lot of CPU cycles and that copying rendered media items utilizes much less CPU cycles. We therefore conclude that rendering media items takes a relatively long time because it takes a lot of CPU power and that copying the rendered media items takes a relatively long time because the process has to wait for responses from the Media Storage.

If we can perform the rendering and copying of rendered media items in parallel, there would be two or more tasks simultaneously and therefore the processing capacity of a dual core processor is used more efficiently. There are also dependencies to be taken into account: media items must be rendered first, before the rendered media items can be copied to the repositories.

The `determineRenderInterval()` function gets the values of `coldmaxmediaid` and `maxmediaid` and uses these to calculate the render interval. Suppose multiple instances of the Autorender Daemon process

```
Tue Mar  7 09:51:40 CET 2006: Daemon render_media started
09:51:41 Creating temp directory
09:51:57 Starting autorender, ids=[12174930,12175429] (500) maxMediaId=12206771; (todo: 31842)
09:51:57 Done 0, now at mediaId 12174930; Memory used: 3.769736*10^6 bytes
09:52:26 Done 50, now at mediaId 12174980; Memory used: 4.637984*10^6 bytes
09:52:57 Done 100, now at mediaId 12175030; Memory used: 5.047328*10^6 bytes
09:53:35 Done 150, now at mediaId 12175080; Memory used: 5.457184*10^6 bytes
09:54:06 Done 200, now at mediaId 12175130; Memory used: 5.863152*10^6 bytes
09:54:23 Done 250, now at mediaId 12175180; Memory used: 6.269024*10^6 bytes
09:54:54 Done 300, now at mediaId 12175230; Memory used: 6.679976*10^6 bytes
09:55:26 Done 350, now at mediaId 12175280; Memory used: 7.085808*10^6 bytes
09:55:58 Done 400, now at mediaId 12175330; Memory used: 7.491864*10^6 bytes
09:56:32 Done 450, now at mediaId 12175380; Memory used: 7.897232*10^6 bytes
09:57:11 Finished rendering
09:57:11 Copying to /var/MEDIARENDEREDTARGET/cl13
10:00:02 Copying to /var/MEDIARENDEREDTARGET/cl15
10:04:37 Copying to /var/MEDIARENDEREDTARGET/cubalibre
10:05:34 Setting coldmediaid
10:05:34 removing temp dir
Tue Mar  7 10:05:35 CET 2006: Daemon render_media stopped (ended with returnvalue 42)
```

Figure 5.7: **Autorender Daemon Log file. In this run of the Autorender Daemon, rendering 500 media items takes approximately 5 minutes and copying the rendered media items approximately 7 minutes. Creating and removing the temporary directories and calculating the render interval takes multiple seconds.**

are running, then they might calculate the same or overlapping render intervals. This would result in rendering the same media items multiple times, which is a waste of CPU power. Furthermore, when multiple Autorender Daemon processes set the value of `coldmaxmediaid`, the processes would set it with respect to their local stored value, not taking into account media items rendered by other processes, which can lead to an incorrect value. Therefore the daemon in this form is not multithread safe.

5.3.3 Requirements

New requirements for the Autorender daemon therefore are:

- rendering and copying of media items should be done in parallel;
- parallel processing should be allowed to optimize the capacity of the machine;
- the Autorender Daemon should allow the use of multiple instances without causing interference of the other instances to enable activation of multiple instances if there is a demand for more performance.

5.3.4 Solutions

To be able to render and copy media items in parallel while taking the render-before-copy dependency into account, we can construct a solution with pipelines [PL]. A pipeline can be understood as an assembly line in which the first step is to render the media item and the second step is to copy the rendered media items to the media storage. If there are multiple processing units, each processing unit can execute one step of the pipeline, and therefore throughput is increased when compared to the initial Autorender Daemon design. The code of the Autorender Daemon should be modified to integrate such a pipeline construction.

To allow multiple instances, the code of the Autorender Daemon should be modified so that each instance of the Autorender Daemon process receives or calculates a unique set of media items to render. Further, the value of `coldmaxmediaid` should be set unambiguously.

5.3.5 Implementation

The design and implementation of a solution that fulfills the requirements above are discussed in Section 5.6.

5.4 NFS

5.4.1 Problem description

NFS is a network file system that enables hosts to share file systems via the network. The Autorender Daemon requests the Media Storage via NFS for media items and stores the rendered media items to the Rendered Media Storage via NFS. Further, the value of `coldmaxmediaid` is also stored in a file, which is placed on a NFS share.

We have seen two problems related to NFS in our analysis:

- stale NFS mounts. In case this problem occurs, suddenly a remote file system is not accessible and the process using remote file system access halts, while the NFS server apparently remains accessible to other clients. Sometimes multiple clients experience the problem at the same time, sometimes only a single client experiences the problem. Sometimes the service is inaccessible for a small period of time and the service is restored automatically. Sometimes the service remains unaccessible and the service has to be restored with intervention;

- slow NFS mounts. Because NFS uses the network, it can suffer from heavily loaded (and sometimes congested) networks. NFS also implies that a machine becomes dependent of how fast NFS requests are processed at the server side. Figure 5.7 shows how a slow NFS mount affects the daemon. Figure 5.8 shows a log file of a Autorender Daemon run in which no slow NFS mount was experienced. Notice that copying rendered media items in Figure 5.7 takes approximately 7 minutes, while in Figure 5.8 it approximately takes 20 seconds.

Both problems stall the Autorender Daemon because of the sequential nature of the Autorender Daemon process, as described in Section 5.3.

```
Fri Apr  7 14:45:59 CEST 2006: Daemon render_media started
14:46:00 Creating temp directory
14:46:00 Starting autorender, ids=[14398900,14399399] (500) maxMediaId=14401533; (todo: 2634)
14:46:00 Done 0, now at mediaId 14398900; Memory used: 7.46272*10^6 bytes
14:46:28 Done 50, now at mediaId 14398950; Memory used: 8.37708*10^6 bytes
14:46:57 Done 100, now at mediaId 14399000; Memory used: 8.843544*10^6 bytes
14:47:20 Done 150, now at mediaId 14399050; Memory used: 9.311224*10^6 bytes
14:47:50 Done 200, now at mediaId 14399100; Memory used: 9.77268*10^6 bytes
14:48:28 Done 250, now at mediaId 14399150; Memory used: 10.237456*10^6 bytes
14:48:53 Done 300, now at mediaId 14399200; Memory used: 10.691832*10^6 bytes
14:49:24 Done 350, now at mediaId 14399250; Memory used: 11.15076*10^6 bytes
14:49:42 Done 400, now at mediaId 14399300; Memory used: 11.612056*10^6 bytes
14:50:01 Done 450, now at mediaId 14399350; Memory used: 12.073608*10^6 bytes
14:50:30 Finished rendering
14:50:30 Copying to /var/MEDIARENDEREDTARGET/cl13
14:50:39 Copying to /var/MEDIARENDEREDTARGET/cl15
14:50:48 Copying to /var/MEDIARENDEREDTARGET/cubalibre
14:50:51 Setting coldmediaid
14:50:51 removing temp dir
Fri Apr  7 14:50:52 CEST 2006: Daemon render_media stopped (ended with returnvalue 42)
```

Figure 5.8: **Autorender Daemon Log file. In this run of the Autorender Daemon process, rendering 500 media items takes approximately 4 minutes and copying the rendered media items approximately 20 seconds.**

5.4.2 Analysis

Stale NFS mounts is a problem we experienced on multiple hosts in the Hyves internal network. We and the Hyves system administrators have made several efforts to optimize NFS settings in order to prevent this problem, but at present, no solution has been found to fix this problem.

Stale NFS mounts are local: host A can experience a stale NFS mount to server B, while host C using the same share to server B does not experience a stale NFS mount.

Slow NFS mounts are caused by heavily loaded hosts. In this case, the hosts are the machines that store and serve the rendered media items. Decreasing the load of these machines is discussed in Chapter 4. If a server gets slow, all connected hosts experience the slow NFS mount.

Although there are no solutions to fix this directly, we can anticipate to the situation.

5.4.3 Requirements

From the problems at the Autorender Daemon caused by the use of NFS, we can conclude:

- As a requirement for the new Autorender Daemon, rendering and copying of the media items should be decoupled so a stale or slow NFS mount does not stop the Autorender Daemon entirely;
- Allowing multiple instances of the Autorender Daemon can overcome a local stale NFS mount.

5.4.4 Solutions

For example, it is possible to introduce a queue to hold rendered media items. In case of a slow NFS mount, the Daemon can keep rendering without having to wait for rendered media items to be copied and therefore the processing capacity is more efficiently used. In case of stale NFS mount, the Autorender Daemon can keep rendering, and when the mount is manually restored, the Autorender Daemon can restart copying of the rendered media items. This also implies that we have to monitor such a queue, because it may be possible that numerous media items are rendered, but are never copied to the Rendered Media Storage.

5.4.5 Implementation

The design and implementation of a solution that fulfills the requirements above is discussed in Section 5.6.

5.5 Render Software

5.5.1 Problem description

Sometimes the Autorender Daemon stalls. Symptoms are that the `convert` process, which is spawned for every rendered media item, takes all CPU-time of one core for a long period of time. For example, after three hours of CPU-time a single media item still has not been rendered. Currently, we can only detect this by logging in to the host running the Autorender Daemon and using system analysis tools to request the time spent on the current running `convert` process.

5.5.2 Analysis

To be able to detect these events, we created a script that uses the age of file in which the value of `coldmaxmediaid` is saved. After rendering 500 media items, this value is updated and this is done by generating a new file with the value of `coldmaxmediaid`. When the file is created, the file system saves the date and time of the file creation.

The script compares the timestamp of the file with the current date and time. We combined this script with our monitoring tool. Since rendering of 500 media items should be done within half an hour, we got an alarm from our monitoring tool if the age of the file is more than half an hour.

At each alarm we investigated the log files from the Autorender Daemon and the host that runs the Autorender Daemon. It appeared that the symptoms only occur when the Autorender Daemon is rendering a video file. From the log files we concluded that the `convert` command entered an infinite loop.

The Web site of the used tool does not report a problem similar to the problem described above. A solution would be to analyze which specific type of video files cause the `convert` command to enter an infinite loop (e.g., by debugging) but this would not fit in the scope of our research.

Therefore we design a workaround instead of a solution.

5.5.3 Requirements

In our workaround, we require that the maximum CPU-time the `convert` command is allowed to use, should be limited.

5.5.4 Solution

A workaround has been designed by creating a wrapper that runs the `convert` command and sets a timer each time the `convert` command is started. The wrapper then uses a time-out. If the time-out elapses, the started `convert` command can be killed and the next media item can be rendered. Media items that can not be rendered, can be requested via the Web site, but are not displayed, since there is no rendered media item.

5.5.5 Implementation

Because the Autorender Daemon is written in PHP and uses the Bash [BASH] shell to execute commands, we used the `ulimit` built-in command from Bash to limit the resources for the `convert` command. From the manual of Bash [BASHM]:

> Provides control over the resources available to the shell and to processes started by it, on systems that allow such control. [...] options are interpreted as follows: [...] -t: The maximum amount of CPU time in seconds.

The code of the Autorender Daemon was adapted from starting a Bash shell that starts a `convert` command to starting a Bash shell that limits all started processes to 300 seconds CPU-time and then starts a `convert` command.

5.5.6 Evaluation

The problem was not experienced any longer.

We conclude that although we did not fix the problem, the Autorender Daemon became more robust.

5.6 Design

5.6.1 Problem description

In previous sections we analyzed problems with the Autorender Daemon and proposed solutions to fix the problem, but we postponed the design and implementation of a solution for the Autorender Daemon to this section. Therefore this section contains a new Autorender Daemon design.

5.6.2 Analysis

In Section 5.2 we concluded that the value of `coldmaxmediaid` is set-back by 50.000 media items mainly because the variable is located in a file placed on a NFS share, NFS shares causes the Autorender Daemon to stall and since the Autorender Daemon is not redundant, a failure can cause the service to stop.

In Section 5.3 we concluded that performance of the Autorender Daemon is low because it uses only one of the two processing units. This can be improved if we parallelize the process in order to use both processing units and combine a CPU intensive and a not CPU intensive task. Furthermore, we analyzed the current limitations to create a solution with multiple instances.

In Section 5.4 we analyzed the current NFS problems and how these problems affected the performance of the Autorender Daemon.

In Section 5.5 we analyzed a problem with the used software that renders the media items and we created a workaround for that problem.

5.6.3 Requirements

The following requirements are collected from the previous sections:

- A mechanism other than NFS shares or memory has to be designed for sharing the value of `coldmaxmediaid`, because stale NFS shares caused the loss of this value. Sharing this variable in memory is no option either, because the value would be lost in case of system failure.
- Multiple daemons should enable redundancy to keep the service running.
- Parallel processing to optimize the capacity of the machine.
- The Autorender Daemon should allow the use of multiple instances without causing interference of the other instances to enable activation of multiple instances if there is a demand for more performance.
- Rendering and copying of the media items should be decoupled so a stale or slow NFS mount does not stop the Autorender Daemon entirely.
- Multiple instances should be allowed to overcome a local stale NFS mount.

5.6.4 Solutions

In Section 5.2 we suggest to use the database to store the value of `coldmaxmediaid`. In order to allow multiple instances of the Autorender Daemon to render different sets of media items, we can extend the current media table in the database with two columns. In the first column we maintain the status of a media item so an instance of the Autorender Daemon can claim a media item to render. The different states could be “rendered” for already rendered media items, “not rendered” for media items that are uploaded, but not rendered yet, and “claimed” to indicate that that particular media item is claimed by an instance of the Autorender Daemon. Then we have to modify the part of the Autorender Daemon that calculates which

media items to render. Instead of calculating an interval, the Autorender Daemon can request the database for a set of media ids that have the status “not rendered”. The Autorender Daemon then can claim these media ids by changing their status to “claimed”. If we use transactions to request and claim the media ids, we can prevent other instances to claim the same media ids. After the media items are rendered and the rendered media items are copied to the Rendered Media Storage, the status of the media items can be changed to “rendered”. Suppose an instance of the Autorender Daemon fails (e.g., because of power failure), then the claimed media ids are never released. To prevent this, we could the second added column to store a timestamp of the claim. Then we can use the timestamp to determine the age of the claim, and if a maximum age is reached, the claim is released and other instances can claim the media id to render the media item. We can create a process that checks the claims in the database and releases them if the claim reaches a maximum age, but we also can add this functionality to the Autorender Daemon every time it checks for new media items to render. The value of coldmaxmediaid then can be derived by finding the lowest media id that has not been rendered.

The current Autorender Daemon is written in PHP. From this implementation we know that PHP is not very suitable for creating daemons, because daemons have to run for long times and when a PHP process run for a long time, it suffers from memory leaks. Although multi-threading or creating multiple processes that communicate with each other is possible in PHP, PHP is mainly a Web server scripting language and therefore less suitable for creating multi-threaded or multi process applications when compared to other programming languages.

We can consider other programming or scripting languages, but since the entire Web site Code is created in PHP, it would be difficult to create a program in another language that has to use the same classes, objects and relations as used in the Web site. Furthermore, the Web site Code’s classes and objects are mapped to the database using a mapping tool, which is not available in other programming languages but PHP. A solution would be to create a daemon program in another programming language that is more suitable for this purpose, and let this daemon program call the PHP code to render a single media item. Since in this case, a PHP process would be created to render a media item, the chances of memory leak are reduced.

As a solution to the requirement to allow parallel processing, we can create multiple threads:

- a thread that renders the media items. We refer to this thread as the “RenderThread”;
- a thread that copies the rendered media items to the Rendered Media Storage. We refer to this thread as the “CopyThread”;
- a thread that interacts with the database in order to request for new media ids to render and that updates the database when the media items are rendered and copied. We refer to this thread as the “DatabaseThread”.

Because copying the rendered media items and interaction with the database should not be CPU-intensive, we can use multiple RenderThreads to fully utilize multiple processing units.

To allow interaction between the threads, we suggest the use of queues. The DatabaseThread determines the media ids that should be rendered and stores these in a queue. The RenderThreads monitor the queue and if the queue is not empty, they retrieve the first item in the queue and render it. After rendering, the RenderThread stores the media id of the rendered media items in another queue. This queue is monitored by the CopyThread. If that queue is not empty, the CopyThread retrieves the first item of the queue and copies the rendered media items to the Rendered Media Storage. After copying the rendered media items, the CopyThread stores the media id of the copied items in another queue, which is monitored by the DatabaseThread. The DatabaseThread releases the claim in the database and sets the status of the media item to “rendered”. This solution is displayed in Figure 5.9.

5.6.5 Implementation

We did not implement this design. The design was recommended to the programmers of Hyves, who implemented a similar solution. The differences between the proposed solution and the implemented solution

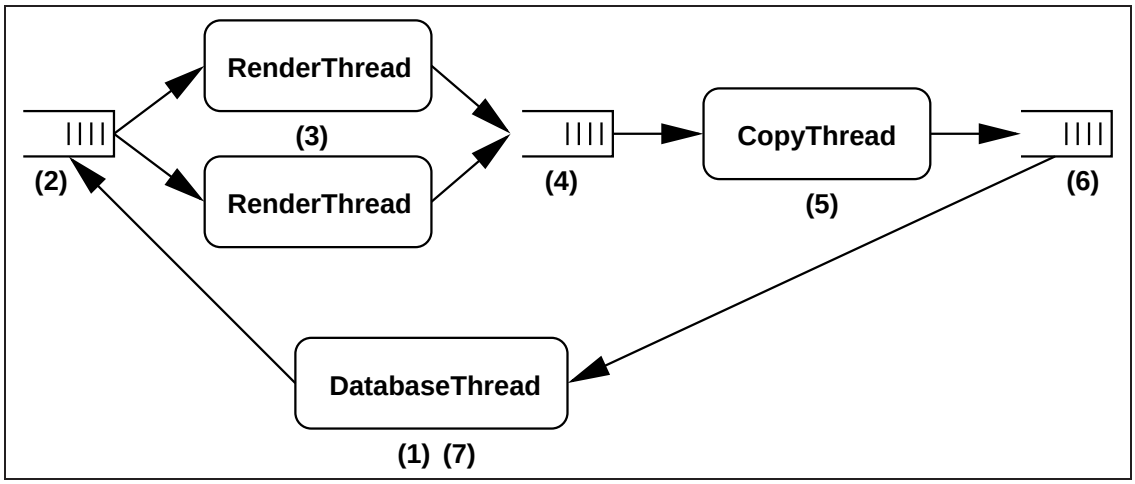

Figure 5.9: **New Autorender Design. (1) The DatabaseThread interacts with the database to determine which media items to render. It claims the media ids. (2) The ids of these media items are stored in a queue. (3) The media items are rendered by the RenderThreads. (4) The media ids of the rendered media items are stored in a queue. (5) The CopyThread copies those items to the Rendered Media Storage. (6) The media ids of the copied rendered media items are stored in a queue. (7) The DatabaseThread interacts with the database to release the claims and to adjust coldmaxmediaid.**

are explained below.

Instead of extending the media table in the database, Hyves decided to extend the database with an additional table. In this additional table, all recently uploaded media items that have not been rendered are stored. If a media item has been rendered, the entry is moved from this additional table to the media table. In the new table, the media item and details concerning possible claims are stored. When an instance of the Autorender Daemon claims a media item, the claim is registered in this table. Hyves developers also decided to create a controller process that monitors the claims and releases them if the claim reached a certain age. Hyves decided for an additional table because otherwise queries for finding media items with a state “not rendered” will consider millions of already rendered media items, while only a small part is not rendered. Hyves decided to create a controller process instead of adding this functionality to the Autorender Daemon because in their opinion creating a controller process is less complex than extending the current Autorender Daemon with functionality to release over-aged claims.

Hyves developers also decided not to extend the current design with multi-threading, but instead they created a solution that allows multiple instances on multiple machines. Therefore it is also possible to start multiple instances on the same machine, and thus multiple processing units are used more efficiently.

In addition, the design of the Web site code was modified and the “Hot and Cold” structure as discussed in 2 was removed. Rendered Media items can only be requested if they are rendered. After uploading a media item, users receive a notification that the uploaded media item will be processed and until that time, the item is not visible in their photo albums.

5.6.6 Evaluation

Using an additional table instead of our proposed solution to extend the current media table with additional columns is more efficient, because with our solution each time all the rows in the media table are considered (currently 35 million rows) and with the solution of the Hyves developers only the newly uploaded media items are considered (on average 1000 rows).

The introduction of an additional controller process instead of adding this functionality to the design of the Autorender Daemon is less robust, because each additional process could be a single point of failure.

Instead of using the multiple Autorender Daemon processes for this functionality, now at least two controller processes have to run.

Instead of multi-threading, allowing multiple instances on the same machine is an elegant solution in our opinion, but a little less efficient. In the case of multiple instances on the same machine, it is still possible that both instances are waiting for copying the rendered media items at the same time, and therefore leaving the processing units idle.

Removing the “Hot and Cold” structure is in our opinion a good design decision. We have identified two options: the Web site renders the uploaded media item to all formats and stores them directly into the Rendered Media Storage and obsoletes the Autorender Daemon, or the Web site does not render at all and the Autorender Daemon renders all the media items. The “Hot and Cold” solution is in between these solutions and has the disadvantages of both solutions: media items are rendered multiple times (by the Web site and the Autorender Daemon) and there is a queue which users are not aware of, which causes confusion why their uploaded media item is not visible.

After the implementation of the redesigned Autorender Daemon and running multiple instances on multiple machines, there has not been a large number of not rendered media items anymore. Multiple machines running multiple instances of the Autorender Daemon software keeps the queue of not rendered media items very short.

5.7 Conclusion

In this chapter we discussed the analysis we did on the Autorender Daemon. First we discussed the problem in which the value of `coldmaxmediaid` was set-back by the amount of 50.000 media items. We identified that this was caused by storing this variable in a file on a NFS share. We also identified that the machine running the Autorender Daemon was not fully utilized because the machine has a so called dual-core architecture, while the Autorender Daemon was not programmed to utilize this feature although there was a demand for more processing performance. Next to this, we identified that slow or stale NFS shares were not anticipated which limited the performance of the Autorender Daemon and for a problem with the used render software a workaround was designed. From all the identified problems with the Autorender Daemon, requirements for a new Autorender Daemon were derived and a new Autorender Daemon was designed and partly implemented. The implemented Autorender Daemon is evaluated.

Chapter 6

Conclusions and Recommendations

In this chapter, we provide an overview of the identified architectural principles and use them to answer our initial research question. Further, we provide recommendations for the implementations and designs that resulted from the architectural principles and our research. Finally, some future work is proposed.

6.1 Architectural Principles

Our initial research question was:

> *Which architectural principles should be applied in the design of a large scale Web site?*

To answer this research question, first these sub-questions need to be answered:

> *What architectural principles should be applied in the design of a storage architecture for a large scale Web site?*
>
> *What architectural principles should be applied in the design of a static content serving architecture for a large scale Web site?*
>
> *What architectural principles should be applied in the design of a processing architecture for a large scale Web site?*

Therefore, we give per sub-question an overview of the architectural principles identified in our research.

6.1.1 Architectural Principles for Storage Architectures

In our case study, the combination of the linear storage pattern and the requests pattern that was not linear, resulted in the situation in which the ratio of the number of items per storage node and the ratio of the number of requests per storage node did not correspond. This caused a difference in load and therefore the storage with the most requests experienced a performance problem because its maximum request capacity was reached.

Storing items with a pattern in which the chance of storing an item at an arbitrary storage node is equal for all storage nodes, balances the number of items stored per storage node. Furthermore, independent of the request pattern, it also balances the number of requests per storage node. Even if there are no performance issues, it is optimal to balance the load, in order to handle peak loads properly.

Using such a storage scheme in which the chance of storing an item at an arbitrary storage node is equal for all storage nodes, the ratio of the number of items per storage node and the ratio of the number of requests per storage node correspond, and therefore there is no difference in load.

We have identified as an architectural principle for designing a storage architecture for a large scale Web site, that the load of storage actions (writes and reads) should be balanced on each storage node. One should design a storage architecture that uses a mechanism or scheme in which the chance that an arbitrary item is stored is equal for all storage nodes and storage items, because the randomness enforces a natural balance suitable for every request pattern. This prevents the situation in which one storage node limits the maximum number of reads and writes and enables to use the aggregate storing and processing capacity of the storage.

In our case study, at first, a NAS with a capacity of 4TB was purchased. Because the NAS was used for storing backups and storing the media items, it was decided that when the capacity of this NAS was not sufficient, to purchase a second NAS with 4TB capacity. In the new situation, the first NAS was used for storing backups, and the second NAS was used for storing media items. When the second NASs capacity appeared insufficient, a larger NAS was purchased. In this situation, the first and the second NAS with a capacity of 4TB were used for storing backups and the third NAS with a capacity of 9TB was used for storing media items. Also the capacity of 9TB appeared insufficient. Additional external disks are used to provide sufficient capacity in combination with the NAS, because currently there are no NASs available with larger capacity. We have seen that it is possible that a large scale Web sites growth outperforms the growth of the capacity of available resources.

We identified that when designing a storage architecture for a large scale Web site, one should find a solution to overcome the limits in capacity provided by one single machine (e.g. processing capacity or storage capacity), because it might be possible that, sooner or later, the growth of the large scale Web site outperforms the growth of the available technology. Such a design should use clusters or pools of machines with disks, in order to overcome the limitations by one single machine with disks. By taking into account that in the future new and/or improved technology might become available (e.g., allowing nodes with different capacities) enables migration to newer technology and phasing out the old technology.

Both architectural principles are not only applicable to the design of a storage architecture, but also to any design in which items of a certain class need to be distributed over items of another certain class. Examples seen in the case study are rendering the media items using multiple Autorender Daemon servers, storing rendered media items in a number of storage nodes and serving Web pages to a number of clients using a number of Web servers.

In our case study, the load-balancing principle was applied at several parts of the service. Examples are load-balancing the users over the Web site Web servers and, not mentioned in this report, load balancing the Web site Web servers over a number of member database slaves. In the parts of the service where this principle was not applied (e.g., the storage), scalability problems occurred that were experienced as performance problems.

We have to mention that there is a essential difference between load-balancing Web servers over a number of database slaves and load-balancing the storage nodes. In the first case any Web server can be binded to any database slave because their content should be equal, in the second case the contents of each storage node are not equal. Therefore additional logic is necessary to combine the request and the destination and wee can not speak of the term “binding”.

In our case study, the principle to use pools or clusters to serve a storage or processing service in order to overcome the limits by using a single unit can be applied at several parts of the service. We already mentioned the issue with the NAS, but it also holds for the Autorender Daemon: its processing capacity was limited and the demand for processing capacity exceeded this limit. Because the service was limited to a single machine, it was not possible to improve the performance without changing the design or the implementation of the Autorender Daemon.

6.1.2 Architectural Principles for Static Content Serving Architectures

In our case study, we have seen that by interpreting the statistics of the requests that were done by clients to the case study's Rendered Media Web servers resulted in the use of a caching solution. Therefore we would like to suggest that the request pattern for a large scale Web site should be analyzed. Although this is not a design principle, creating tools which allows one to analyze the request pattern when designing a large scale Web site is.

Further, we have used a caching solution to cache "popular" rendered media items. Although rendered media items can be popular without any reason, there are also rendered media items popular with a reason. Examples are the rendered media items that appear for a long period of time on the "Open Home Page". The Open Home Page is a page on the Web site showing the most recently uploaded media items, the most popular rendered media items in terms of numbers of requests, numbers of credits or numbers of comments. Introducing such a page on the Web site can be a reason to research the effects for the request pattern and might lead to introducing additional caching strategies.

We have identified as an architectural principle for designing a large scale static content serving architecture, that analysis on the requests pattern should be enabled by allowing monitoring of the requests and by deriving request patterns by monitoring the requests. This can provide deep insight into the use of the application by its users and can provide guidelines when maintaining, extending or improving the performance of the large scale Web site.

We have seen that this architectural principle is not only applicable to a static content serving architecture, but can be applied for every part of the large scale Web sites. We think that when designing a large scale Web site the focus should be on maintaining the large scale Web site also, next to the focus on extending the large scale Web site and maintaining the systems that run the large scale Web site.

6.1.3 Architectural Principles for Processing Architectures

As architectural principle for designing a large scale storage architecture, we have identified that pools or clusters are necessary to provide sufficient capacity and not being limited by one machine or device. Next to overcoming the limits as opposed by using single machines or resources, pools and/or clusters also provide redundancy.

We saw that this architectural principle was applied at several parts in our case study. Some parts were considered not critical and therefore these parts were not redundantly implemented or in pools or clusters. Although some parts were not considered critical, the growth of the large scale Web site caused a change in view. Therefore it is better to apply the principle for every part of the large scale Web site from the start. This prevents future bottlenecks because of being limited to a single machine and this limits the impact of a failure when only one machine is being used for a certain purpose.

We have also seen that most of the redundancy provided by the use of multiple instances (e.g., multiple Web site Web servers) was just a side effect of creating a scalable solution. Therefore we stress redundancy in this section.

As architectural principle for designing processing architectures for large scale Web sites, we have identified that every sub-service of the large scale Web site should be implemented using pools or clusters for providing the necessary redundancy.

6.1.4 General Architectural Principles

Next to the architectural principles identified for designing a large scale storage architecture, a large scale processing architecture and a large scale static content serving architecture, we also identified general architectural principles.

We have identified that tight coupling of some parts of the Hyves Web site lead to limiting the growth of the large scale Web site. Examples of such tight coupling are using the NAS as a backup and file server

simultaneously, which caused both functions to interfere with each other. Another example is the variable `intervalboundary` that has been hard-coded into the application and limited the flexibility of adjusting this value.

We consider loose coupling as an architectural principle for designing large scale Web sites. The examples of tight coupling limits the growth of parts of the large scale Web site and therefore should not be implemented.

We have seen that in our case study system-level monitoring has been implemented and there is also application-level monitoring for the used applications like Apache and MySQL, but there is no application-level monitoring for the in-house written applications and there is no monitoring of the aggregate results. This limits the monitoring of the large scale Web site as a whole. We have also seen an example of application-level monitoring that was implemented for monitoring an in-house written application after performance problems occurred and how this lead to more knowledge on how to handle the problem.

Therefore we have identified that a good architectural principle is to implement application-level monitoring from the start of the design from every storing, processing and/or serving part when designing a large scale Web site.

6.2 Conclusions

Since all sub-questions are answered, we can answer our initial research question:

> *Which architectural principles should be applied in the design of a large scale Web site?*

Our answer is:

> *The principles that we have identified in the sub-questions are mostly identified after performance or capacity problems in several parts of our case study. Therefore the identified architectural principles are applicable to our case study certainly but are not necessarily applicable to the design of any large scale Web site. The identified architectural principles are that redundancy and scalability should be provided by implementing (sub)services of the large scale Web site using pools or clusters, that the requests the large scale Web site processes should be investigated in order to gain more insight to improve or extend the current implementation and/or design of the large scale Web site and application-level monitoring enables a more proactive maintenance. The identified architectural principles are not much different from topics covered in any book on distributed systems, still our research showed the consequences of not applying these concepts at the design of a large scale Web site.*

6.3 Recommendations

- The current caching solution with Squid uses an in-memory cache and an disk cache. Current caching performance can be improved by using more machines, more partitions and by extending the RAM of the machines. Furthermore, there exists interface cards for x86 based servers which can be inserted into a PCI slot of the server (see Figure 6.1). These cards provide a simulated hard disk to the computer. The simulated hard disk is implemented in the RAM of the interface card. According to the specifications, these cards offer lower access times than the current fastest SCSI hard disks. By altering the current caching reverse proxy server setup with an in-memory cache and a disk cache to a setup with in-memory cache, on memory-disk cache and a disk cache, caching performance can be improved. Comparisons of such cards can be found in [DDRX] and [IRAM]. The vendor can be found in [DDRG];

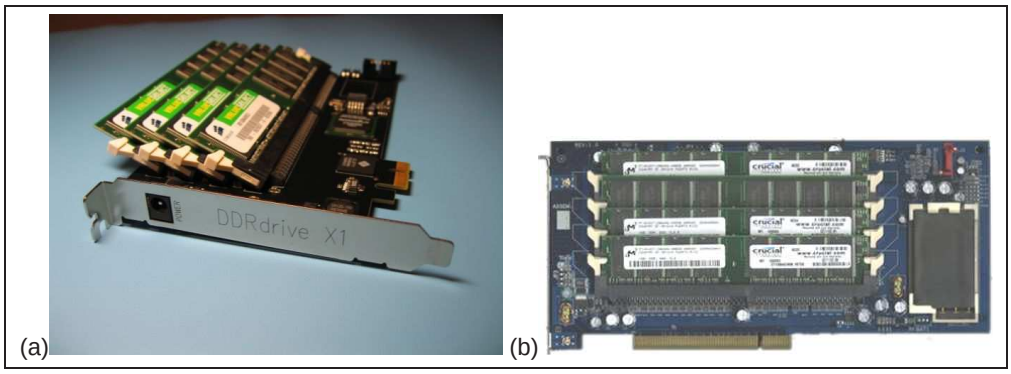

Figure 6.1: **Photograph of (a) a DDR drive X1 and (b) iRAM.**

- Replace the current media storage and the rendered media storage with a setup comparable as discussed in this thesis. Instead of focusing on a large distributed file system like GFS or Panasas, focus on technologies that content delivery networks like Akamai and Coral use;
- In large applications like SAP or ERP administration is divided into two distinct administration areas: system administration and application administration. We think that, now the Hyves Web site has experienced a considerable growth, it it time to introduce application administration. We have to say that to certain levels, application administration is already in progress (e.g., the community managers that delete undesired content from the application like pornography and illegal activities). An application administrator can, using the application level monitoring as suggested, enable a more proactive approach when extending and maintaining the current architecture of the Hyves Web site. For example, the growth of the number of media items can be an indication for when to order additional storage nodes and the growth in the number of page views can be an indication of when to order new rack space and Web servers.

6.4 Future work

- A research why the '200'-resolution is significantly less requested than the other by the application offered resolutions is necessary. Questions to be answered are:
  1. is the resolution unpopular by the users or does there exist another reason for not requesting this resolution?;
  2. depending on the outcome of the question above, reconsideration is necessary if the resolution should be kept in the application or maybe dropped;
  3. Effects of both scenarios could be investigated. Think in terms of decreased render times, decreased storage capacity needs, application reprogramming etc.;
- Statistics on the requests that are received at the Rendered Media Web servers can be used to generate statistics on the number of distinct media items that are requested and how much memory is needed to cache all distinct requests. Using these data together with extended information from the caching solution, one is able to calculate the efficiency of the caching solution. This can be an indication how future improvements take effect on the caching solution (e.g., adding additional caching machines);
- A prototype of the storage solution that is provided in this thesis could be implemented as a proof of concept and evaluated. Perhaps it is possible to determine the desired amount of storage nodes in order to provide enough serving capacity. Perhaps it is possible to find a correlation between the number of users, the number of media items and the number of storage nodes;

- Several times there were ideas at Hyves to implement cache nodes at the networks of large Internet service provides in the Netherlands. Research on how this could be done is necessary and how this could improve the current performance is needed.

References

[AKAMAI] *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, David R. Karger and Eric Lehman and Frank Thomson Leighton and Rina Panigrahy and Matthew S. Levine and Daniel Lewin. STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, 1997, p. 654-663, El Paso, Texas, United States.
`http://www.akamai.com/dl/technical_publications/ConsistenHashingandRandomTreesDistributedCachingprotocolsforrelievingHotSpotsontheworldwideweb.pdf`
Last viewed on November 16th, 2006.

[ALONSO] *Web Services, Concepts, Architectures and Applications*, G. Alonso, F. Casati, H. Kuno, V. Machiraju, Springer-Verlag, 2004, Berlin, Germany.

[ASIQT] *A Short Introduction to Queuing Theory*, Andreas Willig, Technical University Berlin, Telecommunication Networks Group, 1999, Berlin.
`http://www.tkn.tu-berlin.de/curricula/ws0203/ue-kn/qt.pdf`
Last visited November 16th, 2006.

[AVALANCE] *On the design of S-boxes*, A.F. Webster, S.E. Tavares, Department of Electrical Engineering, Queen's University, Kingston, Ont. Canada, p524- 534.
`http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C85/523.PDF`
Last visited November 16th, 2006.

[BASHM] *The GNU Bash Reference Manual, for Bash, Version 2.05b* Free Software Foundation, Inc. 1991-2002, `ulimit` section.
`http://www.gnu.org/software/bash/manual/bashref.html#IDX106`
Last visited November 16th, 2006.

[CHEN02] *A Scalable Cluster-based Web server with Cooperative Caching Support*, G. Chen, C.L. Wang and F.C.M. Lau, Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong. A preliminary version of this paper appeared in Proceedings of IEEE International Conference on Cluster Computing (Cluster 2001), Newport Beach, California, October 2001, under the title "Building a Scalable Web Server with Global Object Space Support on Heterogenous Clusters".
`http://www.cs.hku.hk/~clwang/papers/WebServer-CCPE2002.pdf`
Last visited November 16th, 2006.

[COOKIE] *Computer Networking, A Top-Down Approach Featuring the Internet* James F. Kurose, Keith W. Ross, second edition, international edition, 2003, Addison Wesley, Boston, United States of America. p.100-101.

[DAEMON] *The Linux A-Z*, Phil Cornes, 1997, Prentice Hall, Essex, England, p.352.

[DFS] *Applied Operating System Concepts, first edition*, Abraham Silberschatz, Peter Galvin, Greg Gagne, 2000, New York, United States of America. p541-553.

[DSYS] *Distributed Systems, Principles and Paradigms*, Andrew S. Tanenbaum, Maarten van Steen, 2002, Prentice Hall, Upper Saddle River, NJ. p.33, p.427-432, p.672-676.

[HASH] *Discrete and Combinatorial Mathematics, an Applied Introduction*, Ralph P. Grimaldi, 5th edition, international edition, 2004, Pearson Education, Boston. p.694

[HPLTR1] *Evaluating Content Management Techniques for Web Proxy Caches*, Arlitt, Martin; Cherkasova, Ludmila; Dilley, John; Friedrich, Richard; Jin, Tai, HP Labs Technical Reports, HPL-98-173, 990430
`http://www.hpl.hp.com/techreports/98/HPL-98-173.html`
Last visited November 1st, 2006.

[HPLTR2] *Enhancement and Validation of Squid's Cache Replacement Policy*, Dilley, John; Arlitt, Martin; Perret, Stephane, HP Labs Technical Reports, HPL-1999-69, 990527
`http://www.hpl.hp.com/techreports/1999/HPL-1999-69.html`
Last visited November 1st, 2006.

[IQT] *Introduction to Queuing Theory*, Second Edition, Robert B. Cooper, Computer Systems and Management Science, Florida Atlantic University, Boca Raton, Florida, North-Holland, New York.

[PL] *Logic and Computer Design Fundamentals*, M. Morris Mano, Charles R. Kime, 2nd edition, international edition, Prentice Hall International, Upper Sadle River, NJ, United States. p.453-455.

[SNA97] *Sed & Awk, 2nd edition*, D. Dougherty and A. Robbins, O'Reilly, Sebastopol, CA, UsA

[SQD04] *Squid, The Definitive Guide*, D. Wessels, O'Reilly, Sebastopol, Ca, UsA

[WCCH] *Web caching with consistent hashing*, D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi. MIT Laboraty for Computer Science
`http://www8.org/w8-papers/2a-webserver/caching/paper2.html`
Last visited November 22nd, 2006.

Web References

[AHD00] *The American Heritage Dictionary of the English Language*, Fourth Edition, 2000, Houghton Mifflin Company.
`http://dictionary.reference.com/browse/Web%20site`.
Last visited on November 16th, 2006.

[APACHE] *The Apache Project*
`http://httpd.apache.org`
Last visited on November 16th, 2006.

[AWK] *Gawk - Gnu Project - Free Software Foundation (FSF)*
`http://www.gnu.org/software/gawk`
Last visited on November 16th, 2006.

[BASH] *Bash - Gnu Project - Free Software Foundation (FSF)*
`http://www.gnu.org/software/bash`
Last visited on November 16th, 2006.

[CACTI] *Cacti, the complete rrdtool-based graphing solution*
`http://www.cacti.net`
Last visited on November 16th, 2006.

[CACTIH] *Hyves internal Cacti*
closed source.

[CLARiiON] *EMC Clariion Networked Storage Systems*
`http://www.emc.com/products/systems/clariion.jsp?openfolder=storage_systems`
Last visited November 16th, 2006.

[CISCOCE] *Cisco Cache Engine*
`http://www.cisco.com/univercd/cc/td/doc/product/webscale/webcache/index.htm`
Last visited November 16th, 2006.

[CISCOCSM] *Cisco Content Switching Module*
`http://www.cisco.com/en/US/products/hw/modules/ps2706/ps780/index.html`
Last visited November 16th, 2006.

[CODA] *Coda File System*
`http://www.coda.cs.cmu.edu/`
Last visited November 16th, 2006.

[COOKIE] *Wikipedia on HTTP Cookie*
`http://en.wikipedia.org/wiki/HTTP_cookie`
Last visited November 17th, 2006.

[CORAID] *TrueBit/CoRaid*
http://www.coraid.nl
Last visited November 17th, 2006.

[CORAL] *Coral: The Coral Content Distribution Network*
http://www.coralcdn.org/
Last visited November 17th, 2006.

[CRON] *cron from FOLDOC*
http://foldoc.org/?query=cron
Last visited November 22nd, 2006.

[DDRG] *Gigabyte Computer Products - DDR drive*
http://www.ddrdrive.com/ddrdrive_prototype.html
Last visited November 1st, 2006.

[DDRX] *Tweakers.net - DDR X1 drive*
http://tweakers.net/nieuws/38098
Last visited November 1st, 2006.

[EQUALLOGIC] *EqualLogic Networked Storage*
http://www.equallogic.com/ Last visited November 17th, 2006.

[EWEEK] *Eweek: MySpace and Isone*
http://www.eweek.com/article2/0,1895,1947684,00.aspa
Last visited November 22nd, 2006.

[GENTOO] *Gentoo Linux*
http://www.gentoo.org
Last visited November 16th, 2006.

[GOOGLE] *Google*
http://www.google.com
Last visited November 16th, 2006.

[GLABS] *Google Labs publications*
http://labs.google.com/papers
Last visited November 16th, 2006.

[FLASH] *Macromedia Flash Professional*
http://www.adobe.com/products/flash/flashpro/
Last visited November 16th, 2006.

[HASH] *Wikipedia on Hash Function*
http://en.wikipedia.org/wiki/Hash_function
Last visited November 17th, 2006.

[HYVES] *Hyves.net - Always in touch with your friends*
http://www.hyves.net
Last visited November 16th, 2006.

[IBCFS] *Ian Blenke Clustered File System*
http://ian.blenke.com/projects/xen/cluster
Last visited November 22nd, 2006.

[IBBD] *Ian Blenke Computer Engineer*
http://ian.blenke.com/projects/cornfs/braindump/braindump.html
Last visited November 22nd, 2006.

[IMGMGK] *ImageMagick: Convert, Edit and Compose Images*
http://www.imagemagick.org
Last visited November 22nd, 2006.

[IPVS] *The Linux Virtual Server Project*
http://www.linuxvirtualserver.org
Last visited November 22nd, 2006.

[IRAM] *Tweakers.net - iRAM drive*
http://tweakers.net/nieuws/38256/
Last visited November 1st, 2006.

[JUNIPER] *Juniper Application Acceleration Platform*
http://www.juniper.com
Last visited November 22nd, 2006.

[KTCPVS] *KTCPVS Software: Application-Level Load Balancing*
http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html
Last visited November 22nd, 2006.

[MOGILE] *MogileFS*,
http://www.danga.com/mogilefs/
Last visited November 22nd, 2006.

[MSISA] *Microsoft Internet Security and Acceleration Server*
http://www.microsoft.com/isaserver/prodinfo/previousversions/2004.mspx
Last visited November 22nd, 2006.

[MSN] *MSN.com*
http://www.msn.com
Last visited November 16th, 2006.

[MYSPACE] *MySpace*
http://www.myspace.com
Last visited November 16th, 2006.

[MYSQL] *MySQL*
http://www.mysql.org
Last visited November 16th, 2006.

[NAGIOS] *Nagios Network Monitor*
http://www.nagios.orga
Last visited November 17th, 2006.

[NPS] *Netscape Proxy Server*
http://wp.netscape.com/proxy/v3.5/datasheet/index.html
Last visited November 22nd, 2006.

[ONESTAT] *OneStat*
http://www.onestat.com
Last visited on November 16th, 2006.

[PANASAS] *Panasas - Accelerating Time to Results with Clustered Storage*
http://www.panasas.com
Last visited November 16th, 2006.

[PERL] *The Perl Directory - perl.org*
http://www.perl.org
Last visited November 16th, 2006.

[PET] *PET Cooperative Proxy Cache*
http://pet.sourceforge.net
Last visited November 16th, 2006.

[PHP] *PHP: Hypertext Preprocessor*
http://www.php.net
Last visited on November 16th, 2006.

[POSTFIX] *The Postfix Home Page*
http://www.postfix.org
Last visited on November 16th, 2006.

[RHCS] *Red Hat Cluster Suite*
http://www.redhat.com/docs/manuals/csgfs/pdf/rh-cs-en-4.pdf
Last visited November 22nd, 2006.

[RHGFS] *Red Hat Global File System*
http://www.redhat.com/docs/manuals/csgfs/pdf/rh-gfs-en-6_1.pdf
Last visited November 22nd, 2006.

[SQUID] *Squid Web Proxy Cache*
http://www.squid-cache.org
Last visited November 16th, 2006.

[UVADFS] *uvadfs*
http://staff.science.uva.nl/~delaat/snb-2004-2005/p15/report.pdf
Last visited November 22nd, 2006.

[WIKICH] *Wikipedia on Consistent Hashing*
http://en.wikipedia.org/wiki/Consistent_hashing
Last visited November 22nd, 2006.

[WIKIDHT] *Wikipedia on Distributed Hash Table*
http://en.wikipedia.org/wiki/Distributed_hash_table
Last visited November 22nd, 2006.

[WIKIJS] *Wikipedia on JavaScript*
http://en.wikipedia.org/wiki/JavaScript
Last visited November 22nd, 2006.

[WIKIMD5] *Wikipedia on MD5*
http://en.wikipedia.org/wiki/MD5
Last visited November 22nd, 2006.

[WIKINAS] *Wikipedia on Network Attached Storage*
http://en.wikipedia.org/wiki/Network-attached_storage
Last visited November 22nd, 2006.

[WIKINFS] *Wikipedia on NFS*
http://en.wikipedia.org/wiki/Network_File_System
Last visited November 22nd, 2006.

[WIKIOC] *Wikipedia, Online Community*
http://en.wikipedia.org/wiki/Online_community
Last visited November 22nd, 2006.

[WIKISHA] *Wikipedia on SHA hash functions*
`http://en.wikipedia.org/wiki/SHA_hash_functions`
Last visited November 22nd, 2006.

[WIKIW] *Wikipedia on Website*
`http://en.wikipedia.org/wiki/Website`
Last visited November 22nd, 2006.

[WIKIWP] *Wikipedia on Proxy Server*
`http://en.wikipedia.org/wiki/Proxy_server`
Last visited November 22nd, 2006.

Appendix A

Tools

This appendix contains the source codes of the tools that were written and used for our research.

1. `requeststats_a.awk` see A.1
2. `requeststats_id2plot.sh` see A.2
3. `requeststats_id2plot_uniq.sh` see A.3
4. `requeststats_time2plot.sh` see A.4
5. `tcpdump.sh` see A.5
6. `processtcpdump.sh` see A.6

The requeststats_a.awk script processes Apache- or Squid-logs and generates a summary with statistics. Execute by `awk -f requeststats_a.awk <access.log >result.txt`. In the source, grain, maximum_id and several other options can be adjusted. The output consists of statistics per format, per id, per unique id and per period of time (all configurable), and the number of lines parsed.

Listing A.1: requeststats_a.awk

```
# requeststats.awk
# tristan@hyves.nl

# expects an apache access log as input

BEGIN {
  # config
  max_interval = 26000000;
  #grain = 500000;
  #grain = 4850000;
  grain = 500000;

  timegrain = 1800;
  timemax = 3600 * 24;

  # init counters for requests per resolution
  f50 = 0;
  f75 = 0;
  f120 = 0;
  f200 = 0;
  f500 = 0;
  f700 = 0;
  ftotal = 0;
  utotal = 0;

# init counters for requests per interval
  for (i=0; i*grain < max_interval; i++){
```

```
          icount[i] = 0;
          uniekperinterval[i] = 0;
          f50a[i] = 0;
          f75a[i] = 0;
          f120a[i] = 0;
          f200a[i] = 0;
          f500a[i] = 0;
          f700a[i] = 0;
  }
  itotal=0;

  #init counters for requests per id
  for (i=0; i < max_interval; i++){
    irequest[i] = 0;
  }
  requesttotal=0;

  for (i=0; i * timegrain< timemax; i++){
    timereq[i] = 0;
  }

}

{
  #time stamp is $4
  # strip leading [dd/mmm/yyyy:
  d1 = substr($4,index($4, ":")+1, 99);
  #print "d1 is " d1
  split(d1, datum, ":");
  seconde = datum[3];
  minuut = datum[2];
  uur = datum[1];
  time = seconde + (minuut * 60) + (uur * 3600);

  #request is $7
  #strip leading /
  p1 = substr($7,length("http://interval1.rendered.startpda.net/"),99)
  # strip leading 10...0-15...0/
  p2 = substr(p1, index(p1, "/")+1, 99)
  # strip trailing .jpeg
  p3 = substr(p2, 0, index(p2, ".")-1)
  split(p3, plaatje, "_");
  id = substr(plaatje[1], index(plaatje[1], "/")+1, 99);
  resolutie = plaatje[2]

  if (resolutie == "50") {
    f50++;
        f50a[int(id/grain)]++;
  }
  if (resolutie == "75") {
    f75++;
        f75a[int(id/grain)]++;
  }
  if (resolutie == "120") {
    f120++;
        f120a[int(id/grain)]++;
  }
  if (resolutie == "200") {
    f200++;
        f200a[int(id/grain)]++;
  }
  if (resolutie == "500") {
    f500++;
        f500a[int(id/grain)]++;
  }
  if (resolutie == "700") {
    f700++
        f700a[int(id/grain)]++;
  }

  icount[int(id/grain)]++;
  timereq[int(time/timegrain)]++;
  irequest[id]++;
}
```

```
END {
  ftotal = f50 + f75 + f120 + f200 + f500 + f700;
  for (i=0; i * grain < max_interval; i++){
    itotal += icount[i];
  }

  for (id=0; id < max_interval; id++){
    if (irequest[id] != 0){
      uniekperinterval[int(id/grain)]++;
    }
  }
  for (i=0; i * grain < max_interval; i++){
    utotal += uniekperinterval[i];
    print uniekperinterval[i];
  }

  print NR, "lines parsed";
  printf("\n");
  printf("Resolutie statistieken\n");
  printf("Res    aantal    %%\n");
  printf("50: %8d  %02.2f .\n", f50, f50/ftotal * 100);
  printf("75: %8d  %02.2f .\n", f75, f75/ftotal * 100);
  printf("120:%8d  %02.2f .\n", f120, f120/ftotal * 100);
  printf("200:%8d  %02.2f .\n", f200, f200/ftotal * 100);
  printf("500:%8d  %02.2f .\n", f500, f500/ftotal* 100);
  printf("700:%8d  %02.2f .\n", f700, f700/ftotal * 100);
  printf("\n");
  printf("Id interval statistieken\n");
  for (i=0; i*grain < max_interval; i++){
    printf("%10d-%10d: %10d %2.2f\n", i*grain, (i+1)*grain, icount[i], icount[i]/itotal * 100);
  }
  printf("Interval-resolutieverdeling\n");
  for (i=0; i*grain < max_interval; i++){
    printf("%10d-%10d: %10d (%10d,%10d,%10d,%10d,%10d,%10d)\n",
    i*grain, (i+1)*grain, icount[i], f50a[i], f75a[i], f120a[i], f200a[i],
    f500a[i], f700a[i]);
  }
  printf("Id uniek statistieken\n");
  for (i=0; i * grain < max_interval; i++){
    printf("%10d-%10d: %10d %2.2f\n", i*grain, (i+1)*grain,
    uniekperinterval[i], uniekperinterval[i]/utotal * 100);
  }
  printf("Tijd verdeling\n");
  for (i=0; i*timegrain <timemax; i++){
    printf("%10d-%10d: %10d %2.2f\n", i*timegrain, (i+1)*timegrain, timereq[i],
    timereq[i]/itotal * 100);
  }

}
```

The requeststats_id2plot.sh script processes the result of a requeststats_a.awk script and creates a graph of the number of requests per interval of ids. Execute by `requeststats_id2plot result.txt`. Output is in `graph.eps`. The result can be viewed with a Postscript-viewer like Ghostview.

Listing A.2: requeststats_id2plot.sh

```
#!/bin/bash
#
# tristan@hyves.nl
FILE=$1
PLOTCOMMAND="a_plotcommand"
PERCENTAGES="a_percentages"
INTERVALLEN="a_intervallen"
RUW="a_ruw"

if [ -e $PERCENTAGES ]
then
  rm $PERCENTAGES
fi

if [ -e $INTERVALLEN ]
then
  rm $INTERVALLEN
fi
```

```
if [ -e $RUW ]
then
  rm $RUW
fi

if [ -e $PLOTCOMMAND ]
then
  rm $PLOTCOMMAND
fi

cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$4}'
  > $PERCENTAGES
cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$1$2␣}'
  > $INTERVALLEN
cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$3␣}'
  > $RUW

echo -e "set␣terminal␣postscript␣eps␣color" >> $PLOTCOMMAND
echo -e "set␣grid" >> $PLOTCOMMAND
echo -e "unset␣border" >> $PLOTCOMMAND
echo -e "set␣boxwidth␣0.6␣absolute" >> $PLOTCOMMAND
echo -e "set␣style␣fill␣solid␣0.5␣border" >> $PLOTCOMMAND
echo -e "unset␣xtics" >> $PLOTCOMMAND
echo -e "set␣yrange[0:]" >> $PLOTCOMMAND
echo -e "set␣format␣y␣\"%9.0f\"" >> $PLOTCOMMAND

# make the xtics
i="0"
STRING="set␣xtics␣nomirror␣rotate␣by␣90␣("
for label in ‘cat $INTERVALLEN‘; do
  STRING=‘echo -ne "$STRING\"$label\"␣$i,"‘
  let i=i+1
done
STRING=‘echo -ne $STRING | sed 's/,$//'‘
STRING=$STRING")"
echo $STRING >> $PLOTCOMMAND

#echo "plot␣'$PERCENTAGES'␣title␣'requests'␣with␣boxes" >> $PLOTCOMMAND
echo "plot␣'$RUW'␣title␣'requests'␣with␣boxes" >> $PLOTCOMMAND

gnuplot <$PLOTCOMMAND >graph.eps
```

The requeststats_id2plot_uniq.sh script processes the result of a requeststats_a.awk script and creates a graph of the number of requests per interval of ids and shows only the number of unique requests (the same id number). Execute by `requeststats_id2plot_uniq result.txt`. Output is in `graph.eps`. The result can be viewed with a Postscript-viewer like Ghostview.

Listing A.3: requeststats_id2plot_uniq.sh

```
#!/bin/bash
#
# tristan@hyves.nl
FILE=$1
PLOTCOMMAND="a_plotcommand"
PERCENTAGES="a_percentages"
INTERVALLEN="a_intervallen"
RUW="a_ruw"
URUW="a_uruw"

if [ -e $PERCENTAGES ]
then
  rm $PERCENTAGES
fi
```

```
if [ -e $INTERVALLEN ]
then
  rm $INTERVALLEN
fi

if [ -e $RUW ]
then
  rm $RUW
fi

if [ -e $URUW ]
then
  rm $URUW
fi

if [ -e $PLOTCOMMAND ]
then
  rm $PLOTCOMMAND
fi

cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$4}'
  > $PERCENTAGES
cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$1$2␣}'
  > $INTERVALLEN
cat $FILE |
  sed -n '/Id␣interval␣statistieken/,/Interval-resolutieverdeling/p' |
  sed '/^I.*$/d'|
  awk '{␣print␣$3␣}'
  > $RUW
cat $FILE |
  sed -n '/Id␣uniek␣statistieken/,/Tijd␣verdeling/p' |
  sed '/^I|T.*$/d' |
  awk '{␣print␣$3}'
  > $URUW

echo -e "set␣terminal␣postscript␣eps␣color" >> $PLOTCOMMAND
echo -e "set␣grid" >> $PLOTCOMMAND
echo -e "unset␣border" >> $PLOTCOMMAND
echo -e "set␣boxwidth␣0.6␣absolute" >> $PLOTCOMMAND
echo -e "set␣style␣fill␣solid␣0.5␣border" >> $PLOTCOMMAND
echo -e "unset␣xtics" >> $PLOTCOMMAND
echo -e "set␣yrange[0:]" >> $PLOTCOMMAND
echo -e "set␣format␣y␣\"%9.0f\"" >> $PLOTCOMMAND

# make the xtics
i="0"
STRING="set␣xtics␣nomirror␣rotate␣by␣90␣("
for label in ‘cat $INTERVALLEN’; do
  STRING=‘echo -ne "$STRING\"$label\"␣$i,"’
  let i=i+1
done
STRING=‘echo -ne $STRING | sed 's/,$//’‘
STRING=$STRING")"
echo $STRING >> $PLOTCOMMAND

#echo "plot␣'$PERCENTAGES'␣title␣'requests'␣with␣boxes" >> $PLOTCOMMAND
echo "plot␣'$RUW'␣title␣'requests'␣with␣boxes,
␣␣'$URUW'␣title␣'unique␣requests'␣with␣boxes" >> $PLOTCOMMAND

gnuplot <$PLOTCOMMAND >graph.eps
```

The requeststats_time2plot.sh scripts processes the result of a requeststats_a.awk script and creates a graph of the number of requests per interval of time. Execute by `requeststats_time2plot.sh results.txt`. Output is in `graph.eps`. The result can be viewed with a Postscript-viewer like Ghostview.

Listing A.4: requeststats_time2plot.sh

```
#!/bin/bash
#
# tristan@hyves.nl
FILE=$1
PLOTCOMMAND="time_plotcommand"
PERCENTAGES="time_percentages"
INTERVALLEN="time_intervallen"
RUW="time_ruw"

if [ -e $PERCENTAGES ]
then
  rm $PERCENTAGES
fi

if [ -e $INTERVALLEN ]
then
  rm $INTERVALLEN
fi

if [ -e $RUW ]
then
  rm $RUW
fi

if [ -e $PLOTCOMMAND ]
then
  rm $PLOTCOMMAND
fi

cat $FILE | sed -n '/Tijd verdeling/,$p' | sed '/^T.*$/d'| awk '{ print $4}' > $PERCENTAGES
cat $FILE | sed -n '/Tijd verdeling/,$p' | sed '/^T.*$/d'| awk '{ print $1$2 }' > $INTERVALLEN
cat $FILE | sed -n '/Tijd verdeling/,$p' | sed '/^T.*$/d'| awk '{ print $3 }' > $RUW

echo -e "set terminal postscript eps color" >> $PLOTCOMMAND
echo -e "set grid" >> $PLOTCOMMAND
echo -e "unset border" >> $PLOTCOMMAND
echo -e "set boxwidth 0.6 absolute" >> $PLOTCOMMAND
echo -e "set style fill solid 0.5 border" >> $PLOTCOMMAND
echo -e "unset xtics" >> $PLOTCOMMAND
echo -e "set yrange[0:]" >> $PLOTCOMMAND
echo -e "set format y \"%9.0f\"" >> $PLOTCOMMAND

# make the xtics
i="0"
STRING="set xtics nomirror rotate by 90 ("
for label in `cat $INTERVALLEN`; do
  STRING=`echo -ne "$STRING\"$label\" $i,"`
  let i=i+1
done
STRING=`echo -ne $STRING | sed 's/,$//'`
STRING=$STRING")"
echo $STRING >> $PLOTCOMMAND

#echo "plot '$PERCENTAGES' title 'requests' with boxes" >> $PLOTCOMMAND
echo "plot '$RUW' title 'requests' with boxes" >> $PLOTCOMMAND

gnuplot <$PLOTCOMMAND >timegraph.eps
```

Tcpdump.sh is a script that dumps all the network traffic (in a certain period of time) in a file.

Listing A.5: tcpdump.sh

```
#!/bin/bash

tcpdump -w tcpdump.txt -s 256 &\; sleep 60\; kill %tcpdump
```

Processtcpdump.sh processes the dump made by tcpdump.sh and filters it for HTTP-traffic and sorts it.

Listing A.6: processtcpdump.sh

```
#!/bin/bash

tethereal -r $1 |
  grep "GET " |
```

```
  sed -e 's/\[P.*e\]//' -e '/interval1/d' |
  awk '{print␣$2,␣$3,␣$8}' |
  sed -e's/\/.*\/\([0-9]*\)_.*$/\1/' -e '/Retransmission/d'|
  sed -e '/Previous/d' -e '/images/d' -e '/^$/d'|
  sed -e '/10\.10\.0\.[0-9]*/d' |
  sort -t. -k3,3n -k4,4n -k5,5n -k6,6n -k1,1n
```

Listing A.7: requestsperip.sh

```
#!/bin/bash

export PATH=$PATH:.

processtcpdump.sh $1 |
  awk -f requestsperip.awk |
  awk -f stats.awk
```

Listing A.8: requestsperip.awk

```
BEGIN {
  last = 0
  counter = 0
}

$2 != last {
  if (NR != 1) print counter
  counter = 1
  last = $2
}

$2 == last {
  counter++
}
```

Listing A.9: stats.awk

```
BEGIN{
  count = 0
  total = 0
  total2 = 0
  max = 0
  min = 9999
}

{
  total += $1
  total2 += ($1 * $1)
  count +=  1

  if (max < $1) max = $1

  if (min > $1) min = $1
}

END{
  printf("Cnt␣%d,␣Tot␣%d,␣M1␣%f,␣M2␣%f,␣Var␣%f,␣StDev␣%f,␣Max␣%d,␣Min␣%d\n",
  count,
  total,
  total/count,
  total2/count,
  (total2/count)-((total/count)*(total/count)),
  sqrt((total2/count)-((total/count)*(total/count))),
  max,
  min);
}
```

Listing A.10: requestspersec.awk

```
BEGIN{
  grain = 1
  start = 0
  end = start + grain
```

```
  count = 0
}

{
  #print $1
  if ($1 >= start && $1 < end) {
    count++
  } else {
    print count
    start = end
    end += grain
    count = 1
  }
}
```