A scalable architecture for video-editing web applications

Christiaan Ottow

January 9, 2009

Abstract

A web application for video editing has high demands for processing power, storage and bandwidth. Digital video material has a very high bandwidth and results in large files. Also, editing these files takes a lot of processing capacity. Creating a scalable design for such an application may therefore require more than the standard approach to web application scalability.

In this research we examine current approaches to load balancing in web applications and create a scalable application design for the case study. This design is based on existing approaches but also includes approaches which are specific to web applications for video editing.

A prototype of the application design is created, and tests are run on it to validate that it is scalable. Potential bottlenecks in the architecture are discussed.

The conclusion is reached that the design proposed in this research is scalable, but that a limit on its scalability exists. By extrapolating the test data and comparing the design to a design currently in use in large web application, we conclude that it is not likely that an application for online video editing such as the case study application, would encounter this limit.

Since web applications are more and more replacing desktop applications for all kinds of tasks, it is important that architecture and scalability of such applications is researched. This research adds to that knowledge by suggesting a design and by an in-depth description of current practices.

Contents

List of Figures

Preface

This is the report of the bachelor research project I carried out for my study Telematics at the University of Twente. It started in October 2007 and ended in December 2008. The research is an external assignment from Furthermore B.V., a web application development company in Amersfoort.

The project was supervised by Maarten Wegdam and Aiko Pras for the University of Twente and by Igor van Oostveen for Furthermore.

The audience of this report is expected to have basic knowledge of computer systems, internet, web applications, databases and software engineering. More specific topics as scalability and load balancing are defined and explained.

Christiaan Ottow, January 2009

Chapter 1

Introduction

In this chapter we will describe the research itself. First we will look at the context, then at the research questions and approach.

1.1 Context

Before we discuss the case study, we will discuss what web applications are, and which scalability issues they have. We will show how the case study represents a specific type of web applications, and then discuss the case study itself.

1.1.1 Web applications

Web applications are software programs that are accessed through a web browser and are stored on a web server. As such, they are not installed on a client's computer (although the client may need to install special software in order to run web applications, such as Adobe Flash player).

When a client accesses a web application, part of the application logic is often transferred to the client (Javascript, Flash, Silverlight) while part of the application logic remains on the server. The part that is transferred to the client communicates with the server through the browser. Some web applications however transfer no logic to the client, only data and layout information. Data storage is always done on the server.

The use of this paradigm results in little demands of clients: no (or little) software needs to be installed in order to use a web application and the hardware requirements are low since most of the application is usually run on the server. Also, the environment in which the application runs is predictable to the software maker since the server remains the same. The clients may use different

browsers, but the differences are small compared to the differences 'traditional' software makers need to solve when working on multiple operating systems.

Another characteristic of web applications is that less processing power is needed by the clients. A very load-intensive application can be run by a fast server, with many simple clients connecting to, giving instructions and viewing output. This can also be a disadvantage since lots of proccessing power is needed at the server side while the processing power available at the clients remains unused.

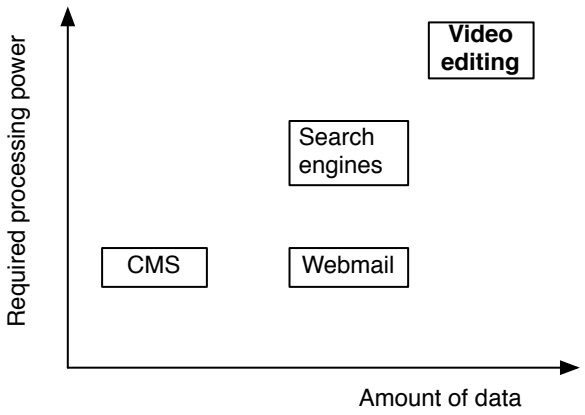
Web applications are being used more and more to take over tasks that were thus-far performed by normal desktop applications. This trend is described as Software as a Service (SaaS[17]). This means that the load (computations) that were done on desktop computers until now, are being moved to the servers of web applications. For instance, an increasing number of people prefers webmail over normal desktop mail, using Gmail or Windows Live Mail (formerly Hotmail) instead of Outlook or Thunderbird, or even some form of online document processing (such as Google docs) instead of a conventional word processor such as Microsoft Word. Also, these software services can be sold not as traditional software packages for which you pay once, but as a service with a periodical fee that you pay as long as you use it. The software vendor controls the service and can deploy new versions and updates without needing to change configurations at the clients of the service.

1.1.2 Web application scalability

So, the demands on web applications have increased. This can create problems in different fields. For applications like Google Search, the problem is how to service hundreds millions of requests per day, and still deliver the search results to each query quickly. Although Google Search deals with lots of collected data, they do not have to transfer this data to clients. For YouTube, the amount of data is a problem in terms of storage and network traffic. Figure 1.1.2 gives some examples of web applications and their required processing power and data transfer.

We will focus on web applications in the right top of the spectrum: those which place high demands on data (storage and throughput) and processing power. An example of this is online video editing. As a web application grows, the demands on the hardware increase. As we will see in chapter 2, this can be dealt with either by replacing the hardware with faster hardware or by duplicating components of the system and distribute the load among them. The first approach ends when one uses the fastest hardware available, leaving only the second approach. However, applications need to be designed with scalability in mind in order for this expansion to be possible. This way of designing what we will focus on in this research.

Figure 1.1: Web application spectrum



In this research, we will look at how an application that places high demand on processing power and storage, can be designed in order to be distributed across different physical systems in order to achieve scalability.

1.1.3 The case study

Furthermore BV has been commissioned for a new project. This project, named “WannaMakeMovies”, is a web application that allows users to create and edit videos. It targets the people for whom Apple’s iMovie and Windows Movie Maker are too complex. They dont need to install any software, and the interface is very simple. Users can upload their media, and combine it with media from the built-in library. They can glue their clips together or split them, and apply simple transitions such as fade to black and hard cut. They can also add text titles to the clips. After this, the content can be published in different ways, like automatically uploading it to YouTube, downloading it, sending it on DVD/CD.

WannaMakeMovies will be a website where users can edit and share their movies, in a typical Web 2.0 community fashion. It will also be a white-label product sold to companies, for whom it will be branded to match their specific purpose. For instance, a travel agency might rent WannaMakeMovies for a monthly fee, and then have customers create their travel movies online. The travel agency will then fill the library with sunsets, airplanes taking off and landing and so on. This service does not have a direct connection to the WannaMakeMovies community website.

For this project, which has an unprecedented scale for them, Furthermore needs a solution for how to organize the application and the data to be able to scale along with the number of users using the system. In their browser, users will work with small lower quality versions of their clips, audio files and images to apply transitions, and titles, and change the order of the clips. Afterwards,

their changes will be applied to the real data on the server, and necessary conversions will be made there. Video editing causes a lot of network traffic due to the large size of the files in question, and a lot of server load when format conversions and video effects are applied.

The project will start with a small number of users, but the system must be designed to be able to grow to a very large number of users without having to modify application structure. Up-scaling the system should ideally be a matter of adding new hardware only.

This research will show how a web application that places high demands on processing power and data flow/storage can best be designed to be scalable. The output will be a design for their application.

1.2 Research questions

As stated in the discussion of the background of this project, the research will focus on web applications that place high demands on data capacity and processing power. A video-editing web application is representative for this category. Therefore, the central research question is:

> How could web applications for online video-editing be designed in terms of application architecture in order to be highly scalable?

From this research question, a number of sub-questions rise that need to be answered in order to answer the main question. These questions are:

- What are the definitions and state of the art of scalability and web application distribution?
- What are the requirements and characteristics for scalability in the case-study?
- How could the application be designed to be scalable to the extent in which it is required in the case study?
- Can we identify potential bottlenecks and verify the scalability of the proposed design using measurements made on a prototype of the proposed design?

1.3 Approach

This research will be conducted in an iterative way, using two iterations. The first iteration serves to explore the subject and get an idea of the possibilities

and pitfalls. The second iteration will deliver the final solution. Each iteration has three phases:

1. Requirements and definitions analysis
2. Creating a design
3. Validation of the design

Definitions, state-of-the-art and requirements analysis
We look at key terms that need to be defined, state-of-the-art approaches to these subjects and requirements to our design.

Creating a design
A solution to the architecture problem is formulated during this phase. This design will be a written idea of a software architecture on an abstract level (not including specific soft/hardware).

Validation of the design
The design must be validated to see if it meets the requirements. We will create a prototype of the design and simulate users in order to run tests, showing if it meets these requirements, and if bottlenecks exist.

1.4 Report structure

This report has the following stucture. Chapter 2 contains definitions and state-of-the-art: results of the literature study. We will then more closely examine the case study to discover requirements, system tasks and system components in chapter 3. In chapter 4 an answer to the research question, in the form of a software design, is presented. It is validated in chapter 5. Finally, we present our conclusions in chapter 6. In this report we do not distinguish between the first and second iteration.

Chapter 2

Definitions and state of the art

This chapter is the result of literature study. First we will define the concepts of scalability and load balancing. Then we will look at what principles play a role in scalability of web systems. After this, we will take a detailed look at how various components of a web system are scaled and the role load balancing plays in this scaling. Finally we will see how virtual machines can be used to achieve scalability.

2.1 Definition of scalability

Scalability is a often used but poorly defined term. In all implicit and explicit definitions, it involes the extensibility of a system, the extent to which it allows for growth.

The LINFO project gives us the following definition of scalability[6]:

> *"Scalable refers to the situation in which the throughput changes roughly in proportion to the change in the number of units of or size of the inputs. It can also be looked at as the cost per unit of output remaining relatively constant with proportional changes in the number of units of or size of the inputs. Scalability refers to the extent to which some system, component or process is scalable."*

This definition approaches scalability as a system in which the relation between resource usage and demands of the system is at most linear. This is even better defined by Brataas et al[3] in the definition that will be used in this research:

> *"An architecture is scalable if it has a linear (or sub-linear) increase in physical resource usage as capacity increases "*

Furthermore, scalability can be mentioned in the context of many specific software qualities, such as performance, mean-time-to-failure, amount of memory usage, reliability, response time.[5]

In our case, the number of users that use the system is proportional to the resource usage. Capacity is defined by the hardware we use. Furthermore, we talk about the potential resource usage, not actual resource usage. If less users than the potential maximum use the system, this does not make the system less scalable.

> *Our architecture is scalable if it has an at least linear increase in users that can use the system as hardware is added.*

So the system is scalable if the number of users that can be added is proportional to the amount of hardware added: twice the hardware should result in twice the user capacity.

Load balancing is a term often used in computing, especially in network-related issues. It is important to us since when attempting to make a system scalable beyond the maximum capacity of a single unit (hardware or software). When using multiple “units”, load will have to be balanced among them.

Load balancing, most literally, is the balancing of load. It is a general term for techniques used to distribute load across multiple systems. Since we are looking for a scalable solution for a web application, we will look at load balancing in this context. We will use the following general definition[2]:

> *Distributing processing and communications activity evenly across a computer network so that no single device is overwhelmed.*

2.2 Principles of scalability in web systems

In the following sections we will look at how scalability is applied in web systems. Before we start, we will describe some terms that are used in this chapter but can have many different meanings.

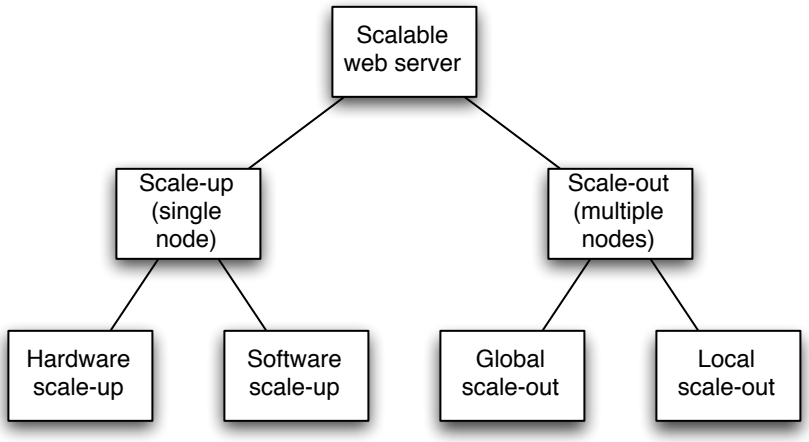
Web application
A web application is a software application that implements a web service and runs on a web system. The web application consists of programming code and digital resources (data), not hardware.

Web system and web service
In this report, a web system is a (possibly distributed) computer system that is designed to provide a web service. A web service is a service provided to users via the internet by means of the HTTP protocol. The web service is

implemented by a web application. A web system consists of hardware. A typical web system consists of HTTP servers (also referred to as web servers or webheads) and a database system. Large web systems also use a storage system for centralized storage. The HTTP servers run the application which uses the database and storage systems for in and output.

There are many approaches to scalability in web applications. We distinguish between scale-up (scaling of a single node) and scale-out (multiple nodes)[7] as in figure 2.1.

Figure 2.1: Architectural solutions for scalable web application systems

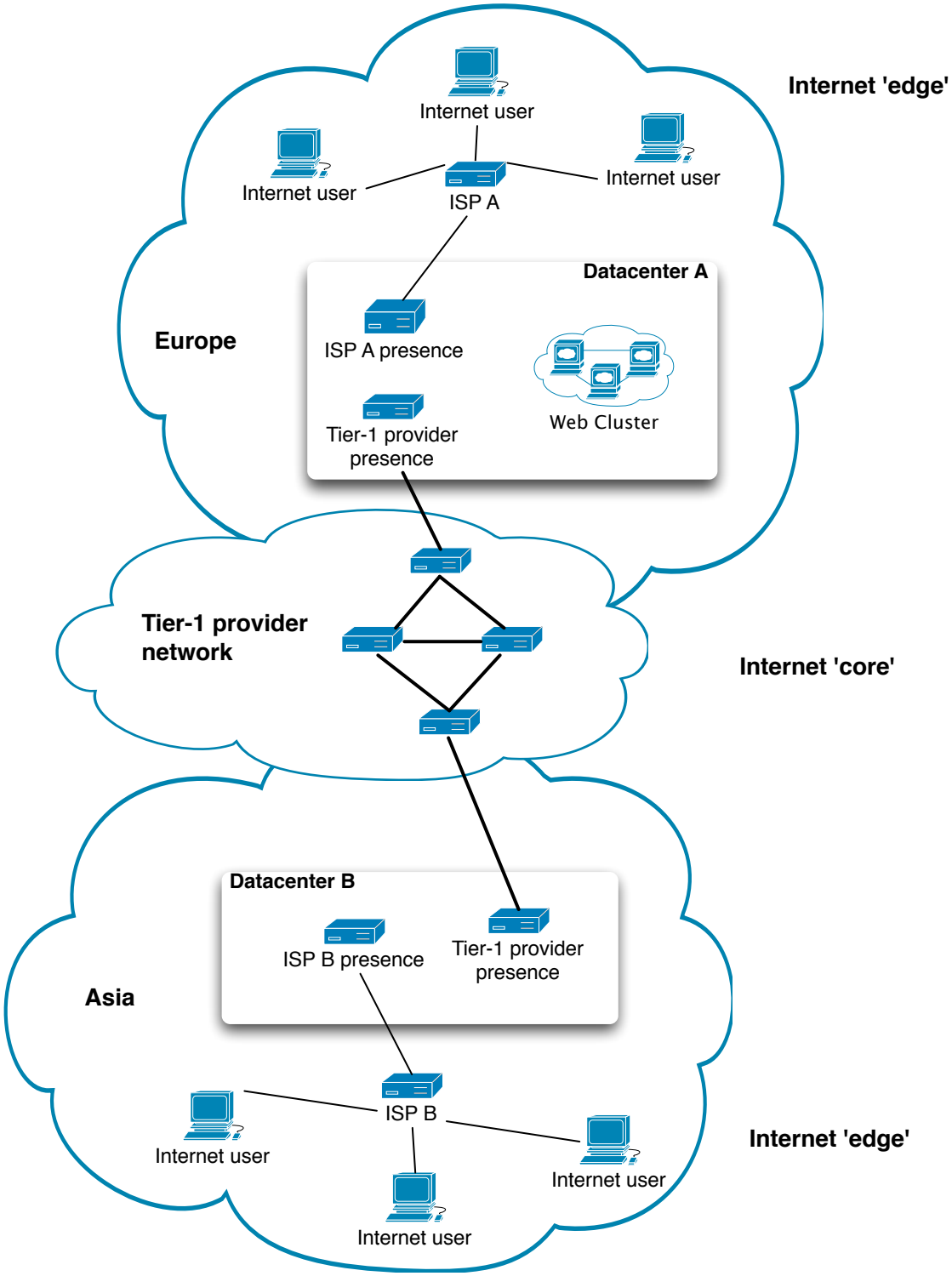


As can be seen in figure 2.1, there are two main approaches to scaling: scale-up and scale-out. While these terms are used with different meanings, we use them to distinguish between trying to increase the capacity of a single server node (scale-up) and using multiple nodes for increased capacity (scale-out). Scale-up can be achieved by upgrading the hardware to hardware with higher performance, or by optimizing the software. In this research however, we do not focus on scale-up but on scale-out. In scale-out, there are two approaches: global and local scale-out. The difference between the two is that in local scale-out the nodes are located at the same geographic location, while in global scale-out they are elsewhere. These two approaches each solve different problems and require different techniques.

2.2.1 Local scale-out

Local scale-out is when multiple nodes, together in a network, are used to increase the capacity of a system. They form a virtual server (or cluster), appearing as one server to the outside world. By using multiple servers, the performance is increased. In section 2.4 we will take a closer look at how such a cluster functions to balance load.

Figure 2.2: Simplified topology of the internet



In local scale-out, the nodes are connected via a local area network, so they have a very high speed network between them. Incoming requests are dispatched among them according to a certain dispatching algorithm (see section 2.4), and all the resources needed to send a response to the request are available to all nodes (database, storage).

The problem that one server could not handle the load is solved by local scale-out: when more capacity is needed, you add more nodes to the virtual server. However, the cluster is located in one physical location. If the network connection to the outside world would fail, the whole cluster would be unavailable. This is solved by multihoming: having multiple redundant uplinks using different providers.

Figure 2.2 shows a simplified topology of the internet, and the place a locally scaled-out web system has. While local scale-out solves the problem of high demand, it still leaves other problems unsolved. Scaling a system using local scale-out will eventually reach a bottleneck, for one of the following reasons:

- Software configuration; maximum number of nodes for an algorithm implementation;
- Architecture; because of a bottleneck in the system adding more nodes is useless;
- Physical space availability;
- Power availability;
- Bandwidth availability

Also, when a web system is accessed from locations which are geographically far away, the round-trip time from client to server is very high for remote clients. The network between these clients and the system may not be very reliable.

Furthermore, the availability of the cluster cannot be higher than that of the data center it is in. This may be a problem for systems requiring very high availability.

2.2.2 Global scale-out

Global scale-out can be used to overcome these problems. With global scale-out, (virtual) servers are spread out across multiple geographical locations. In section 2.4.2 we will discuss the workings of load balancing in global scale-out. The problems mentioned above are solved:

- When the limit in scaling of a cluster is reached, another cluster is introduced;

- When limits in power, bandwidth and physical space of a data center are reached, a new cluster is started in another data center;
- Clusters are placed close to end-users in terms of network hops: they can be placed at the edge of the internet, directly connected to the users' internet providers;
- By placing the clusters in different data centers, the reliability of a single data center is no longer a problem.

Global scale-out simply replicates local scale-out across different physical locations. Apart from increasing capacity, global scale-out is also used to provide users from different countries with different content. For example, when visiting `www.youtube.com` from the Netherlands, you get the Dutch YouTube website which features different videos than the international one does. When we look at figure 2.2 again, global scale-out would be to place another web cluster in datacenter B and make sure users from ISP B are directed to it.

Global scale-out is also used to lower long-distance network usage. Content is kept as close as possible to the end-users so traffic between the different clusters is minimal. Apart from being much more expensive, international traffic also introduces higher delays.

Also, when having multiple clusters, when one cluster is overloaded and the others aren't, they can shift load between one another.

The problem of reaching a bottleneck in architecture, as exists in local scale-out, is only partially solved by global scale-out. Since load balaning is done on a higher level, balancing load between clusters instead of nodes, the system can grow much further, but eventually, if it keeps on growing, will reach an architectural bottleneck again.

Content distribution

Web applications provide different types of content to their users: static and dynamic content. Static content can be any kind of file which is available on the server in the form it is sent to the client, such as images. Dynamic content needs to be assembled or processed by the server before being sent to the client. In the context of web applications, dynamic content is usually a HTML/CSS/Javascript page or piece of XML data that is generated by a server-side application. To generate these pages, the server application may use static content and information from a database.

When looking at the simplified internet topology in figure 2.2, applying global scale-out would mean adding a cluster at datacenter B for users of ISP B. Assuming we want to provide all users of the web application with the same content, we have multiple options on how to distribute the content.

At one end of the spectrum, we would have the full application and all its data at both locations. This would allow both locations to generate the dynamic pages

needed. They would have to keep the data in sync, and when the application code changed it would have to be updated at both locations. The advantage of this approach is that the content served to the users is always fresh. Synchronization of data between the locations however might be a problem.

At the other end, we could appoint one location as 'origin location' and only have a large cache at the other location. Changes in the master location would bubble into the caches delayed. The lifetime of pages in the cache should not be too long since their content is dynamic. Requests that change the data would still have to be forwarded to the origin location. The advantage of this approach is that content does not need to be synchronized, but the disadvantage is that caching dynamic pages leads to inconsistencies or slow updates.

There are ways to combine the two extremes. For example, using Edge Side Includes (ESI[1]), different parts of a webpage can have a different cache lifetime. An image may have a longer lifetime than a piece of text showing the number of users that are online. By using ESI, bandwidth to the origin location can be reduced by 95 to 99 percent for dynamic sites[4].

2.3 Scaling of database and storage systems

Web applications need storage space to store content like images, videos, HTML files and user information. Usually, part of this information is stored in a database and part is stored directly on a filesystem. Information that comes in files and only needs to be passed through to the user (like images, videos and HTML files) can be stored on a filesystem, avoiding the overhead of a database system. Information like user accounts, parts of website content, forum/weblog posts however aren't files, and often need to be searched through, combined and updated. This information is usually stored in a relational database management system (DBMS) like MySQL or Oracle.

Both systems (filesystems and DBMSs) require a different approach to scalability, although similarities can be found due to the read/write nature of both systems. Databases and filesystem storage systems are the 'write' part of a web system. The component responsible for execution of the web application itself, the web servers, only read data and present it to the client. Therefore, scaling these systems takes a different approach.

2.3.1 Scalability of file storage systems

In this section we will discuss the scalability of filesystem storage, which we will now simple call 'storage' (as opposed to 'databases' or 'DBMSs').

[1] `http://www.esi.org`

Network File System

Storage systems can be connected to the webheads in several ways, on different layers. A popular solution is by using the Network File System (NFS) [16] protocol, an open standard specified in the RFCs 1094, 1813 and 3530. It is an application layer protocol, operating on top of TCP (or UDP in older versions). Clients can mount NFS shares and access them as if it were a local filesystem. It supports locking and access control. NFS servers can be normal servers running open-source NFS software.

NAS and SAN systems

An NFS server is a form of Network Attached Storage (NAS)[15]. NAS systems are storage systems that are attached to an TCP/IP network, offering storage services on file level through NFS, CIFS (Microsofts file sharing protocol, also known als SMB), AFP and other protocols. Apart from installing server software on a normal server to turn it into a NAS system, one can purchase specific NAS hardware. These commercial NAS systems often offer scalability and redundancy.

Another way of using central storage in a server cluster is a Storage Area Network (SAN)[18]. In a SAN, storage in central like in a NAS only it is not accessed at file level, but at block level. Clients tunnel their I/O commands through a storage network (usually with iSCSI[12] or Fibre Channel[11]) to the SAN system. Using iSCSI, the network is connected using normal ethernet networking, allowing for speeds up to 1 Gbps. When Fibre Channel is applied, speeds up to 10 Gbps can theoretically be reached.

Many commercial NAS and SAN systems exist, allowing for hotplugging of disks, snapshots, redundancy and many other features. The exact technologies used by the different companies are beyond the scope of this research.

Scalable software solution

As said, a network-attached storage system can be created by installing NFS server software on a normal server. One could install other file related service, like an CIFS or FTP server in order to create a NAS. However, these solutions are not scalable beyond the hardware up-scaling of the storage server. Due to the read/write nature of storage (versus the read-only nature of a web service) it is hard to balance load to a storage system among multiple storage nodes.

However, *data partitioning* is a way in which load balancing can be applied. It is not transparent, but must be implemented in the application using the storage system. For example, if we look at a video-hosting web application, many video

files will be on the storage systems. If a database would be created where the meta-information about the video files (like owner, relations, keywords) would be stored, we could also store a "storage tag" there. This storage tag could indicate on which storage system to find the file itself. Or in another example, user home directories on a central storage. We could have different storage servers and setup the software so that it would look for user data of usernames starting with one of the letters '*a*' through '*e*' on server one, '*f*' through '*j*' on server 2 and so on. Although this method does not provide in transparent load balancing, it is usable. It scales the capacity (by using more servers) and also the throughput, provided that the content is spread across the servers with respect to how often it is accessed. If the most-accessed data would all be on one server, it would still be a bottleneck.

Another software solution is using a "clustered filesystem"[10] or "shared-disk filesystem". Such a filesystem is accessed from multiple servers. Concurrency is regulated on a lower level than files, permitting concurrent access to the same chunk of data. Each system using the clustered filesystem is presented a serializable view of the total filesystem. Examples of open source clustered filesystems are Coda (`http://www.coda.cs.cmu.edu`) and GlusterFS (`http://www.gluster.org/`). Such a clustered filesystem can be used by multiple NFS servers in order to create a scalable storage cluster. Open-source clustered filesystems are being developed but not very mature yet. Proprietary clustered filesystems are used by SAN vendors.

Some storage products offer a scalable solution by using both NAS and SAN. Webheads will access an NFS server, which has spread the actual data across multiple SAN nodes. The SAN nodes together form a virtual harddisk partition.

2.3.2 Scaling of database systems

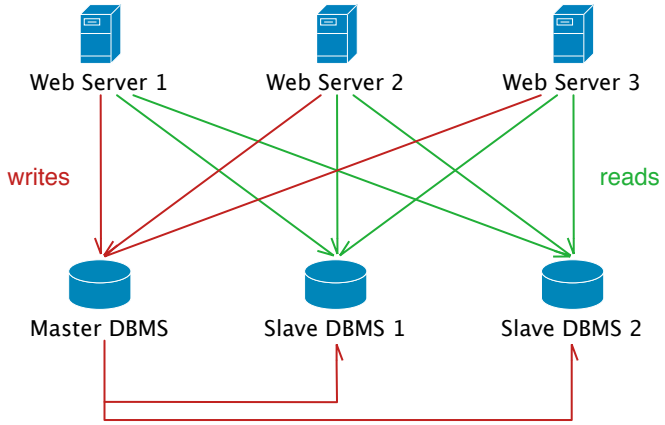
For databases, almost the same problem as for storage exists. It is difficult to balance load across multiple servers, since that would mean that multiple servers need to synchronize the data with each other. With webheads this is not a problem since the data isn't changed by the client requests, the web requests lead to changes in the database and storage. However, software-based solutions exist in the area of databases. We will look at the most common ones.

Replication

Most Relational Database Management Systems (RDBMSs) support a form of replication to achieve higher scalablity. With replication, queries to one server are "replicated" one or more other servers (see figure 2.3). In replication, a database node be either a master or a slave. A slave watches the master, and queries to the master that update the data (UPDATE, DELETE, ALTER etc)

are replicated to the slave. They are not executed right away on the slave, but delayed to a convenient time. Queries that only read from the data are ignored. Several slaves can watch a master, and masters can be slaves watching other masters.

Figure 2.3: Database replication



This may not seem like a solution, since write queries are still executed on all nodes. However, read queries are usually the most demanding queries in a RDBMS. They often involve combining all the rows of several tables (joining) and comparing all of them against a set of criteria. Write queries never involve multiple tables. Furthermore, web applications usually involve mostly read queries to present content to a visitor, and little write queries. Using replication, these read queries can be balanced across many servers.

Replication has an issue with consistency, however. The writes to the slave database are not executed instantly, so after a write to the master the system is in an inconsisten state until the write has been executed on all the slaves. This issue is not solved by the replication model.

Data partitioning

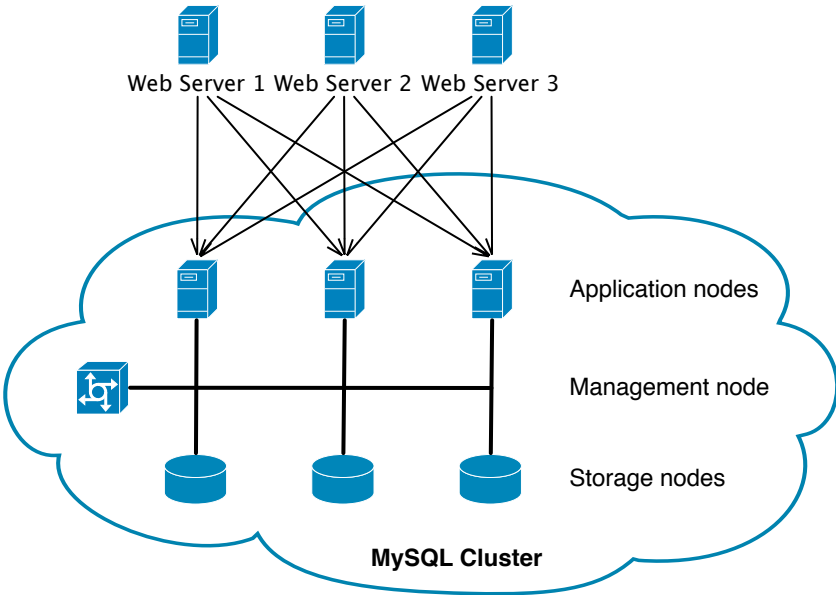
Of course, data partitioning is possible in the same way it is possible for storage systems. One could use multiple database servers, and store different datasets in them. The problem with this approach is that often queries will span across multiple tables, which using partitioning might be on different servers. Queries will need to be split up into smaller queries to different servers.

However, just as with data partitioning in storage systems, the frequency at which certain parts of the data are accessed must be considered when creating the partition. Placing a lot of frequently accessed data in one partition will still create a bottleneck.

Clustering

MySQL (a popular open-source RDBMS) supports another method of scaling: clustering. While clusteirng is a general term, in MySQL terms it refers to a certain setup of database nodes running a special version of the MySQL software. The setup is shown in figure 2.4.

Figure 2.4: MySQL cluster



The actual data is partitioned and stored in the storage nodes. They also synchronize data with each other to provide in an active/active failover system. When a data node fails, there is always at least one other data node which has the same information. The application nodes form the interface to the data. They can be MySQL servers but also MySQL APIs used by other services. There is no interdependence between the application and data nodes. The management nodes maintain cluster information, and are used when a node wants to join the cluster or when there is a cluster reconfiguration. They can be started and stopped without affecting the clusters operation.

The MySQL cluster offers scalability since data nodes can be added to the system to increase storage space, and application nodes can be added to increase performance.

2.4 Scaling of HTTP systems

We have seen how scalability is achieved in two components of web systems: database systems and storage systems. In this section we will see how scala-

bility is achieved in HTTP systems, the central 'hub' of the web system.

2.4.1 Load balancing in local scale-out

Inside a virtual server, the incoming requests need to be distributed among the nodes. The most common way to do this is by assigning one node the special function of load balancer, receiving the requests from clients and dispatching them to the other nodes (the webheads). Furthermore, if the web application needs storage and database systems, they need to be present as well. The scaling of storage and database systems is discussed in sections 2.3.1 and 2.3.2 respectively.

A load balancer can operate in different levels of the OSI Model[14]. It can also use different algorithms to decide which webhead to pick, and the interconnection between load balancer and webheads can be designed in a few different ways that have great influence on scalability and bottlenecks of the system. Furthermore, web services are often state-aware, which requires the load balancers to be so to. In this section we will discuss common approaches for each of these four areas.

Layer of operation

A HTTP load balancer receives the requests the clients send by having a socket on port 80 (the port used for HTTP), and accepting connections and requests to there as if it were a normal web server. The client and load balancer first establish a TCP connection (layer 4, transport layer) after which a HTTP request (layer 7, application layer) is sent by the client[7][13].

The load balancer however can already decide which server to forward the request to before it is received. As soon as a connection attempt is made by the client, the load balancer can pick a web server and leave the receiving and handling of the HTTP request to it. If a load balancer does this, it is operating in layer 4. This means the load balancer doesn't even have to evaluate the actual HTTP request.

Another option for the load balancer is to accept the connection from the client, read the HTTP request and choose a server based on the contents of the request. It is then operating in layer 7.

The choice for a layer of operation is closely related to the dispatching algorithm chosen.

Dispatching algorithms

A load balancer can use a static algorithm to determine which server to dispatch a request to, like random or round-robin. However, it can also base its decision on more factors using a dynamic dispatching algorithm. Factors that can be weighed in to come to a dispatching decision include:

- Load of each webhead;
- Number of requests each webhead is processing;
- Round-trip time to each webhead (availability);
- State parameters (see 'affinity' below)

Many more factors can be thought of. Most of them require the webheads to give feedback to the load balancer, while with static algorithms the load balancer can decide on its own.

Affinity

An important issue specific to HTTP which influences how the load balancer should operate, is affinity. HTTP is a stateless protocol, it does not define a way for the server to track users across different requests. This feature however is required by many websites, for example for authentication and preferences. This problem is usually solved by giving the user a HTTP cookie, a piece of information it sends with every request it performs from the moment it receives the cookie, until the cookie expires. The server can store authentication information or preferences in this cookie, but for security and performance reasons usually it only stores an identifier. The actual data is kept on the server and acessed through the identifier. Since this information is kept on the server (a specific webhead in the case of a cluster), the user's requests will need to be dispatched to that same webhead from the moment it receives the identifier on. This we call affinity: the relation between a specific server and a client.

A layer 4 load balancer can perform a very basic form of affinity: storing the dispatching decision it made the first time, and from then on sending every request from a client to the same webhead. This can be done without knowledge of the higher-level state information: the cookie. However, it also means that on a higher level, a session might not be established at all or might have been destroyed, while the load balancer still bases its dispatching decisions on the 'state information' it maintains.

Also, some ISPs use proxy servers to put all the HTTP requests of their clients through. This means that a very large number of internet users accesses the

internet through a very small number of IP addresses. For the load balancer, this means that it might overload a webhead because all the requests coming from a certain ISP are dispatched to that webhead based on client IP.

Furthermore, an IP address is not unique to a user. Many users may share one IP address, but sometimes a user can have multiple IP addresses because its ISP dynamically allocates IP addresses and they change from time to time. So, while affinity at layer 4 is an easy solution to the state problem, it is not waterproof.

Taking care of affinity at layer 7 is reliable, since this is also the layer where the state is important. It is also more complex, and slows down the dispatching decision by the load balancer. The load balancer now not only needs to read the HTTP request for its normal decision parameters, but also for state information. This state information may differ from application to application: session cookies may have different names or not be used at all.

Network topology

There are several ways to implement a locally distributed web cluster. Cardellini et al [7] distinguish three types of these systems:

- Web cluster
- Virtual web cluster
- Distributed web system

Although all of them are locally distributed, there are some important differences between the three. The first, the web cluster, is shown in figure 2.5.

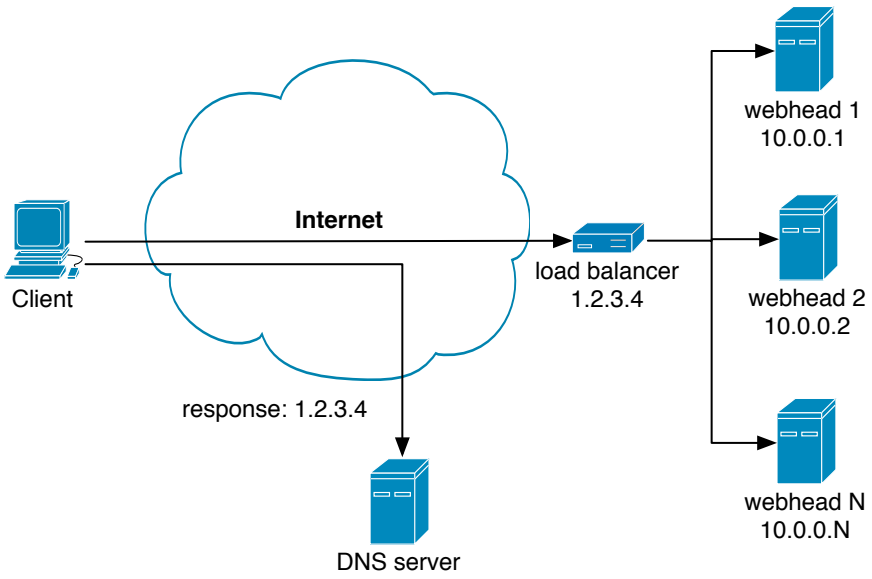
As can be seen, the web cluster uses a load balancer, a function we have discussed earlier on. This setup is the most common one.

The virtual web cluster is much like the web cluster, in that it has only one IP address to the outside world, functioning as one big server. As can be seen in figure 2.6, the virtual web cluster does not have a load balancer. All the webheads share the VIP (virtual IP), and they all receive all requests. They have an algorithm for determining who answers to what request.

The last architecture is what Cardellini et al call a distributed web system. This system uses many IP addresses for the same number of webheads. Request dispatching is done through DNS: the DNS server simply returns a different IP address for different requests (figure 2.7). This is the most simple form of load balancing, the request is 'dispatched' before it is made.

In a distributed web system, the network layout is simple: the webheads are all connected to an internet uplink and all have their own IP address. In the virtual

Figure 2.5: Architecture of a web cluster



web cluster, all webheads have a virtual network device that has the VIP. The web cluster introduces an extra network hop, the load balancer, and according to Zhang[19], a few designs are possible from there.

Network Address Translation
Using network address translation (NAT), the incoming packets are re-written and sent out to the local network, to the webhead. The load balancer replaces the destination IP address (which contains the VIP) with the IP address of the selected webhead. The webhead processes the request, and since the load balancer is its default gateway, it sends the response back there again. The load balancer replaces the source address (which is the address of the webhead) with the VIP, and sends the reponse back to the client.

IP encapsulation
IP tunneling (also called IP encapsulation) is a technique to encapsulate IP datagrams within IP datagrams, which allows datagrams destined for one IP address to be wrapped and redirected to another IP address. This technique is often used in VPN connections. For our web cluster, this means that the load balancer maintains a tunnel with each of the webheads. When it dispatches a requets to one of them, it is sent through the tunnel. The webhead receives the original packet that the client sent to the load balancer through the tunnel, and is therefore aware of the client IP address. To send the response, it uses this address and changes the packets it sends to have the VIP as source address.

Direct Routing
Direct Routing is similar to tunneling. All the webheads have a virtual network interface configured with the VIP. However, they do not broadcast their address and don't use ARP, so the switch they are on is not aware of them having this

Figure 2.6: Architecture of a virtual web cluster

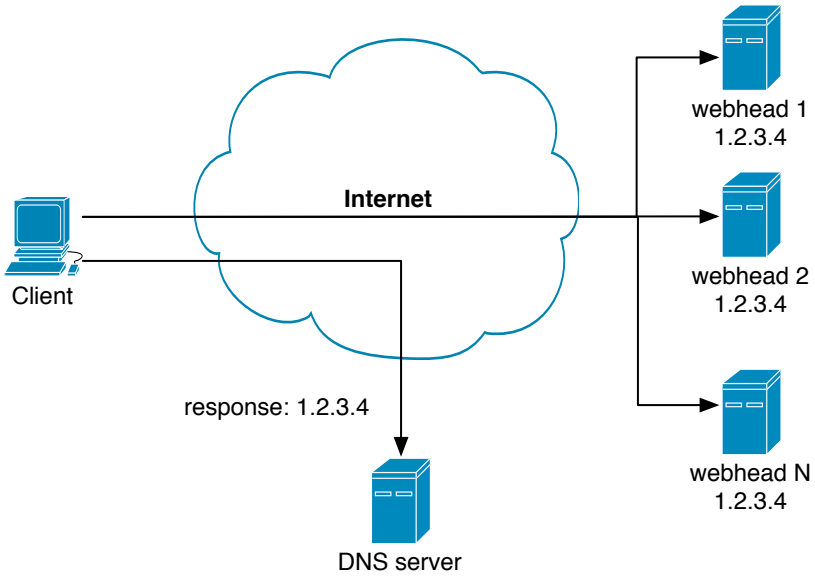
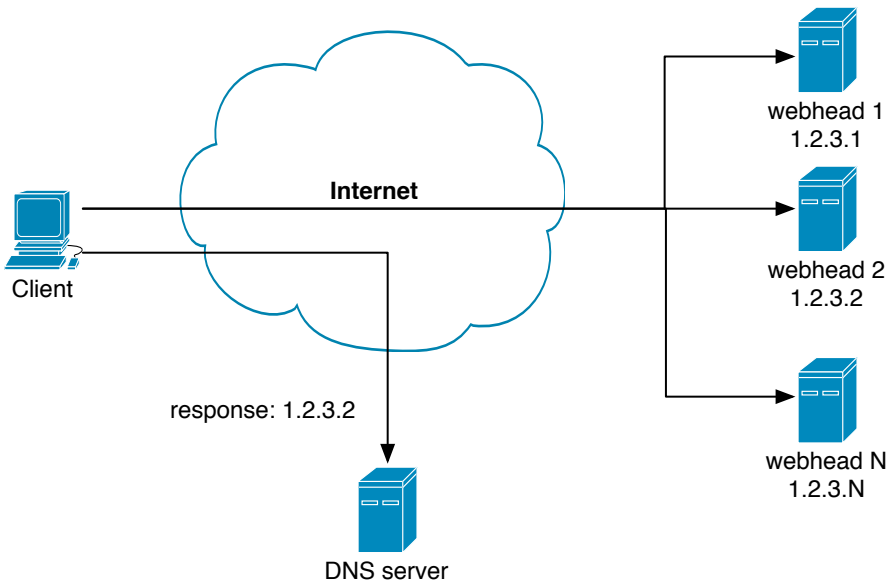


Figure 2.7: Architecture of a distributed web system



address. The load balancer doesn't rewrite the IP packet when it dispatches it, but changes the ethernet address to that of the selected webhead. This requires all webheads and the load balancer to be on the same network together, at link level. The response is returned in the same way as with tunneling.

NAT is the most simple technique. Nothing needs to be configured at the load balancers (except for the default gateway, but they need one anyway). A big disadvantage is that all traffic that flows back to the client must also pass through the load balancer. Especially in case of large responses (for example, video files) this can keep the load balancer unnecessarily busy. Tunneling and direct routing are almost the same. However, tunneling requires more configuration work and slightly more computing by all nodes in order to encapsulate packets. Direct routing is the fastest, but requires all the nodes to be on the same ethernet network. With tunneling, nodes can be moved around more since they are connected at IP level.

2.4.2 Load balancing in global scale-out

Global scale-out requires different load balancing techniques than local scale-out. We could apply the same techniques, however, we would not benefit from the advantages of global scale-out if we did. For example, if we would use a web cluster as desribed in the previous section, and place the different webheads around the globe, clients would still always need to connect to the load balancer. Therefore, when using global scale-out, dispatching of clients needs to take place in an earlier stage than their actual HTTP request.

A common way to achieve this, is by using DNS to select a (virtual) server the client is sent to[7]. A client will perform a DNS request when a URL is entered into the browser, and the DNS server can use any algorithm it wants to return an IP address. Also, by using a very low time-to-live (TTL) for the record it returns, which encourages frequent refreshes of the returned information. In order to make better dispatching decisions, the DNS server can be aware of the health and load of the pool of servers it redirects users to. The DNS server acts as a load balancer with a dynamic dispatching algorithm[4]. The Akamai case study B is an example of this technique.

Another way to send users to a server near them is by using a "landing page" where a user selects a country, and is then sent to the server closest to his location. This method is referred to as URL redirection. Sometimes this process is automated by looking up the country the users IP address has been registered in, or by using the information the browser sends with the request. When using one of these approaches, the architecture of the distributed web system is clear to the user since the user is redirected to a different URL. For example, when one would visit www.example.com from the Netherlands, one could be redirected to www.nl.example.com. Since this is a different hostname, it would

resolve to a different IP address, which could be that of the European or Dutch web cluster for example.com.

Chapter 3

The case study application

In the previous chapter we looked at the state of the art in scalability and load balancing of web applications. This chapter takes a closer look at the case study at hand: its tasks, requirements and components. The term “visitor” will be used to refer to someone viewing content, and “authors” to account holders who edit and create videos using.

3.1 System tasks

Before a solution to the problem can be defined, requirements to the solution need to be established and the functional components of the system discerned. We will now examine the tasks the system has to perform in order to understand the requirements and system components.

As said before, users will use the system to:

- Upload their audio and video materials;
- Edit the materials by combining them with each other and library materials, adding transitions and titles and re-ordering fragments of material. These tasks are performed on their computer by the client-side web-application using low-quality samples of the real materials;
- Download the results of their work and/or have them published to the online archive.

The uploaded materials are in Windows Media Video (WMV) format or Quicktime Movie (MOV) from compact camera’s and Digital Video (DV) format from camcorders. DV material is the largest in size, about 100 MB per minute. These materials need to be transcoded to low-quality Flash Video (FLV) for

the client-side application to work with. The web application performs this transcoding and also keeps the high-quality materials for later use.

The materials are stored on the server and linked to the authors accounts. The authors will then over a period of time log in incidentally and edit their materials, perhaps adding more to their library. After a while, they will have produced a decision list of actions that must be applied to the high-quality materials in order to produce the desired result. When they have arrived at this point, they will give the system the command to produce the final result, and the system will decode, edit and encode the bits of material into a final video. It is not yet clear what will be done with this final video. It might be downloaded by the users, placed on a website, sent to them on DVD or anything else. For this research we will assume that the video will be placed on the website (and viewed there). This website is similar to YouTube, visitors can view the productions of the authors and comment to them. It will also have community functionality like user profiles et cetera.

At a more detailed level, the system will have to perform the following tasks:

- Allow for visitors to watch video productions;
- Allow for visitors to create an account;
- Allow for authors to edit their profile/information;
- Authenticate authors;
- Allow for visitors to view author's pages and interact through comments/messages;
- Transcode and store audio/video materials uploaded by authors;
- Offer the authors a web application for video editing;
- Offer the authors a library of audio/video materials;
- Produce videos from the uploaded materials and library materials according to a decision list;
- Offer the produced results for download and viewing through the website

3.2 Requirements

In order to measure wether a solution to the scalability problem is scalable, we will need to have criteria the system must meet. These criteria come from the commissioner, and Furthermore BV. The requirements we place on a solution in order to be valid, should guarantee that the objective is satisfied: clients are served. So, the requirements are a list of the conditions that makes that users can be served. These requirements are:

1. Responses to requests to the community site must be delivered within:
   - 1 second for 90% of the requests
   - 2 seconds for 98% of the requests
   - 30 seconds for 99.5% of the requests
2. Every visitor viewing a video should be given at least 70 KB/s bandwidth. This ensures that loading a video doesn't take too long;
3. Every video should be compiled on the server in a reasonable amount of time. This amount of time is at most the video length plus 30 minutes.

The system is not required to be able to scale out infinitely. The commissioner nor Furthermore BV have hard requirements on the maximum size, but if the application does extremely well it will have to be able to serve 100.000 users per day. When validating the design in the validation phase we will use this number to validate if the system can scale-out far enough.

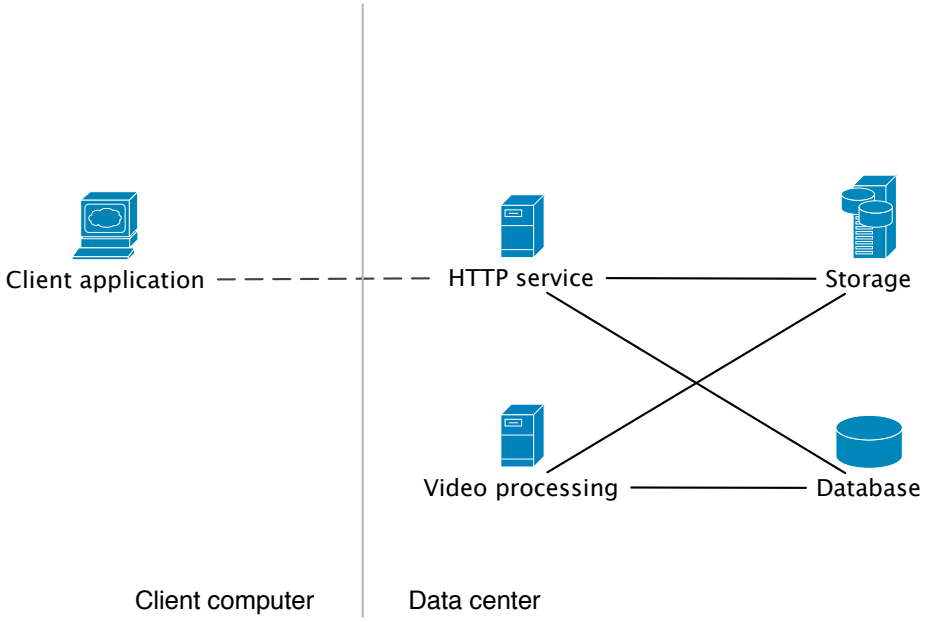
3.3 Components

Now we will discern the functional system components, in order to be able to map them to physical components later. These components are parts of the system with a unified task or function, which can be separated of the rest of the system to some extent. These components can use different approaches to scalability. The components are:

- **HTTP service**
  The clients access the web application through HTTP servers. These servers run the server-side application, retrieve and store data from and to the database and storage components, and send the results back to the client.
- **Client application**
  The client application is downloaded from by the client through the HTTP service and run on the client computer. It communicates with the HTTP service.
- **Database**
  The database contains all data except for static content and audio/video files. Information about users, videos and the dynamic content of the web pages is stored here.
- **Video processing**
  Videos must be converted, edited and stored. The part of the system that handles the audio/video editing tasks can be seen apart from the rest of the system.

- **Storage**
  The audio/video files (library materials, uploaded materials and results produced) and other content (client application, materials for the website such as images and stylesheets) are stored here so HTTP service can access them.

Figure 3.1: System components and relations



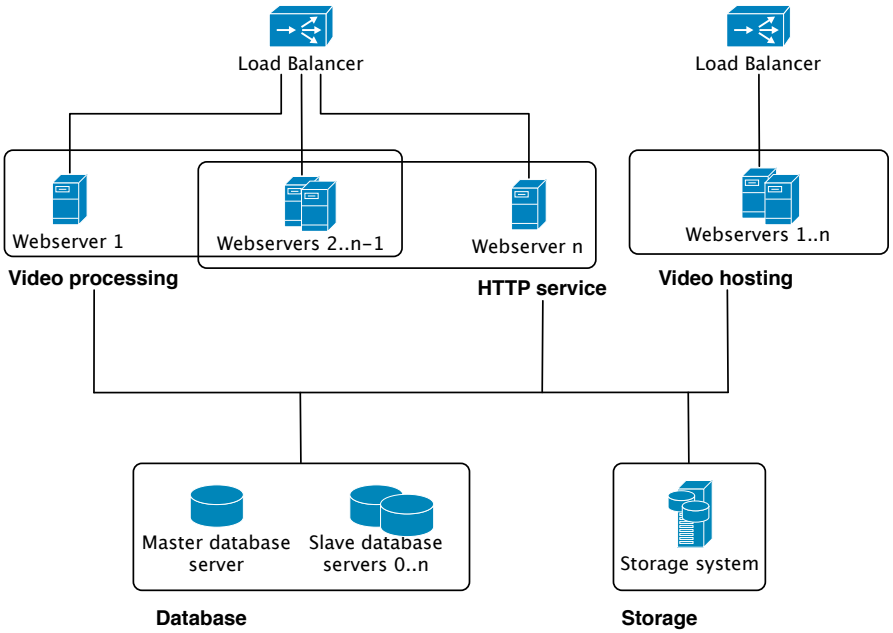
Chapter 4

System design

In this chapter we describe a design that meets the scalability requirements, as well as the other requirements. We describe the design as a whole, and then discuss the specific choices made for each of the components.

4.1 Overview

Figure 4.1: Architecture



The complete design is shown in figure 4.1. It contains all the components discussed earlier, except for the client application, and adds extra components. The client application, from a scalability perspective, is not relevant to the de-

sign of the system since the number of clients increases linearly with the number of client applications. Therefore we will leave it out of the design.

The figure shows the HTTP service and video processing component behind one load balancer. The boxes for the video processing component and the HTTP service have overlap, sharing some servers. The load balancer determines how many servers should be for video processing and how many for the web service and assigns based on current demand. The servers have only one role at any time, but this role may change as demand changes.

The components are linked together by a high-capacity local network, the black lines. This can start with a 100mbit network and be increased as demands increase. Currently, 10Gbit networks are available though expensive.

The solution uses a separate component voor video hosting. Video content is streamed to clients by this component instead of by the server-side application component. Figure 4.2 shows the setup in case geo-distribution is necessary. This shows the use of the video hosting component, which will be discussed in more detail in section 4.3.

4.2 HTTP service component

In figure 4.1 the video processing component has an overlap with the HTTP service component and is located behind the same load balancer as the HTTP service is. This shows that the video processing component and HTTP service are using the same servers.

This works as follows. At any time, there is a pool of x servers available. Of these servers, $n \geq 1$ are designated webservers for the HTTP service, and $x - n \geq 1$ servers are designated video processing servers. Once the load changes, and a larger amount of servers is required to serve HTTP requests to visitors, n can increase, re-assigning video processing servers to the HTTP service. When the visiting peak is over, servers can be re-assigned to the video processing component. Also, the compilation of videos does not need to take place right after the user requests it: there is a threshold that can range from minutes to days. If for example the threshold is 24 hours and at night load on the HTTP service is low, more servers could be assigned to the video processing component to get all the video work done quickly.

The re-assigning of servers will be done by the load balancer, since it can monitor the number of incoming requests. It will need to retrieve the load from the video processing servers (and perhaps from the webservers, depending on the criteria used for re-allocation).

Load balancing must be done at layer 3, the network layer. The session information must be stored in the database by the webservers. A client can be

served by server 1 for his/her first request and by server 2 for the second since both webservers can read and update the session information. The load balancer is oblivious to this statefulness. Central session storage is necessary since a webserver can suddenly disappear from the HTTP service when it is re-assigned to the video processing component. No user sessions will suffer from this reassignment with this approach. Layer 7 load balancing could be used in combination with central session storage to solve the same problem, but then load balancing at layer 7 would not add value since it doesn't matter which webserver serves the request. It would cost resources of the load balancer, so we choose layer 3 load balancing.

By choosing this design we introduce a possible bottleneck: the load balancer. When web servers are added, the capacity of the load balancer is not scaled. We assume this does not limit scalability in practice for applications similar to our case study, since the requirements on the resources of the load balancer are very low. This assumption will be validated in the validation phase.

Combining the hardware for HTTP service and the video processing does not increase scalability. It does however provide a solution for applications that require heavy processing capacity to utilize their hardware more efficiently than when the hardware of these components would not be combined.

4.3 Video hosting component

This component is new compared to the components described in chapter 3.3. The video hosting component sends video files to visitors viewing videos. When a visitor visits a page on the website that includes a video, the video itself will be loaded from the video hosting component. In other words, it only serves HTTP requests for the individual video files. These files are stored in the storage component.

The component includes a load balancer (layer 3, affinity is not required) to spread the load across multiple webservers.

By using this component, the HTTP service component does not have to deal with the video files. Requests for video files are very different from other HTTP service requests, since they require little server-side processing but have a long transmission time to the client. HTTP service requests are generally small in size, as are the responses (compared to video files), but they require much more work on the server. Separating these types of requests allows for more effective load balancing, since requests are more equal in resource usage. The real use of this component however, becomes visible when the content must be made accessible from more locations. One can take a copy of the video hosting component, include a storage system in it that receives updates from the main storage system, and place it somewhere else (for example, in

another country using global scale-out). In this way, new content travels to this location only once, and is served to visitors there from the “satellite” video hosting component. These visitors would still have to use the central location for browsing of other pages and video-editing however. An example of this setup is shown in figure 4.2.

The usage of this component also makes it possible for the owner of the system to use multiple colocating/hosting parties with different uplinks (and prices) for the core system and the video hosting service. It increases the scalability of the total network throughput capacity of the system: there doesn't need to be a single transit uplink that has enough capacity for the whole system; two uplinks can be used (one for the core system and one for the video hosting component). Also, clients can be forced to use a certain video hosting component by the HTTP service; which video hosting component the will use is based on what URL they get for the video file from the HTTP service.

For this load balancer we also assume that its inability to scale will not be a problem. We will validate this assumption in the validation phase.

4.4 Video processing component

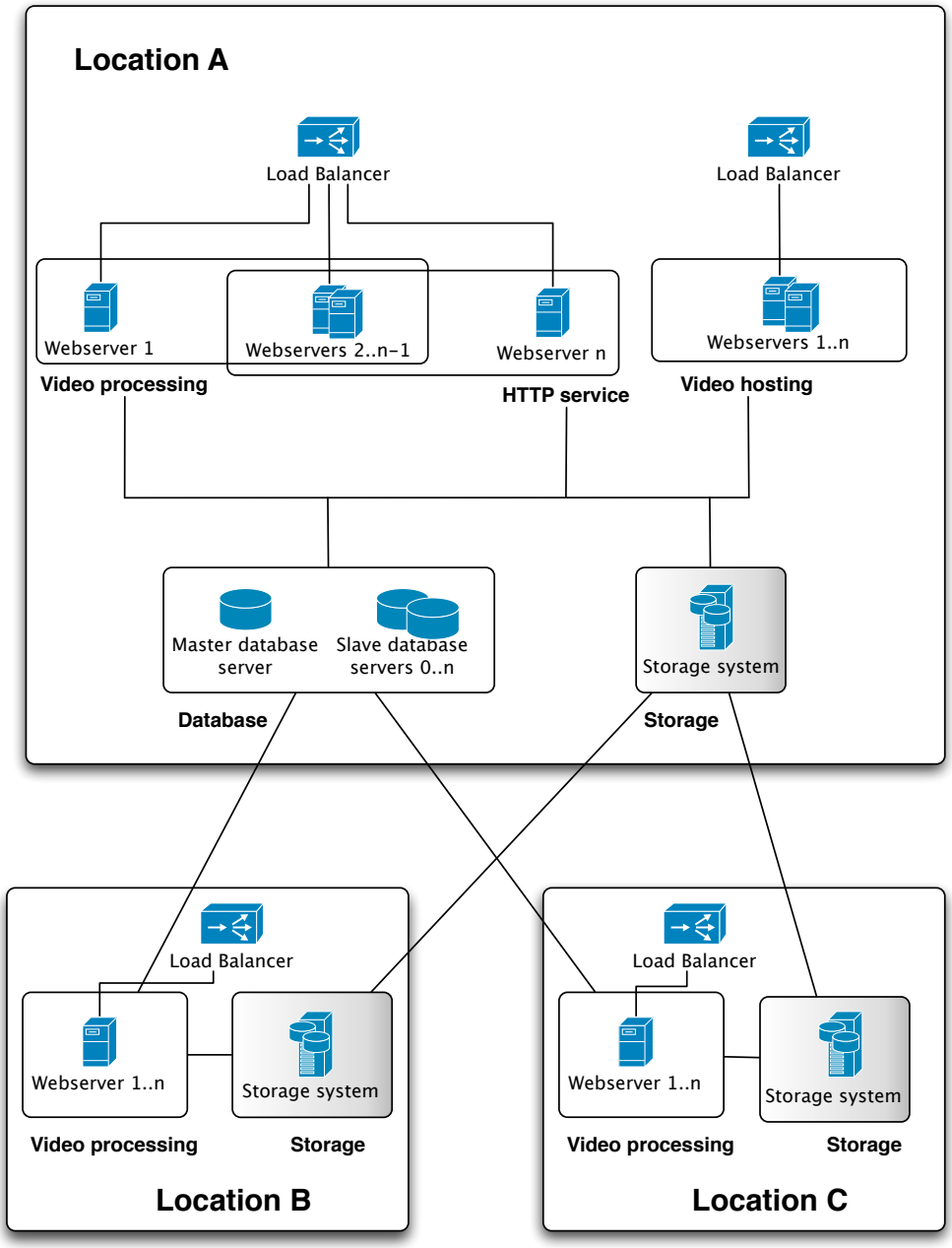
The video processing component consists of one or more servers runing software that performs video manipulation. This software reads the video data from the storage component, and the actions that must be performed from the database component. Jobs for this component are added in a queue in the database by the HTTP service, and processed by the video processing servers on a FIFO basis.

We chose to create a separate component for this task since having a component with only this single task makes it very simple to implement and scale. Would we have assigned this task to the HTTP service servers, where the requests are received, load balancing would become much more complex. A webserver would become unresponsive to HTTP requests as soon as it had a video processing task. As for scalability, since there is no centralized element in the component, it scales by adding more servers.

4.5 Database component

The database component consists of multiple servers; one master and the rest slaves. The write queries (INSERT, UPDATE, DELETE) go to the master server, the read queries (SELECT) which are by far more in number than write queries, are distributed over the slave servers.

Figure 4.2: Extended architecture



Among the options were this setup (master-slave replication), a database cluster (like mysql cluster) and data partitioning (all discussed in section 2.3.2). We have selected the master-slave replication setup since we assume that it provides enough scalability for the case study and doesn't have the complexity of a cluster setup. We assume that a master that only parses writes, in a web application (which typically have very little write queries), can be scaled using hardware scale-up enough in order not to become a bottleneck. In the unlikely situation that is becomes a bottleneck, a second master can be added which synchronizes with the first master using master-master replication. All queries will still be executed on both masters, but with relaxed timing to gain performance. Our assumptions regarding the master server will be validated in the validation phase.

4.6 Storage component

For a storage system, there are several options: simple NFS servers using data partitioning, commercial NAS and SAN systems, and a scalable software solution using general-purpose servers.

We looked into a scalable software solution using open source clustered filesystems, but found these implementations too immature to be used in production environment. Data partitioning could be a solution, but has important drawbacks as discussed in section 2.3.1.

We reached the conclusion that the storage must be handled by a storage product that natively offers possibilities for scaling. More specificly, it should offer the possibility to bundle multiple storage system together in a load-balancing kind of setup (scale-out). An example of such a product is Netapp's line of products using their Data ONTAP GX operating system. This version of the Data ONTAP operating system supports pooling multiple storage systems from the FAS series (FAS3000, FAS6000) into one virtual storage server with load balancing. Virtualization is done on NAS level: the clients connect to a virtual NAS server consisting of multiple FAS systems, much like a virtual HTTP server with HTTP load balancer. Another candidate is HP PolyServe's FSU system. This system achieves virtualization in the accessing servers, on filesystem level, using a clustered filesystem.

These products are scalable in the desired way, and offer many more features like synchronization with other locations (for example, when using off-site video hosting components with their own storage), snapshots, backups, redundancy and more.

Chapter 5

Validation of the solution

The design specified in the previous chapter needs to be validated in order to make sure that it indeed meets the requirements. In this chapter we will describe the process of validation and the results.

5.1 Approach

After validation, we should be sure that the design meets the requirements. The first of these requirements is the scalability requirement. From the literature research (chapter 2.1):

> *Our architecture is scalable if it has an at least linear increase in users that can use the system as hardware is added.*

To validate this requirement, we will use the following approach. First we will create a prototype of the system, following the design from chapter 4. We will then measure how many users can use the system by simulating user interaction. Should we now double the amount of hardware, we should be able to serve around twice the amount of users. So, we will create a prototypes of the web application and simulate the users.

In order to measure how many users the system can serve, we will use the criteria from chapter 3.2. These are:

1. Responses to requests to the community site must be delivered within:
    - 1 second for 90% of the requests
    - 2 seconds for 98% of the requests
    - 30 seconds for 99.5% of the requests

2. Every visitor viewing a video should be given at least 70 KB/s bandwidth. This ensures that loading a video doesn't take too long.[1]

3. Every video should be compiled on the server in a reasonable amount of time. This amount of time is at most the video length plus 30 minutes.

We will start with a small number of users, and then increase until either the response times are too high, the bandwidth cannot be given to users or video compilation takes too long.

Of course, the linear increase between hardware and number of users served cannot go on perpetually; every system will encounter some bottleneck (even if it be available space on our planet). We need to know which potential bottlenecks there are, and estimate if they will be a problem if our design would be implemented.

To locate potential bottlenecks, we will monitor different aspects of the system during the testing. To get as much data as we can on potential bottlenecks, the performance of all servers (CPU utilization, disk throughput and network throughput) will be monitored. When the testing is finished, we will analyze at this data to see if bottlenecks existed in the system, or if parts of the system are likely to become bottlenecks.

5.1.1 Prototype of the system

To validate the design, we need to either model the design in a simulator or prototype it. The advantage of prototyping over simulation is that the prototype is more likely to resemble a real-world system since it contains similar components. In a simulation, not all side effects can be taken into account, while in a prototype they will surface. This is also the drawback of prototyping: not all side effects that show in a prototype are relevant, because in the 'real' system they might be different. However, we have chosen to create a prototype. The scope of this project is limited, and since we already have much of the knowledge needed to setup a cluster of servers, prototyping will be faster than simulating. We have explored the simulation possibilities, and did not find a simulation tool with features for modelling servers ánd the network instead of just the network.

Also, in order to find potential bottlenecks that we didn't think of ourselves, a simulation or an analytical model would not suffice.

For practical reasons we are leaving some components out of the prototype: the storage system and the video hosting component. The storage system for instance cannot be prototyped without using a special storage hardware,

[1] This requirement will be dropped, see section 5.1.1.

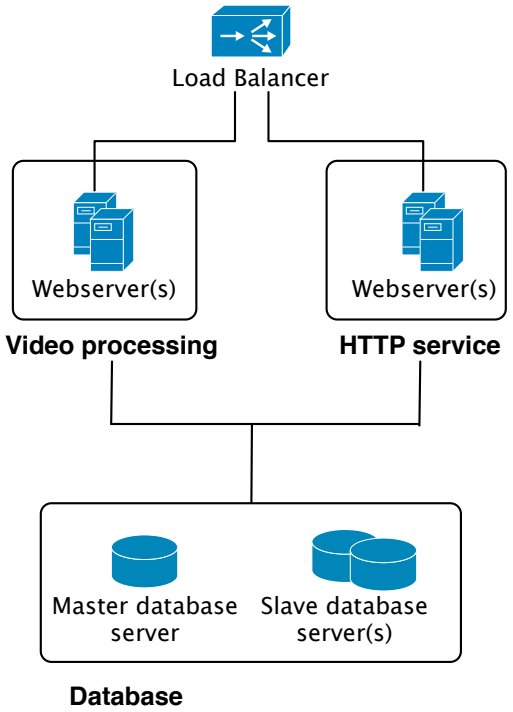
which is not available for this research. If we would replace this component with normal hardware and lower the load on it, we could include it in the prototype, but it wouldn't tell us anything about the real scalability of the system.

Also, the dynamic aspect of the server-side component, assigning servers different roles according to the needs at the moment, is left out. It does not affect the scalability of the application, it only increases server utilization (efficient use of the resources) and costs extra time and effort to implement.

Also, the video hosting component will be left out of the prototype. It functions separate from the rest of the system. This component would primarily interact with the storage component, which is being left out. Furthermore, it functions in the same way as the HTTP service: a load balancer with websevers behind it. Due to pratical limitations (availability of time and servers) we have to keep the prototype small. By looking at the server-side application we can see if the load balancing setup indeed can serve twice the amount of users with twice the amount of hardware. Since the video hosting component will be omitted from the prototype, we also drop the bandwidth requirement. Video files are downloaded from this component, so bandwidth should be guaranteed by it.

Summarizing, our validation concerns the database component, the video processing component and the server-side application component. The prototype for validation is shown in figure 5.1.

Figure 5.1: Prototype components



Hardware setup

The machines used for the prototype are all identical. They each have a 800 MHz Pentium III CPU, 256 MB RAM and a 16 GB SCSI harddrive. The systems all have Debian GNU/Linux installed on them, with debian-patched kernel version 2.6.18-5-686. No modifications have been made to the OS, other than the IPVS kernel module on the load balancer.

All machines have a 3com 3c905C network card, connecting them to an HP ProCurve 2626 managed switch.

The Apache version used on the web servers is 2.2.3. The database servers use MySQL version 5.0. The version of ffmpeg used on the transcoding servers is SVN-r10959.

The visitor simulation application is run on a Apple Macbook with a 2.4GHz Intel Core 2 Duo CPU, 4GB of RAM and a 250GB harddisk. It is connected to the same switch as the test machines.

Database component

The database component is prototyped according to the design using a master-slave replication system. The software used is the open source database server MySQL. The first setup will include one master and one slave, for the second setup a slave will be added. Strictly this does not double the hardware but this is the way the database system scales up with respect to the amount of reads it can handle, leaving the master as a potential bottleneck.

Video processing component

The video processing component is prototyped by a PHP script which invokes “ffmpeg”, an open source commandline video transcoding and editing tool. The script reads what needs to be done from the queue in the database, loads the video file from the harddisk of the processing server (since the storage component isn’t prototyped) and writes the result back there. The script is periodically executed by each server. In the first setup one server will run this script, in the second setup there will be two.

The video processing component needs to perform two different tasks: transcode uploaded materials into low-quality FLV files and combine a lot of uploaded materials into one FLV file for the final video file a user creates. The exact input and output for the video processing script does not really matter, since the output of the video processing component is never used in the prototype. One video file is used as input to both tasks. To simulate video transcoding, it is transcoded to FLV once. To simulate creating the final video, the same job is

run three times. This may not be an accurate representation of the compilation of a video project, but in our test it matters only that load is generated by users, and that twice the amount of users will generate twice the load.

HTTP service component

The server-side application component is prototyped by a PHP program which simulates typical CMS behaviour. It performs a number of heavy SELECT queries on the database, inserts some data into the database, creates PHP objects and variables, modified some video-metadata and includes other PHP files. This PHP application runs on a single web server behind a load balancer in the first setup, and on two webservers in the second setup.

5.1.2 Simulation of the users

Users, from the perspective of this system, can be defined in two types: visitors and authors. Visitors are those who access the community website and view videos. Authors create content by adding their own materials and combining library materials into new videos.

Both user types need to be simulated. The client-side application however does not need to be simulated, neither does its usage. Since the number of users is per definition in linear relation with the available hardware for the client application, we can omit it. The downloading of the client application from the server-side application and the sending and processing of requests from the client however needs to be included in the simulation.

We have written a Java program that simulates users and logs the timing results. It sends requests to the server-side application, waits for a certain amount of time, and then sends a request again to simulate a browsing user. To simulate multiple users, multiple threads are used. Using Java on Mac OS X, about 1000 threads can be created without a problem. For a higher number of users, two instances of the program must be run. A user viewing videos is simulated using the same program, but by changing the request URL. An author is simulated also by changing the request URL. An author will download small FLV video files every now and then, and request transcoding of files. Also, the program uploads video files to simulate authors submitting new content. The program limits the throughput to simulate the bandwidth a normal client would have (typically between 50 and 100 KByte/s upstream). Each thread uses a unique IP address on the LAN to avoid host-based restrictions like number of requests and throughput.

We have to make estimations for the numbers of users of different kinds that will be online at the same time and how often they will request content during

normal usage. We want to know if the application is scalable in the way we want it to, so the question at hand is if the way in which we model the users affects the scalability. If we estimate that for every author online there are about 100 viewing users online, we will get a higher load on the server-side application than if we say there are only 50. This doesn't affect the scalability of the system; it will only change the bottleneck for serving more users.

5.1.3 Testing parameters

We need to establish testing parameters: what values we will choose for the variables in the testing sequences. The value of V_{max} will be determined prior to the test runs.

Single hardware setup

Number of visitors ($= n$):	V_{max}
Number of authors:	$\frac{1}{40} \cdot n$
Visitor delay between website requests:	10 seconds
Visitor delay between video requests:	5 minutes
Author delay between video finalize requests:	60 minutes
Author delay between requests for upload & transcode:	10 minutes

Double hardware setup

Number of visitors ($= n$):	$2 \cdot V_{max}$
Number of authors:	$\frac{1}{40} \cdot n$
Visitor delay between website requests:	10 seconds
Visitor delay between video requests:	5 minutes
Author delay between video finalize requests:	60 minutes
Author delay between requests for upload & transcode:	10 minutes

5.1.4 Test data

The database system must be filled with test data in order to run tests. We created test data representing users. This test data consists of tables and records for projects, comments on videos, and user profiles including pictures. The amount of testdata matches the number of users currently being tested (total of users of the system, not users concurrently online). This is important, since a database system is under much heavier load when it has to combine two tables of each 1000 records than with tables of 500 records.

For video conversion, a Quicktime video file of 3 minutes and 20 seconds. Its video stream is encoded with MJPEG at 640x480 with 30 frames per second, the audio stream with PCM 8 kHz mono. We assume it resembles a typical home video in codecs, size and quality. Video was converted to FLV format, 25 frames per second, with 64 kbyte/s audio in MP3 format at 22050 Hz. For future reference, the complete ffmpeg command used for transcoding is: `ffmpeg -i <infile> -vcodec flv -r 25 -ar 22050 -sameq -y <outfile>`

5.1.5 Accuracy

The prototype is very rough, and the environment on the linux machines plays a role in the performance on the systems. During tests, all kinds of system processes can start to claim resources leading to higher response times. Ideally, we would run the tests for a long period, preferrably at least 5 hours to balance the effects of system jobs requiring system resources. Due to time constraints for this research, we will run each tests 3 times for half an hour. If the 3 tests are too diverse to draw conclusions, we will run more tests. By 'too diverse' we mean that the results show flapping: some of the tests fail to pass the requirements while some do, and the largest difference between results is more than 10%.

Furthermore, preliminary tests have shown that the cluster will probably service a very small number of users; somewhere around 50. The differene between 50 and 51 users is insignificant; we will consider differences from 5% up.

5.2 Results

In this section we will discuss the measurements taken on the prototype.

5.2.1 Hardware setup 1

The figure shows the response times for the first test. The first column shows the response time in ms that 90% of the responses were received in, the second column shows the same for 98%, and the last for 99.5%.

On the single hardware setup tests were run with an increasing number of users until a limit was reached. The videos were transcoded in time for all of the tests. This was verified by inspecting the transcoding queue after each test. The HTTP response times are shown in figure 5.2.

Figure 5.2: HTTP response times setup 1

	90% $\leq$ (ms)	**98% $\leq$ (ms)**	**99.5% $\leq$ (ms)**
45 users (1)	836	1316	1837
45 users (2)	821	1317	1827
45 users (3)	779	1138	1466
47 users (1)	924	1368	1833
47 users (2)	910	1387	1836
47 users (3)	987	1494	2045
49 users (1)	1046	1600	2315
49 users (2)	1000	1502	2066
49 users (3)	1063	1676	2373
50 users (1)	1127	1784	2537
50 users (2)	1137	1733	2511
50 users (3)	1158	1765	2590

From this data we learn that the system can serve at most 47 users at a time. For 47 users, all three tests show response times within limits, while for 49 users the response times exceed the limit of the 90% requirement. We haven't tested with 48 users, but as described in the previous section, this difference is insignificant. We will test if the second hardware setup can serve 94 users. We don't need to test other amounts of users, since we have an answer to the scalability question if the second setup can server 94 users.

During the test, due to an error the transcoding times of videos weren't saved while they should have been to validate the requirements. However, the transcoding times were saved in the second setup with double the amount of hardware. We have decided not to re-run the first test to obtain this data, for the following reason. There are two options for the queue times: either they were okay and the bottleneck to the number of users is the 90% HTTP response time requirement, or they were not okay and we could serve actually less users than the 47 as determined by the HTTP response times. Either way, if the second setup can actually serve 94 users in both HTTP response times and queue times, this proves our solution is scalable. In the first case it is clear; the HTTP response times were the bottleneck for the first test. In the second case, the bottleneck should actually have been reached earlier: we should be serving less than 94 users since the queue times didn't meet their requirements. By serving 94 users successfully anyway we show that our solution is more than scalable.

5.2.2 Hardware setup 2

The length of the video sample used for conversion is 3:20. It is transcoded three times to resemble compilation of a larger video file. The requirement

Figure 5.3: HTTP response times setup 2

	90% $\leq$ (ms)	**98% $\leq$ (ms)**	**99.5% $\leq$ (ms)**
94 users (1)	948	1388	1903
94 users (2)	946	1543	2567
94 users (3)	903	1284	1656

Figure 5.4: Video queue times setup 2

	Time(s)	**Time(m)**
94 users (1)	1110	19
	1594	27
	759	13
94 users (2)	1605	27
	779	13
94 users (3)	1236	21
	746	12
	1566	26

is that video files be compiled within 30 minutes plus their length, so for this sample that is 40 minutes.

The HTTP response times for the second test setup are shown in figure 5.3. The times the video processing component took to process transcoding requests, are shown in figure 5.4. Each row represents a request in the queue. These times are only the times for requests by authors for compilation of a video. The video processing component was also busy with transcoding uploaded files, which was a much lighter task. Since we did not set requirements for this task, the results aren't shown here. Not every test run generated the same amount of requests due to random intervals between requests. The time in seconds in the first data column is the time between insertion in the queue and completion of transcoding of a video file. The time between transcoding start and transcoding end is not relevant and thus not shown. The time in minutes is rounded, and only there for easier reading.

From this second testrun we learn that doubling the amount of hardware enables the system to serve twice the amount of users while staying within specified limits for response times and queue times. We need to look at bottlenecks before we can conclude that the design is scalable.

5.2.3 Bottlenecks

There are a few elements of the prototype that were not scaled between setup 1 and setup 2. They are by design not scaled, although they enabled scaling of other parts of the design. These elements are the load balancer, the

master database server and the network itself (network interfaces, cables and switches). To see if these elements could become a bottleneck, we monitored the state of all the servers during the test runs. By state, we mean CPU, disk throughput and network throughput.

We cannot draw hard conclusions about these bottlenecks. If their resource usage was very high during the tests, we can conclude that the system cannot scale to the double amount of users again, and thus for sure is not scalable enough. If however the resource usage of the load balancer and database master server is low or normal, we cannot draw conclusions on how far the system could scale out.

A way to still give an idea of the scalability limits (and idea, not a definite answer) is by looking at web application which use similar setups for these components, and serve large amounts of users.

Load balancer

Figure 5.5: Load balancer CPU load with 47 users

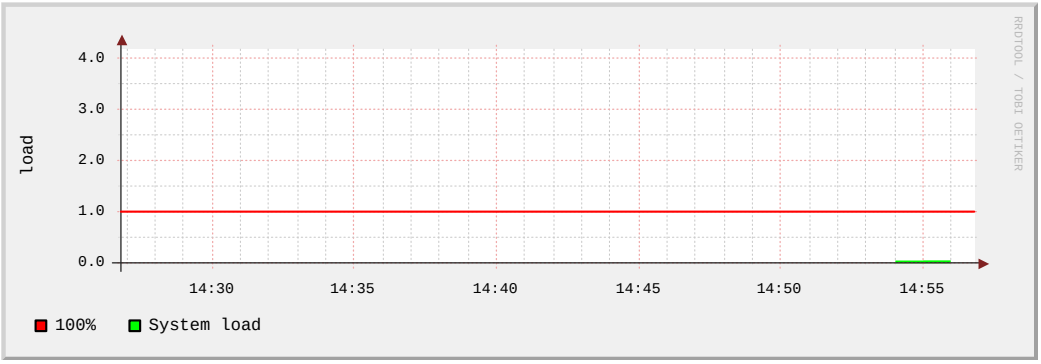
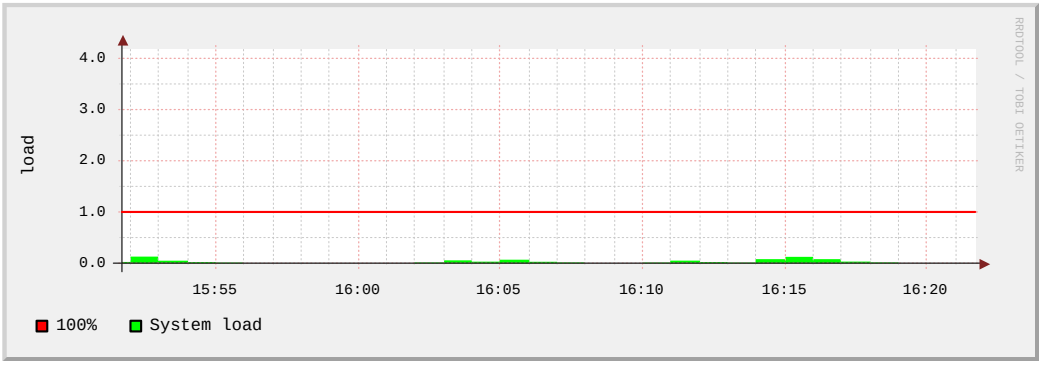


Figure 5.6: Load balancer CPU load with 94 users



As for the load balancer: figure 5.5 and 5.6 show the CPU usage during one of the tests in round 1 and one of the tests in round 2. The vertical axis indicates the load, the horizontal axis the time. In Unix terminology, load is defined in

number of processes having to wait for execution time. So, a load of 1.0 (the red line) indicates that one process is waiting for CPU access. At this point the system is considered to be 'full' although it can continue to accept more load. From this point on however, processes will suffer under the load as they have to wait.

Figure 5.7: Load balancer network throughput 47 users

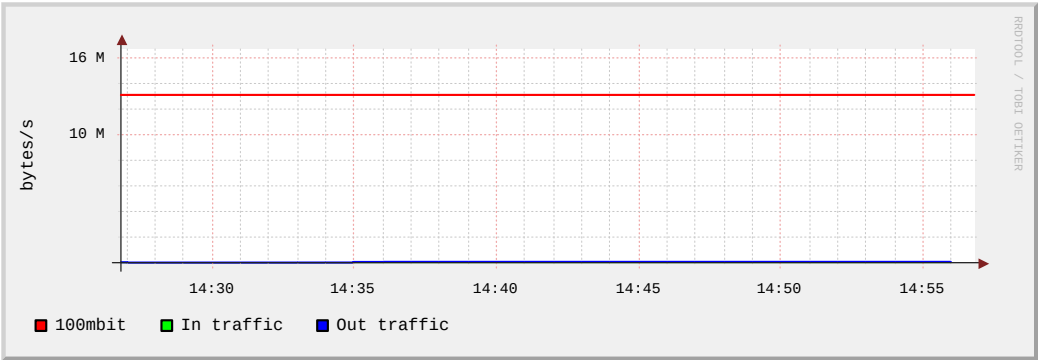
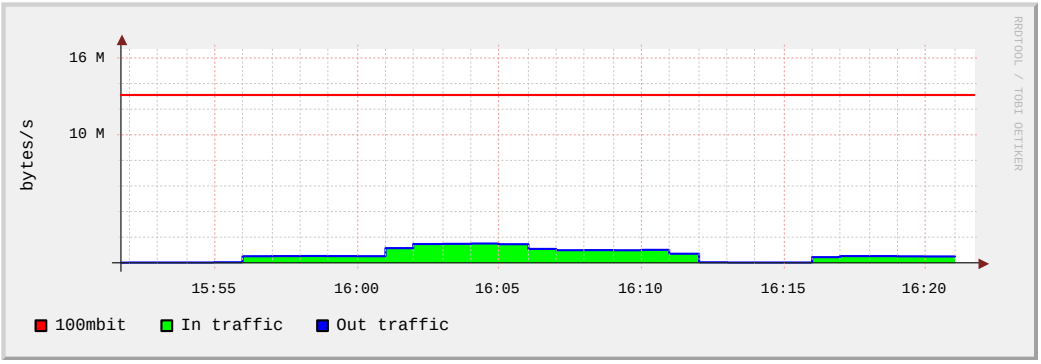


Figure 5.8: Load balancer network throughput with 94 users



The disk showed no significant activity at all during both tests.

The network activity is remarkable (figures 5.7 and 5.8). During the first round of tests, none of the graphs showed network usage. During the second round, the usage was significant. I have no explanation for this other than an error in the measurements during the first test round. The red line in the graphs shows the maximum throughput (theoretical maximum) of the network interfaces.

Database master

The CPU load of the database master is shown in figures 5.9 and 5.10. As can be seen in these graphs, the load was very low. The disk activity and network activity were close to zero (not visible in graph).

Figure 5.9: Database master CPU load with 47 users

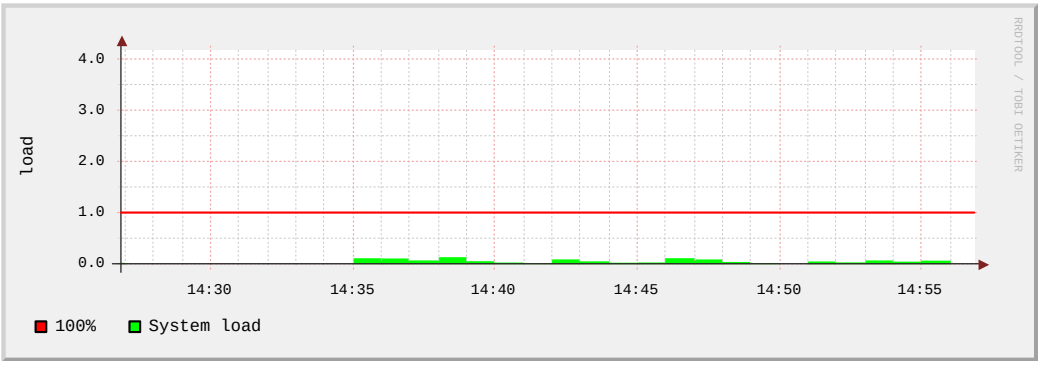
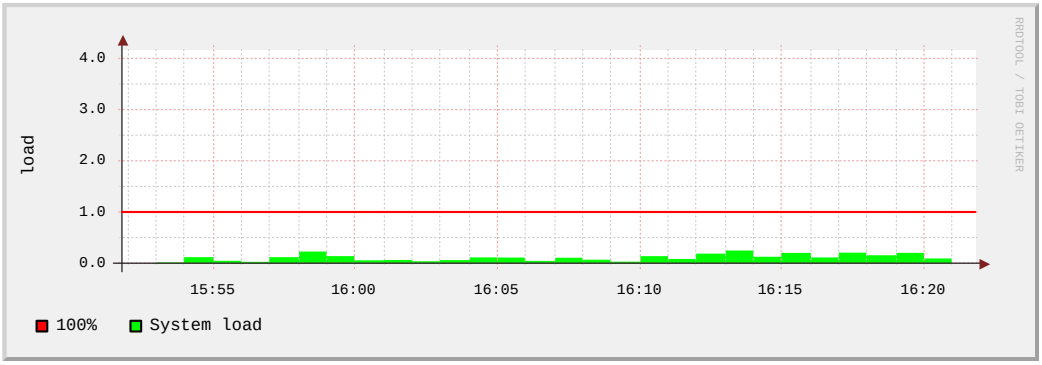


Figure 5.10: Database master CPU load with 94 users



5.3 Discussion

We have created a prototype of the design for validation, we have run tests on it and gathered and analyzed the results. We will now discuss what can be learned about the scalability of the prototype (and thus part of the design) based on these tests.

We have seen that doubling the amount of hardware leads to twice the user capacity. Response times stayed withing limits when placing twice the load on the system. As for video compilation times, in the second setup the compile times were within the requirements which means that we probably could have had more users as far as the video processing component is concerned, since the HTTP response times were the bottleneck in adding users.

As for the bottlenecks, we can conclude from the graphs that a load balancer of equal hardware as the web servers, can service 2 web servers while keeping its CPU load below 0.1 (as visible in figure 5.6). This means that for the load balancer to reach a load of 1.0, we would need 20 webservers in this setup (assuming a linear increase in load). A safe estimate of the performance increase that could be realized by software optimization is 50%, changing the number of servers to 30. The estimate is inaccurate because the optimization

has not been tested or theoretically explained, and the linear increase between load and webserver is also untested and unexplained. A 50% performance increase due to optimization seems very high, but the OS can be finetuned to perform this specific task at the cost of other tasks in multiple ways (process priority, preemptive resource management, stripping of other tasks/services). As for the network throughput, this potential bottleneck can be put out of range by using a 1Gbps network instead of 100Mbps.

Apart from this rough estimate, we can look at other real-worlds examples of this setup. Slashdot (case study in appendix A) is an example of a large web application (3.000.000 pageviews per day on weekdays) using HTTP load balancing. We estimate our case study application will have to handle about 500.000 pageviews per day: 100.000 users, each viewing 5 pages. Slashdot uses a single load balancer to distribute the requests among 16 webservers.

For the database server, the load is low as well. How many users could be served on the current hardware is again impossible to tell accurately. However, based on the load with 94 users, showing load peaks of 0.2, the system should be able to handle 5 times the amount of users (470) while incidentally peaking to 100% load. Looking at an example again, Slashdot uses two master database servers replicating each other. However, all the write queries are directed to one of them (the master-master setup is for safe failover) so effectively they use one master server, as is done in our design.

So, the potential bottlenecks in our design didn't turn out to limit scalability during the tests, and are also not likely to limit scalability until the application surpasses an application like Slashdot in size.

Chapter 6

Conclusions

This research started with the following question:

> How could web applications for online video-editing be designed in terms of application architecture in order to be highly scalable?

A number of sub-questions were stated in order to answer the research question:

- What are the definitions and state of the art of scalability and web application distribution?
- What are the requirements and characteristics for scalability in the case-study?
- How could the application be designed to be scalable to the extent in which it is required in the case study?
- Can we identify potential bottlenecks and verify the scalability of the proposed design using measurements made on a prototype of the proposed design?

We will now discuss how these questions were answered in the previous chapters.

What are the definitions and state of the art of scalability and web application distribution?

In chapter 2 several existing approaches to scalability and web application distribution were discussed. Different components of web applications are designed to be scalable in different ways. For components that provide read-only

acess to resources (such as web servers typically do) load balancers play an important role. They distribute the requests among the participating nodes, based on different kinds of conditions. For read-write components such as database and storage, simply distributing requests among the participating nodes does not suffice; there needs to be some form of synchronization between the nodes. Different approaches to this problem were discussed.

What are the requirements and characteristics for scalability in the case-study?

This question was answered by examining the case study: its tasks, the requirements that are placed on it by the commissioner and Furthermore BV, and its components. These components determine the scalability characteristics since different types of components require different approaches to scalability.

How could the application be designed to be scalable to the extent in which it is required in the case study?

A design was presented in chapter 4, using approaches found in the literature study and some custom additions that improve scalability or performance in the type of web application that is being researched.

Can we identify potential bottlenecks and verify the scalability of the proposed design using measurements made on a prototype of the proposed design?

The scalability of the design was validated in chapter 5. We concluded that a prototype implementing our design is scalable. When doubling the amount of hardware in the prototype, the amount of users that can be served also doubles. Bottlenecks in scalability were not reached, but potential bottlenecks were identified.

In order to answer the research question, there is one more step we need to make. The fact that the prototype is scalable and does not encounter bottlenecks in our test, does not guarantee scalability for a larger scale implementation of the design. So the question is: how far should a system that implements our design be able to scale out in order to be called scalable? This question cannot be answered by the definition of scalability; we will answer it using our case study.

Extrapolation of the testdata showed how far the system could be scaled up before the different bottlenecks would be reached. By comparing this scaling

space to a large webapplication currently in use, the conclusion was drawn that this scaling space would be sufficient for the case study application.

This conclusion is untested. Where the limit in scaling of the design lies exactly, is not determined by the validation phase nor will we explore it here. Based on this research the conclusion can be drawn that the design is scalable with a limit, and that the case study application is not *likely* to reach this limit.

A larger application, for example something the scale of Youtube, could not use our design. Bottlenecks that would be reached first are the load balancer and the database master server. However, for web application for online video-editing somewhere between the size of the case study and Youtube, the design is scalable. Where this limit lies depends on many factors, and is hard to predict on beforehand. Some recommendations will be made in the next section to enlarge the scalability window.

The conclusion from this research is that for web applications with high demands on processing power, storage and throughput, such as applications for online video editing, scalability can be achived, with an unknown but existing limit, using the design presented in this research.

6.1 Recommendations

In the case that bottlenecks in the scalability of the system would be reached, the system is designed modular enough to replace individual components to increase scalability. Recommendations will be made for replacing the database, storage and network components. Furthermore, we will show how the system as a whole can be virtualized for a low-cost implementation.

Database

The database component could be replaced with a clustered database. Various database manufacturers offer software to do this. An example is the Cluster edition of the popular open source MySQL database, described in section 2.3.2.

Storage

For the storage system, there are many ways to scale. The way included in the design is by accessing the storage on file-level as a NAS, and bundling multiple storage systems together behind this NAS. Another possibility, one with greater throughput, would be to use a SAN (Storage Area Network): networked storage on block level instead of file level. A SAN is typically accessed by iSCSI (SCSI over TCP/IP) or, for even more throughput, Fibre Channel.

Network

We recommend that the implementation of the design use a 1Gbps network to begin with instead of 100Mbps. The 100Mbit limit is quickly reached using central storage, whereas 1Gbps will be enough for a long time.

Virtualization

Furthermore, the number of servers for the initial implementation can be reduced by using hardware virtualization. Using a virtualization system such as Xen or VMware, a physical server runs a number of virtual servers which behave as normal servers, with even near-native performance. The HTTP service component could be virtualized into a single (hardware) server, while still having the architecture of a load balancer with multiple web servers, making scaling easier. The same goes for the database component and video hosting component. When scaling up, an extra physical server could be added, taking over some of the virtual servers.

Appendix A

Case study of Slashdot

In these appendices we will take a look at a few cases of large web applications and how they handle scalability. The first one is slashdot.org.

A.1 Introduction

Slashdot.org is a website that offers “news for nerds, stuff that matters”. It is a news site that focusses on news on technology, internet, security and related issues. Users can submit stories, ask questions and vote for stories which will then rise in the rankings. Slashdot is owned by Sourceforge Inc. It is difficult to gather traffic and visitor statistics for slashdot.org, but in the FAQ they state:

> *Slashdot typically serves 80 million pages per month. We serve around 3 million pages on weekdays, and slightly less on weekends.*

Source for this case study is a series of articles by slashdot on their network infrastructure, available at `http://meta.slashdot.org/article.pl?sid=07/10/18/1641203&tid=124`.

A.2 Application profile

Slashdot is a news site, which means it contains lots of text data (articles) and comments. Users can read, comment, search and post. Sine it is mainly about serving text pages, slashdot is mainly about webheads accessing databases, and implementing smart caching in between.

A.3 Infrastructure

The Slashdot network infrastructure is shown in figure A.1. Incoming traffic is received by the load balancer. What isn't shown in the picture, is that six of the webheads have an extra function; they act as layer 7 load balancers. The layer 3 load balancer balances the load among the six layer 7 load balancers. They dispatch the requests among the webheads according to the requested page. They also redirect registered users to special servers, granting them a better response time. The functionality of the website is segregated across servers so that if a specific part of the website has a performance problem or suffers from a DDoS attack, the rest of the website will still function normally.

The database system uses a multiple master setup. Two masters replicate each other. One of them acts as the single write database; it is the only database queries are written to. Both masters are replicated by a slave. The webheads select one of these four servers for their queries. The masters can be switched easily in case of failure; the second master (which is not used for write queries) can easily be used as write master because it is already configured as master and replicated by a slave and the other master.

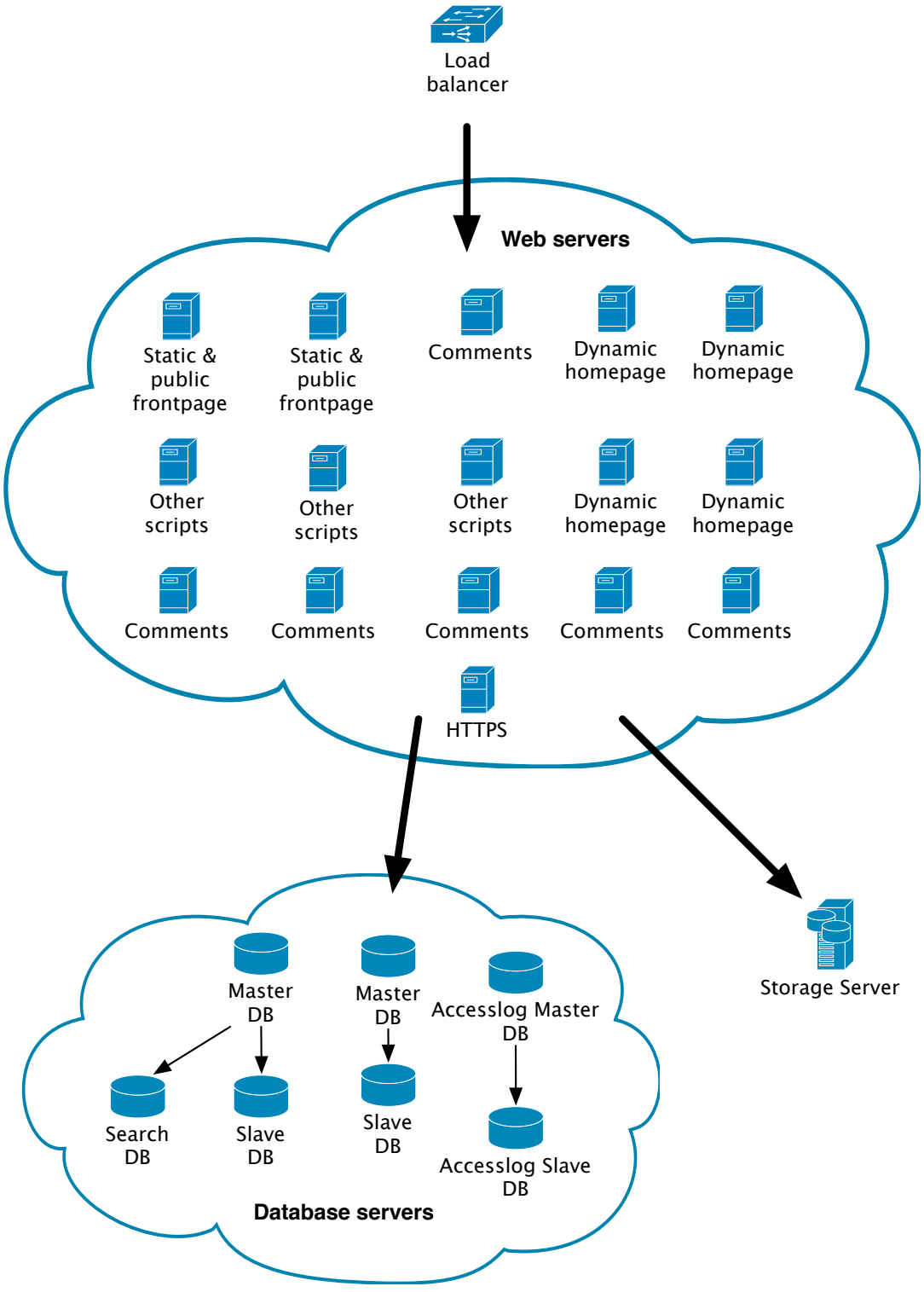
The access logs are stored in a separate database server, this is a form of data partitioning to increase performance. Since slashdot uses the access logs intensively for moderation and filtering, a separate slave database is used for read queries from the access logs. Furthermore, a separate database is used for search queries.

A single NFS server is the storage server for all static content. The webheads mount it in read-only mode for better performance (no concurrency issues).

A.4 Conclusions

Of the case studies, Slashdot comes closest to a standard web application. They have a huge number of 'read' actions compared to the 'write' actions. They use the three standard web system components: webheads, database, storage server. Since the dynamic content is all in the database, the storage system is read-only. They have solved the challenges that the large number of visitors bring by using custom software solutions; using open-source software and own software (that they have open-sourced) they optimize load distribution. Although their two-level load balancing solution is interesting, we are not likely to be faced with the same issues in the case study application.

Figure A.1: Slashdot network infrastructure



Appendix B

Case study of Akamai

In this second case study, we will take a look at Akamai, a Content Distribution Network company.

B.1 Introduction

Akamai's core business is to distribute content for their clients. When a company has a website that attracts many visitors from across the globe, and they cannot or will not increase their own hosting capacity, the company can hire Akamai to distribute its content. Akamai provides global scale-out as discussed in section 2.2.2. Source for this case study is an article by Akamai on their distribution network [4].

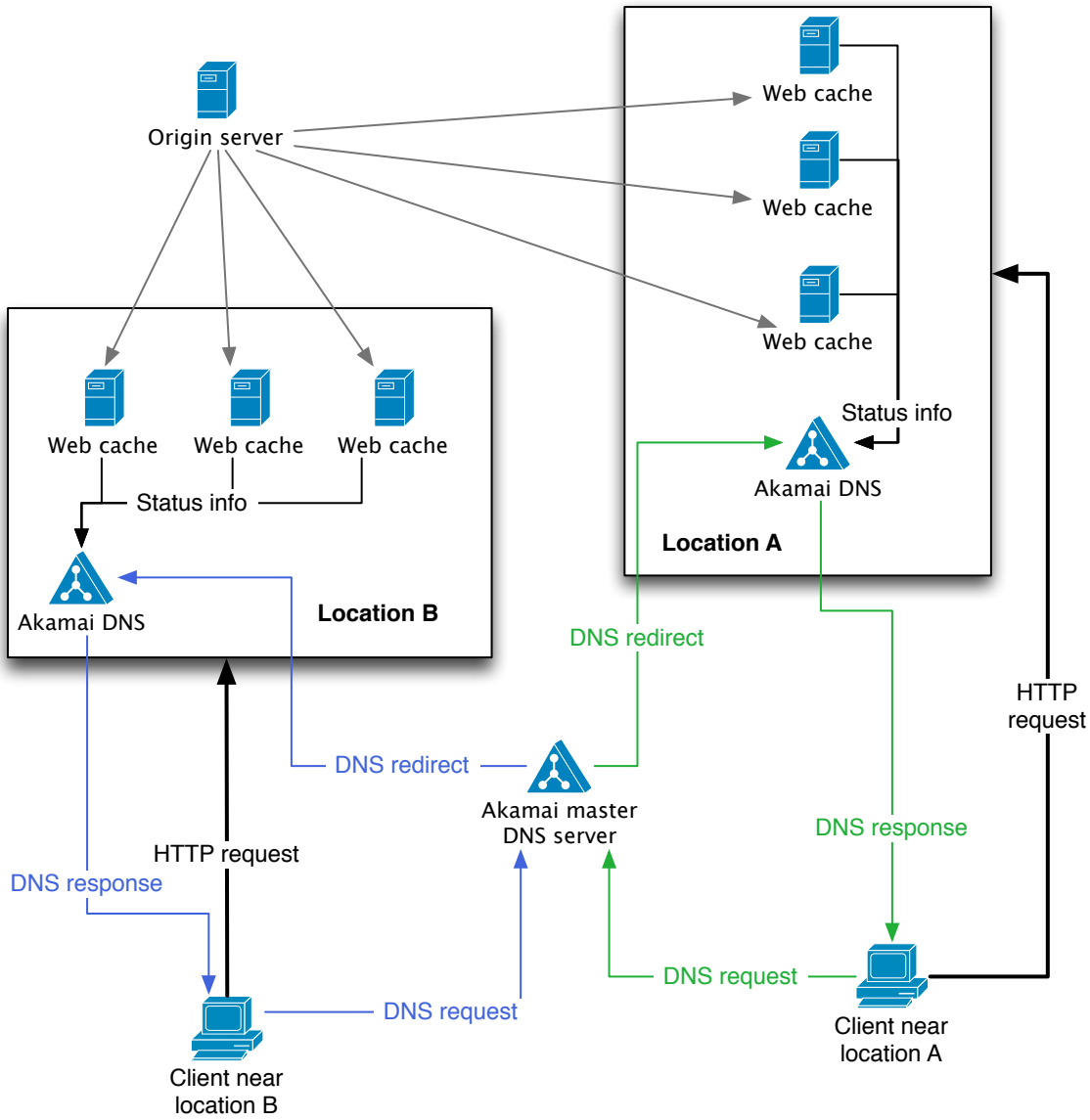
B.2 Application profile

When Akamai distributes your content, you still need to host the content yourself. Akamai picks up the content from these "origin servers" and takes care of distribution from there. This makes it easy for customers to change their content.

B.3 Infrastructure

The simplified Akamai network infrastructure is shown in figure B.1. The origin server is the system the original content is on, and where it is updated. When the content changes, the web caches shown in location A and location B will also be updated. These two locations are datacenters somewhere on the globe. In the figure there is a client near A and a client near B. When

Figure B.1: Akamai network infrastructure



they request the hosted content, for example `www.example.org`, they get redirected to the Akaimai master DNS server by the .org nameserver (we will not explain the DNS system in detail here). The Akamai master DNS server can tell from the clients IP address what its approximate physical location is. Based on this location, it responds by redirecting the clients request to a nameserver near them: at location A for client A and at location B for client B. This local Akamai nameserver collects status information on the Akamai webheads (or clusters) at the location. Based on this status information (health, availability, load etc) it responds with the IP address of one of the webheads to DNS requests. These DNS responses have a very low time-to-live (TTL) so that they will be re-requested soon. When a webhead becomes unavailable or overloaded, the local DNS server will simply take it out of its list of possible responses.

B.4 Conclusions

Akamai, as a CDN, is specialized in the global scale-out of websites and web applications. They use DNS for global load balancing, which can be very interesting to our case study. Their local scale-out method also uses DNS, which might be less suitable to our case study. In our case, we probably need more fine-grained control over the load balancing process than DNS (with its caching) can offer.

Appendix C

Case study of Google Search

The last case study is on Google Search, the most well-known service of Google Inc.

C.1 Introduction

In the early 2000's, Google became very popular for its internet search service. Though there were already many search services, Google managed to become the number one search site. Part of the secret of their success is the search algorithm, which can search enormous amounts of data very fast. Alghough the algorithm still is kept secret, Google has revealed its search infrastructure in an article which is used for this case study[1][1].

C.2 Application profile

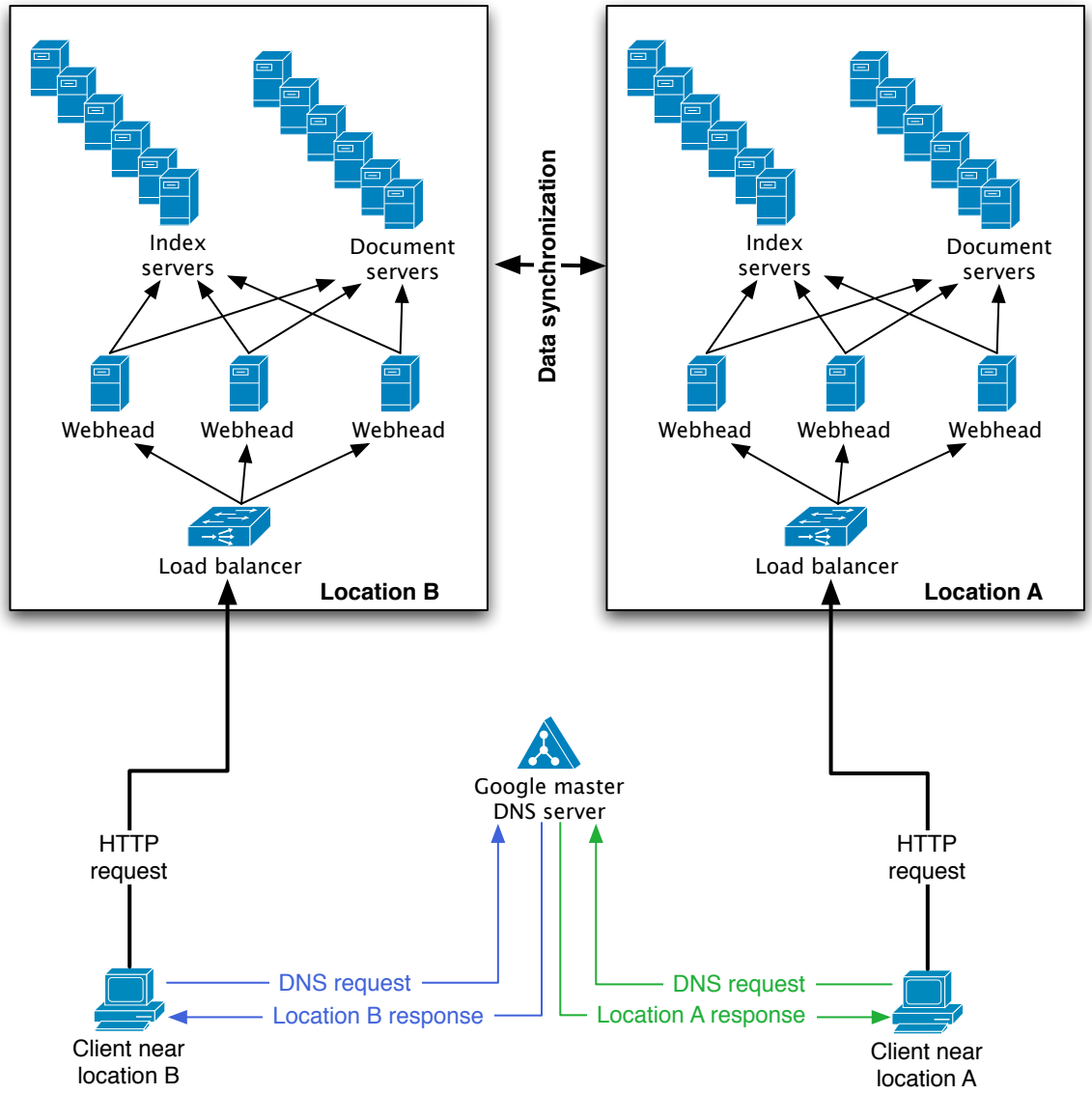
The Google web search application is an internet search engine: one types some keywords into the textbox, clicks on the "submit" button and a page is served with web pages that have a relation to the keywords. The strongest relations are presented at the top of the first page of results.

Google serves this search application on some 15,000 commodity PCs (in 2003), which together have the processing power of a large supercomputer but just a fraction of the price. Commodity hardware is not very reliable, so fault-tolerance is built in at software level. All data available in the system is replicated over multiple nodes to ensure availability.

[1] The article is from 2003. Though the infrastructure will probably be much the same now, absolute numbers might be different

C.3 Infrastructure

Figure C.1: Google Search infrastructure



The google search infrastructure is shown in figure C.1. Google uses DNS for global request dispatching, selecting a Google search cluster near the client. There the request is dispatched to a webhead using a HTTP load balancer. These webheads do not answer the search query themselves. Instead, it coordinates the answering of the question, formats it and sends it back to the client.

The answering of the query itself comes in two phases. In the first phase, the index servers are presented with the keywords. They have a reverse index that

maps every keyword to a list of matching documents (the hitlist). They intersect the hitlists of the individual keywords to determine the relevance of hits. This relevance determines the order on the output page. The total amount of data in the raw documents is tens of terabytes of data, and the index itself is terabytes to. To be able to search through this, the index is sliced into pieces (index shards). Each of these shards has a randomly chosen set from the full index. A pool of machines serves requests for each shards, and the overall index cluster contains one pool for each shard. When a request needs to be served, a load balancer selects an index server for each shard that is needed. The index cluster itself is a load balanced cluster of subclusters, each subcluster serving queries for one shard of the index.

The first phase has a list of document IDs (docids) as a result. The second phase is to send this list of docids to the document servers, which come up with the URLs, titles and summaries of each of these documents. The document servers are setup in a way similar to the index servers. The total set of document information is sliced into shards, and each shard is represented in a document server cluster by multiple servers. Requests to the document servers again are routed through load balancers.

When the document information has been gathered, the webhead creates an output page in HTML, invokes an ad server for relevant ads and a spell-checker to do spell checking on the query. When all output is gatherd, the response page is sent to the client.

C.4 Conclusions

Google Search, like Akamai, uses DNS for global scale-out. This can be very interesting to our case study. Unlike Akamai, Google Search uses HTTP load balancing for local scale-out. This approach will probably fit our case study best. Google Search, like our case study, performs a very specific task. The local cluster of Google Search isn't like a standard web cluster which has a web server, a database server and storage. In our case study, we will probably also use non-standard components. Google seems to be using normal HTTP with HTTP loadbalancing to access their non-standard services. An interesting part of the Google Search architecture that is not described, is now the document servers stay in sync. We discuss this problem in section 2.2.2.

Bibliography

[1] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, March-April 2003.

[2] Strawberry Online Hosting Consultancy. Hosting definitions. `http://www.strawberryonline.co.uk/hosting-terms.htm`.

[3] Brataas G and Hughes P. Exploring architectural scalability. In *Proc. 4th WOSP*, pages 125–129, 2004.

[4] Dilley J, Maggs B, Parikh J, Prokop H, Sitaraman R, and Weihl B. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, September/October 2002.

[5] Duboc L, Rosenblum D.S, and Wicks T. A framework for characterization and analysis of software system scalability. *ESEC/FSE'07*, September 2007.

[6] The Linux Information Project. Scalable definition. `http://www.linfo.org/scalable.html`, March 2006.

[7] Cardellini V, Casalicchio E, Colajanni M, and Yu P. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263311, 2002.

[8] Inc VirtualIron. Virtualiron livecapacity. `http://www.virtualiron.com/fusetalk/blog/blogpost.cfm?threadid=106&catid=22`. [Online; accessed 10-April-2008].

[9] Inc VMware. Vmware drs. `http://www.vmware.com/products/vi/vc/drs.html`. [Online; accessed 10-April-2008].

[10] Wikipedia. Clustered filesystem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Shared_disk_file_system`. [Online; accessed 7-April-2008].

[11] Wikipedia. Fibre channel — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Fibre_Channel`. [Online; accessed 22-February-2008].

[12] Wikipedia. iscsi — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/iSCSI. [Online; accessed 22-February-2008].

[13] Wikipedia. Load balancing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Load_balancing_%28computing%29. [Online; accessed 28-January-2008].

[14] Wikipedia. Multilayer switch — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Layer_4_router#Layer_4-7_switch.2C_web-switch.2C_content-switch. [Online, accessed 22-February-2008].

[15] Wikipedia. Network attached storage (nas) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Network-attached_storage. [Online; accessed 22-February-2008].

[16] Wikipedia. Network file system (nfs) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Network_File_System_%28protocol%29. [Online; accessed 22-February-2008].

[17] Wikipedia. Software as a service — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Software_as_a_Service. [Online; accessed 7-April-2008].

[18] Wikipedia. Storage area network (san) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/SAN. [Online; accessed 22-February-2008].

[19] W Zhang. Linux virtual server clusters. *Linux Magazine*, November, 2003.