BLITZ

A distributed CMS based on LAMP

Mads Hvelplund <mandrax@diku.dk>
Philippe Bonnet (Advisor) <bonnet@diku.dk>

March 8, 2006

Abstract

This thesis will discuss a compromise between the expensive professional web server cluster, and the cheap single-server solution based on a shared LAMP (Linux/Apache/MySQL/PHP) hosting environment. By using only the non-specialised components available in every basic LAMP setup, it is possible to build a system that provides some of the benefits otherwise only available to high-end systems at a fraction of their cost. To achieve this, the system uses multiple, independently hosted servers, that cooperate in a distributed network to form a single CMS driven web site. In the event of an error, the individual nodes can function independently to a large extent. A prototype has been developed to demonstrate the replication and synchronisation mechanisms, and while there is still much work to do, initial progress indicates that the project goal is feasible.

Contents

Chapter 1

Introduction

Anyone who has spent time on the Internet has come across a Content Management System (CMS), whether they knew it or not at the time. From the humble blog to the sprawling company homepage, CMS's are everywhere. In essence, a CMS is a way to distance the content providers from the physical storage and representation of the content. It allows people with minimal knowledge of database systems to store articles and information in a framework that allows the content to be searched, indexed, and correlated.

A minimal CMS traditionally consists of a Database Management System (DBMS) and a web server running some sort of script language interpreter. When the user connects to a CMS-driven web site, he[1] is served a script-page that retrieves the requested article from the DBMS or the file-system. The article data is then formatted by the script to provide an HTML document that can be viewed in the user's browser. This is called dynamic content.

Dynamic content is created by request as opposed to static content, that is simply a file located on the file-system. Dynamic content allows the user to receive content tailored to his request, but comes at a cost in efficiency. Modern web servers are highly efficient when it comes to serving static content. Popular content can be cached in memory on both server and the client, or on proxies along the way. Dynamic content by its very nature cannot be stored as readily. The contents of a dynamic page might depend on a user's profile, or be a compilation of articles that match a search query. To generate the dynamic content, the web server must load a parser, compile and execute the script, connect to the DBMS, and finally generate the content based on the query result. All this adds up to more time spent processing the request and serving the page to the visitor. It also consumes more memory and CPU resources.

While there are ways to optimise the production of dynamic content, it is ultimately a question a tradeoff between speed and ease of administration. On a small web site with less than twenty HTML documents and no more than a 100 files in all, a single web master can manage the site by hand. Content can be produced using web authoring tools like Microsoft FrontPage or Macromedia Dreamweaver, and added to the site manually. When a new article is added, the web master must amend menus and add references on the other pages referring to the new page. With a little forethought, page layout across the site can

[1]The male pronoun is used throughout this thesis without intended gender bias.

be managed with Cascading Style Sheets (CSS), reducing the amount of work needed when the site changes its layout.

As the site's complexity grows, the static solution will scale poorly for the web master. It will be harder to maintain referential integrity between the various pages and their individual hyper-links. Again, while there is software to check link integrity, the problem is systemic. Small errors on one page among thousands are next to impossible to detect. This is where the dynamic solution shines. The CMS controls all areas not related to writing the actual content, maintains referential integrity in links between pages, and allows the web administrator to use templates to implement his web layout.

Since a purely static solution does not scale well, it is necessary to think of ways to improve the efficiency of a dynamic solution. To boil the issue down to essentials, it's all a matter of speed. The faster content can be delivered through a limited transport medium, the *more* content can be delivered. More content delivered translates to more customers served.

When the amount of traffic to a server increases, bottlenecks are formed by the constraints of the server's hardware:

- Bandwidth
- Storage layer speed
- CPU speed

When a bottleneck forms performance drops, and fewer customers are served. Alleviating these problems is big business, and there are many commercial solutions available. For the purpose of this thesis however, we will confine our interest to those available on Open Source LAMP[2] systems.

1.1 Problem Definition

The traditional approach to creating a cluster has been to use multiple redundant database- and web servers connected by a high speed LAN and connected to the Internet via one or more dedicated load balancing servers (fig. 1.1). Performance of the individual servers can be improved by adding custom kernel modules that spread traffic across database servers in a locality aware fashion[47; 30], ensuring a minimum of cache misses. Script efficiency on web servers can also be improved dramatically by using systems that cache the interpreted script code as machine code[2; 6].

To reduce bandwidth bottlenecks and read latency, the next step is to create multiple data centres dispersed geographically and use custom DNS servers to ensure that visitors always reach the closest server. Depending on the needs of the web site provider, data can either be fully replicated between data centres, or located on a distributed DBMS.

All of the above can be accomplished with well known Open Source components. But even though the software is free, the setup above certainly won't be. Hardware costs alone would add up to thousands of dollars per data centre, and

[2]Systems built on a combination of **L**inux, **A**pache, **M**ySQL, and **P**HP are often referred to as LAMP systems.

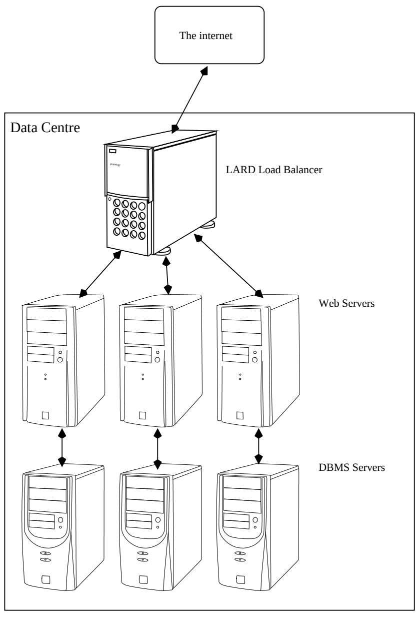

Figure 1.1: The high-end setup

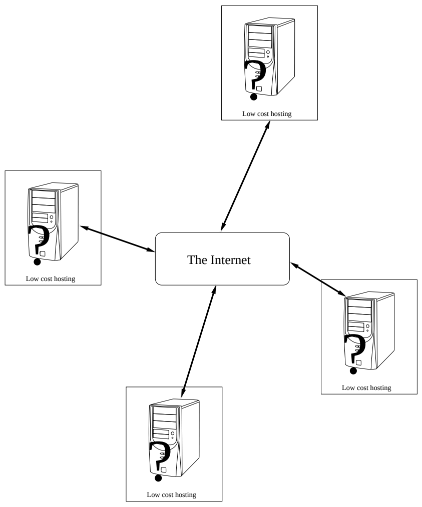

Figure 1.2: The “budget” cluster

the skilled man power needed to install and maintain the system at peak performance with multiple geographically distributed data centres, is prohibitive to all but the largest companies.

For individuals, smaller companies, and non-profit organisations, the Open Source alternative is to use a single data centre located on a cheap hosting service, sharing the same physical server with other customers. In that market, the price competition between providers is fierce. Prices for a LAMP host range from 30$ to 100$ a year. Depending on local providers, bandwidth may be rated by traffic or be free, but the cost should still be well below what a commercial line would cost.

This thesis will discuss strategies for developing a compromise. Using only the non-specialised components available in the low cost hosting environment we will develop a system that provides some of the benefits otherwise available only to the high-end version. To achieve this the system uses multiple independently hosted servers that cooperate in a distributed network to form a single CMS driven web site (fig. 1.2). In the event of an error, the individual nodes will be able to function independently to a large extent.

1.2 Design Space

The central premise of this thesis is “low cost scalability”. This means that by using a basic platform that is cheap and widely available, we can create a high availability Content Delivery Network at a fraction of the cost that commercial systems cost.

Having a system that runs on components that are exclusively Open Source is the first step. But we can cut the price of each node even further, by relying only on the bare minimum set of software that can be found in most low cost hosting environments across the world. Such a setup should include relatively recent versions of Linux, Apache, MySQL, and PHP.

For the purpose of this thesis we assume a platform of Linux 2.4.x, Apache 1.3.x, MySQL 4.1.x, and PHP 4.3.x. The specific version numbers are arbitrary, but are uniformly not the newest version of the given software[3]. As long as our proposed system does not rely on any deprecated mechanisms in these packages, performance should only improve with newer versions of the software.

In a shared hosting environment there are certain restrictions. Scripts are often run with limited rights in “sandbox” environments. We discuss these restrictions in chapter 2. For now it suffices to say that the proposed system must account for these as well.

Running on budget hardware in shady parts of the world means that the system must be relatively robust, and be able to handle loosing a node or two, either temporarily or permanently. At the same time, some low budget hosting might offer more advanced services at a reasonable price. Therefore the system should be able to coexist with other performance enhancers, such as proxies, load balancers and multiple local database servers.

1.3 Contributions

Designing and implementing a fully featured CMS is beyond the scope of this project. Instead we focus on the design of a core framework for a scalable CMS. We call this framework Blitz. We will discuss the features and design decisions necessary to implement a Blitz in a restricted low cost hosting environment, and as a proof of concept, implement the parts of this design related to data replication among nodes.

1.4 Etymology

The CMS framework developed in this thesis is somewhat optimistically named Blitz. The word “blitz” is German, and means lightning. The name originated early in the project, when the main drive of the project was to make a CMS that could deal with flash crowds. While the name itself has many connotations, including speed and rapid attacks, it was mostly chosen because “a blitz is a fast lamp”, a pun on the Danish word for a camera flash.

1.5 Outline

The chapters of the thesis are divided as follows:

- Chapter 2 explores the design constraints further and discusses the features a CMS framework should provide.
- Chapter 3 provides a discussion of the challenges in building a distributed system, and discusses how to keep the nodes in the cluster synchronised.

[3]At the time of writing: Linux 2.6, Apache 2.0, MySQL 5.1, and PHP 5.1

- Chapter 4 builds on the design discussions of the two previous chapters and goes into details on the actual implementation of a prototype.
- Chapter 5 reflects on the prototype test results.
- Chapter 6 concludes with an overview of the project and discusses the lessons learned from the prototype.

Chapter 2

Design

This chapter further explores the design space we introduced in the previous chapter, and deals with the initial design of our CMS. In this chapter we will only look at the design of a single-server CMS. In chapter 3 we extend this design to encompass multiple nodes working in unison.

2.1 The platform

As mentioned in the introduction, everything in this thesis presupposes a specific platform; LAMP.

Linux is a mature operating system based on Linus Torvalds original kernel. Since its inception in 1991, it has steadily gained ground on its competitors, maturing from an OS for hobbyists to a professional OS embraced by major corporations such as IBM. In 2005 it was the 3^{rd} most common OS bundled with new servers[28] second only to Windows and Unix.

Apache is likewise a market leader. According to a Netcraft[14] survey, Apache has been the most popular web server since 1996, with a market share of 68% at the beginning of 2006.

MySQL claimed a 44% percent market share[20] at the end of 2005, mainly due to its association with LAMP.

While PHP started as an offshoot of Perl, it has come a long way since then, becoming a fully featured Object Oriented language with a large following. Apache's own statistics say that 43% of all Apache servers run the PHP module[1] making it the single most popular Apache extension with more than 20 million installations[26].

The combined platform has become so popular that even business giants like Microsoft are beginning to pay attention[11]. It is in short: efficient, free, and ubiquitous. As a platform it should need no further recommendation to form the basis of our CMS.

[1]This only includes the Apache module version of PHP. PHP also runs on other web servers, and is available in a CGI variant as well.

2.2 Design Constraints

Working in a shared hosted environment presents limitations that a dedicated server does not have. With ownership of a server, the developer can install extra binary software packages, and have processes running in parallel with the web server doing various maintenance tasks. For example, we could let the job of keeping nodes synchronised be handled by a native process. Apart from the speedup related to moving from PHP's interpreted language to something written in native code, it would also mean that each request to the web server would need to include less PHP code, and execute fewer instructions.

Unfortunately this is rarely, or never, going to be an option in a shared hosted environment. To cut costs, hosting providers will put many customers on the same physical server. To insulate the customers' files from other customers, various sandbox and root-jail schemes are used, essentially limiting each customer to viewing their own files. To work on a large scale, the individual sandboxes must be identical and have as few extraneous files as possible.

2.2.1 Server Extensions

Both the Apache web server, and the PHP scripting engine can be extended with 3rd party modules, but again, we cannot assume that a given module will be present. Most likely the only modules present will be those needed by the minimal setup. Fortunately, the standard configuration of PHP comes with support for XML[33] manipulation and OpenSSL encryption.

2.2.2 Shared Memory

By default PHP allows programs to access shared memory on the host. This can be used to exchange data between threads and would be very useful for the semaphore mechanisms we need in chapter 3. Most hosting providers choose to run their PHP servers in what is called "safe-mode". This is a further restriction of the PHP-scripts' rights. Among other things, it disables shared memory access. Therefore we will have to rely on other methods of inter-process communication.

2.2.3 Reactive vs. Proactive

The final constraint we will cover is inherent to all web applications. In short; all operations must be initiated by a remote user. PHP scripts are only executed when someone requests that script from the web server. In most cases the script is also terminated as soon as the visitor at the other end terminates his connection to the web server. The latter can be avoided to some extent by telling PHP to finish script execution even after a connection is reset, but this may be overridden by the hosting environment. A common limitation is to limit the overall execution time to a set time-span. In our test environment, three nodes had a 2 minute timeout, while the last three had no limitations.

2.2.4 Working around Constraints

Most of the problems listed so far can be overcome by sacrificing efficiency to emulate the missing feature. In most cases, using a component written in PHP will be slower than a specialised module written in native code, but the PHP version offers greater flexibility. The PEAR[15] project provides a large library of pre-made components[2].

PEAR provides a shell-based installer that allows the user to download and install packages from the Internet while maintaining dependencies. Packages can also be downloaded to a local directory and copied to a remote server. This allows us to bundle the necessary PEAR packages with our own system for installation in a hosting environment that does not allow shell access[3].

Another challenge posed by the restrictive environment is interprocess communication and synchronisation. Unfortunately the work around here must be to use disk storage which can be very slow. One way to handle this would be to have a database table which contains the shared information. The DBMS server can prevent one process from reading a record while another process is updating it, but table locking causes a significant overhead, and only has an effect on the local node. The problems related to distributed locking will be discussed in chapter 3.

2.3 Content Management Systems

The term "Content Management System" covers a wide range of software systems that fulfil very different roles, not all related to serving Internet content.

A typical CMS provides an abstraction layer to a collection of documents, allowing users to search, retrieve, and edit the documents. Documents can be actual files or records in a database depending on the purpose of the CMS. In the following we will focus on the type of CMS used on the Internet to organise and deliver web content.

2.3.1 Example - an Online Newspaper

Creating a web site is an involved process that includes creating content and formatting it in a way that can be read by a browser. Pages need to be organised in relation to each other, and placed where the web server can access them. In a larger web site, such as an online newspaper, this process will involve many people. A design agency will provide the layout specifications and graphics, and web designers or programmers will then translate the layout into an HTML compliant[29] presentation. Journalists then write articles, that have to be edited and approved before being published to the site, laid out in the proper design, and placed in the correct sections of the web site with digitised images taken by photographers. Each of these tasks requires specialised knowledge to perform, and no single employee can have all the skills required. To put it bluntly, the artists can't spell, the programmers can't draw, and the journalists don't know how to use a computer except as a word processor.

[2]Similar to what CPAN does for Perl.

[3]Shell access means allowing a user to log on to a server and execute programs on the server. Very few hosting environments allow this.

This is where a CMS becomes so valuable. By allowing every person in the pipeline to work on their designated task in parallel instead of waiting for the previous step in the process to finish. The following example assumes that the CMS uses a common pattern called Model / View / Controller, or simply MVC[13]. The MVC pattern separates data access, business logic, and data presentation, making it easier to modify a part of a system without breaking the other parts.

First a web designer creates a layout using mockups and drawings. When it is approved, he makes a set of “template” pages. These templates do not need to depend on the actual implementation of the document storage. A template simply contains a layout description with blank spots for content. Software engineers then link the templates to the CMS, and create the necessary code to pull content from the backing storage to fill in the blank spots.

Now the journalists, photographers and editors can do their work. Journalists are presented with a simplified interface that allows them to paste their articles from their word processor of choice. The articles remain invisible until they are edited and approved by the editor, at which time he picks a spot in the virtual hierarchy of the web site for each article.

Only when an article has been “published”, does it become visible to visitors. The front page of the web site automatically displays the newest articles in order, using snippets of text, or “teasers”, drawn from the articles. From the front page or a section page the visitor can follow links to the main articles or search the entire archive. Searching can further be incorporated into the actual articles to display a sidebar with other articles relevant to the same subject.

Once the site structure is in place, programmers and engineers generally need not interfere in the daily workings. But should a situation arise where the code or layout behind the newspaper’s web site needs to be changed, it is a simple matter to upload new code or templates, since the content in theory does not depend on the presentation.

In the example, we did not specify a storage model. That is because it did not matter to anyone but the software engineers. Articles could be stored in XML files, in database records, or a combination thereof. All that matters here is that the CMS knows how to read, write, and search the storage layer.

2.4 The CMS Framework

Now we turn to the design of our own CMS framework. The example above gives us a good view of what is needed:

- Application Logic (the “Model”)
- User Interface (the “View”)
- Storage Model (the “Controller”)
- Security Model

The latter part was only implied in the example, but is still necessary. To stay in the frame of our example, what would happen if visitors could change articles they didn’t like, or if journalists could publish unapproved articles. The result would probably be much like the Wikipedia[27].

Of the other three, we leave the View and the Model largely to the developer that will use the CMS framework. That leaves the Controller.

2.4.1 Storage Model

The core of a CMS is its storage model, or "Controller". In choosing our storage model we have to consider our needs. We need to store our data in a manner that is first and foremost easy to search. One of the hallmarks of a good CMS is its ability to organise and extract related data. Second, we want to be able to annotate the documents with comments and notes. Examples of annotations to a document are a log of notes from an editor to a writer, or a list of references to related articles. Finally we want to be able to indicate ownership of documents and to restrict access to them based on a user's rights.

Documents as Objects

So far we have not discussed what a document is in our frame of reference. A document is any resource that can be transmitted via the HTTP[38] protocol. To represent documents in our CMS it makes sense to view a document as an object in the Object Oriented Programming sense of the word. Documents are instances of a document class. The class is given methods that allows it to retrieve and store its associated data. By viewing the document as an object we do not have to store complete HTML versions of all our documents. Instead we store the parts of the document that are not layout or format dependent, and provide the controlling class with a "View" that can render the object instance on demand. Similar approaches are used in other Content Management Systems such as Globule[49] and Fabricata[7].

Files

We now look at various ways to store our document objects. The simplest method is to use plain text files on the web server. Files are efficient for some of our purposes. Web servers are optimised to handle small files and can cache them intelligently, but when it comes to finding random data in a large nested file structure, performance drops. Furthermore, most file systems do not provide a way to annotate files[4]. One solution is to store annotations as files along side the document (fig. 2.1). In the example the document consists of all the files stored in the directory. The main text of the document is located in one file while each of the meta-data fields are represented by a separate file. This is similar to how Apache's WebDAV[41] module stores properties on files.

```
blitz:~# find /var/www/articles/
/var/www/articles/Employment%20is%20up
/var/www/articles/Employment%20is%20up/author.txt
/var/www/articles/Employment%20is%20up/body.txt
/var/www/articles/Employment%20is%20up/relatedarticles.txt
```

Figure 2.1: Files and annotations stored in a file system

[4]Aside from assigning each file a name.

This model has several advantages. It is very easy to implement, and searching the repository can be handled with the *nix `grep` command. The downside is, that searches generate a lot of disk activity and take a time proportional to the combined size of all properties! Even with file system caching, this is not efficient.

A popular variation on the above scheme, is to store all properties pertaining to a document in an XML file. The wealth of software available to handle XML files makes this easy to implement as well. Depending on the CMS' implementation, rendering the document so a visitor can see it, can be as simple as performing a set of XSLT[36] transformations to turn the XML document into a HTML document. But as a stand-alone solution, it fails for the same reasons as the flat file system. If we want search responses that are faster we need to turn to databases.

Databases

Storing documents in a relational database allows us to use the database to index properties, but we still have to settle on a scheme. The simplest scheme is to have a table where each record corresponds to a document, and each column corresponds to property. This means that all document types must have the same database scheme. If we want to add a property to one type of documents, we have to add it to all of them.

A compromise would be to have columns only corresponding to common properties, and put all the extra properties in a single column, perhaps encoded in XML (table 2.1). This is how the Fabricata CMS [7] handles documents. Both models are simple and reasonably efficient, if all documents are similar, we seldom need to search the properties that are stored in a single column.

Table 2.1: *"One table to rule them all . . . "*

Field	**Type**	**Null**	**Default**
id	int(10)	No	Auto increment
parent_id	int(10)	No	0
modified	date	No	0000-00-00
image	blob	Yes	NULL
data	blob	Yes	NULL

This brings us to the last model we will present, and the one we ultimately choose. In order to fully leverage the advantages of a relational database we can split the data into multiple tables. Making the individual tables more compact and easier to index.

We assume that all document types share certain common book keeping information. Henceforth we will refer to this data as the document's attributes, where as the user modifiable fields on every document are called properties. This follows the naming convention adopted by WebDAV[41]. Since all documents share the same set of attributes, these are grouped in a table (see table 2.2 to 2.4). What constitutes attributes will be covered later.

Since every document is represented as an instance of a class, all documents of the same class will share their set of properties. Therefore we further add a table for each class to store properties. To link an object's attributes with its

Table 2.2: Structure of table attributes

Field	Type	Null	Default
id	int(10)	No	Auto increment
parent_id	int(10)	No	0
modified	date	No	0000-00-00

Table 2.3: Structure of table properties_article

Field	Type	Null	Default
id	int(11)	No	0
author	varchar(64)	Yes	NULL
article_body	text	Yes	NULL
related_articles	text	Yes	NULL

Table 2.4: Structure of table properties_image

Field	Type	Null	Default
id	int(11)	No	0
photographer	varchar(64)	Yes	NULL
image	blob	Yes	NULL

properties we assign all objects a unique id. In a single server setup, a simple incrementing id will suffice. We also need an attribute that lists each object's PHP class, so we know which table to look for the properties in.

Since our objects mimic ordinary files we need a way to indicate their relation to each other. Thus we add a "parent id" attribute. Now objects can be nested below each other like files in a directory. Next we add a modification time stamp and a filename attribute. Since auto generated ids are hard to remember for humans, we want each object to have a human readable URI. The filename then forms a URN[5], and a URI is generated by concatenating the URN's of parent objects. Objects with no parent id can be considered to lie in the "root directory".

The actual implementation of this model in Blitz has a few more attributes (see appendix A) but these are unimportant for now. Searching the split tables can be faster than the single table model if we assume that we will generally be looking for either objects of the same type (stored in their class table) or looking for relations between object (stored in the attribute table). The query process is covered in more detail in section 2.4.5.

2.4.2 Application Logic

Returning to the original MVC pattern we now come to the "Model". Since the application logic of a site depends on the developer's needs, this lies outside the scope of the CMS framework. To make his application logic, the developer only has to extend the basic object class, and add his the appropriate methods.

[5]URN: Uniform Resource Name, URI: Uniform Resource Identifier[32].

2.4.3 User Interface

In presenting a "View" of the CMS to a visitor, the CMS must translate objects to documents. When PHP first evolved it was common to mix the HTML code with the PHP code. While this is still possible, it scales poorly when sites become more complex. Instead developers today use various forms of templating systems.

One example is the popular Smarty template system[24]. Smarty keeps HTML in separate files with tags marking where PHP values should be imbedded. To speed up parsing, Smarty intelligently caches fragments to static code making it very fast and flexible.

The CMS should have two distinct interfaces. One which it presents to the visitors, and one which content providers use to fill in content. The visitor interface is easily implemented using a something like Smarty or Savant[22] or even traditional mixed HTML/PHP code. This is left for the developer to decide.

What our framework must provide is a form of "page dispatcher". The dispatcher is a script that presents the interface to the visitor, by parsing the visitor's request, and loading the correct object. The object can then render itself according to the dictates of the developer's "View" code.

As an example, we return to our online newspaper. Each article in their database has some sort of unique id. If the dispatch script is `article.php`, loading a page could be achieved by adding the unique id of the article to the URI (`http://www.freespeech.cn/article.php?id=19770901`). This is easy to implement, but not very elegant. Humans have a hard time remembering obscure numbers[6].

A more elegant solution is provided by Content Management Systems such as Fabricata CMS. This CMS uses a feature in Apache that allows a script to behave as a directory.

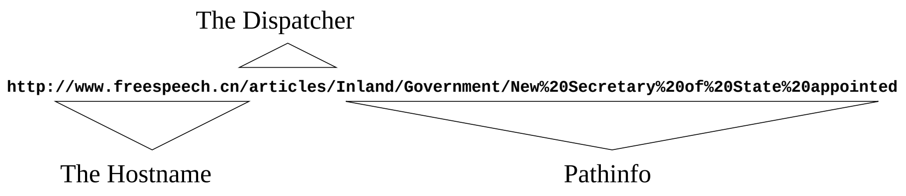
The Dispatcher
http://www.freespeech.cn/articles/Inland/Government/New%20Secretary%20of%20State%20appointed
The Hostname
Pathinfo

Figure 2.2: An example of a dispatch script

When Apache evaluates the HTTP request it looks at each element of the path, and displays the first one that is not a directory. If this is a script, the script is executed, and the remainder of the URI is passed to the script in a variable[7]. We'll call this script the dispatcher (fig. 2.2). In the example, the dispatch script is called "articles" and is located in the document root of the web site. Depending on their setup, some Apache servers will guess that "articles" is actually "articles.php", but we can also instruct the server to treat a file without the ".php" suffix as a script file.

When the dispatch script is called it treats the remainder of the path as a URI for an object in the database. In the example, that would mean that

[6]Blitz uses a 128 bit hexadecimal identifier

[7]PHP scripts can access it through `$_SERVER["PATH_INFO"]` .

the dispatch script should load the an object named “New Secretary of State appointed”[8] nested in an object named “Government”, nested in an object named “Inland”, and so forth.

URI’s of this type are more attractive and user friendly, allowing people to recognise the URI at a glance, and perhaps remember it. Since the names of objects are part of the URI, some search engines will also give the pages preference because the URI is part of their weighting algorithm. The downside is that deeply nested website can theoretically “run out of space”. While there is no standard that specifies the maximum length of a URI, some popular browsers put a limit on it anyway[9]. This is normally only a concern if the page uses GET[10] based forms.

The Administrative Interface

Since Blitz is only a network communication prototype it won’t feature any administrative interface in the normal sense. If it had one, the interface should allow the content provider to view parts of the database and edit them, while at the same time obscuring the actual data representation to avoid confusion. There are various possible approaches, but they boil down to two non-exclusive alternatives. The web interface and the external interface. Web interfaces use DHTML to create an environment where the user can see the structure of objects on the server, and edit selected fields using a form based interface. Input may or may not be validated on the client before being submitted. The other approach provides an interface for external programs to access internal processes, using technology such as SOAP[42] or XML-RPC[56]. The latter is a precursor to SOAP, but widely implemented and somewhat simpler in structure.

SOAP on the other hand is far from simple[11]. It is a complex and powerful protocol to allow one machine to execute functions on a remote machine and retrieve the results. It is widely used in Microsoft products, and fully integrated in all Microsoft development tools. That makes it an ideal candidate for a CMS interface, while developing a browser based application with DHTML is tiresome because no two browsers interpret the HTML standards in the exactly same way. SOAP has also been implemented in PHP, making it possible to build both web interfaces and client-side interfaces using the same handles. In theory developers could create specialised interfaces catering to the needs of each group of content providers.

2.4.4 Security

The users of a CMS are generally divided into content providers and content consumers. Consumers should be kept away from the part of the CMS that allows the user to add content. Neither should content providers need access to all areas of the administrative interface. Finally consumers may be divided along such lines as paying customers and and idle visitors, and the content available to one may not be available to the other. A security framework is needed to handle these divisions.

[8] “%20” is how spaces are encoded according to the standards in RFC2396[32].

[9] In some cases as low as 255 characters, according to RFC2396.

[10] POST and GET are request types specified in the HTTP[38] standard.

[11] Ironically, SOAP is an acronym for Simple Object Access Protocol.

While our prototype does not implement a security system, the problem has been given some thought. As a brief outline, we propose to create a set of PHP classes to handle security: Group, User, Member, Restriction, and Access. For reasons that are discussed later, most of the classes should not have any properties, relying instead on their attributes to store all information. Only the User class should have properties since it will need to store additional information about the user such as a password and possibly an email address.

Groups and Users are straight forward, they represent authorities with different rights. To make a User a member of a Group, we create a Member object with the Group as its parent, and set the URN of the Member object to the Id of the User.

To restrict access to an object we create a Restriction object as a child of the restricted object. The URN of the Restriction object contains a binary flag similar to owner rights in a file system. This can be as simple as read/write, or as complex as desired. To indicate that a user has the right to read the object, we would make Restriction object with the "read"-flag set in the URN.

Beneath the Restriction object we then add a list of people and groups with this particular right represented as Access objects. The Access objects store the Id of the authority they represent in their URN.

By using the above outline it could quickly be ascertained if the access to an object is restricted, and if so it would only take one query per restriction set to find the id's of users and groups with access. If there are no Restriction objects, the site's default access policy would apply.

2.4.5 Querying the Document Store

When a visitor requests a document from our website, the CMS loads the object representing the document from the storage layer. The Controller part of the CMS executes a series of SQL statements and interpret the results to generate a PHP object that is then processed by the Model and View components in the upper layer of the CMS.

Loading objects is an example of a query. The Model asks the Controller for all objects matching certain criteria. When loading an object, the attribute table is searched for an object matching a specific identifier. Blitz also uses version numbers, and together with the id the version forms a unique identifier for each object. The version system is discussed in section 2.4.8.

If no version number is specified at load time, we try to load the first object that matches the identifier and isn't marked as deleted. If a match is found, the Controller loads the properties for the object from the property table assigned to current object's class. The aggregated result is then returned as an object instance.

A query that specifies a set of conditions instead of a specific Id is called a range query. We group range queries into two categories: those that depend solely on an object's attributes (attribute queries) and those that depend on at least one property (property queries). Attributes contain information about the location of objects in relation to others in the CMS hierarchy. They also contain versioning information and time stamps. Properties are defined by the developer, which makes it hard to say much about their use. Presumably the properties will store document content. The reason for this division becomes clear in chapter 3.

To get an idea of the ratio between attribute queries and property queries in a running system, we look at a small business website running on the Fabricata CMS. The site consists of 213 "objects". Fabricata caches objects in memory, meaning that subsequent loads during the same session return the cached object. Each page on the site references several objects, meaning that requesting a page causes an average of 50 queries, the majority of which are memory cache hits. Table 2.5 displays an example of a query distribution from a complete site download. Cached hits are disregarded in the "Count" and "Percentage" figures.

Type	Count	Percentage
Load by name (total 929)	929	8.7%
Load by id (total 24564)	6973	65.4%
Attribute query	2452	23.0%
Property query	315	2.9%

Table 2.5: Query distribution

The statistics show that only a small amount of the queries are in the form of property dependent range queries. The reason for this is, that by far the most queries are initiated by the CMS to draw various forms of menus and to generate URI's for HTML links. All the data necessary for most menus are located Blitz' attributes, so this should also be the case here. Fabricata does not allow us to easily distinguish how many of the load operations could be accomplished using only attributes, but given that most of the queries are used to draw menus by loading the object and extracting a name and some hierarchical information, it is likely that the majority of the loads do not depend on properties.

2.4.6 Caching Search Queries

Searching the database is slower than loading a list from a file on disk. The obvious solution is to cache the result of queries, but simply caching *all* property based range queries is probably not going to do a lot of good. The reason is that property queries will not display the same characteristic patterns as attribute queries. An object's attributes are well defined for all objects, and are primarily used to find relations between objects. Properties contain developer-defined data. An article might have a "body_text" property that will be searched with regular expressions, or an "editor" property that is searched by direct comparison. As an example of what property searches will be used for, we can imagine our online newspaper having a search box on each page allowing the visitor to search the article archives using a simple text query. The results from these queries depend not only on the visitor's interests, but on his way of phrasing that interest, and even on his spelling! Caching a query for the word "censorship" won't help us if the next visitor searches for "censership". The article, "A Day In The Life Of BBCi Search"[31], provides a study of the search patterns on a major online news network. It shows that 80% of all searches were unique, due in part to unorthodox spelling. The BBCi's search uses a number of highly domain specific methods to deal with this. The simple conclusion is that building a fully featured search engine is a subject that requires a deeper understanding of the web site's intended usage. It does therefore not belong in the CMS framework itself.

2.4.7 Write Operations

So far we have only discussed reading data from the storage layer. Obviously we need to be able to write data as well. When an object is saved, the object's attributes to the attribute table. Then the CMS picks a property table based on the object's class. When a new object is created it should be assigned a unique identifier. In a single server setup this is easy. If the first object has $id = n$ then the next object created would be $id = n + 1$. MySQL even supplies an integrated `AUTO_INCREMENT` function for this.

2.4.8 Rollback

A cynic might say, that whenever you allow people to make changes, you also allow them to make mistakes. Therefore most desktop software allows the user to undo changes to a certain degree. This makes sense in a CMS as well. One way to allow undo, is to store a journal of changes, in the manner of version control systems like CVS[5] & Subversion [25]. Another way is to keep older versions until they are explicitly deleted. Each version of a document is then given a version number. When a user deletes and object it is marked as deleted, and filtered out of subsequent range queries. The CMS operation that load objects from storage will load the current version unless given an explicit version number, in which case it loads the older version. This is simple to implement, but requires that the system maintainers clean out obsolete copies from time to time.

2.5 Performance Enhancement

So far we have not devoted much time to discussing the cost in speed and computing resources inherent in the CMS approach to building web sites. Modern web servers, including Apache, cache files in memory. Cached files can be returned at at a much faster rate than files that need to be retrieved from disk.

While Apache can cache the actual PHP scripts in memory, it still has to invoke the script parser to generate the script's output. The reason for this is simple. Output may depend on factors that are unknown to the web server. Therefore it cannot simply blindly cache the output and serve it in subsequent requests.

While the overhead of invoking the PHP parser is reduced when the parser is used as a server module instead of a CGI script, it is still significant. Under heavy workloads it becomes even more marked as each PHP instance consumes memory and processor time, where a static file can simply be streamed unaltered from disk or memory.

A partial solution, is to assign responsibility for caching to the CMS. Presumably the CMS is better qualified to know when output from a script can be safely cached and when it must be regenerated. While this still incurs the initial overhead of loading the PHP parser, execution time can sometimes be greatly reduced by caching the parts of a scripts output that requires costly computation, such as database queries and XML parsing.

2.5.1 Caching Strategies

Section 2.4.6 touched briefly on the benefits of caching property queries. This section will explore caching in general.

When we cache something we have two initial choices. One way is to make objects responsible for removing cache entries that become invalid when the object is modified. This is difficult to implement in the case where the View of one object depends on the contents of others that may not be known in advance.

As an example, we take a web page containing a top menu with links to all other pages in the same category. To render this page the CMS must first build a list of other articles in the same section, and extract their titles and a link to each. Whenever one of the pages in the same section of the site as this page changes its title, the cached version of this page will be invalid. To implement a system where a page registered with the other pages in its category for update notifications would require that the page that wished to be cached also kept track of which of its neighbours knew that it depended on them. The overhead involved as well as the extra responsibility to the developer makes this solution impractical.

The alternative is to accept that a visitor is not always going to get the current version of an object. Instead we guarantee that the version of the object that a visitor receives will have a set maximum age. This means that cache entries grow “stale” after an amount of time, after which they must be regenerated.

The decision now becomes to make a set of rules that govern when a cached object is stale. To decide on a cache strategy for Blitz, we make some assumptions based on practical observation. Objects that wrap static files are unlikely to change very often once they have been uploaded. This is because they generally have to be created and edited off-line, and even the minimal work involved in uploading and downloading the object to edit it, is likely to make a content provider think harder about changes.

On the other hand we have documents that are stored fully in the database, such as articles, blog entries or product descriptions. The access pattern on these will differ from the static files. These objects are often at least partially edited using the CMS’ administrative interface, making it easy to change something and observe the results. Therefore it is likely that such objects will have several minor changes shortly after being created, but will then remain mostly unchanged for the remainder of their existence.

The final type of content considered, is generated content. While everything falls into this category to some degree, we are referring to objects that are created dynamically based on the site’s visitor interaction. This could be objects representing the visitor’s session. Since the representation and content of these objects may change quickly and/or depend on other objects, they are difficult to cache properly.

Since different object types are going to have different needs, we let the objects themselves decide how long to be cached, by giving all objects a method that returns the Time-To-Live (TTL) for an object instance of this type. The default value is an hour, but a developer can override this in his own classes.

When a part of Blitz wants to cache something, it will ask the object how long it should be kept in the cache. An object that depends on several sub objects could iterate through these and return the lowest value as it's own TTL.

In the example above, 60 minutes is probably a good default for static files, but this value could easily be larger. For the articles, TTL could be a function of how long it has been since the object was modified[35], so that an object has a short TTL if it was modified recently, and a long TTL if it has not been modified in long time. For the generated objects TTL could be 0, meaning that they would not be cached.

2.5.2 Caching Pitfalls

Exactly what is to be cached is left to the developer, with the exception of range query results (see section 2.4.5). This is because it is difficult to make general rules for when using a cache will provide a real performance boost.

There are also cases when the content of a page is specific to the visitor. A site might store the visitor's preferences and format output to fit these. In this case two different visitors may see a very different version of the same object.

While some of these pitfalls can be overcome, they are best handled by the developer during site development.

2.6 Related Work

Open Source Content Management Systems are a teeming mass of different systems with more or less well thought out designs. Sites like `http://www.opensourcecms.com/` and `http://www.cmsmatrix.org/` provide a comparative overview of the various offerings.

A quick look at the CMS Matrix shows us that there are more than a dozen CMS's that are based on LAMP or LAMP-like setups. But trimming the list down to those that are Open Source and free to use, presents a more depressing result. Furthermore, none of these CMS's use a distributed approach. At best they will coexist with load balancers and distributed databases, as long as the interface to these is transparent. When a system is confined to a single host environment, like the one assumed by Blitz, it will not scale well.

In general many of the "finished" Open Source CMS's suffer from a degree of specialisation that makes them unsuitable for general purposes. Systems like PHP-Nuke[16] and its countless offspring exemplify this by saddling the user with a rigid default setup, that presupposes a certain structure and layout of the site.

Ultimately it may be a matter of the developer's preferences. Is a pre-made CMS "good enough" to serve the purpose, then it will undeniably save time in the initial setup. But if the pre-made CMS has to be extensively modified to fit its role, then the developer is probably better off building on top of a CMS framework like Midgard [12].

2.7 Discussion

In this chapter we outlined some of the design considerations for our platform. To sum up, the main points were:

Within our chosen constraints we must rely on software components that can stand alone. We choose to rely on the PEAR software library because it provides stable and modular implementations of several important components that we need such as HTTP communication and caching.

Documents are treated as objects and stored in a DBMS. Object information is split into multiple tables based on a distinction between attributes and properties. The former are primarily system related information, while the latter are developer defined content.

Based our evaluation of the challenges and gains of caching range queries we decide that caching property based range queries is not practical, while caching attribute queries is.

The Blitz CMS will be a framework, rather than a finished CMS product, meaning that the system provides the developer with a data abstraction and a set of basic components that can be expanded.

Chapter 3

The Distributed System

Chapter 2 dealt with the design choices for a single server CMS, choices that would be applicable to any CMS. This chapter will look at the design choices specific to a distributed system. Among other things, the procedure for searching the database must be reevaluated for the distributed environment.

3.1 Why Complicate a Simple Problem by Multiplying It?

In the introduction to the book "Distributed Operating Systems" [54], the author lists five reasons for using distributed systems. All of them are applicable in our case:

- **Economics** – The economical reasons have already been accounted for in chapter 1. Multiple cheap nodes provide an affordable alternative for smaller organisations and businesses.
- **Speed** – Multiple nodes can serve a larger amount of people simultaneously, where a single server must time-slice its resources to achieve the same effect. By increasing the number of servers the load saturation of individual servers is reduced, increasing the speed of request handling. Distribution also increases the amount of bandwidth and CPU power available.
- **Inherent distribution** – When it comes to inherent distribution of problems, they don't come much more distributed than the World Wide Web. By locating nodes geographically close to customers we reduce the amount of network latency, and in some cases the actual cost of the bandwidth[1].
- **Reliability** – By spreading the website across multiple servers, we remove any single point of failure and ensure a higher level of availability.
- **Incremental growth** – Finally, by allowing administrators to add servers at will, we can allow an incremental growth of the business' web presence based on demand.

[1]Most service providers charge by the amount of bandwidth used, and in some cases charge a premium on bandwidth that uses lines crossing national or administrative boundaries.

Given these considerations, it makes good sense to have multiple machines that share a workload.

3.2 The Distributed CMS

When we set out to design Blitz as a distributed CMS, we did not specify precisely what this implied, apart from the obvious: the CMS would occupy more than one machine. It is time to narrow this down to a more precise definition.

As stated, the CMS cluster will consist of a geographically distributed, non-homogeneous [46] group of servers. The amount of nodes is limited only by practical and economical concerns, not by design. All the same, it is reasonable to assume that the amount of nodes will not be more than a dozen servers at most.

Once a server joins the network, it is unlikely to leave the network voluntarily, except temporarily as the result of a breakdown. However, when your stated goal is to work with the lowest bidder, this raises legitimate concerns. In short, we have to expect downtime on individual nodes, both short term as the workload on a node makes it temporarily unresponsive, and for longer periods, when servers break down. Empirically, there is no reason why low cost hosting shouldn't be stable, given that the technology involved is comparatively simple to maintain once properly deployed. The point here is, that it never hurts to plan for the worst case scenario.

Given that we can't rely on a stable platform, our data needs a large degree of redundancy, and no single server can be trusted enough to have a dominant role. In a multiple contingency situation, each node should have an essentially complete version of the website so as to function in a stand-alone manner, until the cluster is rebuilt. Full or real-time replication of data is traditionally expensive, but as we demonstrate in section 3.6.2, this won't be necessary.

The system we propose is essentially a distributed cache. Each node has a full set of CMS code, and as much of the documents as necessary. We assume that the disk capacity of the cluster equals the lowest common denominator. If the smallest amount of disk space available on one node is 500 Mb, then the other nodes cannot exceed this. Bandwidth and CPU power equals the sum of all nodes, minus the synchronisation and load balancing overhead.

3.3 Trust

Preventing servers from being compromised is always a concern. Once a hacker has gained access, he can subvert the node's behaviour, using it to modify data and distribute it to the non corrupted nodes.

The article PS01[50] proposes a model for establishing trust relationships between nodes in a network. This allows network nodes that are not controlled by the same administrative entity to cooperate without fear that one bad node can hijack the cluster.

The nodes build a graph of trust relationships prior to cooperating, based on recommendations. If A has a trusted server B, which trusts another server

C, then an implied trust relation ship exists from A to C. Multiple implied relationships reinforce each other.

This combination of propagation and aggregation forms the function $Trust(x)$. If $Trust(x)$ is above a certain threshold, x is trusted, otherwise it is doubted.

The authors note that their trust model does not handle what they call "nasty" attacks, e.g. attacks where a node pretends to be a good node until everyone trusts it, and then attacks the rest of the cluster without warning. Unfortunately, this is a close approximation of what happens when a node is hijacked, which makes the practical aspects of the model doubtful.

Detecting compromised nodes is a project in itself, and beyond the scope of this thesis. But we can prevent unauthorised users from sending synchronisation messages to the nodes by making sure that nodes only accept data encrypted with certain known keys from known hosts. This prevents a malicious user from injecting instructions into the inter-node synchronisation.

A partial solution to the compromised node problem would be to extend the previously outlined security model (see section 2.4.4) to govern which operations and objects, nodes have access to. While this would not completely prevent damage, it would allow uncompromised nodes to compartmentalise damage from a compromised node. But as noted previously, that is beyond the scope of this thesis. For now our CMS shares the same problems as other web sites. If the server is compromised, the content may be changed. Our web site is simply hosted on more than one server.

3.4 Expanding the Single Server Model

Chapter 2 covered the steps needed to read and write objects to and from storage. When the CMS spans more than one node this is not so simple. Distributed caches and databases often choose to partition data across the nodes, meaning that related data can be grouped where it is most often needed. Doing this, also allows the nodes to combine their storage capacity to form a larger whole, but it also means that one server will not have a usable set of data should it be disconnected from the cluster. As stated in the introduction, our goal is to have servers that are for the most part able to work alone. To facilitate this objects must be replicated across several nodes.

3.5 Reading and Writing Objects

Before we tackle the various possible replication schemes, it is necessary to take a look at the actual process of reading and writing data. Both Coral [39; 4] and GlobeDB[52] use a concept where every document has an authoritative owner that can either solve disputes about content or handle replication. This introduces a single point of failure that we would like to avoid in Blitz.

When a node updates its copy of an object, it has to notify the rest of the cluster in some manner to invalidate any replicas that are now obsolete. When this happens there is a possibility of a race condition. What happens if two nodes create a new object at the same time and assign it the same id?

The model for assigning ids that we outlined in section 2.4.7 is fine for single host systems, but in a distributed environment it won't work. Nodes should have a way of assigning id's that don't overlap without resorting to a distributed locking scheme. One way would be to have a normal incrementing counter on each node. To make a unique id the node could then concatenate the counter with it's node name. The id "n2.23" would be the 23^{rd} object on the node "n2". Another way is to take make an id using the MD5 hash of something random. This gives a very large set of possible keys (2^{128}). Of course this approach is only as good as the random number generator. This is the option chosen in Blitz, as it presents an attractive uniform id name-space.

3.5.1 Locking and Mutual Exclusion

After dealing with the global namespace, another related obstacle presents itself. What happens if two nodes update the same document and distribute it at the same time? The answer is that the same thing happens as when two people edit a file on a network share and save it. This is relatively straight forward because objects are loaded into memory in an atomic database operation, and saved to the local database in the same manner. The person who saves last, overwrites the earlier version. This eliminates some of the problems found in distributed file systems where a program with a pointer to a place in the file is confused by sudden changes to the file.

In some cases this will not be possible, e.g. when the object in the database is only a wrapper for an actual file on the web server. On a desktop OS it is usually possible to open a file in exclusive mode, and denying others write access to it while the file is open. To achieve the same result in a distributed environment would require a semaphore mechanism which is difficult to implement in the limited environment. Since we do not want any single node to have special responsibilities, the node wishing to lock a resource would have to notify every other node that it was locking a document. Even disregarding that one node may not be aware of all other nodes, the whole process suffers from a race condition if two nodes try to lock the same document and start notifying the other nodes in a different order.

To work around this it should be possible to lock the object wrapping the file. This can be done with a local semaphore on the node. If the node then receives an update broadcast that would affect the locked object, it simply ignores it, allowing the process that has acquired the semaphore to finish its work. This allows wrapped files to function as ordinary objects. This form of consistency is sometimes called publish consistency[34] as opposed to sequential consistency where every change to a file is made immediately available.

3.5.2 Time

Keeping a synchronous clock between nodes is a classic problem in distributed systems. Unfortunately, the hardware clocks on each nodes is beyond the CMS' control. So how important is this? When changes to an object are propagated, each object includes a time stamp that tells the receiving node when this version of the object was modified. If two nodes make changes simultaneously it is likely that their individual update broadcasts will arrive at other nodes in a random sequence. This causes a problem if the clocks are skewed on the originating nodes.

Let us say that one node is an hour behind on the clock. If the other nodes discard updates that appear to be outdated, they will not accept any updates to objects issued by the node with the bad clock if another object seems to be newer because of the skewed time stamps.

Most modern hosts use the NTP[45] system to keep their local clocks up to date, meaning that only actual, near simultaneous updates, can cause inconsistencies as described, unless the local clock is off by more than 60 minutes.

In practice, clocks that are off by a few minutes will probably have little influence on the system. The reason is that most updates to an object that occur within a short time are likely to occur on the same host, as a content provider logs in to the administrative interface and makes changes. Since the changes originate from the same node their individual time stamps will be consistent[2]

With a little forethought on the developer's part it should be easy to avoid situations where visitors across the cluster can cause simultaneous changes to the same object. For example, we can imagine tracking changes to a shared object by creating nested objects with a random naming scheme like the identifier scheme listed in section 3.5. This would prevent overlapping changes to the actual object, and prevent accidental name overlap of the nested objects.

3.6 Replication

To present a consistent view of the web site on every node without constantly loading data from one master node, each node must replicate the site's full data set locally eventually. The key here is "eventually". If node A creates an object it should notify the rest of the cluster of the object's existence, but it might not be necessary to send the actual object to every node in the cluster. If the object is changed on the origin node before any visitors request it, the time spent copying an unneeded object to every node would be wasted. To avoid this waste we should replicate data selectively.

3.6.1 Read/Write Ratios

Before making a decision about our content replication strategy, we should look at the expected read/write ratio. To do this we use a major Danish newspaper as an example[18]. Their web site has six sections: inland, foreign affairs, business, IT, social events, and sports. Each section has a menu at the bottom conveniently listing the last 25 articles along with their publishing date. On an ordinary day[3] with no major news stories this log showed that the news paper published an average of 3.4 articles per hour, with the majority of the articles being inland news, and the least amount being related to social events.

Let us then assume that each article is saved twice, once by the journalist, and once by the editor. Each article also has an image that likewise has to be uploaded and saved. Disregarding the facts that images are probably already located in a large stock photo database, and the fact that articles won't need to be replicated before they are marked as "published", we arrive at a rough estimate of 4 saves times 81 articles for a total of 324 write operations in a day.

[2]Barring absurd situations where the hardware clock is going backwards because it is broken, or a server administrator is fiddling with it.

[3]March 1st, 2006

The paper's visitor statistics are confidential, but the paper version is printed in 461.000 copies every day and read by 527.000 people[10]. Even if we triple the estimated number of write operations and assume that only 1% of the people who read the newspaper do so on-line[4], we will still have significantly more read operations, than write operations! Since the newspaper is a good example of a CMS put to use, this should carry over to other sites as well.

3.6.2 Replication Strategies

In section 2.4.5 we discussed how we distinguish between property queries and attribute queries. The reason can be explained now. Since the content of an object is stored in its properties, and these have no upper bound on size, it might take a while to fetch the object when a user requests it. That is why data should be fetched once and stored as long as it is valid. As we discussed in section 2.4.5, the information needed about objects up until the actual page has to be displayed is mostly the Id, and URI information. This is conveniently stored in attributes, and has a known upper bound in size. We also know that each page may depend on the attributes of several other objects and to only to a lesser degree, their properties.

If a node only has the attribute data it can still satisfy most range queries. If a visitor requests an object either by URI or a specific Id, the node can immediately determine if this is an actual object without resorting to a broadcast query. When the visitor requests a valid object that is not available locally, *then* the node can fetch it from the cluster.

We now come to the actual replication strategy. Other work[48; 40] has shown that not all object types have the same need for replication, and that treating all objects in the same manner is inefficient. Instead individual resources should have differing replication strategies, determined dynamically by analysing access patterns. In their implementation, they have a separate process on each node that monitors access patterns. This is not possible in Blitz for reasons already covered.

Instead Blitz would have to make the decision based on what the developer has told the CMS about the object. To keep things simple we will only deal with two kinds of objects. Objects that are important to the inner workings of the system and need to be fully replicated, and user made objects that can be replicated on demand.

Fully replicated objects should include security information, and database scheme information, and information about which objects exist in the cluster, in short: data that needs to be the same on every server at all times. We therefore choose to fully replicate attributes. Whenever an attribute is changed, the entire cluster receives a copy of it. If we replicate all attributes, it would be attractive to store the information about security and database schemes in the attribute part of an object too. Section 2.4.4 gave a sneak peek at how this could be done for security information and as it turns out, database schemes can be handled in the same way, allowing us to have fully replicated objects without adding different replication schemes.

As for the properties of user objects, the replication strategy should be to send objects when they are requested. Once a visitor has forced a node to

[4]This is a very conservative estimate.

replicate an object, it will in all likelihood be valid for a long time. Averaged over the amount of visitors to the amount of writers, the overhead felt by one visitor is amortised over subsequent visitor requests.

To perform replication in a network where nodes only know their peers and not the rest of the network, an object will potentially have to pass through several nodes on its way to the target. Since these nodes receive a copy of the properties before they pass them on, they should take the time to update their own replicas after passing the properties on down the line.

3.7 Failure Detection and Recovery

In section 3.2 we stated that the underlying network should allow for multiple failures. This leaves the question of how to detect a failure and how to recover from it.

In section 3.3 we concluded that detecting compromised nodes was beyond the scope of this thesis. Therefore remaining failures will fall in the following categories:

- Hardware failure.
- Temporary disconnection

Both failure categories have the same symptom. When contacted by a functioning node, they won't respond. Eventually this will cause the connection to time-out. When this happens, the working node has no way of knowing if it is suffering from a hardware failure of its own or if the remote is temporarily disconnected or damaged. In most cases the node itself will be working. After all, it just received a request. We'll ignore the case where the network interface fails right as the node is trying to respond.

The next option is that either the remote is broken, disconnected, or so busy that it can't respond. One way to determine this would be to keep track of round trip times (RTT) for all peers. While this would include a little extra bookkeeping, it would allow us to make an estimate about the remote nodes condition. If the average RTT over time begins to rise the local node should reduce the frequency of its requests to that node. This saves pointless waiting, and has the added benefit of reducing the load on a perhaps already overloaded node. When some time has passed since the communication problems were detected, the local node could resume its attempts to contact the remote.

Looking at the problem from the damaged node, the node is either powered down or not connected to the other nodes. If the cause is a hardware failure there is a chance that its data may have been lost. To allow for this case, we amend our replication strategy from section 3.6.2. When an object is saved it should not only broadcast the object's attributes, it should also send a full copy of the object's properties to it's peers. This ensures that there will always be more than one replica of every object eliminating the single point of failure.

Since a node only "senses" it's surroundings when it receives a request, broken nodes won't know that they have been off-line for a period of time. To deal with this, we propose to let all attributes grow "stale" over a period of time. The attribute begins to go stale from the moment it is updated on the remote node. When it passes a threshold, the node should ask its peers if they have a newer version, and update its own as necessary. If not, the update timer is reset, and the object begins to grow stale again[5]. Only asking the peers will save time, and should solve most cases where a node for some reason has been off-line for a while.

The last mechanism is only to be considered a backup plan. If a node goes off-line the developer in charge of the CMS should take steps to verify the content of the server when it returns to normal operation, preferably by disconnecting the node, and rolling out a clean database from one of the other nodes before reconnecting it.

3.8 Network Topology

As mentioned in section 3.2, a Blitz network is unlikely to be much larger than a dozen nodes. The reason is that at some point the price of web hosting accumulates to a point where it exceeds the developer's economical pain threshold. The implications of this is that a small cluster of machines can be hand configured, so that nodes are organised in a sensible manner. This is why our prototype does not feature self-organising nodes.

When building a cluster the developer should take Blitz' replication strategy into consideration. The two considerations that have to be balanced is the number of peers each node has vs. the maximum distance in intervening nodes from a node to node.

Having many peers increases the write latency, but will in most cases reduce the average read latency, by increasing the chance that a node close by has a replica of an object when it is needed. On the other hand having few peers, and thus presumably longer paths, has the effect of increasing the general level of replication in the cluster because replicas are stored along the query path when fetching replicas.

The optimal configuration ultimately depends on the site's read/write patterns and can be determined experimentally for each site by modifying node connections on the fly.

3.8.1 Self Organising Networks

Hand configuring the network topology becomes impractical after more than a dozen nodes, which is why traditional distributed caches are often self-organising. A distributed cache like Coral[39; 4] has been designed to handle hundreds of nodes. Their current deployment features 225 nodes scattered across the globe.

Coral organises nodes into clusters based on distributed sloppy hash tables. Each nodes belong to several hash tables divided into layers. The layers divide nodes into clusters and super clusters based on the round trip time (RTT) for a message between individual nodes. Nodes with very fast mutual RTT are grouped in low level clusters, which are then grouped with other clusters based on the average RTT between the clusters.

When a new node wants to join the network it must first learn about an existing node. It can then try to join the existing node's cluster. This will only succeed if 90% of the RTT's between the node and the nodes in the cluster it

[5]In times of high load factors, nodes should ignore if the local objects are stale, and only check when the load-level returns to normal.

wants to join, falls below a threshold. If it fails the node creates a new cluster with itself as the sole occupant, and notifies the Coral network of its existence. Later it can join more suitable clusters as it learns about other nodes from the network.

A model for self organisation would be essential for a large deployment of Blitz nodes, but is left as possible further work.

3.9 Range Queries

Section 2.4.5 described the procedure for loading objects. This is fairly simple in a single-node setup, but when we add more nodes we run into trouble. While a node should always have an up-to-date list of attributes, it may not have all the properties for every object, implying that merely searching the local property tables will not always give a complete result. If our search needs to look at properties, it therefore follows that a search query cannot always be handled exclusively at the local node. Broadcasting queries to the cluster and aggregating results is very slow compared to a purely local database search. To overcome this we should look at ways of reducing the amount of property queries.

As discussed in section 2.4.6, caching property queries is unlikely to provide a performance boost in it self, but in some cases it might be possible to avoid a network spanning search. Consider the case where a previous property query generated a list of all objects with the property "class = Book & author = Herman Melville", and then subsequently loaded all these objects, causing them to be locally replicated. A subsequent search might not need to look at other nodes, if we could establish that all objects of the class "Book" had a local replica.

3.10 Load Distribution

To take advantage of the distributed structure of Blitz, traffic needs to be distributed fairly across nodes. Fairly does not necessarily imply evenly, since not all nodes are created equal. The prototype described in chapter 4 does not implement load balancing, but the final system should.

3.10.1 Flash Crowds

A good load balancer becomes especially important in the advent of a sudden spike in traffic. This phenomenon is often referred to as the "Slashdot effect"[23], named after a popular web portal. It happens when a link is posted on the portal, leading readers of the portal to flood into the linked site within a very short period of time after the link is posted. FKW04[37] documents such a case. Within minutes of the link being posted on Slashdot, the server server went from less than 20 requests per second to over a hundred requests per second. The spike lasted about an hour with a peak of 215 requests per second. Traffic levels like this will act as a "Denial of Service" attack on an unprepared web server, as memory, disk, and CPU capacity are completely saturated over a short period of time. In some cases the server will even crash. The article notes some further interesting patterns, including a smaller "warning" spike before the main spike, and a series of aftershocks following the main attack. The pre-spike is caused by the link being posted to less popular but trend setting portals. The main spike comes when someone posts the link to Slashdot, and the subsequent aftershocks result from people who are not frequent Slashdot readers, or live in different time zones and read the article after the main rush.

3.10.2 Load Detection

Based on their findings, Felber et al. propose a novel method of predicting the hot-spots by looking for the pre-spikes. They use an algorithm that measures the average interval between requests to look for signs of a coming spike. When a spike is detected, the affected data is replicated to backup servers and a load balancing system is started. The system stays active until the average request interval drops below a certain threshold again.

3.10.3 Implementation Challenges

The system in FKW04[37] uses shared memory to store interval data. They note that even using shared memory instead of disk access, this presents a significant overhead. As mentioned in section 2.2.2, Blitz cannot rely on shared memory being available. In fact, measuring server load without actively contributing to it is going to be a problem. Neither Apache nor MySQL exposes information about server load, and even something as basic as reading the server load from standard Linux `/proc/loadavg` interface will be impossible in a sandbox environment.

The problem is exacerbated by the fact that visitors are likely to favour some nodes over others on their initial visit. If one node is named `http://www.freespeech.cn` while other nodes have names like `http://node6.freespeech.cn`, the first server is likely to receive the most traffic.

There are three simple ways to approach the problem, but none of them even come close to an optimal solution. The first option is to dedicate one node as the load balancer. Picking the node with the best hardware/bandwidth configuration, and only serving the redirection script from it should outweigh some of the costs of keeping statistics on slow disks. Depending on Apache's file handling, the files may even be mapped to memory on after frequent use.

Another, more subtle solution, would be to change the way Blitz generates URI's. Instead of returning a relative URI (`/Frontpage/Article`), Blitz could return an absolute path to an object, but with the server part being chosen by random (`http://node4.freespeech.cn/Frontpage/Article`). The random server choice can be weighted to account for some servers being more powerful than others. While this method is less wasteful than a dedicated load balancer, it is far from perfect. Furthermore the strange inter-linking is likely to confuse some search engines[6].

As it stands, the best solution is to use a round robin DNS[21] solution where the same CNAME corresponds to multiple A records, with a short TTL. This would be possible without breaking the platform constraints, but is not as good as a dedicated locality aware load balancer. Since redirection happens transparently to both the client and the CMS when a DNS lookup times out,

[6]Actually it might also help, as the inter-linking could function as a form of "Google-bombing"[9].

the system would have to store all user session data in a fully replicated manner to ensure that user interaction did not suffer when the client was redirected.

Finding a good way to balance the load fairly must fall in the category of further work, as the above solutions are not satisfactory in the long run.

3.10.4 LARD

LARD stands for Locality Aware Resource Distribution [47; 30]. Locality awareness covers a number of concepts, but when used in conjunction with th routing of web page requests, it means redirecting requests to a location where they are likely to be handled in the fastest possible way. This in turn is combination of load balancing to avoid load saturation on individual servers, and keeping track of which server received similar requests in the past. The latter means redirecting similar requests to the same server if possible, to maximise the efficiency of caching on the host. Incorporating LARD into Blitz would face the same problems as those described for load balancing, but could give a significant boost in performance. Even if local nodes have replicas of every document, there is still a performance gain from redirecting to nodes that have recently served the object, as the underlying DBMS is likely to have cached the queries performed by the document.

3.10.5 Locality Prediction

Another form of locality awareness is related to detecting the geographical location of a visitor and redirecting him to the nearest possible server. Some use route tracing[39; 44] to pick a node that has the shortest route to the client. Others like Kademlia[43] attempt to discern location by comparing the IP address of the visitor with the IP addresses of the nodes. If load balancing and LARD were implemented, it should be simple to use Kademlia's strategy for the initial choice of a server to handle the visitor.

3.11 Related Work

There are several examples of Open Source initiatives to make distributed databases. But working in a WAN environment poses several challenges that don't exist in a secure LAN environment. The following systems are designed to account for the challenges of a WAN network.

3.11.1 Globule

Globule is a distributed system for improving quality-of-service (QoS) by caching and replication. Unlike dedicated Content Distribution Networks (CDN) such as Akamai[1] it does not focus on streaming data, but aims to handle all web content. It is based on Globe [55], a platform for large distributed objects.

As mentioned in section 2.4.1, Globule wraps all resources in objects that have the methods to replicate themselves between participating servers. The replication strategies are determined during runtime based on access patterns.

The Globule nodes exchange encapsulating document objects along with the physical files. Globule nodes "lease" storage space to their peers, and can host several independent sites by sharing their resources. Despite allocating space to other peers, the originator of the object retains control of it. Thus the Globule setup is not fully distributed. When a client requests an object, it is redirected by the intervening nodes to the nearest replica of the object using HTTP redirection and DNS redirection using a custom DNS system that assigns short TTL's to responses[51], making sure that one server does not receive all incoming requests for extended periods.

Globule proposes to handle dynamic pages by encapsulating both the code (PHP,ASP, etc.) and the data needed by the code (from the database). To do this, the system must be able to analyse dynamic documents to detect which data is needed by the code. Dynamic pages that can't be analysed are always served by the master site. This seems difficult to handle efficiently in reality, and the authors do not go into detail on how they plan to accomplish it.

Like Blitz, server nodes are not "fully connected" but know only a number of peers, and like Blitz, messages are propagated through the network from peer to peer so that all nodes receive the message eventually. Since servers can be owned and operated by different administrative entities, servers use a trust based system to detect defective or malignant nodes (see section 3.3). Objects containing dynamic code are also run in a sandbox environment to prevent damage from malignant code.

Each node must "lease" space on other servers for its replicated data. This allows servers to apply a cost to each operation.

3.11.2 GlobeDB

GlobeDB is a DBMS developed specifically as a back-end for web based content managements systems. It is built on top of the PostgreSQL[19] DBMS. It is implemented as a database driver for PHP and provides a completely transparent interface to the programmer. Building on earlier work like Globule, it stores information about database access patterns on participating nodes and replicates records to the nodes where they are requested. Database records that are regularly used in conjunction are clustered together for efficiency.

As mentioned, concurrency control is tricky in a distributed database without using global locks that degrade performance. GlobeDB has chosen to handle this by relaxing the consistency constraints of a traditional database to what they refer to as "eventual consistency". Read operations use PostgreSQL's snapshot system to prevent reading inconsistent data on the individual nodes and the replication system ensures that newer versions of the stored records eventually percolate to to every node where they are needed. To handle timing issues with simultaneous updates from several nodes, all records have a master server, which is responsible for keeping track of replicas, and handling updates. When an update occurs, the master pushes the change to all known replicas of the record.

To balance the tradeoff between replicating objects to more servers at the cost of higher write latency vs. less read latency, GlobeDB uses a cost function that allows the administrator to prioritise various aspects of replication according to their needs. The cost function can assign importance to "read latency", "write latency" and "bandwidth usage".

GlobeDB does not account for network or master failures, but leaves it as an exercise for the prospective programmer.

3.11.3 Distributed Databases

Where GlobeDB mainly focuses on distributing the data to where it is needed, other systems put the focus on distributing the work. Mariposa[53] uses a radically different approach. Where GlobeDB uses a somewhat socialist approach, sharing resources data and resources between all nodes, Mariposa is decidedly capitalist in its approach. Nodes use a complicated economic model where they advertise their knowledge of parts of the database and sell their power to perform queries in exchange for credit that can be used to perform queries on other hosts or store data remotely. This approach favours computing networks where many different administrative entities share their computing power, but do not necessarily agree on a common purpose.

Perhaps the most common Distributed DBMS is the Internet's DNS service, that provides a common way to translate names into IP addresses. Other examples of distributed databases include peer to peer networks (Gnutella, ED2K, Kademlia, etc.) that use various forms of distributed databases to share blocks of data. The systems used are often very advanced but domain specific, and restricted to storing data hashes. They are therefore ill suited for our purposes.

3.12 Discussion

In this chapter we have extended the single server outline from the previous chapter by amending the naming scheme for objects to use non-consecutive, unique keys, avoiding the necessity for a centralised key-server.

The initial design has evolved into a distributed cache where objects are replicated among nodes to avoid data loss in case of server failure, and to ensure that an object can be reached on an alternative node if other nodes are too busy to respond.

The proposed replication scheme will fully replicate object attributes. Properties are only partially replicated initially, and then replicated on demand later. This also leads to some changes in the way range queries across multiple nodes in the cluster are handled, since queries depending on properties can no longer be satisfied locally in all cases.

Some design areas that were covered will not be part of the prototype in the next chapter. Failure detection and recovery is not handled, and our only concession to security is to encrypt all inter-node communication with a two-key encryption scheme. Since failure detection and recovery will be not provided for in the prototype, there is no need to have objects grow "stale" over time either.

By looking at the expected update patterns we can also take a relaxed attitude to some timing, locking and mutual exclusion issues. Only the bare necessities will be provided by the prototype.

Finally the issues of load balancing and a self-organising network topology was covered. However, no satisfactory solution was found, mainly due to the restrictions of our chosen environment.

Chapter 4

Blitz

Building on previous chaptere, we can now describe the prototype.

The prototype was implemented in pure PHP using an Object Oriented approach. All classes, functions and global variables have the prefix *BLZ*[1]. To the best of our knowledge their are no other widely distributed PHP packages with this namespace. It should be noted that namespace does not refer to namespaces in a C++ sense, since PHP versions below version 5 do not support this concept. Instead it is common practice to emulate namespaces by adding a prefix to all variable, class, and function names.

Blitz makes extensive use of PEAR's[15] error handling, cache, and HTTP request packages. The latter package was modified to allow non-blocking requests.

Blitz also Alex Poole's OpenSSL[17] Open Source wrapper class as an interface to PHP's built-in OpenSSL support.

4.1 Objects

The core of Blitz is the *BLZObject* class. This class provides methods for serialisation and deserialisation, as well as deletion. Object propagation and replication is controlled by *BLZObject* but the actual implementation is delegated to the *BLZClient* class.

Each object in Blitz is an instance of an object derived from *BLZObject*. Derived classes can affect the serialisation processes by overriding the main methods (*save*, *loadId*, and *delete*), as long as they remember to call the parent version as well.

Objects can be loaded either based on Id or on a URI. when loading the object by URI with the *BLZLoadURI()* function, the object loader has to unravel the URI to determine object's id. This causes some database activity on the local node, but once the Id has been found the result is cached.

If the Id of an object is known in advance, objects can be loaded either with *BLZLoadId()* or *BLZLoadIdAttributes()*. The latter function disables propagation meaning that missing replicas are not fetched. This is useful if the developer knows in advance that all the information needed is kept in the object's attributes.

[1]Unfortunately the BZ namespace was already used by the Bzip2 extension

Apart from caching URI to Id relations, objects are not cached them selves. This has several reasons. For one, the actual load procedure is very simple, and it is doubtful if serialising the object to a cache file would make for faster load times. The second reason is that objects will probably cached by the developer once the object's view has rendered the object.

4.1.1 Atomicity

It was said in section 3.5.1 that load and save operations are atomic. This requires a little more explanation. Both operations take place in two parts. First the attributes are processed then the properties. When an object is saved, one of the first things that happens is that the new version is assigned a version vector in the form of a modification date specified by the *mdate* and *usec* fields. Since no other object will have the same version vector, subsequent read and write oprations to the properties are safe. There is no guarantee that a set of properties will not be superceded before they are serialised by another process marking the attending attributes as deleted, but that is the rules of the game. Each version of an object will be consistent.

4.2 Object Lists

Queries to the database are wrapped in the *BLZObjectList* class which also handles the cases where the search query has to be sent to other nodes to get a result.

To search the database the developer creates an *BLZObjectList* instance. by default a blank object list contains *all* objects in the database. To narrow the list down, the developer adds conditions with the *attribute*- and *property* methods. Conditions can be any valid MySQL comparison.

Based on the conditions the object list object can deduce the minimum number of SQL queries necessary to get a result. If the developer has set a property condition "`text LIKE '%Moby%'`" it means that the list must search all property tables for classes with a property called *text*. The query space can be narrowed further by adding more conditions or by specifying a specific class to be search, either with an attribute condition or by specifying it in the property condition (eg: `book.text LIKE '%Moby%'` narrows the possible classes to the "book" class). When the developer is satisfied with his query, he can retrieve an array of results with the *getResult()* method. If the object list can tell in advance that the query has conflicting conditions the result is automatically an empty array. Otherwise the minimum amount of SQL queries are compiled and executed.

Complex queries can be built by combining the results from multiple queries. Query results are always returned in an array of attributes. Currently the prototype only has placeholders for ORDER, OFFSET, and LIMIT instructions. How range queries are handled is covered in section 4.5.3.

4.3 Client and Server

Inter-node communication is handled by the classes *BLZClientBase*, *BLZServer-Base*, and the *BLZServeRequests()* function.

When a node wishes to initiate communication it calls one of the static methods in a class derived from *BLZClientBase*. Each method matches a "listener" in the server part.

broadcastUpdateAttributes() – Sends a new attribute-record to all nodes (see section 4.6.1).

sendUpdateProperties() – Sends an object replica to all peers (see section 4.6.2).

broadcastDelete() – Marks an object deleted on all nodes in the cluster (see section 4.6.3).

broadcastFetchReplica() – Used when a node needs a replica for an object (see section 4.6.4).

broadcastRangeQuery() – Broadcasts a property query to all nodes (see section 4.5.3).

When the client makes a call to another node it formats the message as a HTTP request with a few extra headers, and adds an SSL encoded version of the query as a POST variable. Details on the encoding scheme is covered in section 4.5.

All nodes include a call to *BLZServeRequests()* in their dispatch scripts. When an incoming connection is processed, *BLZServeRequests()* looks at the request before the rest of the dispatcher is allowed to see it. If the request comes from another Blitz node, the function creates an instance of the *BLZServer* class and passes the request to the new server instance.

The server instance decodes the message "payload". If the payload can be decoded and verified, the request is processed. Some requests spawn requests to other nodes. If the requests are blocking the server stalls until an answer can be returned or until the request times out. Requests time out after 60 seconds in the prototype, and it does not keep track of round trip times.

When the request is processed, the server returns the answer without passing control back to the dispatcher.

The process is described in greater detail in Appendix B.

4.4 Miscellaneous

Blitz uses a variety of support functions that are not part of the classes. One of the most commonly used functions is *BLZConfig()*. It is used to wrap all global variables used by Blitz in a shared namespace. By wrapping globals we avoid having other packages accidentally overwriting internal state variables.

4.5 Communication

All communication among nodes is based on the common principle of the broadcast. Broadcasts are either "deep" or "shallow". A shallow broadcast only reaches the instigator's peers nodes. A deep broadcast can be viewed as a series of shallow broadcasts. When a node receives a deep broadcast it forwards the request to its peers, then processes it. When the request is processed on the node it waits for its peers to respond before aggregating their responses and returning them to the node that instigated the process.

Broadcasts are not atomic in the sense that they do *not* guarantee that either everyone gets the message or no one.

4.5.1 Tokens

Broadcasts need to have a stop condition, or they will simply keep multiplying in the system until all are swamped by broadcast messages. Blitz uses a tokens to control the flow of broadcasts.

BLZToken objects contain a semaphore mechanism that allows an object to acquire the token. Tokens are stored in the nodes database and can be in one of three states: *unprocessed*, *processing*, and *processed*. Only unprocessed tokens can be acquired. When acquired, the token changes status to *processing*, and when it is released it changes to *processed*.

When a node initiates a request it creates and acquires a token. This token is passed along with the request to other nodes. When the target nodes receive the request they immediately try to acquire the token. If they fail, they return an error to the instigating node. If the token is successfully acquired, the node will process the request and forward the request with the token if required. When the node has processed the request and received any outstanding replies from other nodes, it releases the token and returns its own result aggregated with any results from peers.

The semaphore mechanism in *BLZToken* uses PHP's built-in semaphore mechanism if it is available[2]. If it isn't, Blitz emulates it using PHP's *flock* mechanism to create a file semaphore in the site's cache directory.

4.5.2 Encryption

To provide a level of inter node security, and to help verify the identity of other nodes, Blitz uses SSL certificates based on two-key encryption. SSL support is standard in newer PHP versions and therefore likely to be available. Each node has a private key and a public key, as well as a list of its peers' public keys. Since all messages must be encrypted with a public key, nodes cannot communicate with other nodes unless they have this key. It is possible to disable encryption, this is not recommended.

4.5.3 Range Queries

Propagated range queries happen when a node tries to make a list of objects that would depend on the contents of an object property. When this happens the node sends the query's conditions to its peers and aggregates their answers with its own results. To prevent infinite query loops, nodes along the range query's path will disable their own propagation primitives while handling the search. This means that each node queries will only look at its own properties instead of spawning a parallel range query.

Testing proved that querying all nodes took a prohibitive amount of time. To optimise the process, the developer can define a constant, `local_search_mrr`.

[2]Semaphores are not available on servers running in PHP's safe-mode.

This controls when Blitz will propagate the query, and when it will be handled locally. Property queries are only propagated if the ratio between total possible candidates and local replicas is below `local_search_mrr`. Possible candidates are are deduced from the local attributes, and the replication ratio is calculated by dividing the total number with the number of candidates that are locally replicated. By default Blitz will propagate all queries unless the ratio is greater than or equal to the Minimum Replication Ratio.

In an actual system all objects will be replicated after a while, meaning that a significant speed up is achieved by only propagating queries that are known to depend on external input.

4.6 Object Propagation

Object propagation is handled by a series of matched function handlers, as mentioned in section 4.3. These functions handle the work of updating objects, deleting them, and replicating them.

4.6.1 Attributes

Attribute updates happen when one node makes changes to its version of an object. It is only possible for a node without a replica to make changes to the object, if the object does not normally have any properties[3].

The attribute update is a broadcast to all to all peers, which in turn propagate the broadcast to their peers. Nodes only accept versions that are newer than their own version. In theory the attribute change should be non blocking since the instigator does not need to hear any responses, but in practice this quickly causes race conditions to occur, where one set of attributes depend upon another. To give an example. When a new class is defined by creating a *BLZClass* object, it can be assigned properties as part of the call to the class' constructor. This means that the new *BLZClass* object is broadcast almost simultaneously with the new *BLZProperty* objects, nearly always causing a race condition where the remote node does not have a class property table ready when the property try to add themselves to the class definition. The Blitz prototype handles this poorly, by simply making the attribute update a blocking call.

4.6.2 Properties

When an object is created, or an existing object is modified, its *replica* attribute is set to 1. Then its new attributes are broadcast to all nodes as in the previous section, however the broadcasted attribute-record has a *replica* flag set to 0. This means that even if remote nodes had a replica before, it becomes invalid because it belongs to previous version. After this broadcast, the updating node sends a replica to its peers in the form of a property-record. When a peer has successfully received and decoded the properties, it changes its replica flag to 1 to show that it now has a valid replica.

[3]Objects without properties are explained in section 3.6.2

4.6.3 Object Deletion

To allow the content providers to undo changes, no object attribute set is ever deleted, as long as its class and property scheme remains unchanged. Instead objects that become obsolete, or are deleted by users, are marked as deleted. Over time deleted objects may pile up and some sort of garbage collection strategy might be in order. The only instance where data is lost, is if the developer changes a class by removing a property or the entire class. In the first case all data located in the deleted property is purged. If the class itself is deleted, all objects of that type are purged as well.

When an object is deleted, the instigating node broadcasts a deletion message to the network, and other nodes then also mark the object as deleted. Ideally cached data relying on the deleted item should be invalidated, but this is difficult to implement for the reasons discussed in chapter 2.

4.6.4 Replica Loading

When a node tries to load an object using the *loadId* or *loadURI* methods that is not locally replicated, it sends out a *broadcastFetchReplica()* call to its peers. The request traverses the cluster until one node returns an object and stops forwarding the request. Each node along the query path will store a the replica as well, if the retrieved replica supercedes its own. The node that initiated the replica search will wait for all peers to return, and then pick the newest replica.

4.7 Physical Files

Some files such as large multimedia files, are impractical to store in a database. To facilitate these, Blitz provides the *BLZFile*. This class breaks a number of the “rules” in Blitz’ replication scheme and should be treated with care. *BLZFile* instances create a shadow object that tracks the file. The actual file is not replicated with the object since copying a 300 Mb WMV movie as a side effect of saving a change to it’s URN is a little harsh. This means that physical files are an exception to the rule that all objects have replicas on at least two nodes.

Instead, files are only replicated on demand. When a visitor begins downloading the file that is not locally replicated, the node begins streaming it to the user while simultaneously writing blocks of it to disk on the local node. Files have a property that contains the URI of the file on the originating node. If the file is fully downloaded to the local node, the local node changes the property on it’s replica so that the URI points to its own version, without changing the versioning vector for the object. This means that other nodes that get the replica will try to fetch the physical file from the node they got the shadow object from. This breaks the rule that says that nodes will only communicate using secure channels with known peers, and presents a potential security issue. It also breaks the atomic update mechanism described in section 4.1.1 which is *bad*. In short, file handling needs more work to be safe.

4.8 Discussion

This chapter only outlines the most important implementation details. With all projects there comes a time, when development must be frozen so the project can move on to testing and release. The Blitz prototype is no different, and certain features did not make the final cut. As it stands, message encoding sacrifices efficiency for a generic solution. While this was good in the beginning of the development cycle, it is no longer efficient. The cycle of double serialisation, encryption, and compression to send messages, and its corresponding reverse at the receiving end is too slow.

On the same note, the current method of updating attributes is cumbersome, and one of the things slows down the whole system. Since updates are blocking, a lot of time is wasted waiting for what should be a “fire-and-forget” call, while the attribute update traverses every node. If we took more note of failure detection and recovery, it might be reasonable to at least check if the attribute update was received, but in the current setup it could be avoided. The reason for the blocking method, has already been mentioned in section 4.6.1. A solution could be to allow a developer to group selected updates into sequences. Each sequenced message should then have a sequence number indicating its position in realtion to previous packages[4]. On receipt, sequenced messages should be stacked, and executed in the correct order as soon as possible. All non-sequenced messages could now be non-blocking. This however, remains a project for later versions.

The current prototype filters out obsolete objects by default, but can list them if needed. On the other hand, the object loading routines don’t support loading objects that are not the current version. This is a minor concern at best, but does prevent the developer rolling back changes without resorting to direct database administration. The feature was left out intentionally along with a user friendly editor interface, and a security model due to time constraints.

As noted, the prototype does not account for node failures, track round trip times between nodes, or change the default time-outs on HTTP requests. Neither does it track the staleness of attributes, since attributes are always up-to-date in the eternal sunshine of the failure free cluster.

With these caveats in mind we can now test the prototype.

[4]Not unlike TCP/IP packages.

Chapter 5

Test results

To test the prototype, a cluster with six nodes was constructed. The nodes are located geographically close[1], so network latency is generally below 10ms between nodes and will not play any part in the tests. It is possible to simulate greater distances by adding a short sleep time to the script that processes incoming requests from other nodes, but this would only obscure the test results further.

Each node has a slightly different setup. Software versions are summarised in table 5.1[2]. Node 1, 2, and 3 are hosted with low cost hosting providers, and there is no way of knowing exactly what hardware they are located on. Node 4 and 5 are high-end production servers[3]. Node 6 is not. All servers are multi-processor servers.

The information in the table was extracted using PHP's `phpinfo()` function. From the extended kernel version information we can see that the machines all use an SMP enabled kernel, so presumably they all have multiple processors. The actual software versions are not important, but it should be noted that node 6 was not a originally intended as a web server, and therefore did not not have a very efficient setup.

Server	Kernel	Apache	MySQL	PHP	Safe Mode	OpenSSL
www	2.6.12.5	?	4.0.24	4.4.1	On	0.9.7e
node2	2.6.12.6	?	4.0.24	4.4.1	On	0.9.7e
node3	2.6.13.4	1.3.24	4.1.15	4.4.2	On	0.9.6b
node4	2.6.8	1.3.34-1	4.0.23	4.3.10-15	Off	0.9.7e
node5	2.6.8	1.3.33	4.1.11	4.4.0-4	Off	0.9.8a
node6	2.4.16	1.3.33	3.23.49-3	4.3.10-16	Off	0.9.7e

Table 5.1: Software Versions

Network overhead is related to several factors. One of these is how well connected the nodes are. The more peers a node has the longer some operations will take. On the other hand the longest path a for a broadcast message would presumably be smaller if each node is well connected.

[1]In the same city.

[2]Node names have been changed to protect the innocent.

[3]Graciously provided by Brandhouse a/s.

We use three setups for our tests. One where each node has 2 peers (fig. 5.1), one where each has 3 peers (fig. 5.2), and one where each has 4 peers (fig. 5.3).

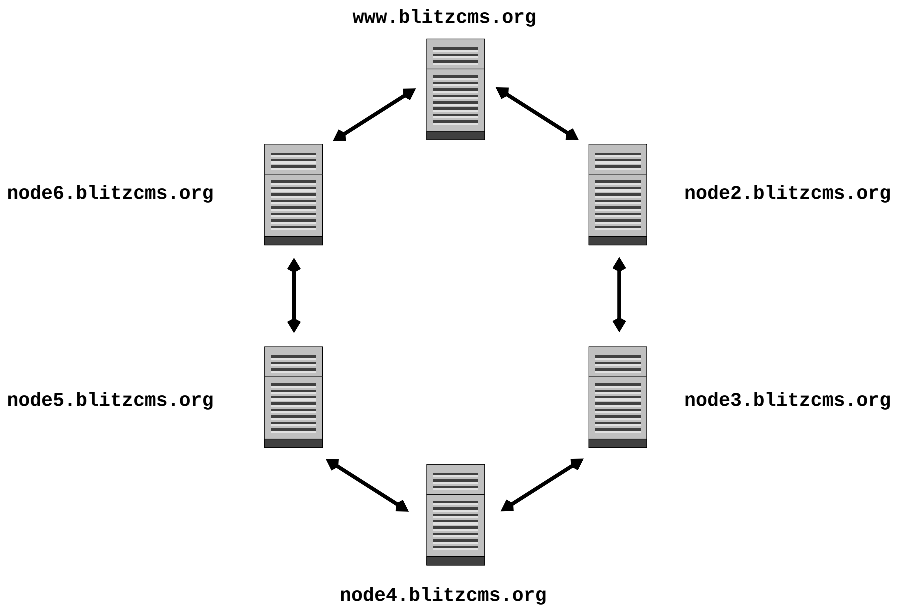

Figure 5.1: Setup A

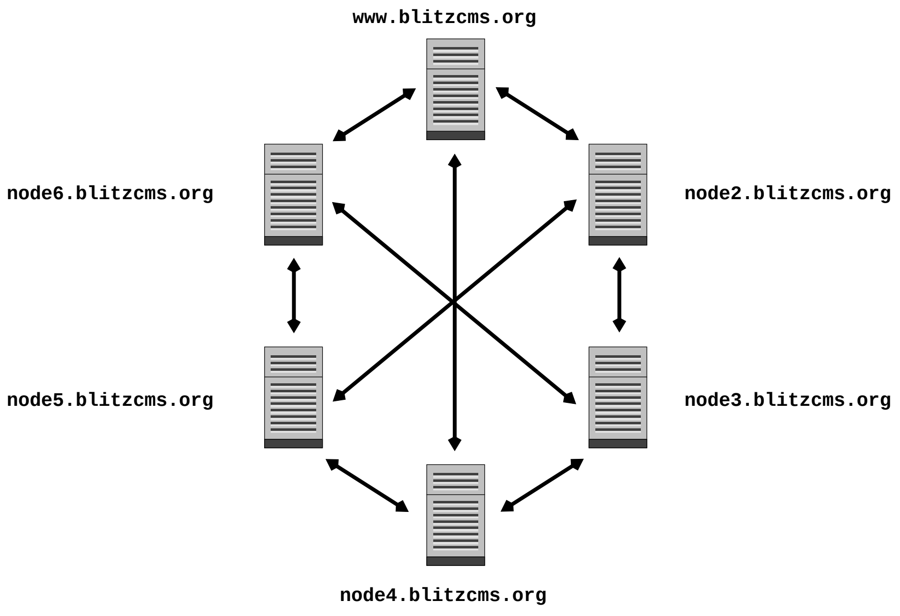

Figure 5.2: Setup B

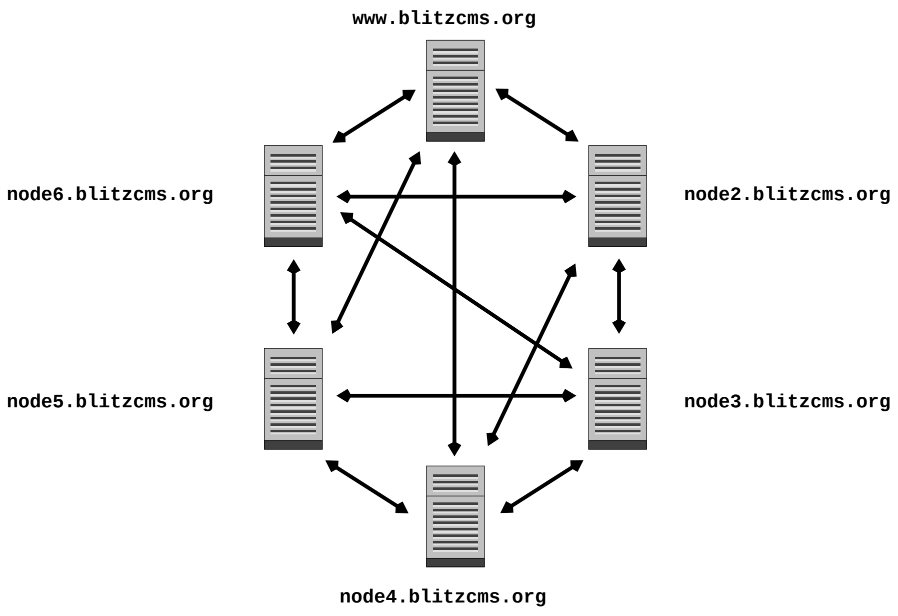

Figure 5.3: Setup C

5.1 Replication

The first test involves the core of Blitz – the replication system that keeps the cluster's nodes synchronised.

During normal operation, replication happens only after an object is modified. When this happens replicas are sent to all peers, and subsequently, to the remaining nodes on demand. Assuming new nodes are "boot-strapped" with a full set of replicas, actual replication and the associated server overhead is spaced out over time. To get an idea of how long it would take to "cold start" the system, each node was seeded with a database containing the full attribute set, but only node 1 was given replicas of the objects.

The test web site imitated an online newspaper. The data set consisted of two different object types: Article and Image, with 500 of each. Articles were placed in a randomly generated and deeply nested structure. Each Article had exactly one nested Image in addition to any nested Articles. All Articles consisted of randomly generated HTML data ranging in size from between 3.5 Kb to 6 Kb. Images consisted of a small "shadow" object based on the *BLZFile* class with an attached image file, averaging 50 Kb in total size.

In a Cold Start, replication only happens when visitors try to access objects that aren't replicated. When all nodes were fully traversed simultaneously from multiple clients[4], the resulting overhead of inter-node communication quickly saturated the slower nodes causing the whole cluster to slow down. Table 5.2 shows the final time for each setup. The more inter-node connections, the slower the Cold Start.

Figure 5.4 shows the amount of inter-node communication. *Queries* repre-

[4]This was done using GNU Wget[8] from three clients with individual bandwidth in excess of 10 Mbit.

Setup	Minutes
A	33
B	51
C	77

Table 5.2: "Cold Start" – Full Replication Time

sent the total number of messages sent from one node to another, while *Replies* are the number of responses received. The reason for the discrepancy between the two numbers is that nodes will ignore duplicate messages which become more common as the number of node connections increase.

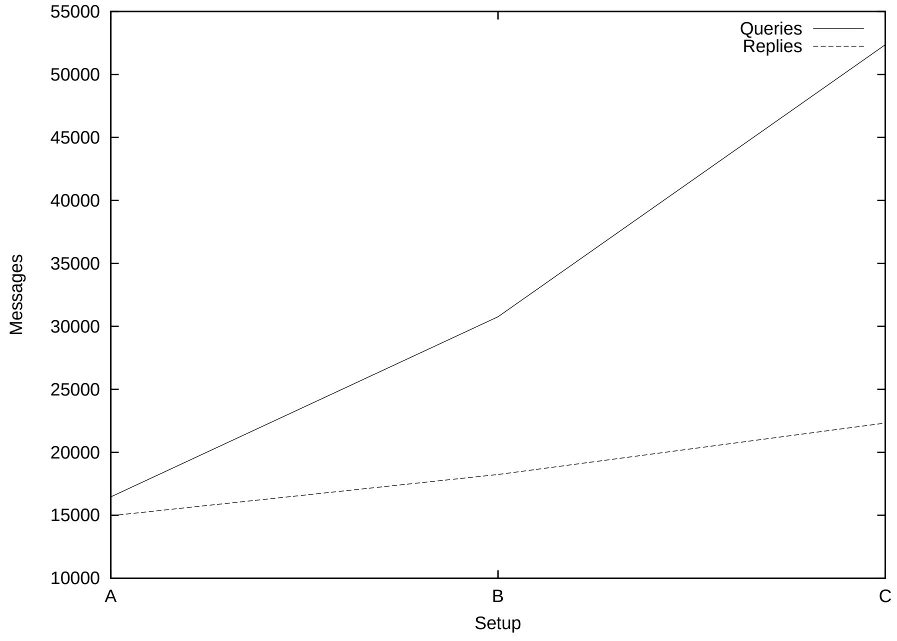

Figure 5.4: "Cold Start" – Full Replication Overhead

Individual replication times fluctuated wildly depending on server saturation, and distance to the nearest replica. The Blitz prototype currently waits for the "best" replica, meaning that more node connections cause higher wait times. Table 5.3 shows the actual delays. The average time is skewed by the very long wait times caused by server saturation. The actual maximum wait time seems more related to server load, than to how inter-connected the nodes are. The actual average times show a clear progression.

In hindsight, it seems wasteful to have a node wait for the best result, when the first result should also be the best result, which holds true unless the cluster contains multiple different replicas of the same object, all claiming to be the current version.

Setup	Average	Min	Max
A	**1.98**	0.0009	7.62
B	**3.06**	0.0017	16.33
C	**4.62**	0.0030	12.05

Table 5.3: Time spent Retrieving a Replica (seconds)

5.2 Read/Write

Figure 5.5 summarises read/write performance. Operations were executed on node 1, and the response times were logged. *Create* is the time taken to make an object, replicate it on peers, and propagate the attributes to the rest of the cluster. *Update* is the time taken when only attributes are updated. The *Create* value for setup C is an anomaly that can be explained by a temporary drop in overall server load on the test node during the test cycle. The times for *Delete* also seem skewed, this time on the value for setup B. The important observation is that operations that require a node to inform the other nodes take longer when there are more peers. Loading a local object is nearly instantaneous, which accounts for the flat line in graph two.

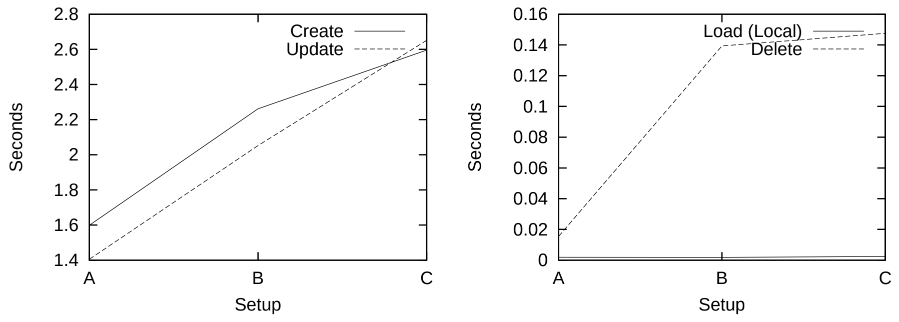

Figure 5.5: Read/Write Performance

5.3 Range Queries

Range queries come in two flavours. The first is attribute and property queries that can be handled locally. They average to 0.029 seconds, but the actual query times fluctuate between 0.0163 and 0.0441 seconds. Local property queries are those where the local node can deduce in advance that it already has enough local replicas to perform the query (see section 4.5.3).

Distributed property queries happen when the local node might not have enough replicas to give a satisfactory answer. Response times for these can only be labelled as abysmal (fig. 5.6). Query time actually increases with the number of peers, for the same reasons as those listed under section 5.1. The main culprit in the case of the long query times can probably be found in the way Blitz encodes messages. This was discussed in section 4.8. In short the cumbersome encoding will be especially punishing for large results, in part due to PHP's slow serialisation method.

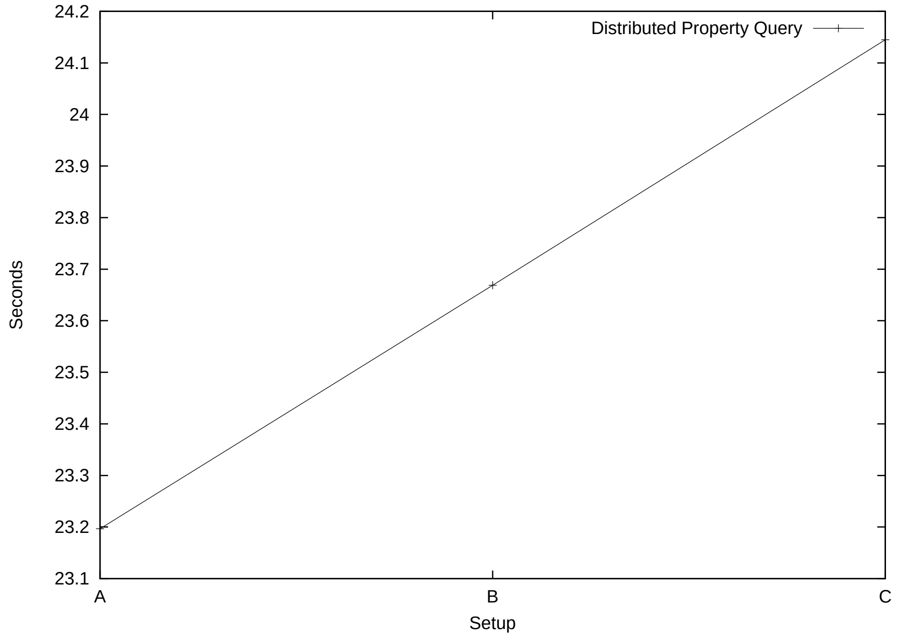

Figure 5.6: Property Queries

The numbers clearly show that a domain specific cache strategy is needed, in the cases where it is not enough to set `local_search_mrr` to a value below 1.0.

5.4 Throughput

In the introduction to the thesis it was noted that a static solution would always out-perform a dynamic solution, and as discussed in chapter 2 and chapter 3, part of the solution is to reduce a dynamic site to mainly static components by caching selected parts.

Figure 5.7 shows how a minimal dispatch script (see Appendix C) performs against a static HTML file with the same output. The results were produced using Apache Bench[3]. Given the two files, node 1 was able to serve the static version roughly twice as fast as the dynamic version, with performance declining at high amounts of concurrent requests.

On *average*, performance for a more sophisticated site will not be far from the test dispatcher, if rendered HTML is cached and reused.

5.5 Discussion

Testing the system in an actual low cost hosting environment inadvertently proved many of our negative assumptions about node stability in the shared hosting environment. Test results are averaged over 50 to 100 samplings. This

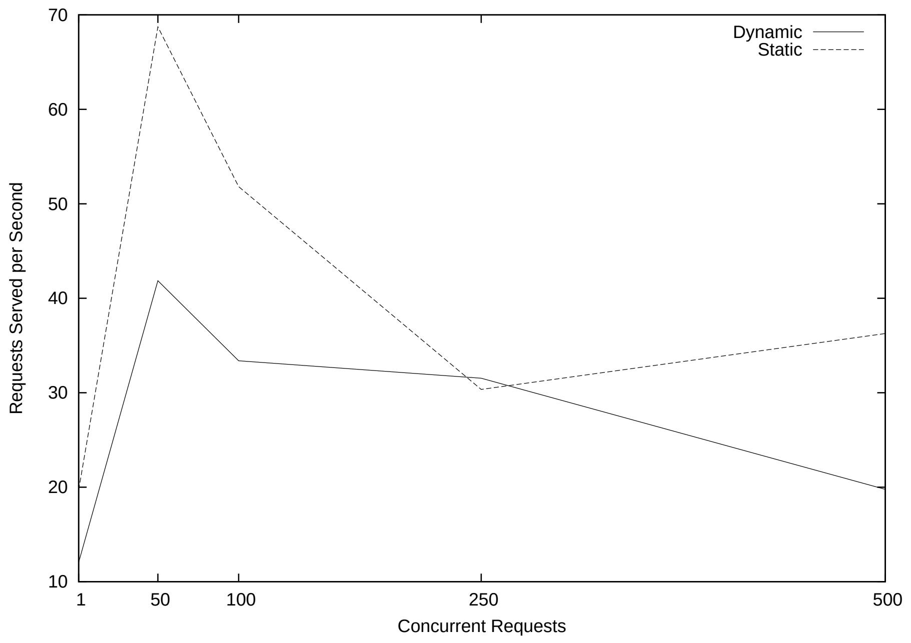

Figure 5.7: Throughput Comparison of Dynamic vs. Static Content

was necessary because individual node performance fluctuated significantly with no apparent pattern, even when test cycles were performed at a times that are traditionally low on traffic[5].

Despite this, the test results give broad hint, not only of the troubles of using low cost servers, but also of the possible performance bottle-necks. Distributed searches are too slow for practical use, and their use needs to be finely regulated by the developer. Furthermore, the result of using non-blocking updates (see section 4.6.1) effectively erodes what should have been the benefits of a highly connected cluster.

The end result is that the developer needs to be aware of the inner workings of the CMS to use it efficiently, which was not the original intent.

[5]Websites often have a traffic increase around lunchtime during business hours, and another one before and after dinner time in the evening.

Chapter 6

Conclusion

The purpose of this thesis was to design a distributed CMS framework that made no assumptions about platform capabilities except for what would be provided by a standard LAMP system. By looking at the constraints posed by such a system, we have outlined the major problems and provided workable solutions for them. Building on that design, we have implemented a prototype of the core mechanisms: replication and object serialisation. The prototype also supports searching the objects whether they are available locally or must be found on other nodes.

The resulting system is stable but not very developer friendly. To be usable by more casual developers, a framework should be as transparent as possible, which the Blitz prototype is not. Testing showed that search queries which could not be satisfied locally would consume large amounts of time. While in most cases this can be overcome by careful query construction and class design on the part of the developer, it is still needs work.

6.1 Further Work

Apart from the missing features listed in previous chapters, the primary component missing from the current system, to put it into production, is a load balancer. If a way could be found to write a load balancer within the constraints of the project, the system would be workable. Unfortunately, writing a proper load balancer in pure PHP is not feasible. A simple look at the test results in the previous chapter should suffice to convince any one. A peak performance of 50 requests won't do a lot of good when some one links the site to a Slashdot post, resulting in loads in excess of 200 requests per second. As it stands, the best solution is to use a solution based on round robin DNS, as described in chapter 3.

Another possible addition would be to use the classes used to wrap physical files, to control code updates in a consistent manner. As it stands, all nodes must either be stopped or risk inconsistency during a code update.

6.2 Bending the Rules

During the design phase, the constraints of the self-imposed "platform dogma" proved to be a recurring problem with no general solution. It became increasingly obvious that a compromise would be necessary to provide an efficient solution to problems such as load balancing. The simple fact was that having just one trusted server with unrestricted access would remove or circumvent almost all the major problems. A single dedicated server could run load balancing and DNS services for the remainder of the cluster. If it had the excess capacity, the trusted node could also run periodical backups, monitor remote nodes, and ensure cluster integrity after a remote node failure. If the trusted node was not part of the distributed cache itself, the other nodes would still function if the trusted node was disconnected. The benefits of having one trusted node outweighed all the drawbacks except one: its cost.

There was a time when having a server with a fixed IP address was expensive and rare. Today, most Internet providers can supply such a line to private users at a negligible monthly cost. The problem is, that ADSL lines are symmetrical in their bandwidth, and high bandwidth symmetrical lines are still very expensive. On the other hand if bandwidth prices keep falling at the same rate as they have for the last ten years, this won't be a problem for long.

6.3 Fin

As a final thought, consider this:

You are never going to win a professional race in an old bus. But a fleet of buses are going to bring a lot more people to the finishing line for the same money as a competitive race car.

Appendix A

Database Scheme

The Blitz CMS stores most objects in a relational database. All tables have the prefix “blz_” to reduce the chance that their names overlap with other tables in the same namespace. In a shared hosting environment there is often only one database allotted per domain name, meaning that the object database may have to co-exist with the data stores of other applications.

A.1 Attributes

The attribute table contains all objects in the CMS.

Table A.1: Structure of table blz_attributes

Field	**Type**	**Null**	**Default**
id	varchar(32)	No	
parent_id	varchar(32)	No	
mdate	int(10)	No	0
usec	int(6)	No	0
udate	int(10)	No	0
urn	varchar(128)	No	
class	varchar(64)	No	
replica	int(1)	No	0
deleted	int(1)	No	0
system	int(1)	No	0

id - A 128 bit hexadecimal identifier. Id’s are non-consecutive.

parent_id - Like **id**.

mdate - The objects last modification date. A Unix time time stamp.

usec - The micro second part of the above time stamp. Serves as a version id.

udate - The last time the attributes were synchronised with a server that had a valid replica. A Unix time time stamp.

urn - A human readable name for the object.

class - The name of the object's PHP class. If prefixed by an underscore "_", the object does not have any separate properties.

replica - Is there a local replica of this object? 0/1 value.

deleted - Has the object been marked as deleted or obsolete? 0/1 value.

system - Is the object used by the CMS to store internal state information? 0/1 value.

A.2 Nodes

The table stores information about known peers including cryptographic keys for them and the local node. Each node has a record representing itself. This record is the only one that contains a private key.

Table A.2: Structure of table blz_nodes

Field	**Type**	**Null**	**Default**
id	varchar(32)	No	
server_uri	text	No	
public_key	text	No	
private_key	text	No	
private_pass	text	No	
last_contact	int(10)	No	0
last_rtt	float	No	0
rtt_avg	float	No	0

id - A 128 bit hexadecimal identifier. Id's are non-consecutive.

server_uri - The WWW address of the remote server.

public_key - This is an RSA public key.

private_key - This is an RSA private key.

private_pass - Unused.

last_contact - Unused.

last_rtt - Unused.

rtt_avg - Unused.

A.3 Tokens

The table stores transaction states in the form of tokens.

Table A.3: Structure of table blz_tokens

Field	**Type**	**Null**	**Default**
token	varchar(32)	No	

Table A.3: Structure of table blz_tokens (continued)

Field	**Type**	**Null**	**Default**
status	int(1)	No	0
origin	varchar(128)	No	

token - A 128 bit hexadecimal identifier. Token identifiers are non-consecutive.

status - This describes the current status. 0 - unprocessed, 1 - being processed, 2 - finished. If there is no token, a value of 0 is implied.

origin - Unused.

A.4 A sample property table

This is what a property table for the class "foobar" would look like, if the class had only one property named "text". All other fields are part of the record's key.

Table A.4: Structure of table blz_class_foobar

Field	**Type**	**Null**	**Default**
id	varchar(32)	No	
mdate	int(10)	No	0
usec	int(6)	No	0
text	text	No	

Appendix B

Inter-node Communication Protocol

Inter-node communication happens in a multi-step process.

B.1 The Client

First the sender constructs an HTTP POST request, with a number of custom headers:

User-Agent – This helps servers identify Blitz requests. The user-agent is a string in the form “Blitz/0.4”.

X-Session-Token – A 128 bit hexadecimal token id.

X-Method – The server method being invoked. Parameters are passed with POST variables.

X-Remote-Node-Id – This is the id of the node that initiates the call. It is primarily used for debugging.

X-Broadcast – Either “true” or “false”. If true, the message should be broadcast to all peers, except the one identified by the “X-Remote-Node-Id” header.

The actual method parameters are called “payloads” in the code and are passed along with the message in a POST variable. A payload can be any valid PHP variable.

To transmit a payload the payload is first serialised into a string. The string is then encrypted using the public key of the target.

The result is an array consisting of an SSL encrypted string, and an envelope key.

The array is then serialised and URL encoded. The latter is necessary because the encoded payload has to be passed as a POST variable. To speed transmission of large payloads, the final HTTP request is transmitted using gzip compression.

At the receiving end, the process is reversed, and the payload is decrypted using the target’s private key, and passed to what ever method was called.

B.2 Server

Servers encrypt and decrypt data like the client. When a server receives a request, it first verifies that the request is calling a valid method as specified in the “X-Method” header.

It then tries to acquire the token specified in the “X-Session-Token” header. If unsuccessful, the server stops processing the message, otherwise, it passes the decrypted message payload to the specified method.

When the method returns, the result is encrypted and echoed as a response to the client.

Appendix C

Minimal Dispatch Script

A minimal dispatch script should include the Blitz CMS files, and handle any incoming inter-node traffic.

C.1 Example `dispatch.php`

```
require_once "blitzcms.inc";

BLZSetDB("mysql://localhost/test_blitz");
BLZEnableCache();

class HelloWorld extends BLZObject {
  function render() {
    echo '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>'.$this->urn.'</title>
</head>

<body>
  <p>'.$this->greeting.'</p>
</body>
</html>';
  }
}

BLZServeRequests(); // Serve incoming requests

$uri = $_SERVER["PATH_INFO"];
$page = BLZLoadURI($uri);

if ($page && is_a($page,"HelloWorld"))
  $page->render();
else {
  header("HTTP/1.1 404 Not Found");
  echo "'$uri' not found.";
}
```

Appendix D

Acknowledgements

Writing this thesis was a long process that went through several changes in focus before it arrived at its current form. Along the way the actual prototype underwent several complete rewrites. The first iteration contained only network stubs but had a nice user friendly editor based on SOAP. Later iterations discarded all pretence of user friendliness in exchange for a Java inspired abstraction layer with far too many classes. Closer to the final version, the prototype used a model that relied on a single server to act as a master-server that had to verify all updates and handle replication. Through trial and error, the current version builds on the ideas of all its predecessors, even if does not share a single line of PHP code with them.

In finishing the project I am deeply indebted to the people who sacrificed their time and resources to help me. I would like to thank:

My parents, for soldiering through the first drafts despite acute boredom.

My former colleagues at Brandhouse a/s, Nikolaj and Jonas, for providing me with three of the nodes I needed for my test cluster. Without their blatant disregard for company property, I would not have been able to test my prototype on an interesting scale. Another former colleague, Søren, provided the crisp logo for the cover page, while Martin provided the Gnuplot examples.

My friends Jacob and Kristian who trusted me the login information for their respective web hosts, providing another two test nodes in a low cost shared environment.

Finally I would like to thank Asger, and my brother Jens, for their invaluable aid in proof reading, making suggestions, and last but not least, for coming up with the name for the project.

List of Figures

List of Tables

Bibliography

[1] Akamai web site. URL http://www.akamai.com.

[2] Alternative PHP Cache. URL http://pecl.php.net/package/APC.

[3] Apache HTTP server benchmarking tool. URL http://httpd.apache.org/docs/1.3/programs/ab.html.

[4] CoralCDN. URL http://www.coralcdn.org.

[5] CVS revision control. URL http://www.nongnu.org/cvs.

[6] eAccelerator. URL http://eaccelerator.net.

[7] Fabricata CMS framework. URL http://www.fabricata.com. *The web site is in Danish.*

[8] GNU Wget. URL http://www.gnu.org/software/wget/.

[9] Google bomb (wikipedia entry). URL http://en.wikipedia.org/wiki/Google_bomb.

[10] Mediacom: Nye læsertal – stort fald for de store dagblade. URL http://www.mediacom.dk/internet/web-mcdk.nsf/pageview/4.1.6.0%20-%20Sto%rt%20fald%20for%20de%20store%20dagblade!OpenDocument&Click. The article is in Danish and deals with the falling sales numbers for major news papers in Denmark.

[11] Microsoft looks to extinguish LAMP. URL http://news.com.com/Microsoft+looks+to+extinguish+LAMP/2100-1012_3-5746%549.html.

[12] Midgard CMS. URL http://www.midgard-project.org.

[13] Model View Controller. URL http://www.phpwact.org/pattern/model_view_controller.

[14] Netcraft. URL http://news.netcraft.com.

[15] PEAR - PHP Extension and Application Repository. URL http://pear.php.net.

[16] PHP-Nuke, . URL http://www.phpnuke.org.

[17] Php openssl class. URL http://www.karenandalex.com/php_stuff/classes/Openssl.php.

[18] Politiken. URL http://politiken.dk.

[19] PostgreSQL web site. URL http://www.postgresql.org.

[20] The register (online news site). URL http://www.theregister.co.uk/2005/10/18/mysql_marketshare_numbers/.

[21] Round robin dns. URL http://en.wikipedia.org/wiki/Round_robin_DNS.

[22] Savant. URL http://savant.tigris.org.

[23] Slashdot (online news portal). URL http://www.slashdot.org.

[24] Smarty: Template engine. URL http://smarty.php.net.

[25] Subversion revision control. URL http://subversion.tigris.org.

[26] Usage statistics for PHP, . URL http://www.php.net/usage.php.

[27] Wikipedia, online dictionary. URL http://www.wikipedia.org.

[28] Windows bumps Unix as top server OS. URL http://news.com.com/2100-1016_3-6041804.html?part=rss&tag=6041804&subj=%news.

[29] XHTML - 1.0 the extensible hypertext markup language (second edition). URL http://www.w3.org/TR/xhtml1/.

[30] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. pages 323–336. URL citeseer.ist.psu.edu/aron00scalable.html.

[31] Martin Belam. A day in the life of bbci search, January 2003. URL http://www.currybet.net/articles/day_in_the_life/index.shtml.

[32] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, August 1998. URL ftp://ftp.internic.net/rfc/rfc2396.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2396.txt. Status: DRAFT STANDARD.

[33] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (third edition). Technical Report REC-xml-20040204, 2004. URL http://www.w3.org/TR/2004/REC-xml-20040204.

[34] R. Burns, R. Rees, and D. Long. Consistency and locking for distributing updates to web servers using a file system. In *Workshop on Performance and Architecture of Web Servers*, June 2000. URL citeseer.csail.mit.edu/burns00consistency.html.

[35] Alex V. Cate. Alex: a global file system. In *Proceedings of the Usenix Conference on File Systems*, 1992. URL citeseer.ifi.unizh.ch/cate92alex.html.

[36] James Clark. XSL Transformations (XSLT) version 1.0. Technical Report REC-xml-19980210, W3C, 1998. URL citeseer.nj.nec.com/bray98extensible.html. http://www.w3.org/TR/xslt.

[37] P. Felber, T. Kaldewey, and S. Weiss. Proactive hot spot avoidance for web server dependability. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, pages 309–318, oct 2004. URL `http://members.unine.ch/pascal.felber/publications/SRDS-04a.pdf`.

[38] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report Internet RFC 2616, IETF, 1998. URL `citeseer.nj.nec.com/article/fielding98hypertext.html`. `http://www.ietf.org/rfc/rfc2616.txt`.

[39] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 239–252, mar 2004. URL `http://citeseer.ist.psu.edu/article/freedman04democratizing.html`.

[40] A.S. Tanenbaum G. Pierre, M. van Steen. Dynamically selecting optimal distribution strategies for web documents. Technical Report IR-486, Netherlands, 2001. URL `http://www.cs.vu.nl/pub/papers/globe/IR-486.01.pdf`.

[41] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. RFC 2518: HTTP extensions for distributed authoring — WEBDAV, February 1999. URL `ftp://ftp.internic.net/rfc/rfc2518.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2518.txt`. Status: PROPOSED STANDARD.

[42] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework. Technical Report REC-soap12-part1-20030624, 2003. URL `http://www.w3.org/TR/2003/REC-soap12-part1-20030624`.

[43] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002. URL `citeseer.ist.psu.edu/article/maymounkov02kademlia.html`.

[44] Kevin Shanahan Michael. Locality prediction for oblivious clients. URL `citeseer.ist.psu.edu/728169.html`.

[45] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992. URL `ftp://ftp.math.utah.edu/pub/rfc/rfc1305.txt`. Obsoletes RFC0958, RFC1059, RFC1119. Status: DRAFT STANDARD.

[46] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Trans. on Knowledge and Data Engineering*, 12(4):645–672, 2000. URL `citeseer.ist.psu.edu/nicola00performance.html`.

[47] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for*

Programming Languages and Operating Systems, pages 205–216, 1998. URL `http://citeseer.ist.psu.edu/article/pai98localityaware.html`.

[48] Guillaume Pierre, Ihor Kuz, Maarten van Steen, and Andrew S. Tanenbaum. Differentiated strategies for replicating Web documents. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000. URL `citeseer.ist.psu.edu/pierre00differentiated.html`.

[49] Guillaume Pierre and Maarten van Steen. Globule: A platform for self-replicating Web documents. *Lecture Notes in Computer Science*, 2213:1–??, 2001. URL `citeseer.ist.psu.edu/article/pierre01globule.html`.

[50] Guillaume Pierre and Maarten van Steen. A trust model for peer-to-peer content distribution networks, 2001. URL `http://citeseer.ist.psu.edu/pierre-trust.html`.

[51] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001. URL `citeseer.ist.psu.edu/qiu01placement.html`.

[52] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. Globedb: Autonomic data replication for web applications, 2005. URL `http://citeseer.ist.psu.edu/sivasubramanian05globedb.html`.

[53] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996. URL `citeseer.ist.psu.edu/article/stonebraker96mariposa.html`.

[54] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[55] Marteen van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed system. Technical Report IR-422, Netherlands, 1997. URL `citeseer.ist.psu.edu/vansteen99architectural.html`.

[56] Dave Winer. XML-RPC specification, June 1999. URL `http://www.xmlrpc.com/spec`.