# Benchmarking Database Distribution by Simulating Load on the Web-Tier

**Simulating users based on extracted sessions from web-access logs on existing business software (M4N) using an open source database distribution solution (Sequoia)**

Written by
*Richard Velden*

Under supervision of
*Prof. Dr. H. Wijshoff*

In collaboration with
*M4N, Amsterdam*

This thesis is submitted for obtaining the degree of Master of Computer Sciences at the Leiden Institute of Advanced Computer Sciences (LIACS)

Leiden University

**August 3, 2006**

**Acknowledgments**

I would like to thank Prof. Dr. H. Wijshoff for his excellent guidance and the employees at M4N for all their time helping me in my research.

## Abstract

In this paper we focus on the problems of growing database applications. Growing database needs can be catered for in many different ways. The simplest being to buy a faster server, but eventually, even the fastest server will not be fast enough and database distribution becomes the only (economically viable) remaining option.

We will try out a distributed database solution on M4N, an existing piece of business software. With simulated load tests, based on access log files we will measure the performance and reliability of the current single database solution and compare this with a distributed database configuration.

In the end we hope to come to a conclusion about the stability and performance of the Sequoia distributed database solution.

# Contents

# Chapter 1

# Introduction

This research was conducted as an internship at M4N. They needed to investigate whether database clustering could be applied to their in-house developed web application. The goal set was to make the system more reliable in a cost effective manner. Some important requirements were preferably not to change the software itself and that the system had to handle higher loads in the future. Before diving into details we first explain about the M4N business itself and how this generates high database loads.

## 1.1 M4N Business

M4N is located in Amsterdam and is in the business of online advertising. Their core business is to provide an online marketplace for both advertisers and affiliates as well as to provide view, click and lead registration. An additional field M4N focuses on is online advertising consultancy to increase their customer's advertising effectiveness [1].

### 1.1.1 Online Advertising Stake-holders

- Advertisers: Online businesses that use ads to boost their online sales e.g. Dell and Amazon.
- Affiliates: Websites capable of attracting an audience. These include news websites such as CNN or search engines like Google.
- Customers: Home or business users who are willing to purchase something on the web (from advertisers).

Imagine someone looking to buy a new laptop computer. To find a good laptop the user enters the keyword 'laptop' into the search field of a search engine. Laptop resellers such as Dell, Toshiba and Apple will pay good money to search engines whenever their websites show up in top whenever a user taps in 'laptop'. They will pay even more whenever a user actually clicks on the link and in the case of an actual purchase a reseller might even give the search engine a percentage of the sales profit. Bringing in customers can be a business on its own (fig: 1.1) [2, 3, 4, 5].

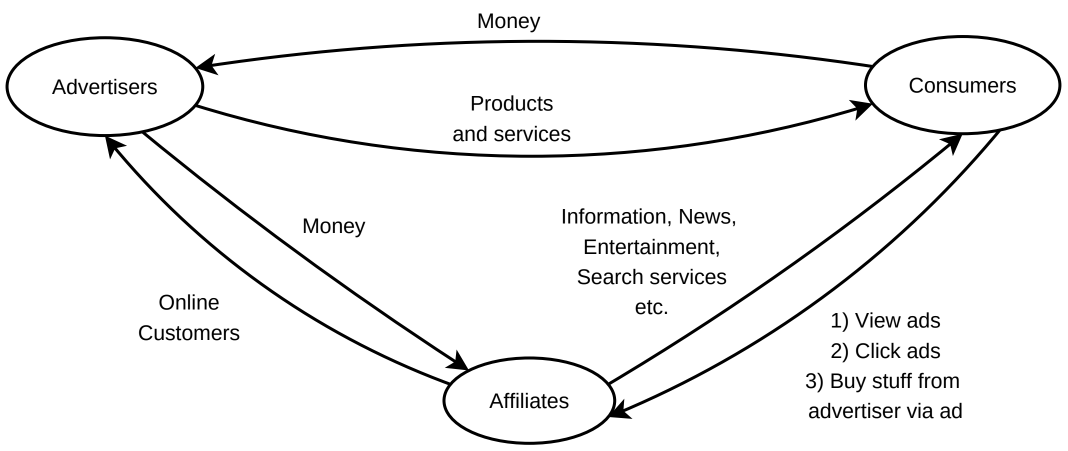


Figure 1.1: Online marketing value chain

### 1.1.2 Online Advertising Reward System

Online advertising generally works on a reward based system in which the advertiser per category determines the amount of money it will issue to the affiliate:

1. View: Whenever a web user views the add of the advertiser on the affiliate website.
2. Click: Whenever the advertiser gets a web visitor via a link of the affiliate website.
3. Lead: Whenever an actual online sale is made by a customer who entered the advertiser's website via a link on the affiliate website.

In the current reward system it is the advertiser who can determine how much to give per category. Dell can for instance issue a view reward of 0.01 cents, a click reward of 4.00 cents and reward actual leads with 5 Euros. It is up to the affiliate whether they want to place this advertisement on their website.

It is easy to see that when for instance Apple offers significant higher rewards it can be more profitable for affiliates to show Apple ads. On the other hand, Dell can outweigh this difference by compensating this per sale difference with higher sales volumes. Why earn 5 times 20 Euros with Apple ads while one can also earn 100 times 5 Euros with the Dell ads? [2, 4, 5]

## 1.2 The M4N System

The M4N system offers a wide range of services. From view, click and lead registration to statistics on how good your ads (and those of others) are doing. We have listed what M4N offers for the various stake-holders (see also fig: 1.2).

- Advertisers: M4N provides the software infrastructure to register views, clicks and leads. Instead of developing or buying such a system and maintaining it themselves, advertisers rely on M4N.
- M4N provides a marketplace in which advertisers can offer their ads and affiliates can find them.
- Security and trust: Affiliates and advertisers can be rated by each other.
- Affiliates: Banner offerings between advertisers can be compared.
- M4N calculates and shows cost effectiveness per banner
- M4N shows how successful each affiliate has been
- M4N offers many other statistics to help boosting online sales for advertisers and increasing affiliate revenue.

In turn for using the view, click and lead registration infrastructure as well as for the other services advertisers pay a fixed monthly fee. Next to this monthly fee M4N also receives either a percentage of the profit made from sales via M4N, a fixed price per view or click, or a combination of these [1, 5].

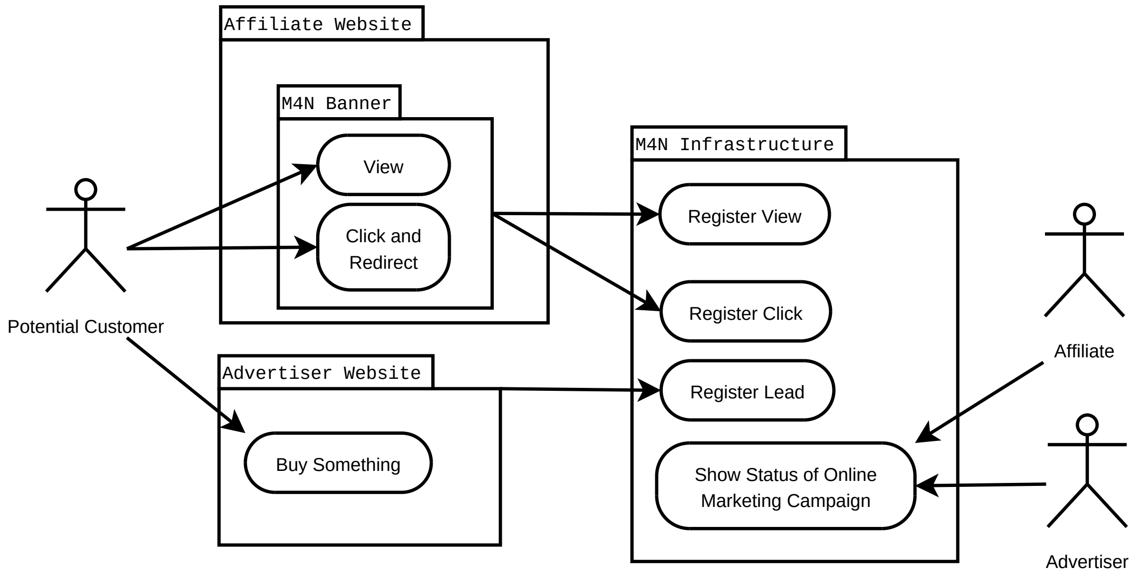


Figure 1.2: M4N Global Architecture

## 1.3 Our Research Project

Imagine the early database driven version of the M4N web application handling a mere 10.000 requests a day. The required database runs on a single CPU system which is easily capable of sustaining the peak and average loads from this web application. A year later however user demand rises enormously to around 3 million requests a day, which equals to an average load of 34 requests per second. A new server with a RAID-5 disk array and quad CPU is installed to cope with this increased load. The next year real problems arose. The database had grown to around a billion records and had to handle 100 million requests a day. It was easy to see that no single system could handle these loads and storage requirements and a more distributed solution had to be found.

Most large applications once started off small and many of them did not calculate in the possibility of up-scaling when choosing a database system. M4N is such a system, which gradually evolved into a big heavy used application.

### 1.3.1 Research Question

Given a specific database driven application like M4N, we want to find out how database clustering can be used to cater for increasing future loads. To do so we have to answer the following questions:

- How does this future load look like?
- How do we simulate this load in a testing environment?
- What database solutions to use for simulating our 'future load' on preferably without changing the software itself?

### 1.3.2 Research Objectives

To answer the fore-mentioned research questions we have put up some global objectives. These can be seen as the phases of our project plan[1].

1. Identify the current bottlenecks in the M4N system (Chapter 2).
2. List potential solutions for solving these bottlenecks (Chapter 3).
3. To benchmark different solutions: Create configurable load tests resembling typical M4N traffic (Chapter 4).
4. Create a mirror infrastructure for testing purposes, in which this typical M4N load can be simulated.
5. Measure reliability and performance of current M4N architecture using the created load tests (benchmarking).
6. Set up and measure reliability and performance of M4N using some distributed database solution (benchmarking).
7. Compare results and draw conclusions on the effectiveness of database distribution.

[1]A indicative project plan with details about what to research was already specified by M4N (see appendix C).

# Chapter 2

# M4N System Architecture

The M4N system basically provides 4 core features. View, click and lead registration being the first three and the website itself as number four (see figure 1.2). In an ideal situation[1] those four components could operate independently. This however is not the case right now.

Due to ill design in the early days of M4N all these components were tightly knit together; a dependency which still forms a threat for M4N's business. Take for instance one critical bug in the website code. This bug can not only crash the website itself but take down the view, click and lead registration as well. Because the risk of these critical bugs is substantially higher in the bigger and more complex website code it is not desired to have the view, click and lead registration run on the same web-server. Solving this entire issue still is a complex job. That is why M4N's developers have only found time to separate the view registration from the rest of the website. However, at the time our thesis project was nearing its end, click registration was added as well for the new version of M4N (2.5.2).

[1]In the case one redesigns M4N from scratch

## 2.1 Global Architecture

The main website runs on *webLive*, a single web-server which because of the fore-mentioned dependencies also handles click and lead registration. This web-server connects to *dbaseLive*, the main database system for all its transactions.[2]

The view registration, because it has been separated from the rest of the system, can be run from any other server. Right now M4N uses *view2Live* and *view1Live*, two separate machines, to handle view registration, writing them to local *'sattelite databases'*. To keep the main database as well as the website updated, the views in the sattelite databases are copied to *dbaseLive* once every half an hour. (see figure 2.1).

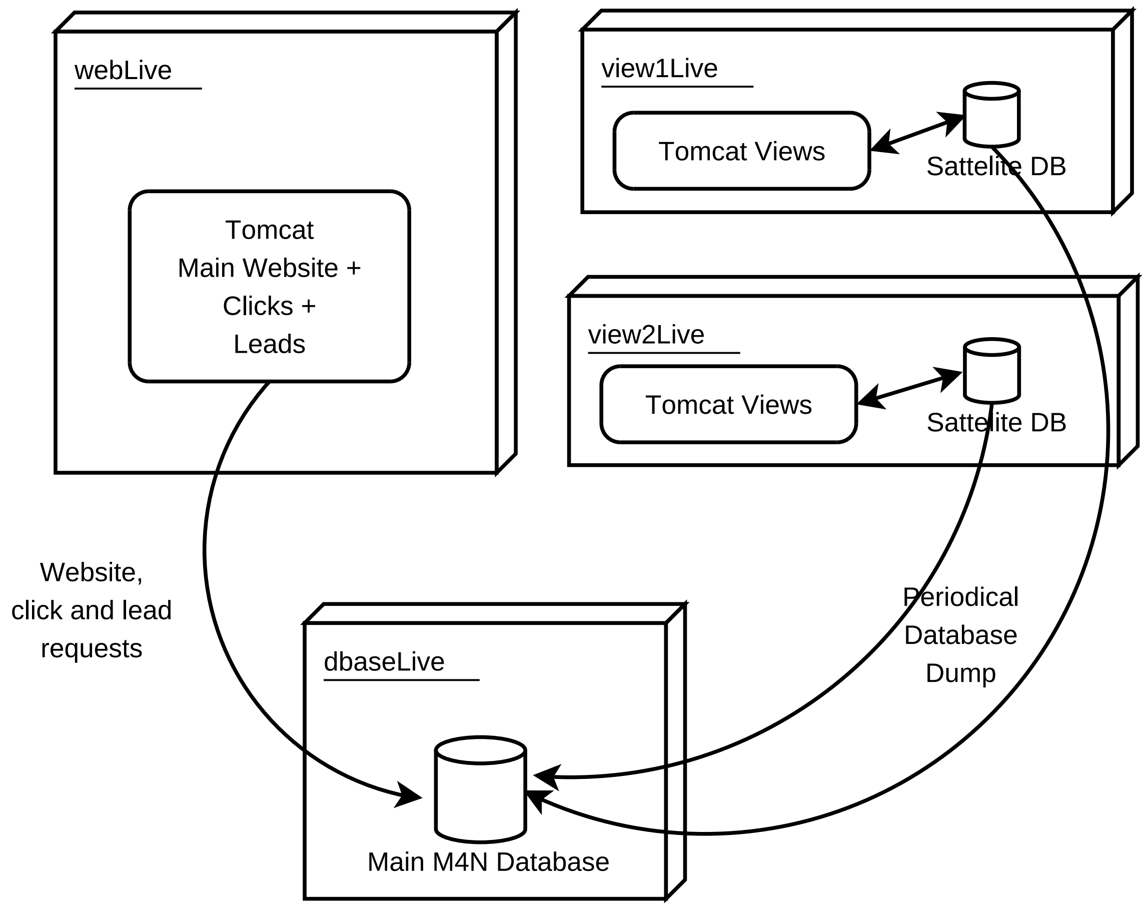


Figure 2.1: *Simplified* M4N System Architecture. Several tasks and services like for instance database backups are obscured from this view.

[2]Some very old view registration code still uses a local 'sattelite-database' on the main web-server. The number of views registered this way however is negligible and will be ignored further in this report.

## 2.2 M4N Live System Specifications

**webLive:**

- CPU: Pentium4 2.8GHz with Hyper-threading
- MEM: 1.5GB

webLive functions as the main web-server for the website as well as doing the click and lead registration.

**dbaseLive:**

- CPU: 2x Intel Xeon 2.4GHz 512K cache, 533MHz FSB
- MEM: 3x 1GB DDR SDRAM 266MHz
- HD: 3x 36GB 10,000rpm 1” U320 SCSI HD

dbaseLive is the main live database for M4N and is directly used by webLive. Once every 6 hours a database dump is made which is used to update the backup database system (view2Live) in case dbaseLive crashes.

**view1Live:**

- CPU: P4 3.2GHz
- MEM: 1GB

view1Live is next to view2Live one of the two view registration servers.

**view2Live:**

- CPU: 2x Intel Xeon 2.4GHz
- MEM: 3GB
- HD: 60GB + 250GB

view2Live is next to view1Live one of the two view registration servers. Apart from registering these views it is also functions as M4N’s backup database system for dbaseLive. In the case dbaseLive crashes, view2Live will be used instead. This does not happen automatically and usually takes around half an hour. [3]

To keep this backup database up to date, view2Live performs a database restore using the latest database dump made by dbaseLive (see figure 2.2). This database restoration is a very intensive job and is performed once every 6 hours, just after a new database dump has been made by dbaseLive. In case the main database server crashes M4N thus loses at most 6 hours of data (3 hours of data on average).

[3]A SMS alert is sent when the server fails to respond. After a manual configuration change view2Live will receive the requests normally going to dbaseLive.

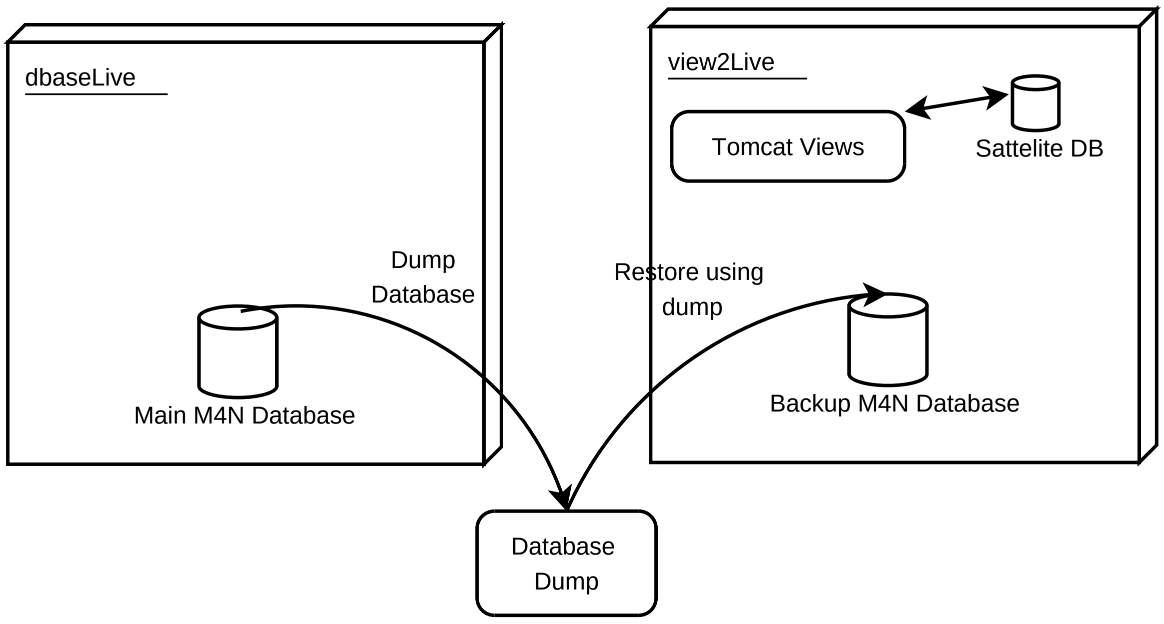


Figure 2.2: Database backups: updating the backup database system.

## 2.3 M4N Bottlenecks

To get a good indication of the performance requirements of M4N we have taken a look at the load statistics of each server over the past year. This way we hope to find under which conditions the database server became the bottleneck of the entire system and what part of the server was the most constraining (CPU, memory or hard-disk).

### 2.3.1 Load Statistics

**webLive:**

| **Last 30 days averages** | |
|---|---|
| Number of Values | 4449 |
| Average | 2.86181724732687 |
| highest | 92.8283333333333 |
| lowest | 0.03 |

According to the 30-day averages, peak loads are hardly observed on the web server. A very low average load and a maximal observed peak of only 92.8% suggest that this server is not a real bottleneck for the M4N system (yet).

The one year statistics in figure 2.5 hint that only marginal loads were experienced. However, one needs to keep in mind that these graphs are based on averages over a certain time period $T$. When the total time span of the graph is long (e.g. 1 year), $T$ will likewise be larger than when the total time span was just 1 day. Once you look at the last two hour statistics (see figure 2.3) of the web-server one finds near-peak loads. Their sporadic nature however keeps them unnoticed when $T$ gets larger. For this reason we argue that webLive does not form a performance bottleneck for the M4N system [4].

[4]It does form a Single Point of Failure on which we will elaborate later on.

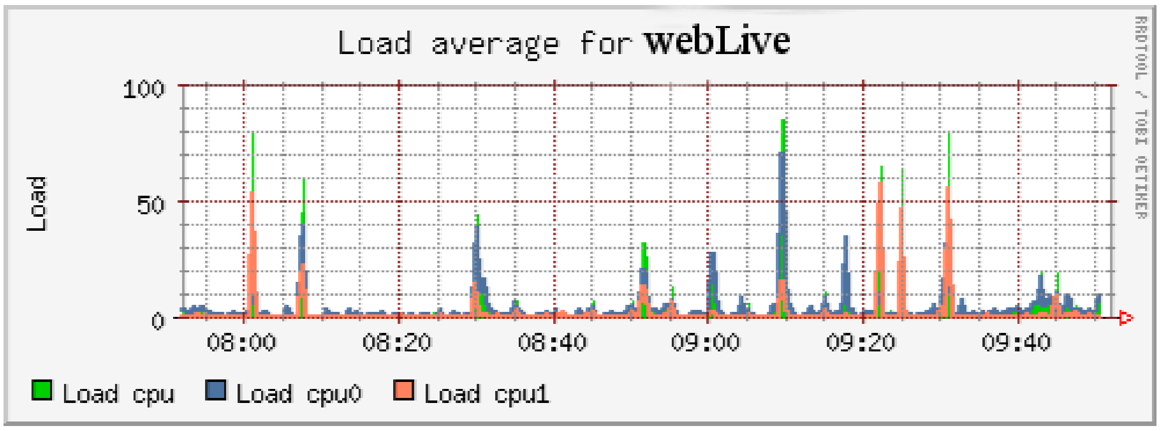


Figure 2.3: webLive last two hour statistics

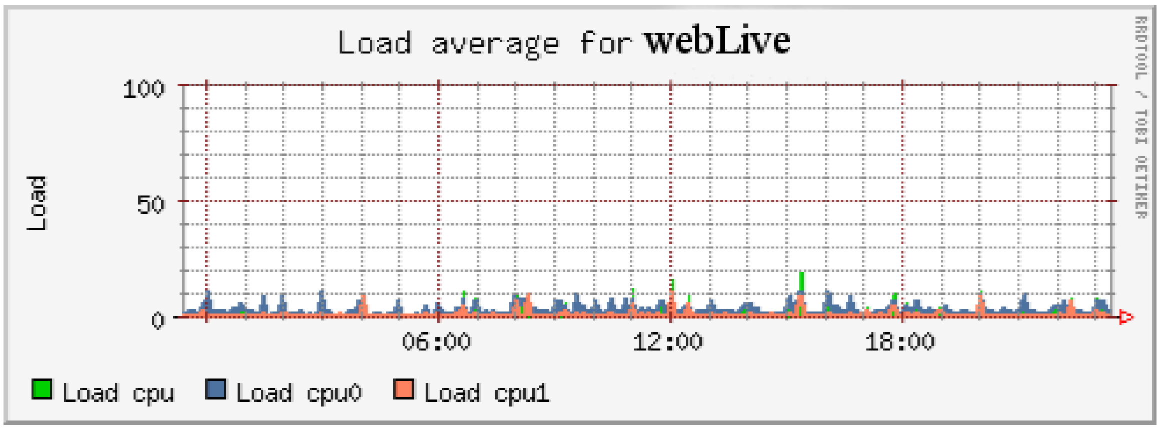


Figure 2.4: webLive 24 hour statistics

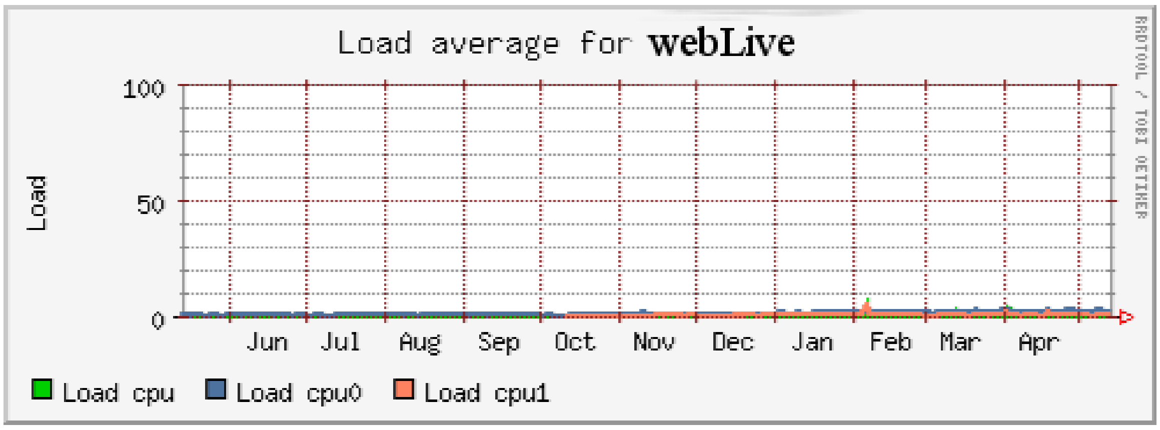


Figure 2.5: webLive one year of statistics

Although webLive occasionally experiences peak loads we can still conclude that it has not been a constraining factor on the total performance of M4N.

**dbaseLive:**

| **Last 30 days averages** | |
|---|---|
| Number of Values | 4462 |
| Average | 26.5889593291582 |
| highest | 100 |
| lowest: | 1.63833333333333 |

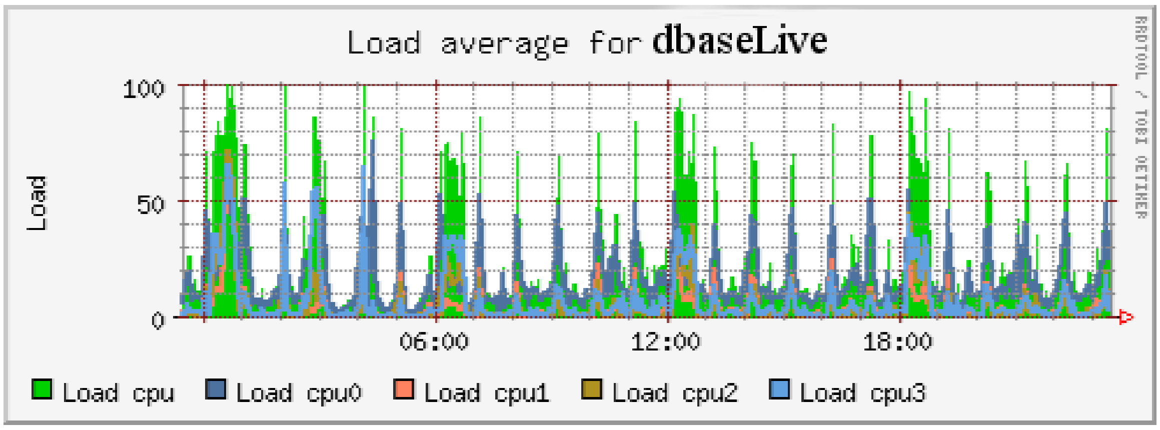


Figure 2.6: dbaseLive 24 hour statistics

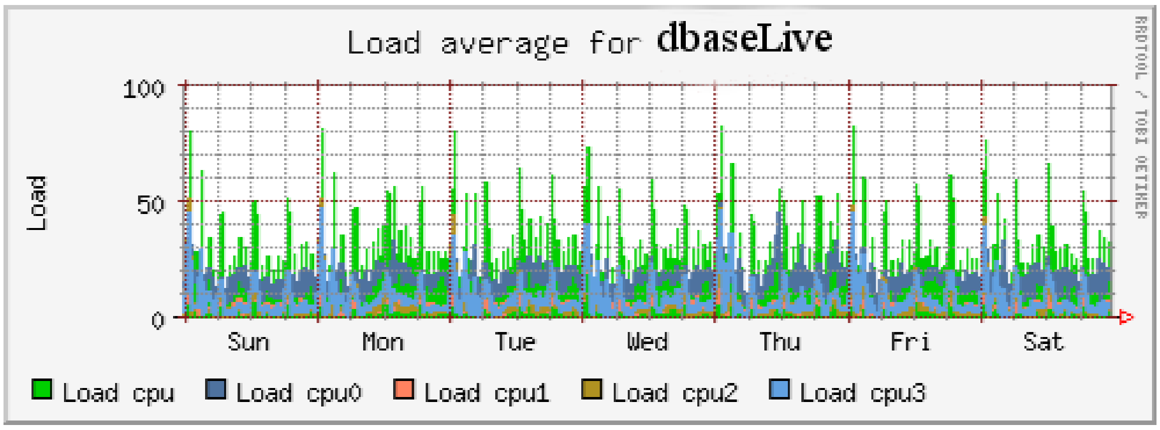


Figure 2.7: dbaseLive one week statistics

dbaseLive experiences major CPU-load spikes once every 6 hours caused by the database dump. The smaller spikes, once every hour are the result of the 'banner-update' script (see figure: 2.6). This script downloads new banners from advertiser's websites and stores them into both the main (dbaseLive) as well as the view (view1Live and view2Live) databases.

Note the widest spike in figure 2.6 between 24:00 and 6:00 which apart from a database backup represents the calculation of *'the statistics'*. Using SQL scripts data is gathered and processed into neat statistics which are greatly valued by M4N's users. These statistics used to be calculated once every four hours. Since December 2005 however, this was reduced to once a day due to a database server crash (notice the gap in the beginning of December in figure: 2.8).

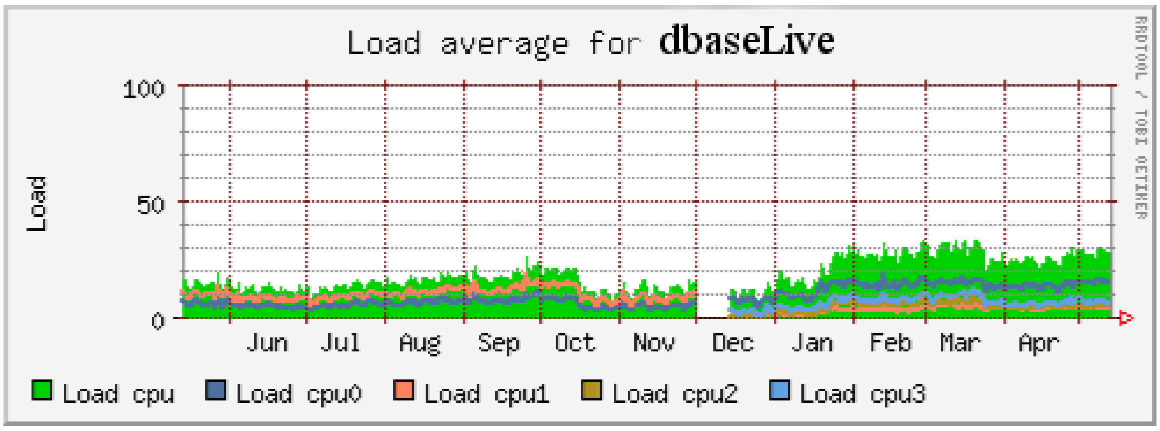


Figure 2.8: dbaseLive one year of statistics

**view2Live:**

| Last 30 days averages | |
|---|---|
| Number of Values | 4462 |
| Average | 22.1503978950517 |
| highest | 99.2733333333333 |
| lowest | 0.0433333333333333 |

An average load of only 22% and a peak load of 99% suggests that the load is not evenly distributed over time. Looking at the statistics we indeed see a spiky load (see figure: 2.9, 2.10). A more careful look reveals two distinct patterns. One pattern of 'large spikes' once every 6 hours, the other spikes are somewhat smaller and occur every hour. The large spikes correspond with a database restoration which is done with the 'just dumped' database version of dbaseLive. The smaller spikes are caused by the so-called banner-update script which retrieves banners from remote sites and puts them in all the databases.

In figure 2.11 we see the loads encountered over the last year. We indeed see a spike in the beginning of December 2005, which corresponds to view2Live's temporary task of taking over from the crashed main database server dbaseLive.

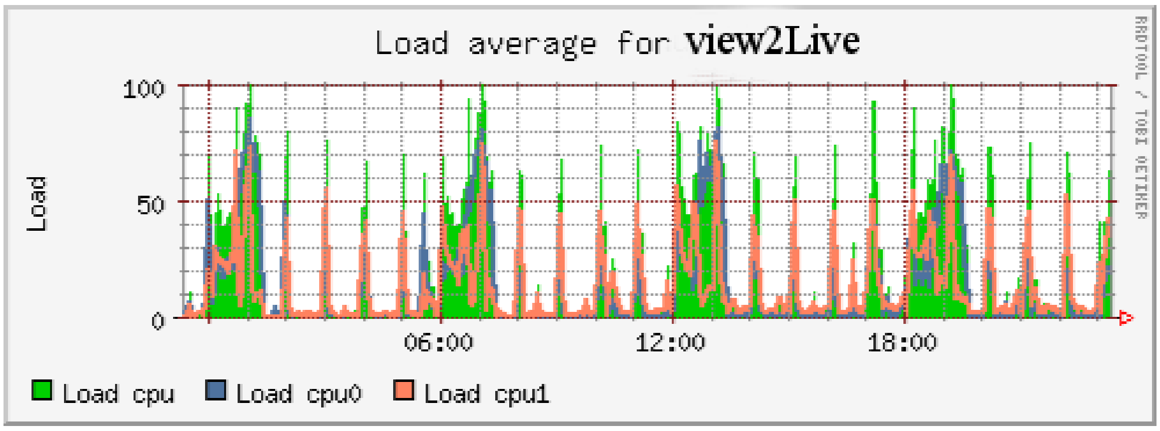


Figure 2.9: view2Live 24 hour statistics

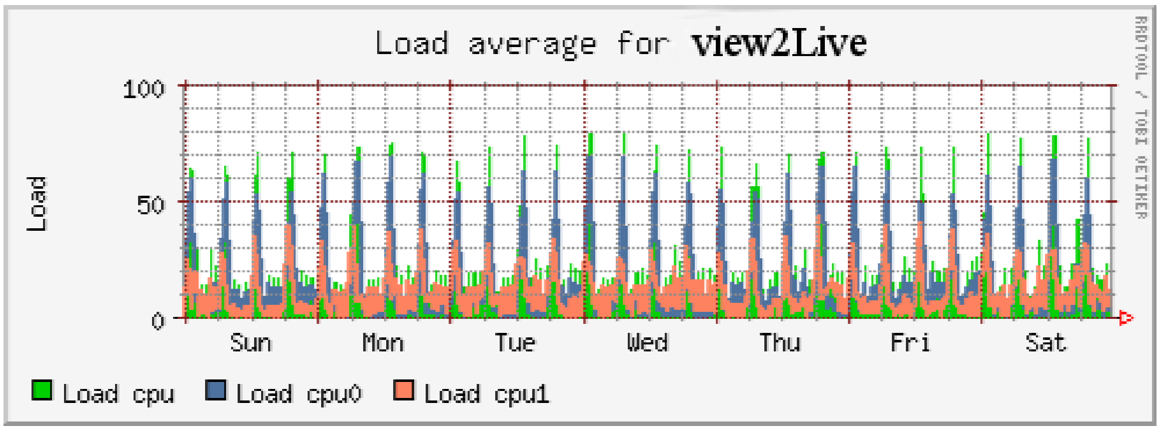


Figure 2.10: view2Live one week statistics

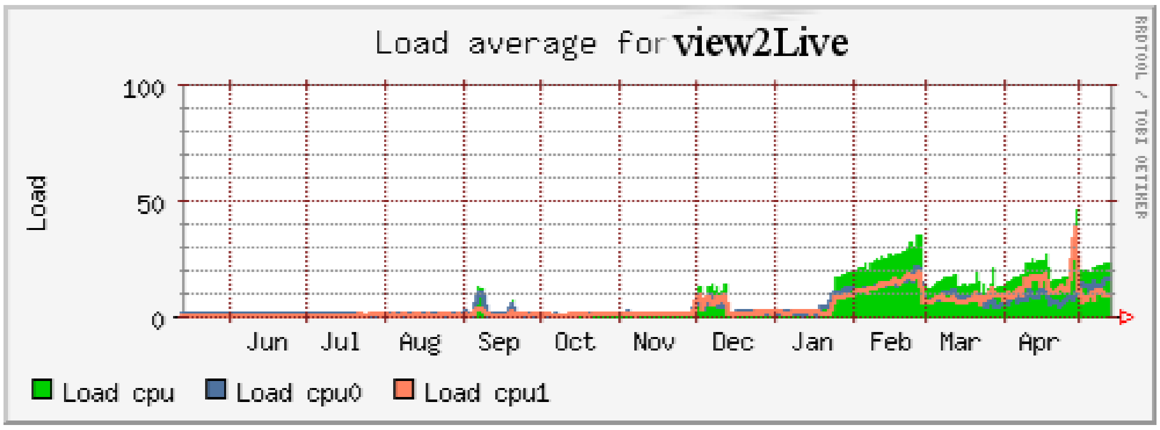


Figure 2.11: view2Live one year of statistics

**view1Live:**

| **Last 30 days averages** | |
|---|---|
| Number of Values | 4462 |
| Average | 11.4077674134223 |
| highest | 95.9783333333333 |
| lowest | 0.43 |

The hourly spikes observed on view1Live are caused by the fore-mentioned banner updates. Furthermore view1Live handles view requests which just generates a very light CPU load (see figure: 2.12, 2.13).

### CPU Load during Database Backup

When using modern disks, especially with SCSI controllers, disk IO does not cause high CPU loads. However, during database backups at dbaseLive and view2Live, we do encounter near 100% load spikes, while one would expect no significant increase in CPU usage at all. Database backups should strictly speaking be only disk IO dependent.

The explanation for this phenomenon at M4N is that the type of backup performed requires every record to go through several checks. We will explain this by using a 'SQL transaction

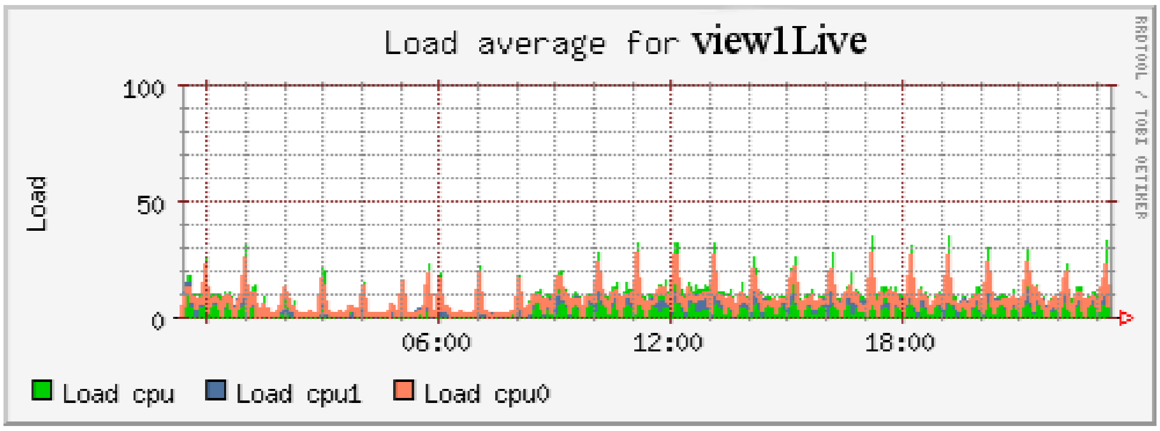


Figure 2.12: view1Live 24 hour statistics

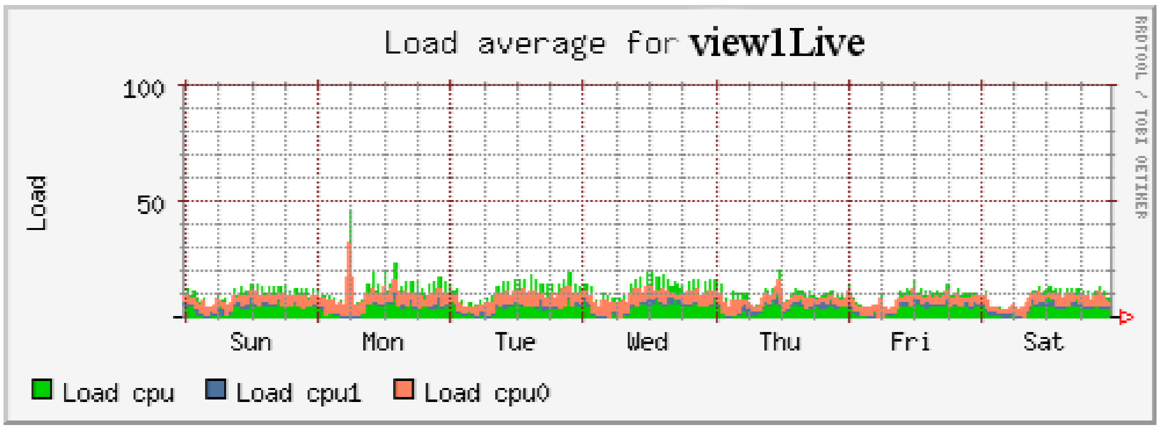


Figure 2.13: view1Live one week statistics

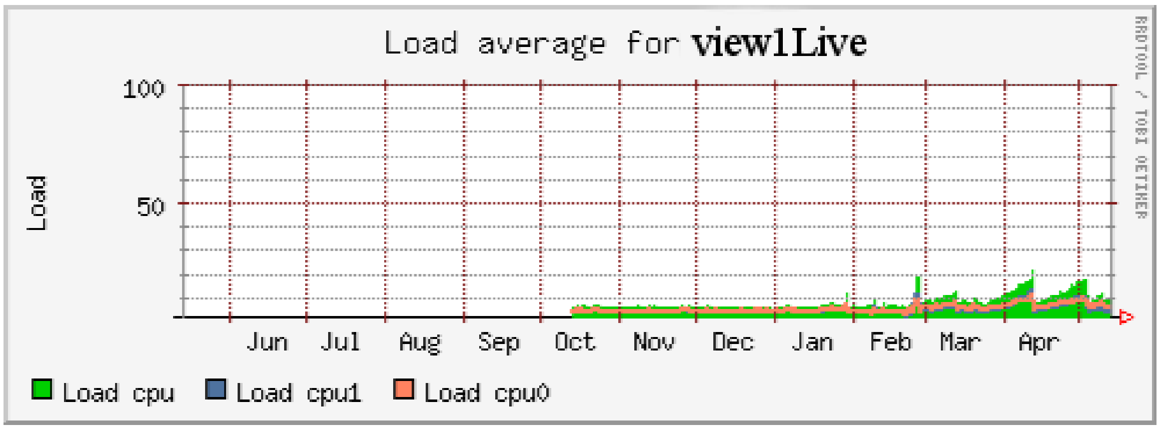


Figure 2.14: view1Live one year of statistics

style' database dump as an example. This type of database dump ensures inter-database compatibility but also requires the CPU to perform some transformation (from database record to SQL statement) for each record (see figure: 2.15). At M4N a Postgres specific COPY command is used to copy the needed tables from the database which is in fact faster than a

’SQL transaction’ dump. Checks however still need to be done during these COPY’s: Checks which require CPU attention and thus cause load spikes during the backups.

**Database record (employees table):**

| id | firstname | lastname | gender |
|---|---|---|---|
| 4 | John | Doe | Male |

**Database dump:**

```
INSERT INTO employees( id, firstname, lastname, gender )
    VALUES (4, ’John’, ’Doe’, ’Male’);
```

Figure 2.15: Example of a SQL transaction based database dump of a single record.

### 2.3.2 Server Task List

Each server at M4N has specific tasks which we have mentioned scattered among some of the previous sections. For clarity we now present these tasks in a short overview.

| | webLive | dbaseLive | view2Live | view1Live |
|---|---|---|---|---|
| Website | X | | | |
| Website backup | | | X | |
| Views | | | X | X |
| Clicks | X | | | |
| Leads | X | | | |
| Clicks, leads Backup | | | X | |
| Main database | | X | | |
| Main database backup | | | X | |
| Calculate nightly statistics | | X | | |
| Database dump (every 6 hours) | | X | | |
| Database restore (every 6 hours) | | | X | |
| Banner updates (every hour) | | X | X | X |
| Sattelite database dump (views) | | | X | X |
| Updating views into main database | | X | | |

## 2.4 M4N System Failure Analysis

There are two distinctive parts. The website and the view, click and lead registration. Failure of the website would only stop users from checking and updating their online advertising campaigns. Not having the website available for an hour or so is not that big a problem. The website is just the interface to maintain and check up on *the actual* service M4N offers, which is the view, click and lead registration.

Sadly however, website failure also results in click and lead registration failure because those components are too tightly knit together (see figure 1.2). Whenever for instance webLive crashes, it does not only take out the website but also the click and lead registration. The current solution is to manually intervene and let view2Live take over these tasks (see figure: 2.16).

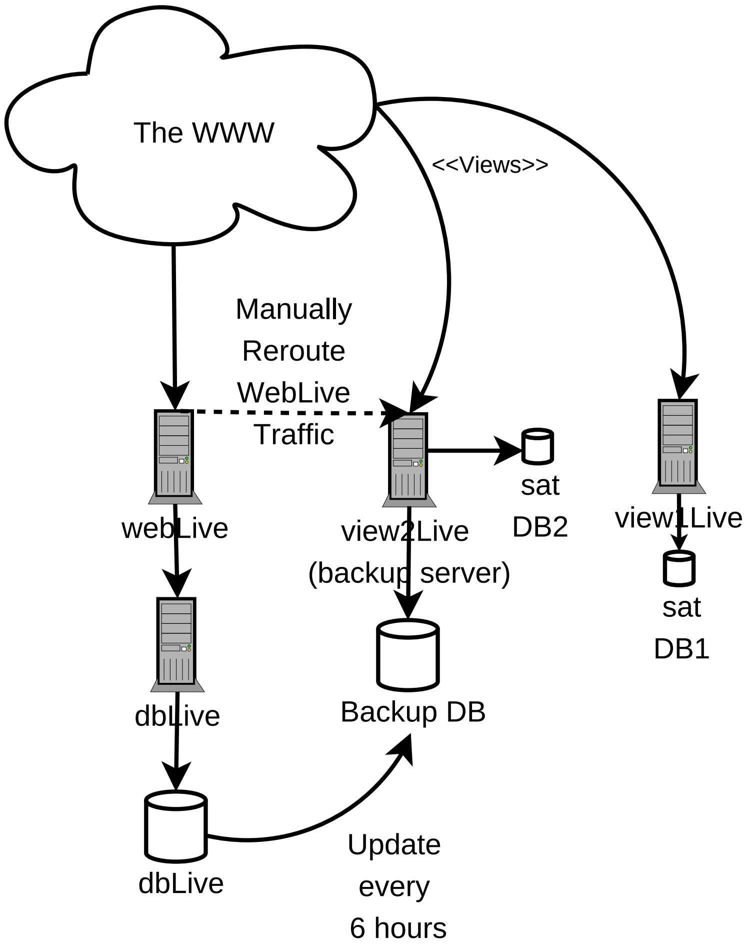


Figure 2.16: Simplified M4N Architecture

In a sense the main website forms a Single Point of Failure (SPF) for these three services. Likewise the main database server (dbaseLive) can be viewed as a SPF for the same services.

When taking a good look at figure 2.16 one sees that at the moment the main database server crashes M4N is forced to use data that is at most 6 hours old. Thus on average, a database crash results in 3 hours of data loss[5].

[5]In reality it is possible to manually restore clicks and leads which have been put in the main database already.

### 2.4.1 Financial Impact

Failure of the view, click and lead registration is the most undesirable type of system failure. It directly affects the revenue of M4N and its affiliates and stops offering banners for advertisers.

Say for instance M4N realizes a revenue of around 100,000 Euros a month from the website alone. This income is divided into three categories:

1. View-revenue: A negligible amount of cash.
2. Click-revenue: Totals to +-30% of the income.
3. Lead-revenue: Covers the remaining 70%.

Failure of the click and lead registration for some time $T$ thus implies that practically no revenue is generated during that time. The *direct* costs $C$ of down-time $T$, given a monthly revenue $R$ for these services are:

$$C = R \times \frac{T}{T_{month}}$$

Improving M4N's architecture which results in decreased yearly down-times can thus actually save money directly.

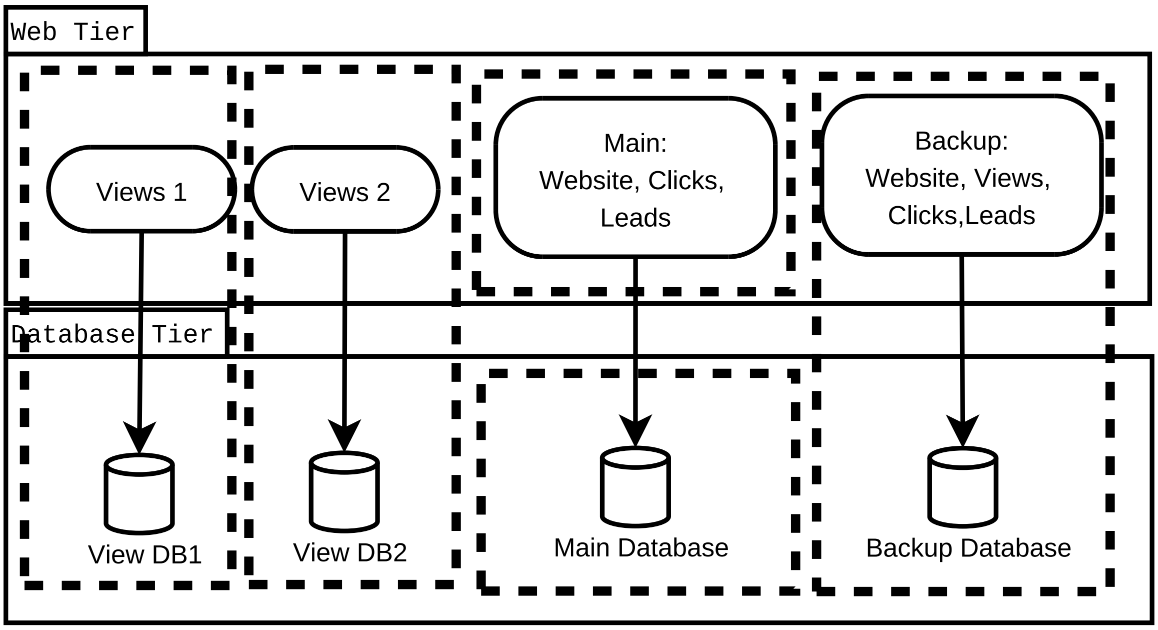
Apart from these direct revenue losses, system down-time also incurs costs in the form of having the systems administrator solve the problem. More importantly, down-time also introduces the cost of losing the confidence of affiliates and advertisers. Advertisers actually lose 'online advertising time', whilst affiliates directly miss revenues which they would otherwise have received from their banners. This last type of revenue loss is hard to measure but potentially the most dangerous effect of down-time. Losing customers (advertisers/affiliates) can have a long-lasting negative effect on the overall business of M4N.

Only in the case a database server burns down (or something equally dramatic) data is actually lost.

### 2.4.2 Single Points of Failure (SPF)

Some servers in M4N both act as database *and* web-server. We thus have to discern between the risk of failure of the separate services (web *or* database) and the risk of machine failure (web *and* database).

A graphical representation of the SPF's in M4N can be viewed in figure 2.17. From this figure we have extracted the following scenarios of things that can go wrong together with the implications and the actions taken to solve the problem:



The dotted lines group the services together according to which machine offers those services.

Figure 2.17: Simplified M4N architecture (SPFs)

1. One of the view server's web-server application crashes: Views will still be handled by the other view server. Normally a restart of the web-server application would be sufficient to recover.

2. Both view server's web-server applications crash: Views will not be registered until at least one of the web-server applications has been restarted. If this can not be done quickly enough one alternative is to register views on the backup system instead.

3. One view server burns down: Views will still be registered on the other view server. A new view server has to be bought or scavenged.

4. Both view servers burn down: No views will be registered until someone manually configures the backup server to register views as well or until new view servers are bought and installed instead.

5. Main database server crashes: The website will be down and no clicks and leads will be registered until this crash is fixed. The quickest solution would be to restart the database application (Postgres) or to reboot the entire server. In the case this can not be done quickly enough (e.g. the server can not be accessed) the remaining option is to configure

the main website to use the backup database instead. In this last case at most 6 hours of data is lost, which can be restored afterwards if this is deemed 'cost efficient'.

6. Main database server burns down: The website will be down and no clicks and leads are registered until another database server is used. The main website needs to be reconfigured to use the backup database server instead. In this case at most 6 hours of data is really lost.

7. Main website crashes: The website as well as the click an lead registration will go down. Restarting the web-server application would suffice in fixing this problem. In the case this still fails one can configure the backup web-server to take over for a while until all problems have been resolved.

8. Main systems are in repair, and backup systems fail: In this rare occasion it is possible to quickly transport one of the development machines to the hosting location to take over some of the tasks.

9. Fire or other disaster at the hosting location: In case of a fire at the hosting location all data of that day is lost and only a backup of the previous day is available at the office (other location). It is possible to transport one of the development servers to the same or a different hosting location.

# Chapter 3

# Bottleneck Solutions for M4N

The objective for removing the bottlenecks in M4N is to achieve increased *reliability* and *scalability* of the system. More concretely; lowering the risk of system failure will result in increased reliability for the services provided by that system. To make M4N more reliable it is therefore important to *minimize* the chance of down-time. One way to do so is to remove the single points of failure by introducing redundancy in the system.

Being able to extend the system more easily whenever more services are required (increased amount of views, clicks and leads to be registered) embodies the *scalability* objective. In other words; to have a system that is able to grow with the business itself. One way is to scale up by purchasing more expensive server systems, or to update the software. At a certain point however single system upgrades become increasingly more expensive and might not be a cost efficient option anymore. In these cases we need other methods of up-scaling, like for instance clustering. In our research we will in particular focus on this second type of scalability.

## 3.1 Removing SPF's: Clustering

To achieve both scalability and reliability we would like to propose some changes. For each server, which has a certain chance $P_c$ to crash we would like to have a second server doing exactly the same. The chance that two servers crash is significantly lower as that of only one server. Furthermore, when both servers can perform tasks *independently* from each other we can in fact do load balancing as well, realizing the scalability objective. The new chance of *'total failure'* now includes that both servers need to crash, instead of just a single one ($P_c^2$ is significantly less than $P_c$).

In our new envisioned future architecture we like to subdivide between the web-tier and the database-tier, the latter will be the focus of our research.

### 3.1.1 Web-Tier

Our suggestion on how to remove the SPFs in the web-tier is to simply run instances of the web-application software on multiple servers instead of just a single one (see figure: 3.1). In practice however, it has not been possible to have multiple instances of the M4N website running at the same time. The reason for this involved a banner update process which did not take in mind the possibility of multiple web-application instances.

Solving this is, according to various developers, a very time consuming job which has been on the 'todo-list' for quite some time. This task has not been finished because the need for new features has always superseded the need for web-application clustering.

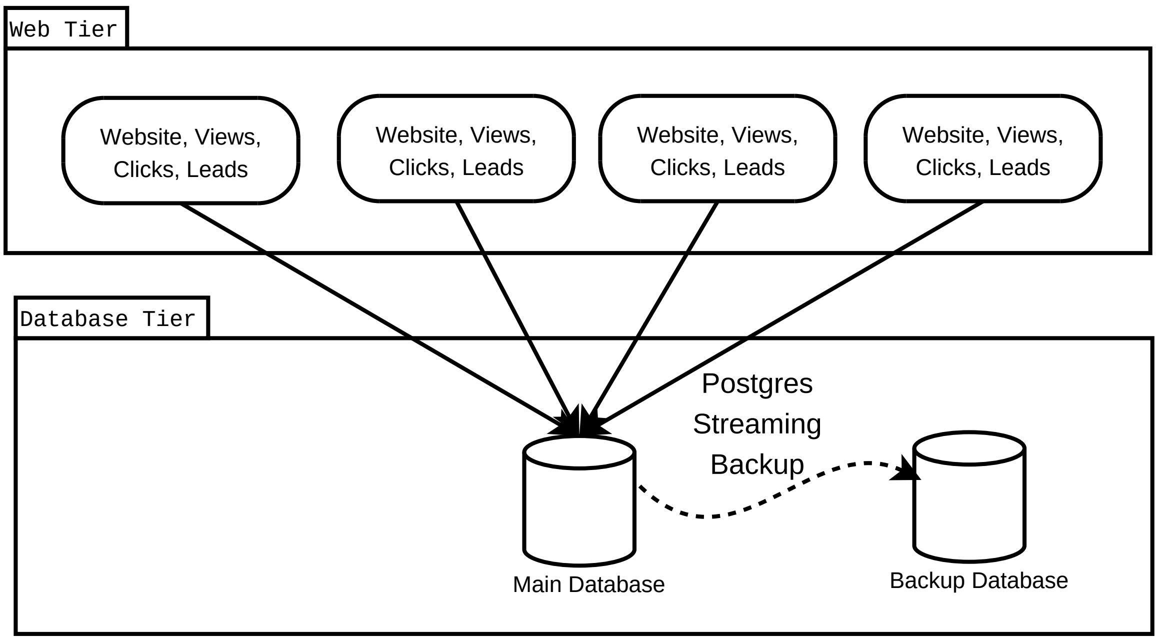

Web Tier
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Database Tier
Postgres Streaming Backup
Main Database
Backup Database

Figure 3.1: *Simplified* M4N Future System Architecture. Clustering the web-tier. By clustering the web-tier we potentially increase the reliability of the entire system. However, in the case the web-tier is not the determining factor for system failure, we gain only a little in reliability. Notice that the main database right now forms the only SPF.

### 3.1.2 Database-Tier

Clustering the web-tier is not enough for a fully redundant and reliable system. Another potential single point of failure is the database system itself. Configuring a cluster of databases can solve this problem. Having a system rely on two databases instead of just one increases reliability. In case one of the databases crash, the other will automatically be used instead.

Several solutions exists for applying database distribution. One of which is to severely adapt the web-application software and to already cluster the information on a logical level (see figure 3.2). This however increases the software complexity because each web-application instance needs the logic to maintain a connection with a multitude of database systems. Therefore we prefer to use a more generic database clustering solution. This type of solution acts as if there were just a single database system which *transparently* accesses a multitude of database systems (see figure 3.3).

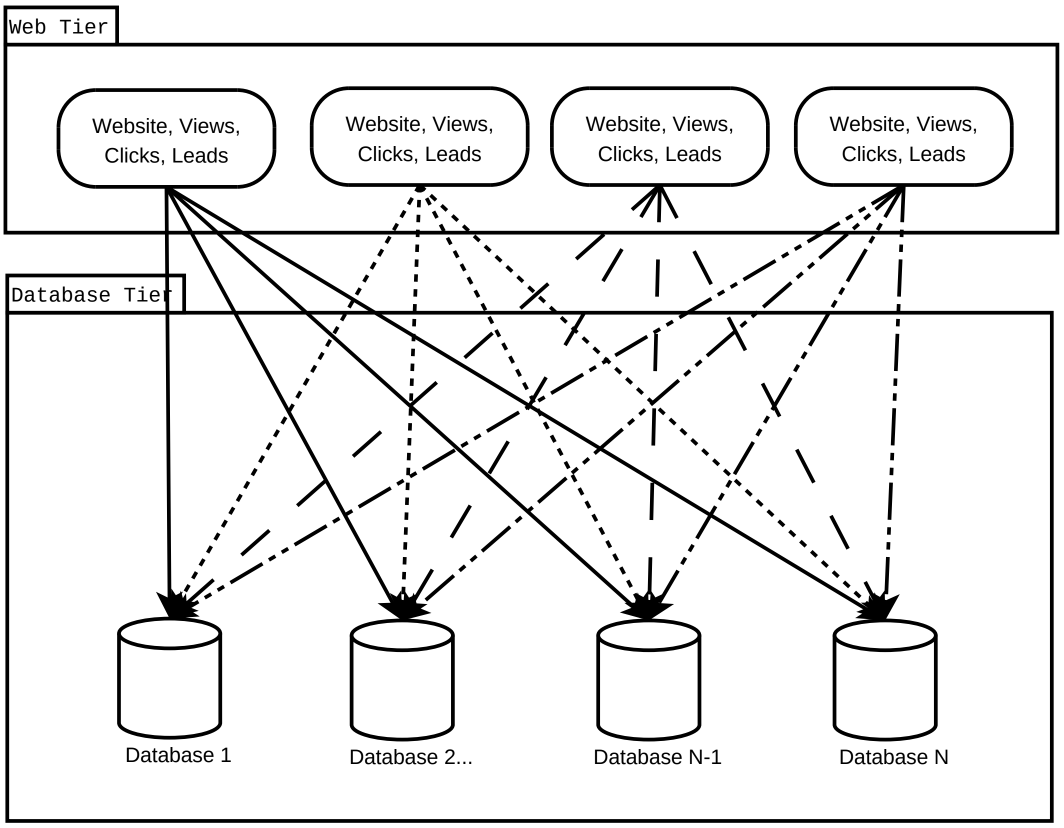

Web Tier
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Database Tier
Database 1
Database 2...
Database N-1
Database N

Figure 3.2: *Simplified* M4N Future System Architecture. Clustering the web- and database-tier

## 3.2 Available Solutions

What should M4N do to keep up with growing performance demands and to increase reliability? One certain thing to do is separating click and lead registration from the main website and getting the M4N web-application running on multiple machines without any problems. Our research however focuses on the database aspect of the problem. We now present a list of options we considered for achieving a more scalable and reliable database backend.

### 3.2.1 Postgres On-line Backup and Point-In-Time Recovery (PITR)

A specific Postgres feature, primarily intended for crash recovery, can also be used to incrementally keep a backup database up to date. Postgres maintains a *write ahead log* (WAL) file which can be configured to start a specific script after such a log file is written. With this script it is for instance possible to copy the WAL files over to a remote backup server and to perform an incremental backup (crash recovery). Specifying a maximal WAL file size of only a few kilobytes ensures you of very frequent updates of the backup database system. In a sense this almost resembles a streaming backup of the database system. Sadly there are some minor issues with this feature which will hopefully be fixed in a next version [36].

This streaming backup feature does not fix the need for manual intervention once the main database crashes. But it does decrease the potential data loss suffered from a crash. Furthermore a system load decrease is expected because no intensive database dumps have to be

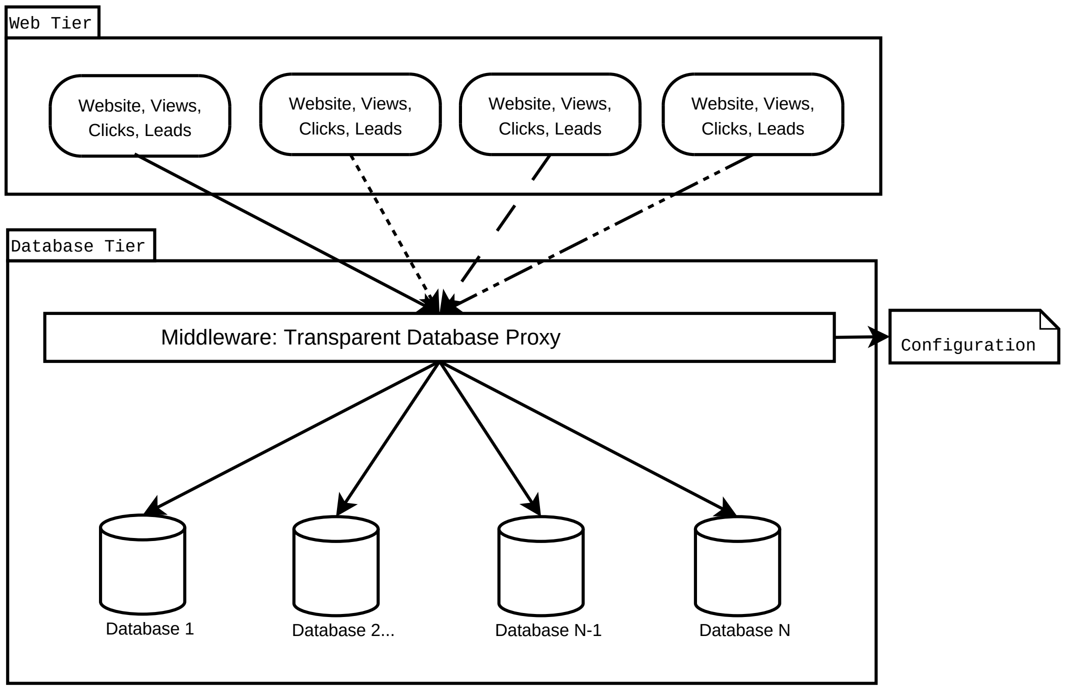

Web Tier
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Website, Views, Clicks, Leads
Database Tier
Middleware: Transparent Database Proxy
Configuration
Database 1
Database 2...
Database N-1
Database N

Figure 3.3: *Simplified* M4N Future System Architecture. Clustering the web- and database-tier using middleware

performed anymore[1].

### 3.2.2 Oracle Real Application Clusters (RAC)

Oracle offers a complete and mature database clustering solution and was one of the options we considered. The only down-sides are that Oracle is very pricy package which might offer too much for a small company. Another problem might be minor incompatibilities on the SQL level. Not all database systems implement the SQL standard completely and sometimes a particular query might work on Postgres but not on other database systems. A final hurdle is the fact that the systems administrators and developers have to manage and work with a new kind of database server. This would cost time and money, a resource which would already be stretched very thin by Oracle licensing costs.

### 3.2.3 Sequoia

Database clustering can also be performed using less expensive or free alternatives. During our research we will try out whether Sequoia (previously known as CJDBC) has grown to the point in which it can be used in a production environment. Although there are some other alternatives we have chosen for Sequoia. Right now (1st July 2006) it claims to be stable enough for production and it seems to offer good administration support together with lots of useful features.

[1]The database-tier of figure 3.1 depicts this particular situation.

Chapter 4

# Simulated Load Test Generation

The objective for our load tests is to measure and compare the speed and reliability of a multitude of different database solutions under similar loads. The intended purpose of our load test is therefore to act as a benchmark which, in the ideal case, actually resembles real system usage.

By simulating real traffic (users, view, click and lead registration etc.) we get a reliable indication on how database system changes will affect the application [22]. For generating these load tests we considered the following *alternatives*:

1. SQL level: Using the SQL log files we replay SQL queries on the database in the same amounts and sequences as they are run on the live system. The generated load test would in this case just test the database system.

2. (Semi-) Automatic browser test generation: Using web access log files we extract the browsing behavior of web users. This type of load testing actually simulates real users on the entire system, rather than just testing the database system.

3. Manual browser test generation: Before deploying a new version of M4N, employees are asked to manually check the web-pages which are relevant to them and their customers. To formalize this process a list has been made of scenarios which have to be executed. By manually converting these scenarios into scripted browser events we can create load tests as well.

Although SQL level testing might have been simpler to implement, the choice was made to do a browser level load tests. The main advantage of browser based tests is that they, apart from the database system, also tests the web application code. These tests could in fact become functional tests as well. Because there were still no real functional tests present in M4N, this load testing project became *the* opportunity to start creating some.

For this functional testing we have chosen the JUnit Testing Framework [1]. We thus have to write our load tests in a similar way as we would write JUnit tests to maintain interoperability (JUnit tests can be used for our load testing and vice versa).

The next choice was between the log file based browser tests and the manually crafted ones based on the predefined scenarios. We chose for the log file based tests. A big advantages of log based testing is that it reflects the actual live system usage. Nevertheless, creating some handmade tests next to the automated ones can still prove useful. Both methods have been reported to complement each other to increase the test coverage [24].

[1]Other packages were considered as well such as Canoo WebTest and Selenium [9, 10]

## 4.1 Response Dependency Problem

When generating load tests from web access log files we only know the timings of the HTTP user requests. Just replaying these request with exactly the same timings as in the logs would seem to be sufficient (log replay). Doing so however could potentially lead to unexpected results. User actions (HTTP requests) in some cases also depend on responses given by the server (see figure 4.1 for an example).

For accurate load testing it is therefore imperative, to use web server responses as a basis for what request to do next. We thus need to get the web response and parse the enclosed HTML document. Then a programmer needs to pinpoint which HTML element contains information for the upcoming request[2]. In the case of figure 4.1 a programmer needs to extract the value X from one of the zones on myZones.jsp and put that value into the next 'gotoZone' request.

**Example session:**

1. Request: Create new advertising zone: POST createZone.jsp
2. Response: createZone.jsp displays success message and a link to myZones.jsp
3. Request: On the createZone.jsp page we click the link: GET myZones.jsp
4. Response: My advertising zones, page containing a list of the user's advertising zones including the newly created zone $X$: myZones.jsp
5. Request: On the myZones.jsp page we click on the just created zone: GET gotoZone.jsp?zoneID=$X$

In a simulation where we only imitate previous *GET* and *POST* commands, we would not be able to find the correct link for zone $X$ which has been generated dynamically. To obtain a correct link we also need to look at the responses coming from the web-server.

Figure 4.1: Example sessions showing dependencies between separate requests.

[2]In an ideal case one wants to emulate browser usage by actually clicking on links just like a real user. This information however can not be extracted from traditional web-logs.

## 4.2 Load Test Requirements & Approach

After discussing the load testing problems with the developers at M4N we have come to a list of requirements (see appendix C for initial project plan).

1. Resemble the real load on the live servers in which three types can be identified:
    - Web site users: Simulate real users. Generating these user tests should be automated preferably from the web access log files.
    - View, Click and Lead registration
    - System Processes: Script performing heavy queries on the database system, e.g. banner updates and statistics generation.
2. Load test must be configurable to fit several future scenarios.
3. Load test have to be in JUnit test format.
4. Test must be executable from multiple clients at a time.
5. Browser simulation: A virtual browser needs to parse the web-page first. Many M4N pages depend on java-script or are dependent on previous page responses, see fig 4.1.

### Load Test Approach

Our load testing framework can be divided into three separate parts; being the website user simulation, the view/click/lead registration and the system processes. We will first dive into the user simulation part, in which we will try to mimic web user behavior based on web access logs.

We then continue with the view, click and lead registration which in fact is just a simple instantiation of the user simulation. E.g. registering a lead is just simulating a user who views a banner, clicks on it and after a while generates a lead. Finally we move to the system processes.

## 4.3 Load Test Generation: User Simulation

We decided to use M4N's custom access log files to obtain typical user sessions. From these we intend to automatically generate functional web tests which can be used for load testing.

Using a regular expression we filtered out the lines of interest, which are characterized by two consecutive lines. The format of these lines is shown in figure 4.2 while figure 4.3 shows which precise data we extracted. See the following list for an explanation of each extracted variable:

- **IP:** IP address of web client.
- **sessionID:** Unique session ID of the web client.
- **username:** Username in case the user has logged in, will be <anonymous> in the other cases.
- **dateTime:** Date and time of the request.
- **HTMLmethod:** Whether the request was of type GET or POST.
- **pageName:** Name of the requested page; e.g. index.jsp
- **GETparameters:** List of GET parameters send with the request.
- **GETandPOSTparameters:** List of GET and POST parameters send with the request.

```
HTTP POST example (typical login web log entry)
66.197.6.234-JGNMNBNMAALK-Oct 30, 2005 3:00:05 AM com.mbuyu.m4n.filters.AuthenticationFilt
INFO: "POST /index.jsp?username=webmaster@somecompany.com&password=secret HTTP/1.1" "usern

HTTP GET example (banner click web log entry)
213.41.87.195-EF5AD0A00FCFAFCB1F9D90062CC819E6-Oct 30, 2005 3:03:44 AM com.mbuyu.m4n.filte
INFO: "GET /genlink.jsp?AffiliateID=190&bannerid=2822 HTTP/1.0" "AffiliateID=190&bannerid=

HTTP GET example (some specific M4N page)
82.134.153.201-09C2E5E50C1DB226BBE03C860D5DD730-rubriek-Oct 30, 2005 7:37:33 PM com.mbuyu.
INFO: "GET /statsaffiliate_orders_per_merchant.jsp HTTP/1.1" "" "http://www.m4n.nl/index.j
```

Figure 4.2: Typical M4N Access Log entries

**IP-sessionID-username-dateTime** *com.mbuyu.m4n.filters.AuthenticationFilter doFilter INFO: "***HTMLmethod /pageName?GETparameters** *HTTP/1.1" "***GETandPOSTparameters***" "referrer" "browser" "pageName" "" "" ""*

Figure 4.3: Template for extracting data from M4N web access log entries

### 4.3.1 Session Extraction

Because the log itself already contains session IDs it becomes pretty trivial to extract all the separate sessions. Per session, we will store each log-entry in chronological order with in between a call to the sleep function. This way the load test will wait an amount of milliseconds equal to the time between the current and the next entry (see fig: 4.4 for an example loadtest).

```
public void loadtestSession1() {
    response = request.get("index.jsp");
    sleep( 8000 );

    response = request.get("createzone.jsp");
    sleep( 47000 );

    response = request.post("createzone.jsp?params");
    sleep( 11000 );

    response = request.get("myZones.jsp");
    sleep( 6000 );

    response = request.get("gotoZone.jsp?zoneID=22" );
    sleep( 29000 );

    // rest of session...
}
```

Figure 4.4: Pseudocode example on how a basic generated load test would look like.

### 4.3.2 User Class Clustering

One of the requirements was the ability to configure the load per user type. Apart from being able to set the overall testing intensity we must also be able to alter the amount of admin users independently from the amount of affiliates or advertisers. For each test session we thus need to know with what type of user we are dealing with.

Luckily, each relevant entry in the web access logs also contained the username of the involved user. With this username we can retrieve the so-called 'user role' from the database [3]. This identifier sort-of indicates what type of user we are dealing with. As a first clustering step we *considered* using the following user roles:

- 0 = Un-subscribed (will not be found in log files because they can not log in)
- 1 = General (anonymous)
- 2 = Affiliate
- 3 = Advertiser
- 4 = TailorMade
- 5 = Admin
- 6 = Administration
- 7 = Finacial
- 8 = m4dart
- 9 = System Administrator
- 101 = Newsletter
- 115 = Newly Registered Users
- ... several other less relevant 'roles'

We noticed that single user can be a member of different user groups. Some 'user roles' were not even real roles in the sense of different user types. For instance the 'newsletter' role which only indicates that the user likes to receive the weekly newsletters from M4N.

The most common double-role is that of users which have the role of advertiser and affiliate. Because almost all advertisers are affiliates as well but not the other way around we have chosen to classify this type of user as advertisers.

By prioritizing these roles we have come to a user role priority graph (see fig 4.5) which will be referred to as the *user classes.* The most outer classes have precedence over the inner ones.

We will use the user classes as the basis for our first clustering step. Based on this clustering we can easily increase traffic for a specific user class by just increasing the amount of traffic to the requested user class.

[3]for this we have used database dump m4n_20060424

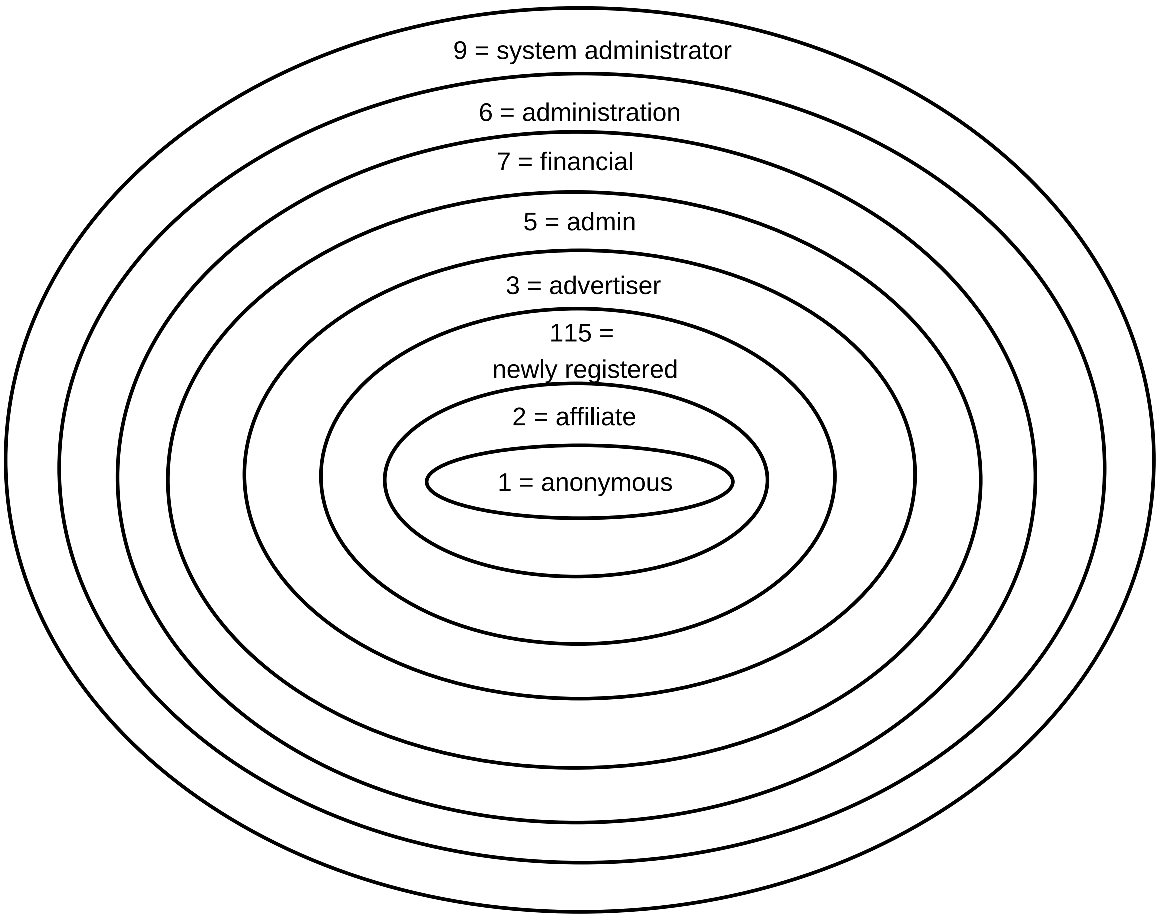


Figure 4.5: User classes used for primary clustering

### 4.3.3 Session Clustering: Typical User Sessions

Each 'user class'-cluster of sessions still contains large amounts of individual user sessions. Before we can use a session for the load test we (programmers) need to check whether we have to pass some variables or links from one response to upcoming requests (see figure: 4.1 for the concept and compare figures 4.4 and 4.6 to see how this intervention would look like in pseudocode). Another example session (see figure 4.7) shows the problems of unique user names. To decrease the amount of manual labor we would like to limit the amount of sessions we have to check. We have chosen to use a clustering algorithm to extract the most *'typical user sessions'* and use those for the load test. We aim to generate a total of around 100 to 150 of these sessions divided between the eight user classes (see figure 4.8).

```
public void loadtestSession1() {
    response = request.get("index.jsp");
    sleep( 8000 );

    response = request.get("createzone.jsp");
    sleep( 47000 );

    response = request.post("createzone.jsp?params");
    sleep( 11000 );

    response = request.get("myZones.jsp");
    sleep( 6000 );

    // programmer intervention
    // zoneID is not the same for each session!
    respons = request.get("gotoZone.jsp?zoneID=22");

    int zoneID = extractVariableFromResponse( response, "zoneID" );
    response = request.get("gotoZone.jsp?zoneID=" + zoneID );
    sleep( 29000 );

    // rest of session...
}
```

Figure 4.6: Pseudocode example on how a generated load test would look like after some programmer intervention.

```
response = request.get("index.jsp");
sleep( 8000 );

response = request.get("register_new_user.jsp");
sleep( 47000 );


String uniqueUser = getUniqueUserName();    // get unique username

response = request.post("confirm_registration.jsp?username=" + uniqueUser + "&otherpar
sleep( 11000 );

response = request.get("registration finished.jsp");
sleep( 6000 );

// rest of session...
```

Figure 4.7: Pseudocode example of a register new user session in the load tests

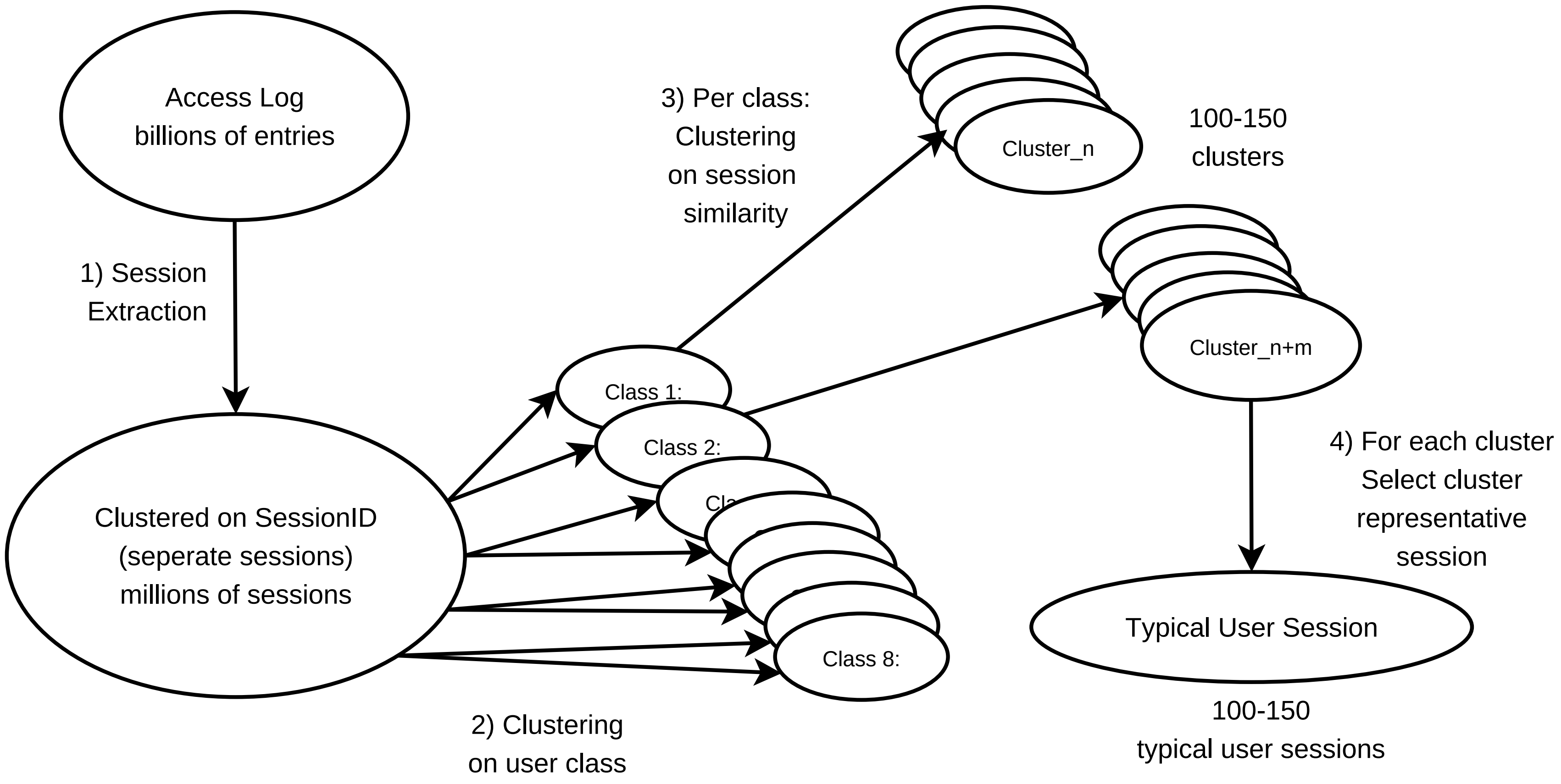


Figure 4.8: Clustering: From ordinary session to typical user session

### 4.3.4 Clustering Algorithm: K-means

We have used the K-means algorithm for clustering our sessions [14, 15]. After trying out different values for $K$ we finally chose one based on the number of sessions ($N$) to be clustered ($K = N^{0.4}$).

Using the K-means clustering algorithm, like any other clustering algorithm, requires a measure for calculating the distance (or inversely the similarity) between the items which have to be clustered. For our session comparison several criteria were considered:

1. Site structure[4]: Pages in the same directory often have related functionality [21, 27]. Example: Which set is more similar, (a,b) or (a,c)?
   (a) `/click_management/index.jsp`
   (b) `/click_management/view_on_hold.jsp`
   (c) `/index.jsp`
2. Pages visited:
   - Only consider the name of the page: doaction.jsp is different from index.jsp.
   - Take page parameters into account as well: `doaction.jsp?cmd=clickbanner` and `doaction.jsp?cmd=viewbanner` share the same page name, but have different parameters.
3. Relative order of pages visited: Visiting pages in the same order as another session makes them more similar [31].
4. Session length (number of pages visited)
5. Session duration: total time spent on the website in this session.

We chose to use page names and session length as our main criteria for calculating the distance between sessions. To save time we started off with this limited comparison function which directly gave nice clustering results. For future research one might consider experimenting with the other criteria as well.

**Limitations:**

The K-means algorithm needs a set of $K$ points defining the cluster centers upon initialization. Thus the quality of the clustering is determined by the choice of these centers. Ideally one wants to have cluster centers which have the least in common with each other (big distances between each other).

In our case we just randomly selected $K$ individuals to form these cluster centers. For future research one could improve this by selecting the top $K$ distances between sessions although this might not guarantee improvement. As an alternative one can use an adapted version of K-mean; the Fuzzy K-means algorithm. The Fuzzy variant has however the same limitations because it is likewise dependent on the chosen weights of the first iteration [14, 15].

Other techniques under consideration were: The Competitive Agglomeration for Relational Data (CARD) algorithm [27], tree structure and cube model based representations for sessions [28, 32], Kohonen self organizing maps to use for data clustering[30, 33, 18] and others [35, 34].

[4]M4N hardly uses any directory structure at all making this criterion virtually useless

### 4.3.5 Clustering Results: Which Sessions to Select?

After clustering the sessions in each user class we came to the challenge of selecting which session(s) to select from each cluster. Our initial expectations were that each cluster would solely consist out of similar typical session which could easily be identified as such. Only after careful study of the separate sessions in each cluster we had to conclude that this was not entirely the case.

In some clusters session diversity existed and we had to select the most common two or three sessions from those clusters. Nevertheless, clustering did seem to work sort-of as expected with the only downside that we had to do some manual selection.

Please refer to appendix A for more detailed information about the clustering application's usage and its multi-threaded performance.

### 4.3.6 Browser Emulation

In all the fore-mentioned load test examples we merely did HTTP requests and caught the raw HTTP responses from which we in some cases extracted needed variables (example fig: 4.6). For a proper browser simulation however more needs to be done. HTML has to be parsed, cookies should be handled and java-script must be executed. Instead of doing this work ourselves we have chosen to use the web browser of HTMLUnit (WebClient). The WebClient is a Java component which can be controlled with simple commands from Java code [8]. As an example we have converted the loadtest shown in figure 4.6 into a version which uses the WebClient (see fig: 4.9).

```
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlPage;

public void loadtestSession1() {
    WebClient browser = new WebClient();
    HtmlPage page = null;

    page = browser.getPage( new URL("index.jsp") );
    sleep( 8000 );

    page = browser.getPage( new URL("createzone.jsp") );
    sleep( 47000 );

    page = browser.postPage( new URL("createzone.jsp?params") );
    sleep( 11000 );

    page = browser.getPage( new URL("myZones.jsp") );
    sleep( 6000 );

    // programmer intervention
    // zoneID is not the same for each session!
    page = browser.getPage( new URL("gotoZone.jsp?zoneID=22") );

    int zoneID = page.getElementByID( "zoneID" );
    page = browser.getPage( new URL("gotoZone.jsp?zoneID=" + zoneID) );
    sleep( 29000 );

    // rest of session...
}
```

Figure 4.9: WebClient pseudocode example on how a generated load test would look like after some programmer intervention.

### 4.3.7 Final JUnit Session Format

Once sessions are created and edited manually (for passing GET/POST parameters) these can be used for load testing. However, upon the introduction of each new M4N version some sessions might be no longer valid. Numerous of changes in M4N can severely affect the correctness of the load-testing sessions; like for instance changed page names or newly added pages. To resolve these issues, programmers again need to search the session files and edit them by hand.

A better option, in our opinion, is to re-create the loadtest sessions altogether using a newer log file (from a development server for instance). This still leaves the need for manually editing the sessions which possibly forces us to re-do a lot of work. To reduce that amount of work and to further automate the manual editing process we adapted the JUnit session file format a little and added a configuration file (see: *codegeneration.props* in the web-log clusterer). The new file format is intended to facilitate a more object oriented approach in determining which value has to be assigned to certain GET and POST parameters. Our approach is to replace each of these parameters in the URLs with calls to the *getText()*-function of so-called *'Handler classes'*. (see figure 4.10 for an example).

```
@LoadTest(123) public void testSession123()
throws Exception {
    WebClient webClient = getBrowser();
    WebRequestSettings webRequest = null;

    parameter.handlers.FromLogHandler submitlogin0 = (parameter.handlers.FromLogHandler)
        Class.forName("parameter.handlers.FromLogHandler").newInstance();
    submitlogin0.setLogValue("login%20>>");

    parameter.handlers.FromLogHandler username1 = (parameter.handlers.FromLogHandler)
        Class.forName("parameter.handlers.FromLogHandler").newInstance();
    username1.setLogValue("user@somewebsite.nl");

    parameter.handlers.FromLogHandler password2 = (parameter.handlers.FromLogHandler)
        Class.forName("parameter.handlers.FromLogHandler").newInstance();
    password2.setLogValue("bigsecret123");


    webRequest = new WebRequestSettings(
        new URL("http://"+ siteurl + "/index.jsp?submitlogin=" + submitlogin0.getText() +
            "&username=" + username1.getText() +
            "&password=" + password2.getText() + "") );
    webRequest.setSubmitMethod( SubmitMethod.POST );
    HtmlPage page0 = (HtmlPage) webClient.getPage( webRequest );
}
```

Figure 4.10: Final file format for our JUnit Test sessions.

By passing each Handler the log-file value as well as the previous page it is possible to either use the log file value or to extract some other arbitrary value from the previous page. It is thus required to create a new handler for each different situation. Still, the question remains on how this will reduce the amount of work which needs to be re-done? To answer this question we take a look at the code generation cycle of the web-log clusterer.

During session generation, the web-log clusterer looks at our *codegenerations.props* file (see figure: 4.11), which contains mappings between pages, parameters and Handler classes. Basically showing which page- and parameter name map to which Handler class. It is thus possible to define general rules on which handler to use for certain situations (page, and GET/POST parameter name combinations). Defining those rules once will save a lot of manual work later on.

```
# properties file for code generation

# format: pagename,*=handler
# format: *,variable=handler
# format: pagename,variable=handler

# default handler
*,*=parameter.handlers.FromLogHandler


# Example entries:
#index.jsp,username=parameter.handlers.UsernameLoginHandler
#index.jsp,password=parameter.handlers.PasswordLoginHandler
```

Figure 4.11: Code generation properties file (codegeneration.props)

We have chosen to use the log-file value as the default action for all parameters. The handler performing this simple operation is our FromLogHandler (see figure 4.12 for the abstract Handler class, and figure 4.13 for the FromLogHandler). Its function is to simply return the log-file value on every *getText()* request.

Other handlers can be added at for other functionality. In the case of the zone-ID examples, a ZoneIDHandler could be written to extract the zone-ID from the previous page and to return that value instead of the log value. In a sense these Handlers are little plug-ins for the session-code generation process.

```
public abstract class Handler {
    protected static Page previousPage = null;
    protected String logValue = null;

    public Handler() {
    }

    /** Set the previous page visited **/
    public static void setPreviousPage( Page previousPage ) {
        Handler.previousPage = previousPage;
    }

        public void setLogValue( String logValue ) {
        this.logValue = logValue;
    }

    // The value which should be inserted
    public abstract String getText();
}
```

Figure 4.12: The abstract Handler class, the basis for each handler

```
public class FromLogHandler extends Handler {
    public FromLogHandler() {
    }

    // The value which should be inserted
    public String getText() {
        return(logValue);
    }
}
```

Figure 4.13: The default handler: FromLogHandler

## 4.4 Load Test Generation: View, Click and Lead Registration

The most critical part of M4N are the view, click and lead registration services. Failure of these services will directly affect revenue. Users who trigger these events are mostly not visitors of the M4N website. In this case we do not want to rely on the web access logs to extract these types of system usages. Loadtest users need to be able to define the amount of views, clicks and leads that will be performed during a loadtest (see fig 4.14 on how to configure).

```
loadtest.viewrate=100
loadtest.view2clickratio=50
loadtest.click2leadratio=200

loadtest.shortlink=50
loadtest.longlink=50
```

Figure 4.14: loadtest.properties file: Defines the view, click and lead testing intensity for our load tester. Notice that we have set it to generate 100 views a second while we generate a click once every 50 views. Likewise after around 200 clicks a lead will be generated. Because there are two distinct types of clicks which can be generated in M4N we can configure the test intensity independently (short vs. long link).

### 4.4.1 Banner Selection

Before simulating views, clicks and leads one first needs to know which banners to use. First we made a list of all[5] the methods in which users of M4N could generate views, clicks and leads. From this list we then concluded that we needed matching pairs of *AffiliateIDs*, *zoneids* and *offids*.

- Views: Uses the AffiliateID and zoneid although zoneid would have been sufficient. The AffiliateID was added to minimize database interaction. URL: `http://views.m4n.nl/genlink.jsp?AffiliateID=1234&zoneid=56789`.
- Clicks, Short links: Only a single zoneid is used (_b='zoneid'). The click will be registered but this does generate more load on the database server compared to the long link type. URL: `http://clicks.m4n.nl/_r?_b=12345`.
- Clicks, Long links: Contain next to the zoneid (called bid_id), a garbled AffiliateID is added to the link. Long links generate less database load as the short links. URL: `http://clicks.m4n.nl/_r?affidsecure=un1234567m6b3EvsbaEnd&m_bid=89012&mbuyuurl=http%3A%2F%2Fwww.advertiser_website.nl%2F%3Fadcampaign2%3Dm4n`.
- Leads: Most easily placed in image sources

[5]There are some other types of clicks and leads as well. These however are for the server tier equivalent to one of the four classes listed here.

```
<img src="http://www.m4n.nl/_l?trid=nocookie&offid=1234&
description1=uniquenumber&description2=priceofpurchase&description3=text1"
width="1" height="1" border="0">
```

We have extracted usable ID's from the M4N database and have put them in a comma separated file (see *AffiliateID.zoneid.offid.csv* in the loadtest scheduler). These values will be used for the view, click and lead simulation during our load tests.

### 4.4.2 Generated Loadtest Sessions

The following loadtest sessions have been generated for the view, click and lead registration. Please notice that for each call we need to pass a new set of ID's.

#### Views

To generate a view one only needs to do a single page request to the server (figure: 4.15).

```
@LoadTest(10000) public void testViewSession1( Integer AffiliateID, Integer zoneid, Intege
    throws Exception {
        browser.getPage( new URL( "http://" + viewurl + "/genlink.jsp?AffiliateID=" +
            AffiliateID + "&zoneid=" + zoneid + "&xml=true") );
    }
```

Figure 4.15: Loadtest for view session

#### Short-links

Generating a click is not just clicking a link. For an accurate simulation we first actually view the banner and wait for some time before we click on it (figure: 4.16).

```
@LoadTest(20000) public void testShortLinkSession1( Integer AffiliateID, Integer zoneid, I
    throws Exception {
        WebClient browser = getBrowser();

        HtmlPage page1 = (HtmlPage) browser.getPage( new URL( "http://" + viewurl + "/genl
            AffiliateID + "&zoneid=" + zoneid + "&xml=true") );

        // At least 4 seconds before we click -> at most 14 seconds
        Thread.sleep( (long) (4000 + 10000 * Math.random()) );

        HtmlPage page2 = (HtmlPage) browser.getPage( new URL("http://" + clickurl + "/_r?_
}
```

Figure 4.16: Loadtest for short link session

### Long-links

For the same reason mentioned as with the short links: For an accurate simulation we first actually view the banner, wait some time and then we click it (figure 4.17). In the long link case however the click URL itself is returned when viewing the banner. It is thus possible to use the response (page1) and click on the first available link (clickLink).

```
@LoadTest(30000) public void testLongLinkSession1( Integer AffiliateID, Integer zoneid, In
    throws Exception {
        WebClient browser = getBrowser();

        HtmlPage page1 = (HtmlPage) browser.getPage( new URL( "http://" + viewurl + "/genl
            AffiliateID + "&zoneid=" + zoneid + "&xml=true") );

            // At least 4 seconds before we click -> at most 14 seconds
            Thread.sleep( (long) (4000 + 10000 * Math.random()) );

            if( page1.getAnchors().size() > 0 ){
                HtmlAnchor clickLink = (HtmlAnchor) page1.getAnchors().get(0);
                HtmlPage page2 = (HtmlPage) clickLink.click();
            } else {
                // No link -> The banner of this ID does not contain click-link
            }
    }
```

Figure 4.17: Loadtest for long link session

**Leads**

Purchasing a product often takes some time sometimes even after a web session has expired. That is why M4N uses browser cookies to still be able to register purchases coming from one of their links. These cookies values are set upon clicking a M4N banner and are needed for the proper functioning of the lead registration. Likewise our loadtest session first performs all the steps a normal user would perform when purchasing something via M4N (figure: 4.18).

```
@LoadTest(40000) public void testLeadSession1( Integer AffiliateID, Integer zoneid, Intege
    throws Exception {
        WebClient browser = getBrowser();

        HtmlPage page1 = (HtmlPage) browser.getPage( new URL( "http://" + viewurl + "/genl
            AffiliateID + "&zoneid=" + zoneid + "&xml=true") );

        // At least 4 seconds before we click -> at most 14 seconds
        Thread.sleep( (long) (4000 + 10000 * Math.random()) );

        HtmlPage page2 = null;
        if( page1.getAnchors().size() > 0 ){        // long link code
            HtmlAnchor clickLink = (HtmlAnchor) page1.getAnchors().get(0);
            page2 = (HtmlPage) clickLink.click();
        } else {            // short link code
            page2 = (HtmlPage) browser.getPage( new URL("http://" + clickurl + "/_r?_b=" +
        }

        // Purchases take long: Between 60 and 120 seconds before we generate lead
        Thread.sleep( (long) (60000 + 60000 * Math.random()) );

        HtmlPage page3 = (HtmlPage) browser.getPage( new URL("http://" + leadurl + "/finis
            "trid=nocookie&offid=" + offid) );
    }
```

Figure 4.18: Loadtest for lead session

## 4.5 Load Test Generation: System Processes

M4N knows many system processes. From the standard ones which are run within the operating system to the more specific M4N related SQL queries which have to be run once an hour or daily.

- Sending mails
- Calculating statistics
- Detecting click-fraud

To support at least the database load generating processes we have added basic support by providing the option to invoke any SQL queries with a given delay. These queries must be stored in a file and can be configured using the loadtest.propterties file (see figure 4.19). We have tested its functionality but will not use this feature during our load test benchmarks.

```
loadtest.dburl=jdbc:postgresql://databaseurl.m4n.nl/m4n_20060424
loadtest.dbdriver=org.postgresql.Driver
loadtest.dbuser=postgres
loadtest.dbpass=passwordX

# Start up queries with a given delay
loadtest.sysjob.job1.query=query1.sql
loadtest.sysjob.job1.delay=10000
loadtest.sysjob.differentjob.query=query2.sql
loadtest.sysjob.differentjob.delay=0
```

Figure 4.19: Example on how one can configure the loadtest scheduler to run a SQL query with a defined delay.

# Chapter 5

# Load Test Execution

One of the requirements was to execute the load test from multiple clients and thus IP addresses. Previous testing experiences at M4N showed that a single client was not always capable of generating enough web traffic to do realistic testing. Moreover we had to circumvent the *'request access limiter'* component build in M4N, which limits the amount of requests that will be handled per IP address.

From a range of alternatives [1]. we have chosen to use the Grinder as the load test scheduling application.

## 5.1 The Grinder

The Grinder is a Java load testing framework which easily enables users to run test scripts in many processes among many machines. The Grinder mainly consists of two applications, one which is used for running the tests (Agent), the other for controlling and monitoring the Agents and their test results (Console).

In a typical Grinder deployment scenario one installs and runs a single Agent application on each test-client machine. Upon startup the agents look in a *grinder.properties* file containing the IP-address and port where they can find and connect to the Grinder Console. Using this console users can send scripts to the agents and tell them to start running them. In turn, the agents communicate back test results back to the grinder.

Apart from relaying where the console can be found, the *grinder.properties* file also tells the Grinder Agent how many simultaneous tests it should run. When running tests each Grinder Agent starts a specified amount of processes which on their turn start an amount of threads. Each thread will independently (except when the script is programmed to communicate with other threads) run the test script (see figure: 5.1). The total number of tests which are run simultaneously is equal to the amount of processes times the amount of threads times the amount of running Agents [6, 7].

[1]We made our decision based on previous experiences and a comparison between load testing tools [11, 12, 13]

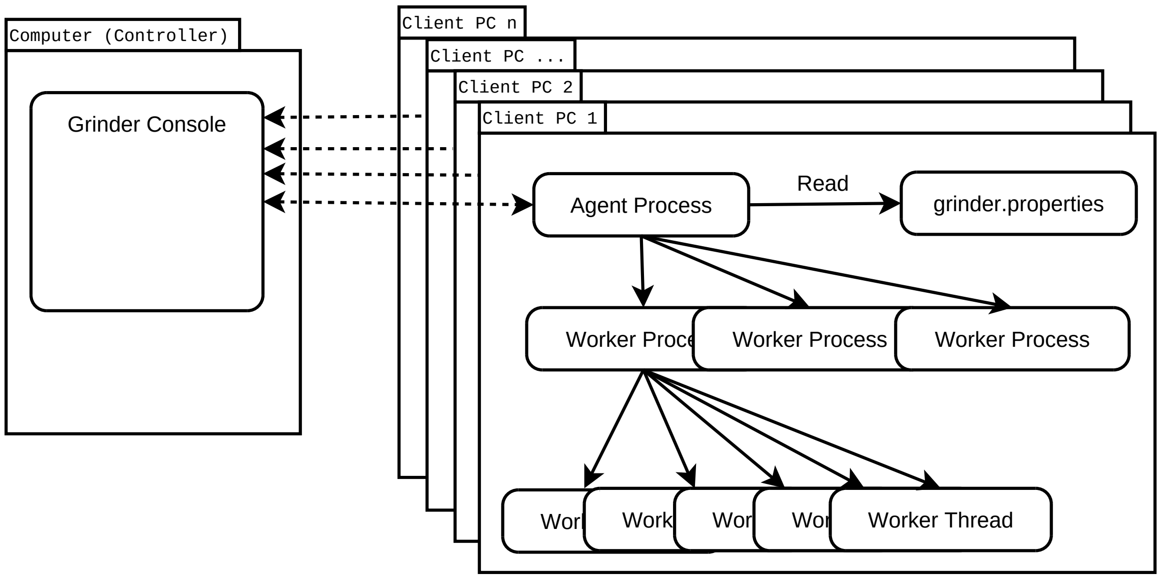


Figure 5.1: Grinder architecture

### 5.1.1 The Grinder HTTP-Plugin

By default the Grinder is shipped with a HTTP-plugin intended for performing HTTP requests and returning the response data. Usage of this plugin is simple: Just wrap a HTTPRequest object into a Grinder Test instance and start doing requests with it (see fig: 5.2).

```
Test testX = new Test( testnum, testname );
HTTPRequest request = new HTTPRequest();
testX.wrap(request);

request.GET("http://localhost/index.jsp");
// statistics are sent back to the Grinder Console with testnum and testname as indicator
```

Figure 5.2: Example usage of the Grinder HTTP-plugin

We decided to use this HTTP-Plugin to do all the HTTP-requests in our loadtest. This way we get timing statistics of each request in our Grinder Console for free. Performing HTTP-requests with this object however can only be done from within Grinder worker threads. This restriction directly forms a problem for our load-testing framework because HTMLUnit starts up new threads for executing java-script. *These (non-Grinder) threads would normally not be able to do HTTP-request via the Grinder Plugin!*

We bypassed this restriction by passing all HTTP-requests to one of the Grinder worker thread. This worker will then perform the actual request on behalf of the requesting thread and will return a result once finished.

## 5.2 Custom Load Test Scheduler

The Grinder comes with a lot of features, one of those is the ability to run JUnit tests. Although this feature at first looked promising, we quickly discovered that it could not be configured in the way we wanted and we thus decided not to use it. The remaining option was to write our own script to schedule which tests to run. Because Grinders scripting language is capable of running any arbitrary Java code and given our preference for the Java language we have chosen to write our load test scheduler in Java.

Whenever a Grinder Agent starts up it will spawn the (see grinder.properties) specified number of process' and threads. Each Grinder worker thread runs an instance of our Jython test script, which in its turns loads our Java-written load test scheduler.

### 5.2.1 Loadtest Configuration And Test Queue Generation

The first job for our scheduler is to check the loadtest.properties file (see figure 5.3). Using this file the scheduler running in the first worker thread[2] then creates a queue of method calls (like the ones in figure: 5.4) representing the tests which have to be executed. The queue is constructed in such a way that, when selecting items at random from that queue, one gets a mix of sessions as specified in the loadtest.properties file.

```
# view click and lead registration
loadtest.viewrate = 40,000
loadtest.view2clickratio = 20
loadtest.click2leadratio = 45

# website
loadtest.sessions = 100
loadtest.anonymous_ratio = 0,12
loadtest.affiliate_ratio = 0,15
loadtest.advertiser_ratio = 0,01
loadtest.admin_ratio = 0,09
...
...

# statistics query
interval = 4 hours
delay = 1 hour
```

Figure 5.3: Example (mockup) loadtest.properties file

[2]Shared test queue: Because of memory considerations we only wanted a single queue per process, instead of per thread.

```
public class AnonymousLoadtests extends LoadTestCase {
    @Loadtest(1) public void testUserClass1() {
        browser = getBrowser();
        page = browser.getPage("some url");
        Thread.sleep( 23 * 1000 ); // 23 seconds
        page = browser.getPage("some other url"); ....etc
    }

    @Loadtest(2) public void testUserClass2() {
        ... session code
    }

    @Loadtest(3) public void testUserClassX....etc...
}
```

Figure 5.4: Example load testing class using annotations (LoadTestCase is an extension of the JUnit TestCase class)

### 5.2.2 Loadtest Startup

#### Web User Simulation

After the test queue has been created, by one of the worker threads, it will proceed with checking whether the web user sessions have already been initialized by another worker thread. The first thread performing this check will start up the configured amount of loadtest user sessions from the test queue. These user sessions are themselves started in new threads (TestRunnerThread.java).

#### View, Click and Lead Registration

A separate thread is started which will be doing all the view, click and lead registrations. This is essentially a loop which generates a view, waits for some time and then starts again. Once every $X$ views however, a click session is started in a separate thread. The same is done for lead sessions, but once in every $Y$ clicks. Both values $X$ and $Y$ derived from the loadtest.properties file as the view2click and click2lead ratios respectively.

#### System Processes

For now separate threads are started which will each execute one of the defined system processes with the configured delay (see *loadtest.properties*). Right now only support has been build in for executing SQL queries. Future updates could include support for running arbitrary shell scripts or so-called Quartz jobs [37].[3]

[3]M4N's system processes were about to be transformed into Quartz jobs which should make it easier to manage them. It would thus be nice if our load-testing application was able to execute those same jobs.

### 5.2.3 Grinder Worker Thread Event Loop

Because the Grinder Worker Thread is the only thread allowed to do the actual web requests, it is necessary for all the other (non-Grinder) threads to pass web requests to a Grinder worker thread. We do this by creating a process-wide job-queue and pass that queue as a reference to each test-running thread.

Whenever a test thread (non-Grinder) needs to do a HTTP-request it will synchronize on that job queue, add a new job, and notify one of the waiting Grinder worker threads. On its turn, one of the waiting Grinder worker threads will remove the first job from the queue, and starts executing that request. Once the request has finished the Grinder worker thread will pass the result back to the test-running thread and starts waiting again for new jobs (see figures 5.5 and 5.6).

**Grinder Worker Thread: Loadtestscheduler.runTests**

```
1. If TestQueue == null: Create new TestQueue.
2. If SessionsStarted == false: Start proper amount of sessions for this process.
3. If SystemProcessesStarted == false: Start sytem processes.
4. If ViewsClicksLeadsStarted == false: Start view, click and lead threads.
5. loop:
    (a) If the job queue is empty, wait on this job queue object.
    (b) If thread has been notified (or the queue was not empty): Remove top job from the
        queue and execute this job.
    (c) Return result to TestRunnerThread.
```

Figure 5.5: Pseudocode explaining internal structure of the LoadTestScheduler.

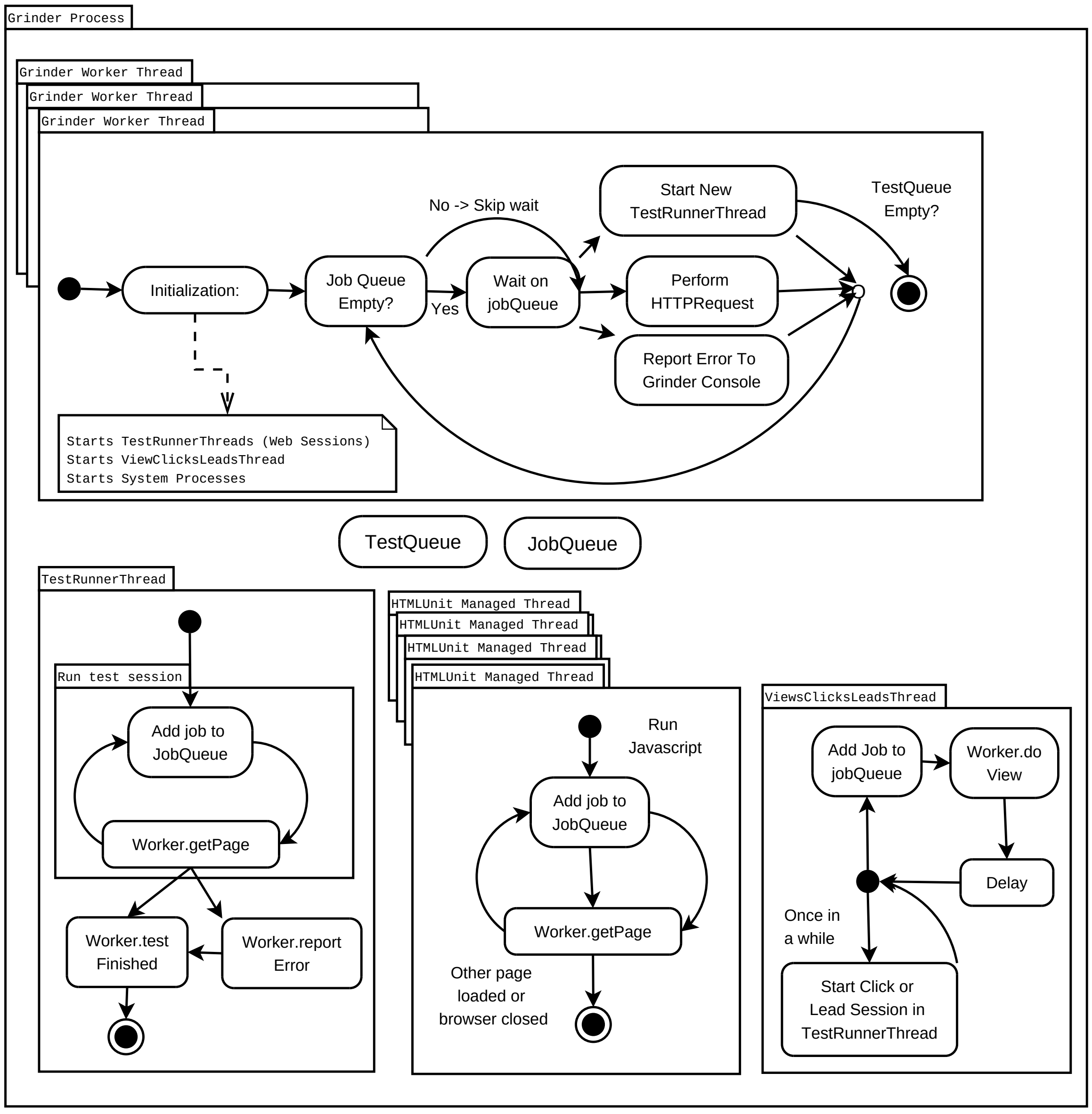


Figure 5.6: Loadtest scheduler internal architecture

## 5.3 Adapting Our Load Tests For The Grinder

### 5.3.1 WebClient: Custom Web Connection Handler

When browsing pages using the HTMLUnit's WebClient however, we do not get any statistics in the Grinder Console as we would have gotten using Grinder's own HTTP-Plugin. To solve this inconvenience we had to pass a custom made Connection Handler to the WebClient. This handler's main function is to return a WebResponse object whenever the getResponse function is invoked. By adapting our own handler to ask a worker thread to do the request instead, using the HTTP-Plugin, we effectively pass all requests statistics to the Grinder Console as well. (see fig: 5.7).

```
WebClient browser = new WebClient();
myHandler = new MyHandler();
browser.setWebConnection( myHandler );

browser.getPage( new URL("http://locahost/index.jsp") );
// inside getPage myHandler.getResponse will be used for retrieving the WebResponse
```

Figure 5.7: Using custom connection handler in HTMLUnit's WebClient

### 5.3.2 JUnit Test Failure

Next to the response time statistics we also had to inform the Grinder Console whenever a JUnit test had failed. We detect these failures by catching the thrown exception from the JUnit assert commands (see figure 5.8). Via a hack we then pass a message to the Grinder telling that this particular test has failed.

```
public void run() {
    // Extract and set mutexes for error reporting
    LoadTestCase theInstance = (LoadTestCase) instance;    // test instance

    try {
        method.invoke( instance, parameters );    // run loadtest method
    } catch ( Exception e ) {
        handleCause( e.getCause() );              // handle JUnit exceptions
    } finally {
        invokeRequest = new InvokeRequest();
        /** ask worker to start next (new) test **/
        if( isUserSession ) {
            invokeRequest.invokeType = InvokeRequest.NEW_TEST_INVOKATION;
        } else {
            invokeRequest.invokeType = InvokeRequest.WORKER_READY;
        }

        synchronized( requestQueue ) {
            requestQueue.add( invokeRequest );        // add request to queue
            requestQueue.notify();                    // notify the queue
        }
    }
}
```

Figure 5.8: TestRunnerThread catching JUnit assertions and other exceptions

### 5.3.3 JUnit Test Transparency

One of the requirements was that each load-test had to be a JUnit-test as well. We thus had to assure that JUnit testing was possible with our load tests and that our LoadTestScheduler had to be able to use ordinary JUnit tests as well.

By creating a LoadTestCase class, which extends the TestCase class we at least ensure that each load test can be used as a JUnit test. In this class we maintain a variable which indicates whether the tests have to be run as a load test or as a JUnit test.

Compared to an ordinary JUnit test we only have to wrap a HTTPRequest object into a Grinder Test object when doing load tests. By using a getBrowser function (see fig 5.9) we ensure this wrapping whenever the loadtest switch has been set.

```
protected WebClient getBrowser() {
    WebClient browser = new WebClient(); // sufficient for doing JUnit tests

    // ********* ONLY FOR LOAD TESTING
    if( testType == LOADTEST ) {
        HTTPRequest request = new HTTPRequest();

        /** Wrap HTTPRequest in test **/
        PyJavaInstance pyRequest = (PyJavaInstance) new Test( getTestNumber(), this.getNam

        /** Create connection handler handler **/
        HtmlUnitWebConnection webConnectionWrapper = new HtmlUnitWebConnection( browser );
        /** Tell our handler to use request and pyRequest for connecting **/
        webConnectionWrapper.setPyHTTPRequest( pyRequest );
        webConnectionWrapper.setHTTPRequest( request );
        webConnectionWrapper.setInvokeMutex( invokeMutex );
        webConnectionWrapper.setResultReady( resultReady );

        /** Tell the web client (browser) to use our connection handler **/
        browser.setWebConnection( webConnectionWrapper );
    }
    // ********************

    return( browser );
}
```

Figure 5.9: LoadTestCase.java: Getting a browser (WebClient object) for either a load test or a JUnit test

# Chapter 6

# Sequoia

*"What is Sequoia? Sequoia is a transparent middleware solution for offering clustering, load balancing and fail-over services for any database. Sequoia is the continuation of the C-JDBC project. The database is distributed and replicated among several nodes and Sequoia balances the queries among these nodes. Sequoia handles node failures and provides support for checkpointing and hot recovery."* [19].

Sequoia in this sense offers the functionality of the middleware distributed database solution suggested in chapter 3 (see figure 6.1).

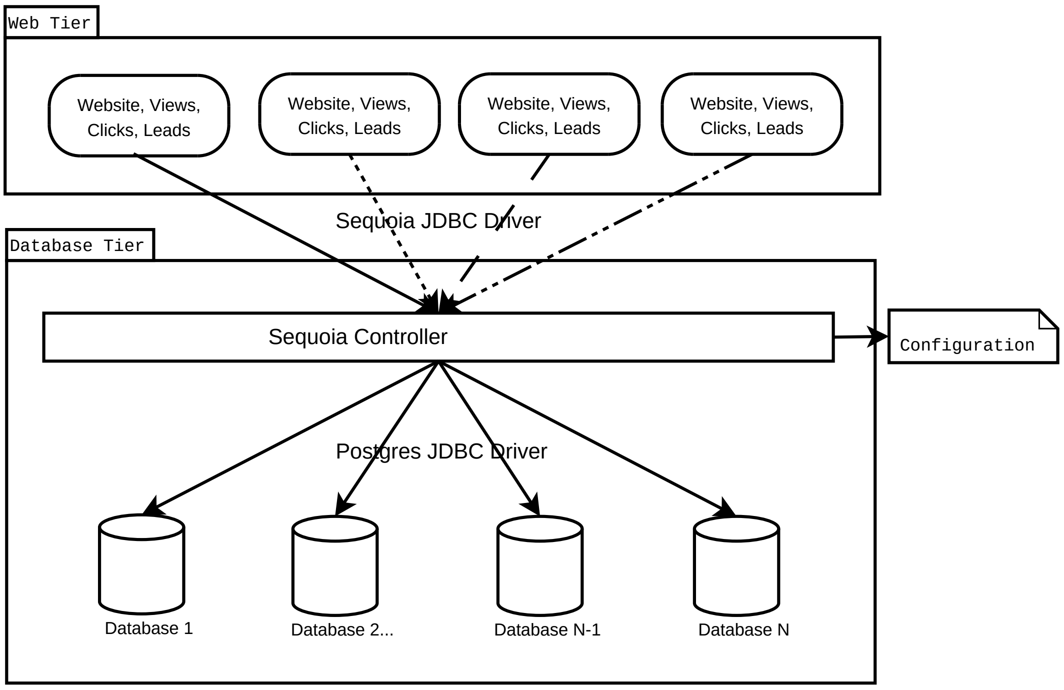


Figure 6.1: *Simplified* M4N Future System Architecture. Clustering the web- and database-tier using Sequoia. Notice how the web-tier communicates with Sequoia using a special Sequoia JDBC driver whilst Sequoia itself uses the ordinary (Postgres) JDBC drivers.

Sequoia is claimed to be stable, and ready for production. Our initial attempts however still showed some minor problems.

## 6.1 Sequoia Deployment

We deployed Sequoia on the web-application server (webNode2). No other machines were available and hosting the Sequoia controller outside the internal network (on for instance university computers) would have caused too much latency.

To get Sequoia running we basically just adapted two of the example configuration files: *controller.xml* and *raidb1.xml*. See appendix F to view our Sequoia configuration files. We also had to adjust the *context.xml* file used by the Tomcat application server. The default datasource used for database connections used the postgres driver which needed to be replaced by the Sequoia driver. Furthermore we needed to replace the Datasource type and factory to use the Sequoia classes instead of the Tomcat ones (see appendix E)[1].

## 6.2 Sequoia Deployment Issues

We have used the latest available version of Sequoia (2.9) for deploying our database cluster. To identify problems and possible debugging we also downloaded the latest Sequoia source from their public CVS.

Our first *test*-deployment attempt was performed on a laptop running Windows. On this machine, we configured both the Tomcat application server as well as a Sequoia controller. This controller served as the middleware between the web-application server and the database systems, relaying all SQL requests to two Postgres database servers.

After starting M4N on the application server we encountered some problems. Problems which were not present when using a regular JDBC driver suggesting that Sequoia caused them. Exceptions were thrown and many pages did not work correctly. To find what was causing these problems we have checked each page in M4N and looked which ones experienced errors. Finally we analyzed what went wrong precisely and listed the exact causes:

- The occurrence of *SELECT <INET-datatype> FROM...* patterns: Selecting INET datatypes resulted in exceptions from Sequoia. It was solved by replacing these types of SELECTS with a *SELECT host(<INET-datatype>) FROM...*

- The occurrence of *SELECT ... FROM<whitespace(non-newline)><newline>...*: The problem was bypassed by replacing the white-spaces and newline with a single newline.

- getBytes function exception: The getBytes function from the Sequoia ResultSet class threw an exception because it could not convert *java.lang.String* datatypes to byte arrays (see figure 6.2 for fix).

- Clicks were not registered correctly because of many unique key constraints violations (*ERROR: duplicate key violates unique constraint "transactions_id_key"*). Normally these unique key values are maintained by so-called SQL-sequences. Once new records are inserted these values are updated using SQL-triggers. Because Sequoia only load balances the select-requests in *RaidB1* mode, this would not have been a problem. Insert statements are performed on all database backends, incrementing the sequence values on each backend.

  In M4N however, the sequence value of the *transaction* table is incremented by means of a *SELECT* statement (*SELECT nextval('transactions_seq') AS transid*), which extracts the

[1]Some of these configuration steps could not be found in the Sequoia manual.

sequence value needed to generate an URL. Because Sequoia load balances select-requests only one of the $N$ database backends in the cluster will update its sequence value. When for instance another backend is used for obtaining the next sequence value that backend will return a value which has already been used.

For our tests we bypassed this issue by incrementing one of the database sequences with 1 billion (*SELECT setval('transactions_seq', oldvalue+1billion)*). With this measure we ensure that sequence values of both database will not overlap with each other for the next 1 billion newly inserted records. To use this fix for testing purposes is fine, but a better solution should be found once deploying Sequoia on a production environment.

```
SEVERE: SQL Error
org.continuent.sequoia.common.exceptions.NotImplementedException: in getBytes(1), don't kn
        at org.continuent.sequoia.driver.DriverResultSet.getBytes(DriverResultSet.java:112
        at org.continuent.sequoia.driver.DriverResultSet.getBytes(DriverResultSet.java:142
        at org.apache.tomcat.dbcp.dbcp.DelegatingResultSet.getBytes(DelegatingResultSet.ja
        at com.mbuyu.m4n.Bannermanager.getHyperText(Bannermanager.java:740)
        at org.apache.jsp.affiliatelinks_jsp._jspService(org.apache.jsp.affiliatelinks_jsp
```

We bypassed this in the M4N code (did not fix the bug in Sequoia) by replacing the line causing the exception.

```
BannerManager.java on line 740:
//banner_text =  new String(rs2.getBytes("banner_text"),"ISO-8859-1");
banner_text =  rs2.getString("banner_text");
```

Figure 6.2: Bypassing getBytes Exception when converting String to byte array

In our second attempt we used webNode2, the Linux server on which we would actually do our load-tests on. Suddenly many of the SQL related bugs were resolved. We suspect that the cause was the different line separator used by Windows (\r\n), because the bugs found were related with the Sequoia SQL parsing. However, the *getBytes()* exception still persisted, so we really had to fix this on webNode2 as well (see figure 6.2).

# Chapter 7

# M4N Benchmark Definitions

We intend to use our load tests as benchmarks for measuring and finding at which loads M4N reaches its peak. In particular we are interested in the types of load which cause peak loads in the *database system* alone. By performing these benchmarks on various database system configurations we intend to create a comparison between those configurations.

The quality of a benchmark relies on several factors. Repeatability for instance which is a prerequisite for all types of experiments. Also recreating similar conditions for each experiment is an important factor because one eliminates the chance of interference from external factors, hence creates a more reliable/qualitative benchmark.

There are various degrees in recreating those similar conditions, which we will try to illustrate by means of a few examples:

1. Using identical server machines and configurations for each separate benchmark run: Due to obvious budget reasons this was not an option for our research thesis.

2. Using the same server machines for each benchmark run: Doing a full re-install of the operating system or use a fresh backup copy of the entire disk for each run.

3. Using the same server machine for each benchmark run: Only perform a full system reboot before each run. Optionally one could remove and restore the database after each run as well.

4. Using the same server machine for each benchmark run: Merely wait some minutes before starting the next benchmark run. Only in some cases reset the Tomcat application or Postgres database server.

Apart from these arbitrarily chosen examples one can imagine numerous of other degrees of repeatability for benchmark experiments. Depending on our time and budget it was up to us to determine to which extreme to go with this.

We chose to keep things simple and to execute the benchmark runs without rebooting the server systems themselves. This was done for the following reasons:

- The servers were sometimes used by other developers as well. Although we chose the 'quiet' hours and indicated to others not to run heavy loads during our tests, we could not guarantee that the used servers were stressed by our benchmark alone.

- Rebooting servers would have been an option if we performed the benchmarks after office hours on a few predefined days, for which other developers could use alternative servers instead. Time however did not permit this option.

- No real measurable side-effects: We noticed that system loads and memory usages returned to 'normal' levels once our benchmark finished. This gave us some certainty that successive tests did not interfere with each other. Backed by our observation that most successive test runs returned similar results we believe that rebooting the servers after each run would not have increased our test reliability by much[1].

## 7.1 General Test Descriptions

Every benchmark type is in some way different from the other benchmarks. Lot of variables however are the same. Here follows a list of these commonalities:

### 7.1.1 Software

- Grinder: Grinder-3.0-beta28
- Tomcat Version: 5.5.15
- Database version: Postgres 8.1.3
- Postgres JDBC driver: postgresql-8.1-404.jdbc3.jar
- Database Dump: The database dump made the 24th of April 2006 (m4n_20060424) was only used for the *regular* view and the click benchmarks (without Sequoia). For all the other benchmarks a newer dump was used (m4n_20060619) because disk storage did not permit any older database dumps on the development servers.
- M4N: The current live version (may 2006) of M4N was used for the load tests. In CVS this was branch m4n_250_0.
- Java VM: JDK 1.5.0.5
- Sequoia: Version 2.9

### 7.1.2 Benchmark System Configurations

For our benchmarking we will compare the following hardware configurations:

1. **M4N Slow**: The M4N web-application hosted by webNode1 and the database server runs on dbaseNode1. For some tests we will use this configuration to show the effect of a faster web-application server.
2. **M4N Default**: The M4N web-application hosted by webNode2 and the database server runs on dbaseNode1. This system serves as an indicator on how the live M4N system would perform under similar loads.
3. **M4N Sequoia1**: The M4N web-application runs on webNode2 which also runs the Sequoia controller[2]. This controller on its turn connects to two database servers dbaseNode1 and dbaseNode2 in a RAID1b configuration (see appendix F for exact configuration settings).

[1]Only in a few cases we needed to restart the Tomcat application server. Logged in users kept being logged in for an hour, consuming a lot of memory due to some M4N specific database caching components.

[2]Using more than two controllers would have been possible if additional servers were available for benchmarking

### 7.1.3 Server and Grinder-Client Systems Specifications

These are the development servers which were used during the load test benchmarks. Here we present a list of system specifications for each machine:

**dbaseNode1**

| | |
|---|---|
| CPU | Intel Xeon 2.0GHz |
| MEM | 768MB |
| OS | Linux version 2.6.11.12-custom-xen-user: (Debian 1:3.3.5-13) |

dbaseNode1 is configured in a special way. In reality dbaseNode1 is a *dual* Xeon 2.0GHz system with 2GB of memory. However by using the XEN [38] virtualisation package we have only been allocated a limited portion of the system resources.

**dbaseNode2 (172.16.26.130)**

| | |
|---|---|
| CPU | AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ |
| MEM | 2GB |
| OS | Linux version 2.6.15-1-k7-smp (Debian 2.6.15-8) |

**webNode2**

| | |
|---|---|
| CPU | 2x Intel Xeon 2.0GHz |
| MEM | 1GB |
| OS | Linux version 2.6.8-2-686-smp (Debian 1:3.3.5-13) |

**webNode1**

| | |
|---|---|
| CPU | AMD Athlon 800MHz |
| MEM | 1GB |
| OS | Linux version 2.6.8 (Debian 1:3.3.4-6sarge1) |

**testClient3**

| | |
|---|---|
| CPU | AMD Athlon 64 X2 Dual Core Processor 3800+ |
| MEM | 1GB |
| OS | Linux version 2.6.15-1-k7-smp (Debian 2.6.15-8) |

**testClient2**

| | |
|---|---|
| CPU | Athlon XP 2000+ |
| MEM | 1GB |
| OS | Linux version 2.6.15-1-k7 (Debian 2.6.15-8) (Debian 4.0.2-9) |

**testClient5**

| | |
|---|---|
| CPU | AMD Athlon(tm) 64 Processor 2800+ |
| MEM | 1.5GB |
| OS | Linux version 2.6.8-11-amd64-k8 (Debian 3.4.3-13) |

**testClient4**

| | |
|---|---|
| CPU | AMD Athlon(TM) XP1800+ |
| MEM | 896MB |
| OS | Linux version 2.6.15-1-486 (Debian 2.6.15-8) |

**testClient1**

| | |
|---|---|
| CPU | Intel(R) Pentium(R) 4 CPU 3.20GHz |
| MEM | 896MB |
| OS | Linux version 2.6.8-2-386 (Debian 1:3.3.5-13) |

## 7.2 View Benchmark

The goal of the view benchmark is to see how many views/second can be handled by M4N on a given configuration.

### Definition

1. Test-run duration: 1 minute
2. Test intensity: Generating 400, 500 and 900 views/second.
3. Test-runs: 3 runs average
4. Startup time: 20 to 30 seconds
5. Machines running The Grinder: Equally dividing the load generation among 4 clients.

### Execution

After configuring the clients and starting the Grinder Console:

1. Start the load using the grinder (by pressing the *start the worker processes* button).
2. Wait for 20 to 30 seconds.
3. Note down the system time, and nearly simulatiously start gathering Grinder statistics (by pressing the *start collecting statistics* button).
4. After 1 minute, note down system time again and simultaneously stop collecting Grinder statistics (by pressing the *stop collecting statistics* button).
5. Store the gathered statistics results in a file before proceeding to the next test.
6. For each test intensity we do these measurements 3 times

## Properties

### grinder.properties

```
grinder.processes=1
grinder.threads=80
grinder.runs=0
grinder.script=loadtest001.py
```

### loadtest.properties

```
loadtest.viewrate=100    # -> Variable
loadtest.view2clickratio=0
loadtest.click2leadratio=0

# Define how much to test of each link type (4 types):
...

# number of concurrent users
loadtest.concurrentUsers=0

# user type configuration
...
```

## 7.3 Click Benchmark

The goal of the click benchmark is to see how many clicks/second can be handled by M4N on a given configuration.

### Definition

1. Test-run duration: 1 minute
2. Test intensity: Apart from comparing the response time results from the 1, 2, 3, 4, 6, 8, 12, 30, 40 clicks/second tests we also try to find the click handling limit for a given configuration.
3. Test-runs: 3 runs average
4. Startup time: 25 to 30 seconds
5. Machines running The Grinder: Single client sufficient for click load generation.
6. Click type: Short-links (see *QuickShortLink.java*)

### Execution

After configuring the clients and starting the Grinder Console:

1. Start the load using the grinder (by pressing the *start the worker processes* button).
2. Wait for 25 to 30 seconds.
3. Note down the system time, and nearly simulatiously start gathering Grinder statistics (by pressing the *start collecting statistics* button).
4. After 1 minute, note down system time again and simultaneously stop collecting Grinder statistics (by pressing the *stop collecting statistics* button).
5. Store the gathered statistics results in a file before proceeding to the next test.
6. For each test intensity we do these measurements 3 times

## Properties

### grinder.properties

```
grinder.processes=1
grinder.threads=80
grinder.runs=0
grinder.script=loadtest001.py
```

### loadtest.properties

```
loadtest.viewrate=1    # -> Variable
loadtest.view2clickratio=1
loadtest.click2leadratio=0

# Define how much to test of each link type (4 types):
loadtest.shortlink=0
loadtest.longlink=0
# The quick types have no wait times (for massive click testing)
loadtest.quickshortlink=50
loadtest.quicklonglink=0


# number of concurrent users
loadtest.concurrentUsers=0

# user type configuration
...
```

## 7.4 Full Simulation Benchmark

The full simulation will generate view, click, lead and website traffic simulation at the same time. Before choosing in which proportions these should be executed we first looked at the typical load encountered on the M4N live servers.

M4N operates only in a single time-zone. Traffic peaks are thus realized during office hours and in the evenings. At night however, when mostly no-one makes use of either the website, or the view, click and lead registration, some other heavy processes are run (like for instance calculating statistics). In an average month, M4N handles around 21 to 33 million views, registers 800, 000 to 1 million clicks, and realizes approximately 8, 000 to 9, 000 sales/leads. In the same period around 600, 000 user-sessions were registered on the web-server.

To determine the most active user-classes we have looked at the sessions after the web-log clustering. Because they are grouped per user-class we can easily see to which degree each user-class should be represented in a typical M4N load. These results can be viewed in the upcoming 'Properties' subsection in the *loadtest.properties* file. Given the assumption that most traffic ($\geq$95%) is generated in only 8 hours per day (24 hours), we come to an average daytime load of:

- 33 million views divided by 30 days and 8 hours equals 137, 500 views/hour which comes to 38 views/second.
- 1 million clicks divided by 30 days and 8 hours equals 4167 clicks/hour which comes to 69 clicks/minute or 1.15 clicks/second.
- 9, 000 leads divided by 30 days and 8 hours equals 37.4 leads/hour.
- 600, 000 sessions divided by 30 days and 8 hours equals 2500 sessions/hour which comes to 42 sessions/minute.

For simulating a typical M4N load we would thus configure (spread among all Grinder Agents):

- View-rate: 38 views/second
- View to click ratio: 33 views/clicks
- Click to lead ratio: 111 clicks/leads
- Simultaneous sessions: 50 logged in users. Proportionate with the 'anonymous/logged' ratio of 1 (number of anonymous users approximately equaled number of logged in users) we add 50 anonymous users bringing a total of 100 simultaneous users.

### Definition

1. Test-run duration: 5 minutes
2. Test intensity: Compare response times of the default, double and quadruple benchmarks, default being the situation described previously.
3. Test-runs: 3 runs average
4. Startup time: 60 seconds
5. Machines running The Grinder: Equally dividing the load generation among 4 clients.
6. After each test run we restart the Tomcat application server to flush the logged in users from memory.
7. JVM Heap size: 1500MB[3].

### Execution

After configuring the clients and starting the Grinder Console:

1. Start the load using the grinder (by pressing the *start the worker processes* button).
2. Wait for 60 seconds.
3. Note down the system time, and nearly simulatiously start gathering Grinder statistics (by pressing the *start collecting statistics* button).
4. After 1 minute, note down system time again and simultaneously stop collecting Grinder statistics (by pressing the *stop collecting statistics* button).
5. Store the gathered statistics results in a file.
6. Restart the Tomcat application server before proceeding to the next test.
7. For each test intensity we do these measurements 3 times

[3]User sessions require a lot of memory. To prevent Java Out of Memory Exceptions we have increased the memory allocation for this benchmark type.

## Properties

### grinder.properties

```
grinder.processes=1
grinder.threads=80
grinder.runs=0
grinder.script=loadtest001.py
```

### loadtest.properties

The mentioned variables need to be divided by the number of Grinder Agents (four in our case) to get the exact configuration for each client.

```
loadtest.viewrate=38       # --> variable (will be doubled/quadruppled)
loadtest.view2clickratio=33
loadtest.click2leadratio=111

# Define how much to test of each link type (4 types):
loadtest.shortlink=50
loadtest.longlink=50
# The quick types have no wait times (for massive click testing)
loadtest.quickshortlink=0
loadtest.quicklonglink=0


# number of concurrent users
loadtest.concurrentUsers=100       # --> variable (will be doubled/quadruppled)

# user type configuration
loadtest.anonymous=15146
loadtest.newuser=300
loadtest.affiliate=11540
loadtest.advertiser=1879
loadtest.admin=408
loadtest.administration=206
loadtest.financial=300
loadtest.sysadmin=580
loadtest.algemeen=635
```

# Chapter 8

# Benchmark Results

We will now per benchmark show and compare the results for each machine configuration. In the end we will draw an overal conclusion.

## 8.1 View Benchmark

Views, although finally stored in the database, are in fact first cached by the web application server. The reason behind this was that storing views one at a time into the database server caused a severe overload once the number of views/second reached a certain limit. Caching them on the web application server and storing them in batches however, reduced the database load significantly. In theory we would thus expect that a view benchmark might not be affected at all by the database-tier. Writing those views (in batches) to the database however will be affected, although *this* will not be part of the view benchmark[1].

**Results on M4N Slow**

| Config: views/second | | 20 | 40 | 50 | 70 | 90 | 110 | 130 |
|---|---|---|---|---|---|---|---|---|
| Views generated | | 4472 | 7347 | 7723 | 8734 | 8184 | 9164 | 9192 |
| Avg TPS | | 75.5 | 123.7 | 129.3 | 146.7 | 139 | 154 | 155.3 |
| Peak TPS | | 115.0 | 181.3 | 186.3 | 214.0 | 198.3 | 187.0 | 187.3 |
| Response times(ms) | Mean | 10.2 | 13.2 | 18.1 | 23.0 | 28.7 | 35.8 | 43.7 |
| | Std. dev. | 6.7 | 10.7 | 22.7 | 26.1 | 34.7 | 65.2 | 105.1 |
| Avg Sys load | webNode1 | 50.20% | 77.80% | 84.40% | 93.70% | 91.60% | 99.80% | 99.99% |
| | dbaseNode1 | 5.74% | 9.55% | 11.68% | 10.72% | 11.89% | 10.58% | 10.42% |

For more detailed load results on the web application server see figures: G.1, G.2 and G.3. Also notice the difference between the *config: views/second* and the *Avg TPS* (average transactions per second) row. The first one shows how we configured the clients while the second one shows the actual (average) performance of the system under test. We see that configuring the clients to generate more than 70 views per second does not result in a significantly higher *AVG TPS*, hence the system has reached a performance limit. This conclusion is confirmed by the high system load encountered on the web-application server (webNode1).

[1]This could become part of a future 'system-job' benchmark.

### Results on M4N Default

| Intensity | | 1 | 2 | 3 |
|---|---|---|---|---|
| Config: views/second | | 400 | 500 | 900 |
| Views generated | | 35379 | 42100 | 41813 |
| Avg TPS | | 593 | 706 | 697 |
| Peak TPS | | 956 | 1135 | 1001 |
| Response times(ms) | Mean | 15.4 | 108 | 92.6 |
| | Std. dev. | 57.3 | 1000 | 1030 |
| Avg Sys load | webNode2 | 61.91% | 83.16% | 82.11% |
| | dbaseNode1 | 21.66% | 21.17% | 18.77% |

During the first benchmark we observed that the main bottleneck for view handling was the CPU of the web application server (webNode1). To see how well views scale on a faster web-application server, we have replaced the webNode1 with the more powerful webNode2. See figures G.4, G.5 and G.6 for detailed system load statistics. Notice that the Avg TPS limit of webNode2 lies around 700 instead of just 150 on webNode1.

### Results on M4N Sequoia1

| Intensity | | 1 | 2 | 3 |
|---|---|---|---|---|
| Config: views/second | | 400 | 500 | 900 |
| Views generated | | 37637 | 42182 | 53137 |
| Avg TPS | | 633 | 728 | 848 |
| Peak TPS | | 846 | 965 | 1220 |
| Response times(ms) | Mean | 20 | 25.8 | 251 |
| | Std. dev. | 186 | 246 | 2567 |
| Avg Sys load | webNode2 | 49.31% | 59.48% | 80.76% |
| | dbaseNode1 | 1.25% | 1.02% | 0.77% |
| | dbaseNode2 | 24.24% | 4.32% | 0.47% |

What strikes us immediately is the relative high load (average of 24.24%) experienced on dbaseNode2 during the 400 views per second benchmarks. We believe however that these were not caused by our view benchmark. Otherwise we would have seen similar or higher loads during the more intensive view benchmarks (500 and 900 views per second). Figure G.11 confirms this was only a random spike probably caused by another developer or system-process. Other statistics for this benchmark can be found in appendix G.

### View Benchmark Conclusion

Because views were handled by the application server alone we expected not to see much difference when we would change the underlying database system. Furthermore we expected that by upgrading the web-application server (from M4N Slow to M4N default) we would get an increased view-registration capacity.

In reality some predictions did not come out as we expected. Although we did see the expected increase of view-registration capacity with the faster web-application server, we did not see exactly the same results comparing the Sequoia and M4N-default configurations. Only after the Sequoia view benchmarks we really noticed that while benchmarking the M4N-default configuration that the database system was put under a considerable load (around 18%-21%). A load which was not present when benchmarking the Sequoia version.

To find out what was causing the load we again did a view benchmark on the M4N default configuration. But again we found a similar database load (around 20 percent) once the tests started, and again no database load whatsoever using the M4N-Sequoia configuration. We are still puzzled about why the database load did not rise when using Sequoia although we suspect that having the Sequoia controller between the web-tier and database-tier must be the determining factor. A wild guess is that is has something to do with the database connection pooling, which is handled by the Sequoia controller instead of by the Tomcat application server (See appendix F for the Sequoia configuration and how it handles database pooling).

## 8.2 Click Benchmark

In terms of database load and response times, M4N knows two distinct types of click registration: The long-link and the short-link. The long-link got its name due to the long url which is used to generate the link. This long url contains (although a bit garbled) two variables. Using these two variables the web-server can easily store the click into the database.

The short-link, as the name suggests is just a very short url containing a single variable. Using this variable the web-server first needs to consult with the database server before it can actually store the click. Although these types of links are easier to type in for M4N's affiliates they are in fact much slower as their long-link counterparts.

Because our goal was to primarily generate load on the database server we have chose to use short-links for our click benchmark.

**Results on M4N Slow**

| Config: clicks/second | | 1 | 2 | 3 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|---|---|
| Clicks generated | | 118 | 239 | 349 | 354 | 345 | 340 | 344 |
| Avg TPS | | 2 | 4 | 5.9 | 6 | 5.8 | 5.7 | 5.8 |
| Peak TPS | | 4 | 5 | 11 | 11 | 10 | 12 | 12 |
| Response times(ms) | Mean | 367 | 350 | 1730 | 3463 | 4837 | 5657 | 4650 |
| | Std. dev. | 26 | 28 | 923 | 2800 | 6730 | 9107 | 6437 |
| Avg sys load | webNode1 | 10.10% | 14.26% | 20.86% | 21.24% | 20.27% | 18.42% | 20.09% |
| | dbaseNode1 | 35% | 75% | 100% | 100% | 100% | 100% | 100% |

webNode1 was used instead of webNode2 for web serving because at that time we could not get access to webNode2. Because the web-application server was not a constraining factor for this test we chose to use the slower web-server as a quick alternative. Notice that the Avg TPS reaches its peak around 6 which corresponds with the 100% load on dbaseNode1.

Just like with M4N's view caching the next version will support click caching as well, which should speed up click handling significantly. Benchmarks performed on the development version of this click-cache have shown that M4N will be able to handle around 120 clicks a second (on webNode2). A lot faster than the +-6 clicks per second of the uncached version. We also found that the web-application server became the bottleneck for the click-cache enabled M4N (instead of the database server).

For our research project we will not ellaborate any further on the click-cache component and will keep using the M4N version without click-cache for the sake of comparison. To show that the web-application server (webNode1) was not the real bottlneck during the benchmark please consult figure H.5. All other statistics related with the click-benchmark can be found in appendix H.

### Results on M4N Sequoia1

| Config: clicks/second | | 1 | 2 | 4 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|---|---|---|---|
| Clicks generated | | 117 | 223 | 473 | 956 | 1455 | 1581 | 1576 |
| Clicks timed out | | 0 | 0 | 0 | 0 | 0 | 10 | 18 |
| Avg TPS | | 1.92 | 3.68 | 7.82 | 15.80 | 24.07 | 26.10 | 26.00 |
| Peak TPS | | 3 | 5 | 13 | 22 | 30 | 38 | 39 |
| Response times(ms) | Mean | 311 | 240 | 275 | 314 | 503 | 12300 | 12167 |
| | Std. dev. | 130 | 120 | 168 | 254 | 528 | 6963 | 7003 |
| Avg sys load | webNode2 | 1.78% | 2.94% | 5.17% | 9.74% | 15.43% | 17.78% | 18.11% |
| | dbaseNode1 | 32.76% | 45.54% | 60.86% | 94.58% | 100% | 100% | 100% |
| | dbaseNode2 | 3.04% | 9.72% | 26.40% | 57.79% | 92.60% | 99.75% | 99.73% |

Notice how the the *AVG TPS* values stop to increase at the limit of 26. From that point the production of additional clicks per second only resulted in higher response times and click time-outs. For detailed load statistics for the click benchmarks please consult appendix H.

### Click Benchmark Conclusion

The clicks benchmark measurements largely correspond with our expected results. Click registration is directly dependant on the database system. Furhtermore, the real performance penalty for this registration is paid when looking up information in the database (*SELECT* statements). Because these statements are load balanced by Sequoia we expected a significant speedup.

The M4N default configuration reaches its peak capacity at an average of 6 transactions per second. At this level we clearly see that the database system (dbaseNode1) is at maximal load and forms the bottleneck for click registration. When using Sequoia however, this peak capacity is reached at an average of 26 transactions per second. It clearly shows that Sequoia efficiently distributes the load over the two database backends (dbaseNode1 and dbaseNode2). In terms of speedup we see that around 4 times as much clicks per seconds (26/6 = 4.33) are registered by Sequoia. This was much higher than we had expected. Logically doubling the database resources should *at most* double the database performance, and not quadrupple.

We hypothesize that the Sequoia controller handles database pooling more efficiently, enabling many requests after each other to be executed faster. This *could* be verified by benchmarking Sequoia with only a single database.

## 8.3 Full Simulation Benchmark

### Results on M4N Default

| Intensity | | 1 | 2 | 3 |
|---|---|---|---|---|
| Successful tests | | 25727 | 35990 | 48388 |
| Failed tests | | 240 | 325 | 464 |
| Avg TPS | | 86 | 120 | 162 |
| Peak TPS | | 441 | 577 | 740 |
| Response times(ms) | Mean | 486 | 883 | 910 |
| | Std. dev. | 1230 | 2240 | 3260 |
| Avg sys load | webNode2 | 10.90% | 17.16% | 23.77% |
| | dbaseNode1 | 59.59% | 95.97% | 87.87% |

See figures I.1, I.2 and I.3 for detailed load statistics during each benchmark.

### Results on M4N Sequoia1

| Intensity | | 1 | 2 | 3 |
|---|---|---|---|---|
| Successful tests | | 30141 | 43578 | 57295 |
| Failed tests | | 243 | 358 | 469 |
| Avg TPS | | 98 | 137 | 175 |
| Peak TPS | | 262 | 485 | 883 |
| Response times(ms) | Mean | 269 | 684 | 675 |
| | Std. dev. | 689 | 1733 | 1687 |
| Avg sys load | webNode2 | 26.85% | 36.32% | 43.09% |
| | dbaseNode1 | 38.46% | 43.52% | 43.62% |
| | dbaseNode2 | 8.82% | 13.41% | 13.78% |

See appendix I for detailed load statistics.

### Full Simulation Benchmark Conclusion

As expected we again see that Sequoia distributes the load among both database backends. Because this load is almost evenly distributed it keeps under 50% for each database. Under the same conditions however, the database (dbaseNode1) in the M4N Default configuration experiences near peak loads. We believe that this peak load is responsible for the higher response times and standard deviations (Std. Dev.) found compared in the M4N Default benchmarks.

What we did not expect was that the web-application server (webNode2) would experience almost twice the load using Sequoia than without Sequoia. Investigation of the load statistics clearly shows that the load generated on webNode2 in the Sequoia benchmarks corresponds with the actual execution of the benchmarks (see figure: I.5). This was not the case with the M4N default benchmark (see figure: I.4). We suspect that the Sequoia controller, running on webNode2, was the cause of this increase. SQL queries from user sessions in particular fetch larger amounts of data than other queries. Therefore the Sequoia controller has to process more data. We speculate that processing large amounts of data could be the cause of the increased load on webNode2.

## 8.4 Overall Benchmark Conclusion

In our benchmarks we have compared the performance of the M4N system with two different database solutions. The first solution being a single database server, while the other was a clustered solution with Sequoia. This Sequoia cluster was composed of two database systems operating in mirrored RAID mode. The purpose of these benchmarks was to find out how each solution performed compared with the other. Our particular interested was to see to which degree database clustering would really benefit M4N in terms of performance.

By segmenting view and click registration we have clearly shown that views, which are cached on the web-application server, are hardly influenced by changing the database system. By using a faster web-application server we have also shown that view-registration greatly benefits from additional processing power (CPU).

Clicks registration on the other hand, greatly benefitted from the additional database offered by the Sequoia cluster. Shortlinks were already known to be slow because they needed to do a select query first before storing the click with an insert statement. It were the select queries which on average took 150ms to 300ms, whilst the inserts were in the 1-10ms range. By load balancing the *performance determining* factor (select queries) a big performance gain was archieved.

Finally we have shown that the overal load experienced by M4N (except for the system processes) also benefits from a clustered database solution. Due to the additional database we observed that the load on the entire database system kept below 50% instead of rising to 90+% on the single database solution. Having a less burdened database-tier directly resulted in lower response times, which gives users a more responsive and faster website.

# Chapter 9

# Results and Conclusions

Summarizing we have achieved the following:

1. Created a framework for simulating web-site users.
2. Created a method for extracting sessions from web-access log files which can be used for simulation.
3. Used this simulation as a benchmark to measure the performance of the M4N system.
4. Deployed M4N in combination with Sequoia in a mirrored RAID configuration.
5. Used the same benchmarks to measure the performance of the M4N system using Sequoia.

From all of this we can conclude that Sequoia is capable to scale up database capacity by means of distribution. We also have shown that Sequoia offers benefits in term of decreased response times and increased throughput for database reliant services.

What we have not shown is some proof whether deploying Sequoia will cause any problems. Although we did manually check the website whether all pages still worked correctly; and we did spot and worked around some problems like the one with SQL sequences, we did not do a full thorough comparison. Comparing for instance the outputs of each SELECT query in M4N with and without Sequoia would have given some certainty about the correctness of Sequoia's implementation. Doing the same for all of M4N's INSERT statements and then checking up whether they are really put into the database would have further increased confidence in Sequoia. In this sense our research project did only give partial evidence about Sequoia's correctness. At this moment we thus can not give a 100% guarantee whether Sequoia is safe for use in combination with M4N.

Another feature which is present in Sequoia is the automatic recovery of database backends. In case one of the two (or more) databases fails, the other database(s) takes over. When a failed database has been restarted it is possible to add it back in the Sequoia cluster. Once added the database can be restored using the latest database dump using the Sequoia console. Using the so-called recovery database, which is maintained by Sequoia, all the records which have been updated and added after the dump are replayed automatically. To test this functionality we have used the click-benchmark:

1. Count the amount of registered clicks for both database backends.
2. Start the click benchmark.

3. After a minute: Stop the Postgres server on one of the backends using the *'kill PID'* command in Linux.
4. After some minutes: Restart the Postgres server.
5. Using the Sequoia console: Restore the crashed backend using the latest database dump. Once completed, re-enable the backend. Notice that re-enabling will automatically update the backend using the recovery log.
6. After some minutes: Stop the click-benchmark.
7. Count the registered clicks on both backends.
8. Check whether the backends registered an equal amount of clicks.

We gladly report that the click-counts corresponded with each other.

## 9.1 Increased Reliability

We measure reliability by looking at the chance to system failure. Say for instance we want to express the stability of the M4N Default configuration. We can define the chance the system is down as $P$. We then define $P(D)$ as the chance of a database failure and $P(W)$ as the chance of web-application server failure. The M4N Default system will go down if one or both of the servers fail, which can be expressed as:

$$P = onlyDBdown + onlyWEBdown + bothDown$$

$$P = P(D) \times (1 - P(W)) + (1 - P(D)) \times P(W) + P(D) \times P(W)$$

The chance for going down is also equal to one minus the chance of all servers being up, which yields the somewhat shorter formula:

$$P = 1 - ((1 - P(D)) \times (1 - P(W)))$$

Now using Sequoia's database distribution we have a different chance $P'$ for system failure. With Sequoia system failure only occurs when either the web-application server goes down, or *all* the database servers crash.

$$P' = 1 - atLeastSingleDBup \times WEBup$$

$$P' = 1 - ((1 - P(D)^n) \times (1 - P(W)))$$

Right now we have only neglected to take $P(S)$ into account which represents the chance the Sequoia controller itself crashes. Because Sequoia controllers can be grouped into clusters as well the Sequoia-tier can only fail when all the $m$ controllers fail.

$$P' = 1 - ((1 - P(S)^m) \times (1 - P(D)^n) \times (1 - P(W)))$$

One sees that using large values for $m$ and $n$ it is easy to decrease the chance of total system failure. In this scenario the chance $P(W)$ of web-application server failure is the constraining factor when $n$ and $m$ are sufficiently large. Running M4N on $o$ multiple servers (some M4N components need to be rewritten) however can further decrease the chance of system failure.

$$P' = 1 - ((1 - P(S)^m) \times (1 - P(D)^n) \times (1 - P(W)^o))$$

This final equation represents M4N in a fully clustered configuration.

## 9.2 Future Research

To facilitate a follow up research at M4N we present a list of possible subjects which could be researched further.

1. Full simulation including system processes
2. Long link benchmark: Less select statements as the short links, so presumably no speedup and perhaps lower performance when using Sequoia.
3. Benchmark other Sequoia RAID configurations: Different raid levels and nested raid levels.
4. Benchmark Sequoia using only a single database: to measure the overhead introduced by Sequoia.
5. Sequoia Scalability: Benchmark Sequoia in mirrored RAID mode, using 3, 4 and more databases to find out whether performance keeps increasing linearly.
6. Additional reliability testing of the automatic database recovery feature of Sequoia: While doing a full simulation benchmark, push the power button of one of the two Sequoia databases and re-enable it after a while. Check whether both databases are consistent again after the benchmark has ended (15 or 30 minutes or so).
7. Validate the correctness of Sequoia: Measure whether each select query gets the same results from Sequoia as from a regular JDBC driver. Might be done by using a script to execute each XPQL query of M4N in a fixed order and writing the output to a file. Doing this for both the Sequoia and the regular JDBC driver will give two files which can be easily compared.
8. Validate whether insert statements are stored correctly.
9. Extra features for the load scheduler and web-log analyzer:
   - Add script execution and Quartz jobs functionality to the load test scheduler.
   - Improve configuration method for web-log sessions generation (XML and passing properties to parameter handlers)
   - Automate or improve interface for web-log sessions generation
   - For the user-session clustering it is possible to further experiment with other clustering parameters and algorithms. Possibly also improving the 'similarity function'.

# Bibliography

[1] *M4N Homepage* http://www.m4n.nl/

[2] *M4N Frequently Asked Questions*, http://www.m4n.nl/faq.jsf

[3] *M4N User Guide*, http://www.m4n.nl/user_guide.jsp

[4] *M4N Table of Definitions*, http://www.m4n.nl/def.jsp

[5] *Wikipedia, Online Advertising*, http://en.wikipedia.org/wiki/Online_advertising

[6] *The Grinder*, http://grinder.sourceforge.net/

[7] *The Grinder: Getting Started*, http://grinder.sourceforge.net/g3/getting-started.html

[8] *HTMLUnit Website*, http://htmlunit.sourceforge.net/, http://htmlunit.sourceforge.net/gettingStarted.html

[9] *Selenium Web Testing Website*, http://www.openqa.org/selenium/

[10] *Canoo Web Testing Whitepaper*, http://webtest.canoo.com/webtest/manual/whitepaper.html

[11] B. Julien, D. Nicolas, *An Overview of Load Test Tools*, http://clif.objectweb.org/load_tools_overview.pdf

[12] *Apache JMeter Website*, http://jakarta.apache.org/jmeter/

[13] *Mercury Loadrunner Website*, http://www.mercury.com/us/products/performance-center/loadrunner/

[14] *Clustering Techniques: The K-means Algorithm*, http://dms.irb.hr/tutorial/tut_clustering_short.php

[15] *Wikipedia, Data Clustering*, http://en.wikipedia.org/wiki/Data_clustering

[16] R. Ali, U. Ghani, A. Saeed, *Data Clustering and Its Applications*, http://members.tripod.com/asim_saeed/paper.htm

[17] M. Perkowitz, *The PageGather Algorithm*, http://www8.org/w8-papers/2b-customizing/towards/node7.html

[18] T. Germano, *Self Organizing Maps*, http://davis.wpi.edu/~matt/courses/soms/

[19] *Sequoia Hompage*, http://sequoia.continuent.org/HomePage

[20] *Sequoia User Guide*, http://sequoia.continuent.org/doc/latest/userGuide/index.html

[21] A. Joshi, K. Joshi, R. Krishnapuram, *On Mining Web Access Logs*, ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000, pages 63-69, http://ebiquity.umbc.edu/get/a/publication/45.pdf

[22] B.M. Subraya, S.V. Subrahmanya, *Object Driven Performance Testing of Web Applications*, 1st Asia-Pacific Conference on Quality Software (APAQS 2000), 30-31 October 2000, Hong Kong, China, Proceedings, pages 17-28, ISBN: 0-7695-0825-1, http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=883774

[23] F. Ricca, P. Tonella, *Analysis and Testing of Web Applications*, Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada, pages 25-34, ISBN: 0-7695-1050-7, http://www.cs.umd.edu/~atif/Teaching/Fall2002/StudentSlides/Adithya.pdf

[24] S. Elbaum, S Karre, G Rethermel, *Improving Web Application Testing with User Session Data*, Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA, pages 49-59, http://esquared.unl.edu/articles/downloadArticle.php?id=76

[25] K. McGarry, A. Martin, M. Addison, J. MacIntyre, *Data Mining and User Profiling for an E-Commerce System* FSDK'02, Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery: Computational Intelligence for the E-Age, 2 Volumes, November 18-22, 2002, Orchid Country Club, Singapore, pages 682-, http://osiris.sunderland.ac.uk/~cs0kmc/FSKD_B_15.pdf

[26] M. Grcar, *User Profiling: Web Usage Mining*, Conference on Data Mining and Warehouses (SiKDD 2004) October 15, 2004, Ljubljana, Slovenia, http://kt.ijs.si/Dunja/SiKDD2004/Papers/MihaGrcar-WebUsageMining.pdf

[27] O. Nasraoui, H. Frigui, A. Joshi, R. Krishnapuram, *Mining Web Access Logs Using Relational Competitive Fuzzy Clustering*, International Journal on Artificial Intelligence Tools, volume 9, number 4, 2000, pages 509-526, http://www.louisville.edu/~o0nasr01/Websites/PAPERS/conference/Nasraoui-IFSA-99-mining-web-access-logs.pdf

[28] J. Pei, J. Han, B. Mortazavi-asl, H. Zhu, *Mining Access Patterns Efficiently from Web Logs*, Knowledge Discovery and Data Mining, Current Issues and New Applications, 4th Pacific-Asia Conference, PADKK 2000, Kyoto, Japan, April 18-20, 2000, Proceedings, pages 396-407, ISBN: 3-540-67382-2, http://portal.acm.org/citation.cfm?id=693333&dl=ACM&coll=ACM

[29] O. Zaane, M. Xin, J. Han, *Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs*, ADL, 1998, pages 19-29, http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=785951

[30] P. Batista, M.J. Silva, *Mining Web Access Logs of an On-line Newspaper*, 2nd International Conference on Adaptive Hypermedia and Adaptive Web Based Systems, 29/5/2002 - 31/5/2002 Malaga, Spain, http://ectrl.itc.it/rpec/RPEC-Papers/11-batista.pdf

[31] W. Wang, O.R. Zaane, *Clustering Web Sessions by Sequence Alignment*, 13th International Workshop on Database and Expert Systems Applications (DEXA 2002), 2-6 September 2002, Aix-en-Provence, France, pages 394-398, ISBN: 0-7695-1668-8, `http://doi.ieeecomputersociety.org/10.1109/DEXA.2002.1045928`

[32] Z. Huang, J. Ng, D.W. Cheung, M.K. Ng, W. Ching, *A Cube Model for Web Access Sessions and Cluster Analysis*, WEBKDD 2001 - Mining Web Log Data Across All Customers Touch Points, Third International Workshop, San Francisco, CA, USA, August 26, 2001, Revised Papers, publisher Springer, Volume 2356, year 2002, pages 48-67, ISBN: 3-540-43969-2, `http://ai.stanford.edu/~ronnyk/WEBKDD2001/huang.pdf`

[33] J. Vesanto, E. Alhoniemi, *Clustering of the Self-Organizing Map*, IEEE Transactions on Neural Networks, Vol. 11, No. 3, May 2000, pages 586-600, `http://lib.tkk.fi/Diss/2002/isbn9512258978/article7.pdf`

[34] E.H. Chi, A. Rosien, J. Heer, *LumberJack: Intelligent Discovery and Analysis of Web User Traffic Composition*, WEBKDD 2002 - MiningWeb Data for Discovering Usage Patterns and Profiles, 4th International Workshop, Edmonton, Canada, July 23, 2002, Revised Papers, publisher Springer, series lecture notes in computer science, volume 2703, year 2003, pages 1-16, ISBN: 3-540-20304-4, `http://citeseer.ist.psu.edu/chi02lumberjack.html`

[35] A. Abraham, V. Ramos, *Web Usage Mining Using Artificial Ant Colony Clustering and Linear Genetic Programming*, in CEC'03 Congres on Evolutionary Computation, IEEE Press, Canberra, Australia, 8-12 Dec. 2003, pages 1384-1391, ISBN: 078-0378-04-0, `http://alfa.ist.utl.pt/~cvrm/staff/vramos/Vramos-CEC03b.pdf`

[36] PostgreSQL 8.1.4 Documentation, *On-line backup and point-in-time recovery (PITR)*, `http://www.postgresql.org/docs/8.1/interactive/backup-online.html`

[37] *Quartz, Enterprise Job Scheduler*, `http://www.opensymphony.com/quartz/`

[38] *The Xen Virtual Machine Monitor* `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`

[39] D. Arthur, S. Vassilvitskii, *How Slow is the k-means Method*, Proceedings of the 2006 Symposium on Computational Geometry (SoCG), Pages: 144 - 153, Year: 2006, ISBN:1-59593-340-9 , `http://www.stanford.edu/~sergeiv/papers/kMeans-socg.pdf`

# Appendix A

# The WeblogClusterer

Adapted version of the WeblogAnalyzer written by Arjan Tijms. The WeblogClusterer is used to extract sessions (in a JUnit test compatible Java format) and to use a clustering algorithm to find out which sessions resemble each other. Aided by this clustering one can select the most frequently occurring types of user sessions.

Usage:

1. Get log files. Catalina.out files using a *custom* format (see figure 4.2).

2. Add log files which we want to use for clustering

3. Click the 'Start Scan' button: The separate sessions will be extracted and stored per user class in the sessions directory in a Java format. These are just java functions with an unique ID number which can be used for load-testing using Grinder and our Load-TestScheduler. This step will furthermore generate session profiles in the *profiles* directory. These profiles are in fact thinned out sessions, only containing page-name and ordering information.

4. Click the preferences button: A window pops up and one can set the iterations and number of threads to use during the K-Means algorithm.

5. Click the *Create Scenarios from Profiles* button: Given these profiles we will try to cluster them using the K-Means algorithm. The result can be found in the *clusters* directory.

## A.1 K-means performance

Normally K-means iterates until some criteria are met (convergence of the algorithm). In practice most problem instantiations of K-means converge extremely quick (far less iterations as points in the dataset). However, for some datasets K-means requires super-polynomial time[39]: $2^{\Omega(\sqrt{n})}$.

Our algorithm does not use convergence as a stop criterion but uses a fixed number of iterations. So for our special case, K-means seems to scale linearly with $K$ and the number of iterations $I$: $O = K \times I$.

To see how well our multi-threaded algorithm scales when given more CPU's we did some tests (see section A.2). These measurements suggest that our K-means algorithm enjoys an average speedup of 1.5 for small $K$ (around 40), and a speedup between 1.8 to 1.9 for larger $K$ values ($K \approx 700$).

## A.2 K-means experiment

### System specs

| | |
|---|---|
| CPU | AMD Athlon 64 X2 Dual Core Processor 3800+ |
| MEM | 1GB |
| OS | Linux version 2.6.15-1-k7-smp (Debian 2.6.15-8) |

### Log Size

For this analysis, as well as for generating the actual load tests we have used the live M4N log files from the 14th of March until the 24th of April 2006. The number of sessions which were found in these logs amounted to 30,604 In total these encompassed 820,576 HTTP requests.

### Clustering Performance Results

#### Small K values

$K = S^{0.4}$, where S is the number of sessions. This yielded a total of around 160 clusters.

| Iterations | 1 thread | 2 threads | 4 threads | 8 threads |
|---|---|---|---|---|
| 40 | 530 sec | 329 sec | 329 sec | 333 sec |
| 80 | 1058 sec | 707 sec | 649 sec | 699 sec |
| 120 | 1540 sec | 1047 sec | 1009 sec | 1103 sec |

#### Bigger K values

$K = S/20$, where S is the number of sessions. This yielded a total of around 1517 clusters divided among 2 large user-classes and several smaller ones.

| Iterations | 1 thread | 2 threads |
|---|---|---|
| 40 | 2996 sec | 1743 sec |
| 80 | 5938 sec | 3163 sec |
| 120 | 9605 sec | 4570 sec |
| 400 | 30858 sec | 16773 sec |

# Appendix B

# Time-line: Activities During Internship

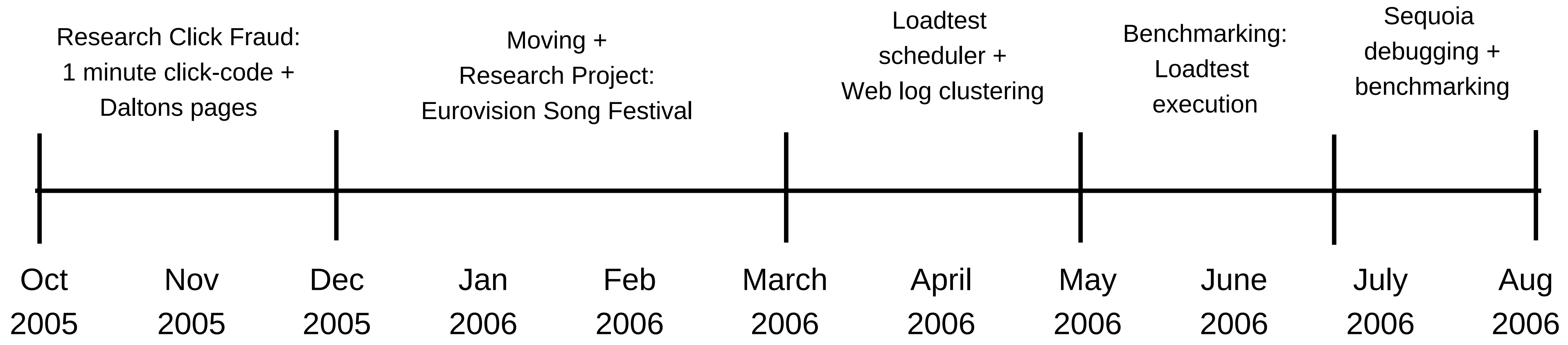


Figure B.1: Activities visualized on a time-line

## B.1 Activity Description

### October

- Researching the 1 minute click-code. Investigate the effect when one automatically disapproves clicks according to the following criteria:
    1. Both clicks have the same IP-address.
    2. They share the same banner_id (or affiliate)
    3. They have occurred within 1 minute from each other
- Research 'daltons' (click fraud): Detecting click fraud by looking at the registered clicks in the database. Also created the so-called Dalton pages in JSP. These pages serve as an front-end for the detection and handling of click-fraud.

### December

Until the start of February I moved into my new house. Once settled I started finalizing my *'Research Project'* about the Eurovision Song Festival.

### March

Creating the Loadtest Scheduler:

- Try out different loadtest applications such as the Grinder, Selenium, JMeter and some others
- Developing the Loadtest Scheduler: Combining HTMLUnit with the Grinder and in particular passing statistics from HTMLUnit back to the Grinder Console, proved to be problematic. This single problem alone cost me two to three weeks.

### April

- Research web log clustering: Tried out and tested some simple Web log clustering algorithms and finally chose to use K-Means. The clustering results yielded our first sessions in a Grinder specific format (*weblogsessions_v1*).
- Load test generation: We had to adapt the loadtest scheduler to properly catch all JUnit assertions and pass those to the Grinder Console as being test failures.
- Rewrote the generated web-log sessions into a JUnit-test format (*weblogsessions_v2*).

### Start of May

Changed our web-log clustering to perform the K-means algorithm in parallel. This increased the speed of the web log clustering significantly (used to take some days). Further optimizations (by profiling the code) improved the clustering even more by short-cutting the session-comparison function[1].

[1]Comparing sessions longer as 100 pages was considered irrelevant. We shortcutted this by only comparing the first 100 pages alone.

## Middle of May/start of June

Performing load test benchmarks:

- View benchmark: The loadtest scheduler was not capable to produce a decent amount of views per second. Adapting the scheduler using a threadpool and some other tweaks greatly increased the load our scheduler could generate.
- Click benchmark
- Session benchmark: Some problems with the development server which required us to dynamically rewrite all login emails with *'test@mbuyu.nl'*. Logins found in the log naturally used real email addresses, which are always renamed in the development database for security reasons.

## End of June

- Adapting the web-log sessions: Extracting the GET parameters from the request URLs and use Java variables instead. An additional requirement was to support a configuration file which couples parameter and page names with pluggable Java handlers for those parameters (*weblogsessies_v3*).
- Getting M4N running on Sequoia: Bugs in the Sequoia's SQL-parser prohibited a quick test run using Sequoia. It should be fixed with a find-and-replace action on all the SQL (.xpql and .sql) files of M4N.

## July

- Perform the full-simulation benchmark on the current M4N infrastructure (non-Sequoia).
- Performing all the benchmarks on M4N using Sequoia (mirror database).

## End of July

Finish the report.

# Appendix C

# Project plan: Database clustering

**Abstract:**

The project concerns the transition of a large web application from a single DB to a clustered DB system.

**Keywords:**

Simulation of query patterns (live testing is not possible because of ongoing business concerns) Optimizing (performance, space requirements, for evaluation a simulation is necessary) Risk management (redundancy, prevent a single point of failure) Cost analysis (trade off -> hardware is not available in unlimited quantities) Development (create specific code patches and/or solve (known) limitations in the involved architectures)

**Project should be based on existing technology:**

1. Stress/load testing: The Grinder -> http://grinder.sourceforge.net The grinder is an existing load testing framework. A server can be tested is a distributed way from multiple clients. Tests can be written in Python (via Jython) or directly in Java code. When necessary a contribution can be made to the framework itself.

2. Database clustering: Sequoia -> https://forge.continuent.org/projects/sequoia/ Sequoia is a raid system for Dbs with a strong academic background. The lead architect is Emmanuel Cecchet, a former PHD and postdoc. Universities as the Rice University and the University of Toronto have made contributions to this project in the past. A (scientific) contribution to the code base of Sequoia that is beneficial to both Mbuyu as the Sequoia project would be ideal.

## C.1 Approach:

### Load testing:

It is the intend that the Mbuyu specific traffic patterns are recognized and extracted from existing server log files. For this we differentiate between traffic generated from a web application (called M4N) and traffic originating from so-called views and clicks servers. This part of the project has a small AI component.

From the recognized traffic patterns, a simulation is to be written that simulates a typical load of the system. For this simulation The Grinder can be used. Using an other test framework or possibly own code is an option.

After the system's load can be tested the actual DB clustering can be considered. For this it is necessary to install the entire M4N system (= the software MBUYU developed). This includes development tools (Eclipse + a set of plug-ins) as well as the software stack to run M4N on: a Java EE server (Tomcat 5.5) and the single DB (Postgresql 8.x).

### DB clustering:

After successfully completing the first phase, the installation of Sequoia can be performed. The first test is to see whether M4N runs at all on Sequoia. Possible problems have to be tracked down using the (Java) source of Sequoia and M4N. This sub phase is quite system management / programming technical oriented.

Once up and running an investigation to the optimal algorithm/configuration for the actual DB clustering can be started. Aspects to take into consideration are one or multiple controllers, two or more DB servers, performance vs safety, scalability (adding a huge amount of extra servers, possibly using a cluster) and all of this plotted against the cost. Through a scheduling plug-in system of Sequoia it is possible to add custom scheduling algorithms with relatively ease. Here too it might be possible that problems/bugs are discovered in either the Sequoia or M4N source code, for which a solution has to be found of course. This sub phase is the most research-technical oriented one.

Optionally an extension on the project is possible, for example when there are little to no problems discovered in the previous (sub) phases. In that case this project may be finished too fast for a scientific master project. This extension could entail the analysis of the existing M4N software architecture; mainly the view and click servers. This architecture can then be changed in order to investigate whether a better architecture can be found in the context of DB clustering. (although the DB clustering itself is transparent, it may be the case that certain other traffic patterns perform better in a clustered environment than the current patterns do.)

# Appendix D

# User Manual: Web-log Clusterer and Loadtest Scheduler

To offer some support on the web-log clusterer and the loadtest scheduler we here provide a small user manual on how to work with both.

The function of these two programs is to extract and convert sessions found in a web-access log into JUnit Tests (Web-log Clusterer) and to execute these sessions on multiple remote computers (Loadtest Scheduler) using the Grinder. To do so the following steps need to be taken:

1. Fetch log files
2. Use weblog-clusterer to create sessions (from logs)
3. Create session clusters using the weblog-clusterer
4. Using these clusters (in weblogclusterer/clusters/) select which sessions (in weblogclusterer/sessions/) to use for load-testing
5. Copy the java code from the selected sessions into the appropriate grinder directory: grinder/lib/sessions/xxx.java
6. Compile the new sessions
7. Run the Grinder

## D.1 User scenario

1. Get web-logs
2. Open Weblog clusterer
3. Optionally: Open codegenerations.props For HTTP GET parameters it is possible not to use the log values but to add custom handlers which uses the previous page as a reference.
4. Go to the preferences:
   - Make sure the database connection info is correct
   - Configure the clustering algorithm: Extra threads come in handy on multi-processing systems. Increasing the number of iterations will result in more accurate clustering (which is an iterative process).
5. Open web-log files to be analyzed
6. Press the analyze button
7. Wait a while: Session files (in <*cwd*>*/sessions/*) will be generated grouped in separate directories on user-class. These can be identified by the user-class combined with an unique number (*sessions/userclass/id.java*). Profile files (in <*cwd*>*/profiles/*) for each user-class will be generated as well.
8. From these profiles the weblog clusterer will perform the next step.
9. Press the cluster profiles button
10. Wait another while
11. Cluster files have been generated (in <*cwd*>*/clusters/*) for each user-class.
12. Open them: One now sees the sessions which belong to a certain cluster (format of each line: ID user-class page1 page2 page3 ... pageN). The sessions which you want to use for the loadtest can be found in: <*cwd*>*/sessions/user-class/ID.java*
13. Select some sessions (Java functions) at random or using common sense and place them into the *grinder/lib/sessions/UserClass-Name.java*. The LoadtestScheduler will use the sessions which are present in these java files found in its sessions directory.
14. Compile java files in *grinder/lib/sessions/*
15. Start the Grinder console (*console*) and a Grinder agent (*startGrinder*)
16. Look in the grinder.properties file and the *loadtest.properties* file for configuring the load-tester.
17. Load-testing can now begin using your new sessions!

# Appendix E

# Context.XML used during Sequoia Tests

During M4N deployment using Sequoia some changes needed to be made in the context.xml in the Tomcat application server. These were not found in the manual and thus will be listed here:

- type="org.continuent...": Use the Sequoia datasource.
- factory="org.continuent...": Use the Sequoia datasource factory.
- user="postgres": The Sequoia datasource factory requires a *'user'* field instead of the username field which was used by the regular Tomcat factory.

```
<Resource name="jdbc/m4n_dba"

    scope="Shareable"
    type="org.continuent.sequoia.driver.DataSource"
    auth="Container"
    factory="org.continuent.sequoia.driver.DataSourceFactory"
    accessToUnderlyingConnectionAllowed="false"

    defaultAutoCommit="true"
    defaultReadOnly="false"
    defaultTransactionIsolation="READ_COMMITTED"

    validationQuery="select 1"
    testOnBorrow="true"
    testOnReturn="false"
    testWhileIdle="false"

    initialSize="20"
    minIdle="0"
    maxIdle="0"
    maxActive="20"
    maxWait="10000"
    removeAbandoned="true"
    removeAbandonedTimeout="10"
    logAbandoned="true"

    timeBetweenEvictionRunsMillis="2000"
    minEvictableIdleTimeMillis="20000"
    numTestsPerEvictionRun="5"

    driverClassName="org.continuent.sequoia.driver.Driver"
    username="postgres" password=""
    user="postgres"
    url="jdbc:sequoia://localhost/m4n_sequoia"
/>
```

# Appendix F

# Sequoia Configuration Files

## F.1 Controller Config

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE SEQUOIA-CONTROLLER PUBLIC "-//Continuent//DTD SEQUOIA-CONTROLLER 2.9//EN"
    "http://sequoia.continuent.org/dtds/sequoia-controller-2.9.dtd">
<SEQUOIA-CONTROLLER>
  <Controller ipAddress="webNode2" port="25322">
    <Report/>
    <JmxSettings>
      <RmiJmxAdaptor/>
    </JmxSettings>
    <VirtualDatabase configFile="m4n-raidb1.xml" virtualDatabaseName="m4n_sequoia"
      autoEnableBackends="force"/>
  </Controller>
</SEQUOIA-CONTROLLER>
```

## F.2 Virtual Database Config

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SEQUOIA PUBLIC "-//Continuent//DTD SEQUOIA 2.9//EN"
    "http://sequoia.continuent.org/dtds/sequoia-2.9.dtd">

<SEQUOIA>
  <VirtualDatabase name="m4n_sequoia">

    <AuthenticationManager>
      <Admin>
        <User username="admin" password=""/>
      </Admin>
      <VirtualUsers>
        <VirtualLogin vLogin="postgres" vPassword=""/>
      </VirtualUsers>
    </AuthenticationManager>
    <DatabaseBackend name="postgresql-dbaseNode11" driver="org.postgresql.Driver"
      url="jdbc:postgresql://dbaseNode1.office.mbuyu.nl/m4n_20060619"
      connectionTestStatement="select now()">
        <ConnectionManager vLogin="postgres" rLogin="postgres" rPassword="">
            <VariablePoolConnectionManager initPoolSize="20" minPoolSize="20"
             maxPoolSize="20" idleTimeout="0" waitTimeout="0"/>
        </ConnectionManager>
    </DatabaseBackend>

    <DatabaseBackend name="postgresql-dbaseNode2" driver="org.postgresql.Driver"
      url="jdbc:postgresql://172.16.26.130/m4n_20060619"
      connectionTestStatement="select now()">
        <ConnectionManager vLogin="postgres" rLogin="richard" rPassword="">
            <VariablePoolConnectionManager initPoolSize="20" minPoolSize="20"
             maxPoolSize="20" idleTimeout="0" waitTimeout="0"/>
        </ConnectionManager>
    </DatabaseBackend>

   <RequestManager caseSensitiveParsing="false">

      <RequestScheduler>
         <RAIDb-1Scheduler level="passThrough"/>
      </RequestScheduler>

      <LoadBalancer>
         <RAIDb-1>
            <WaitForCompletion policy="first"/>
            <RAIDb-1-LeastPendingRequestsFirst/>
         </RAIDb-1>
      </LoadBalancer>
    </RequestManager>
  </VirtualDatabase>
</SEQUOIA>
```

# Appendix G

# View Benchmark Load Graphs

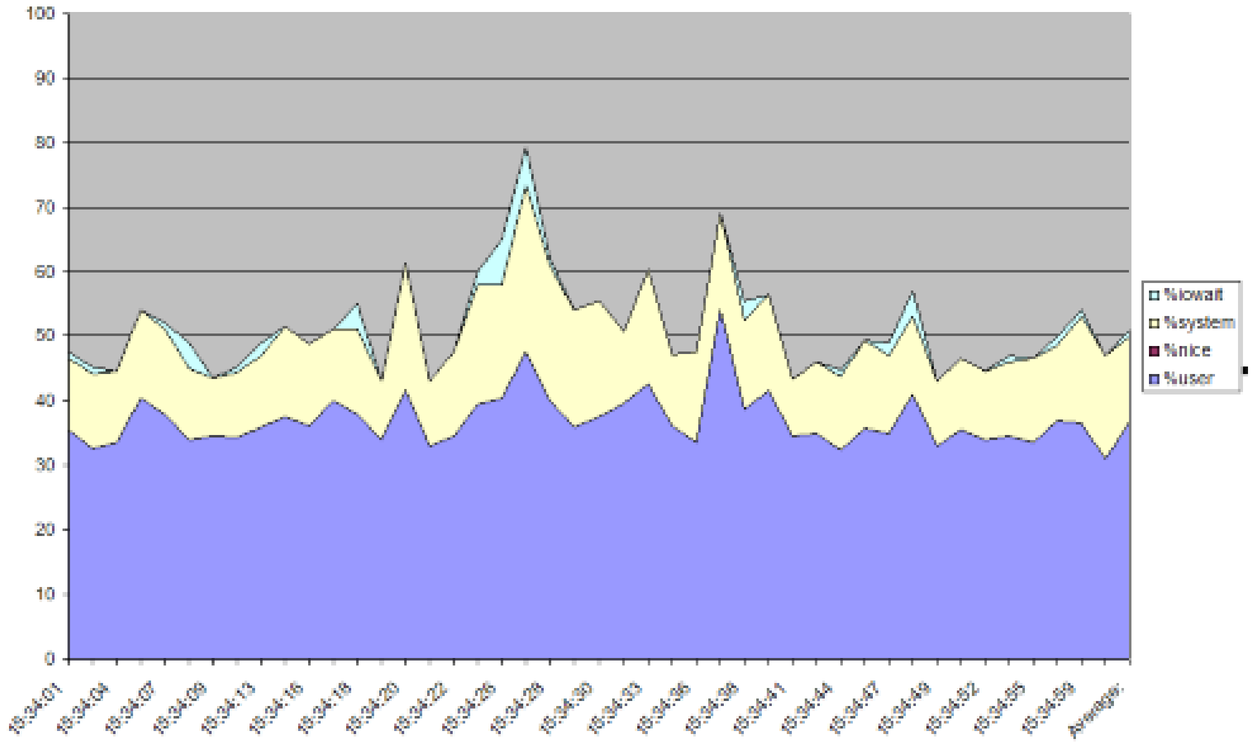


Figure G.1: Load statistics on webNode1 during the M4N Slow view benchmark: 1 Client

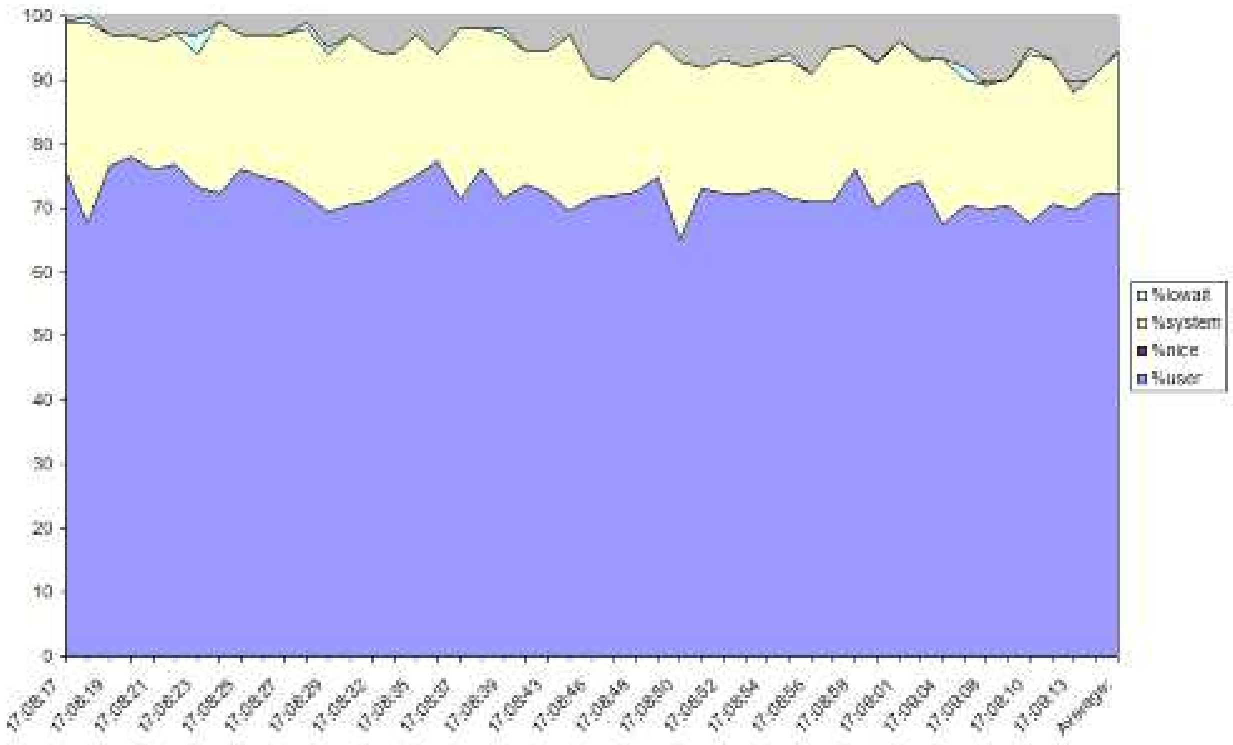


Figure G.2: Load statistics on webNode1 during the M4N Slow view benchmark: 4 Clients

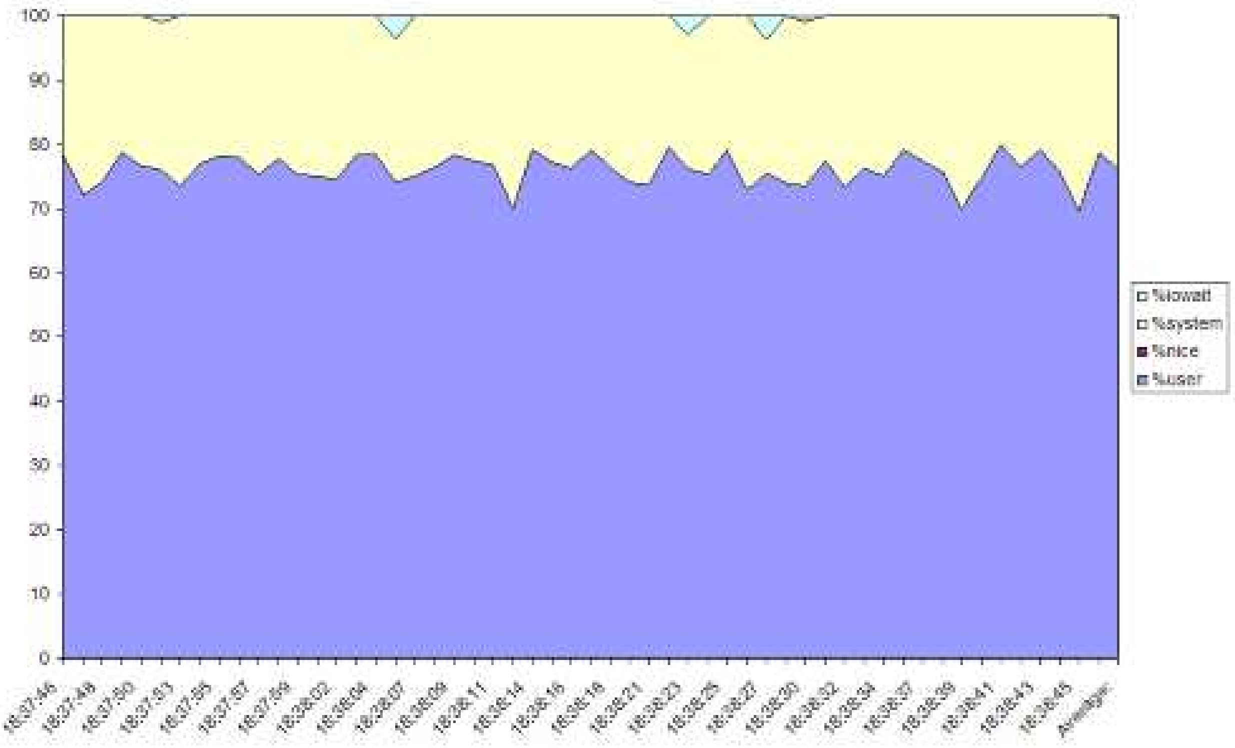


Figure G.3: Load statistics on webNode1 during the M4N Slow view benchmark: 7 Clients

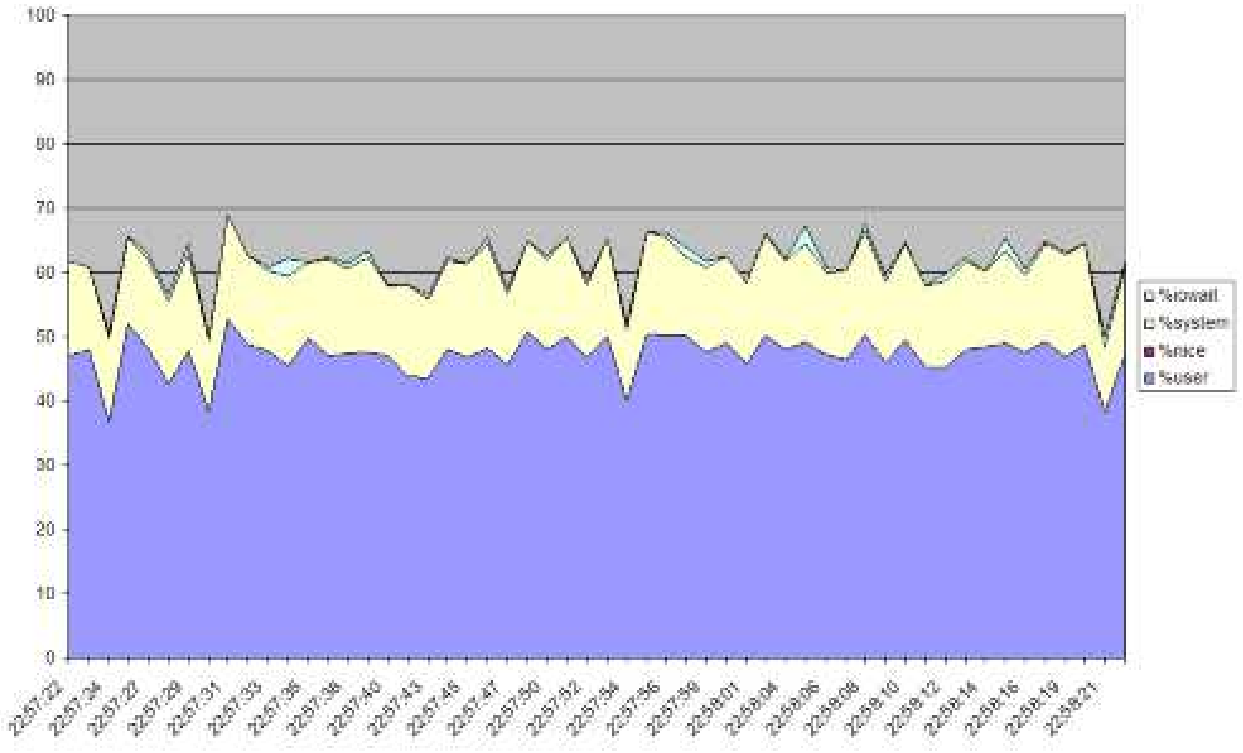


Figure G.4: Load statistics on webNode2 during the M4N Default view benchmark: Intensity 1, 400 views/second

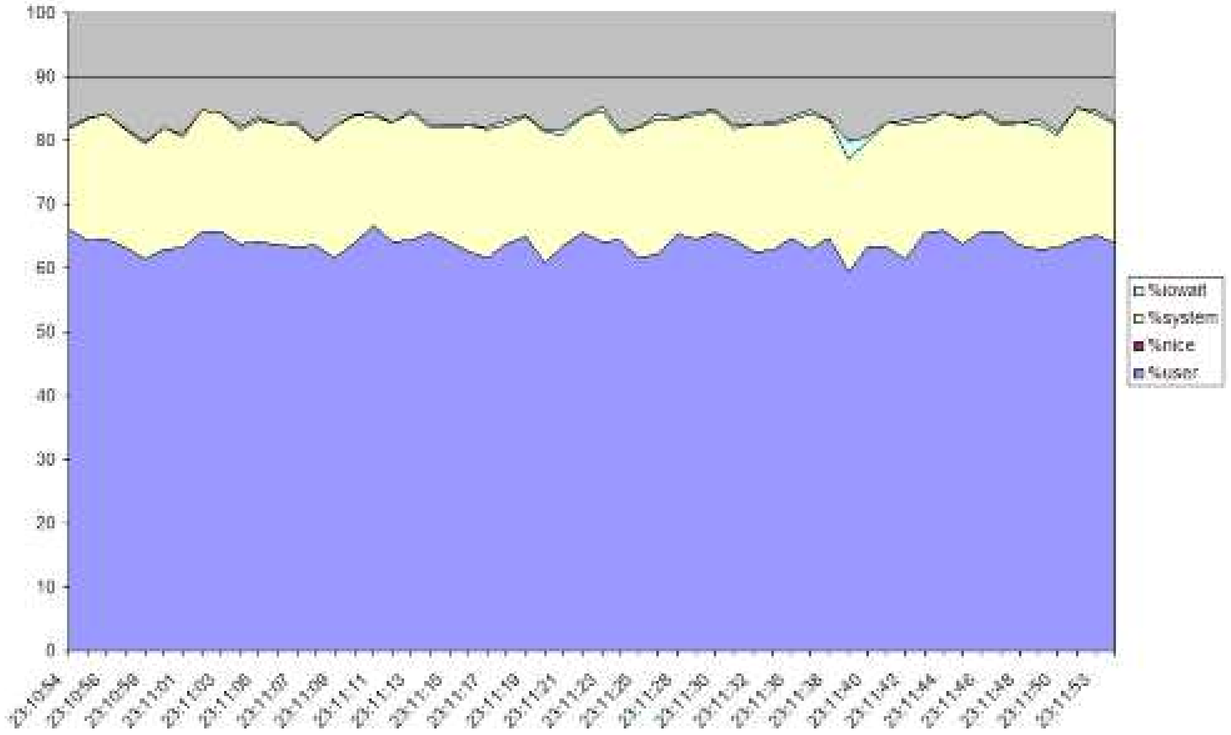


Figure G.5: Load statistics on webNode2 during the M4N Default view benchmark: Intensity 2, 500 views/second

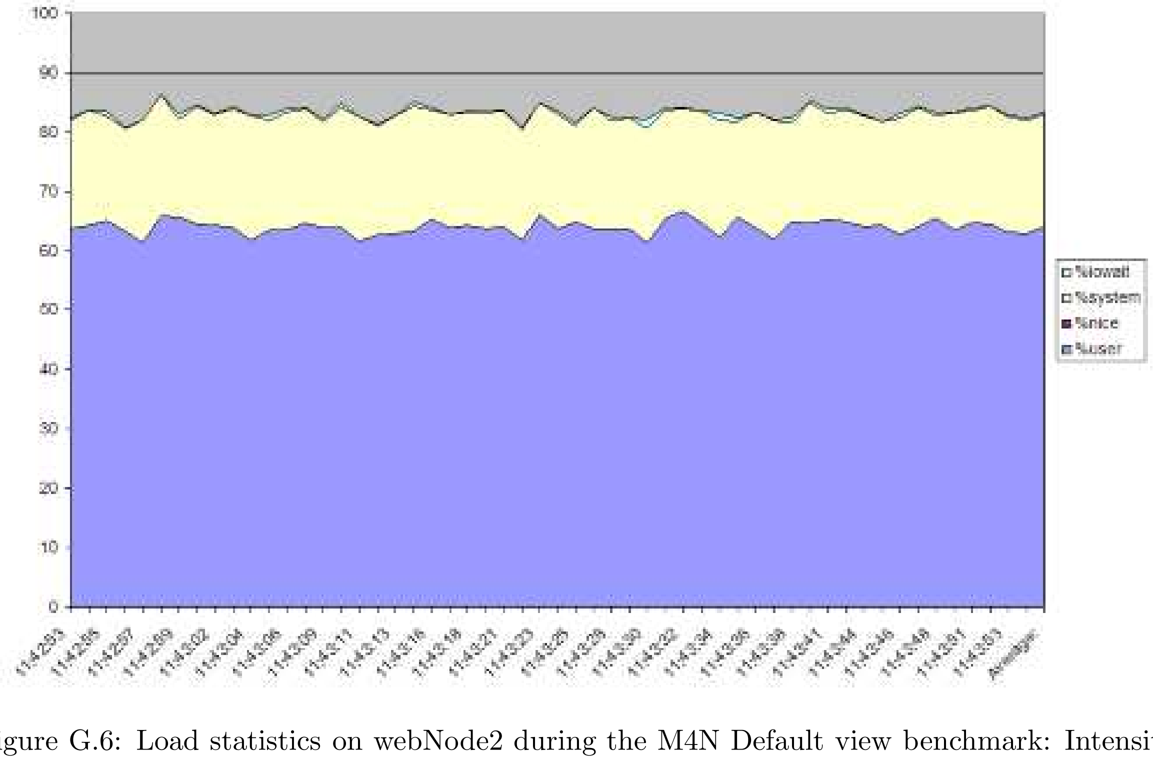


Figure G.6: Load statistics on webNode2 during the M4N Default view benchmark: Intensity 3, 900 views/second.

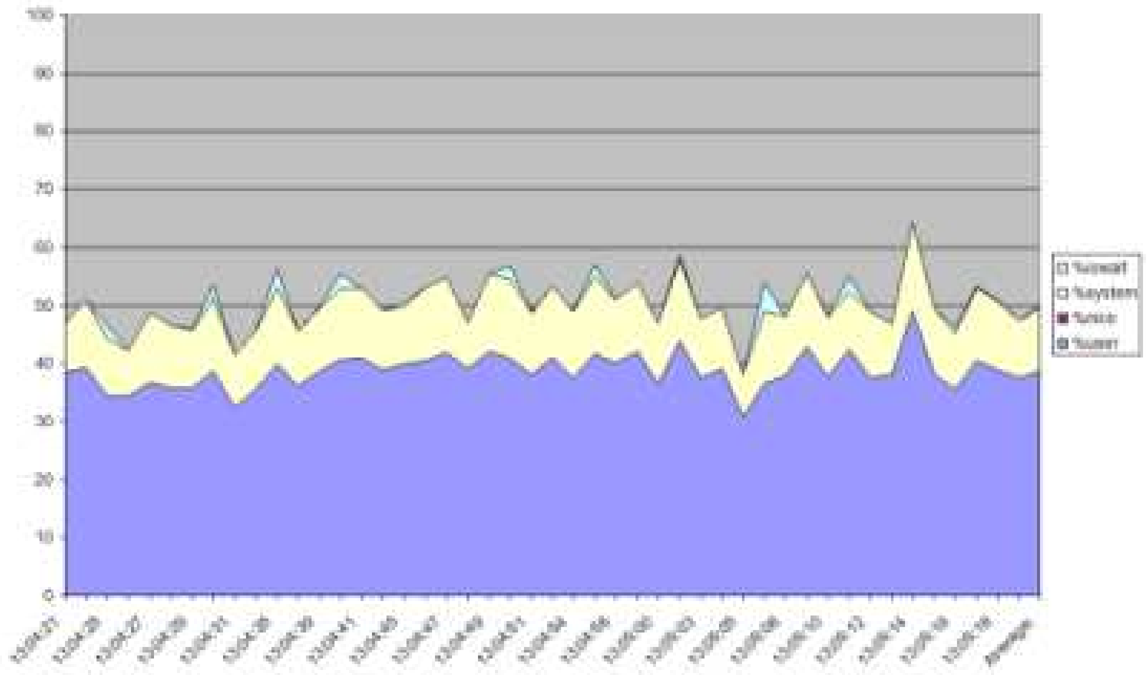


Figure G.7: Load statistics on webNode2 during the M4N Sequoia view benchmark configured with 400 views/second.

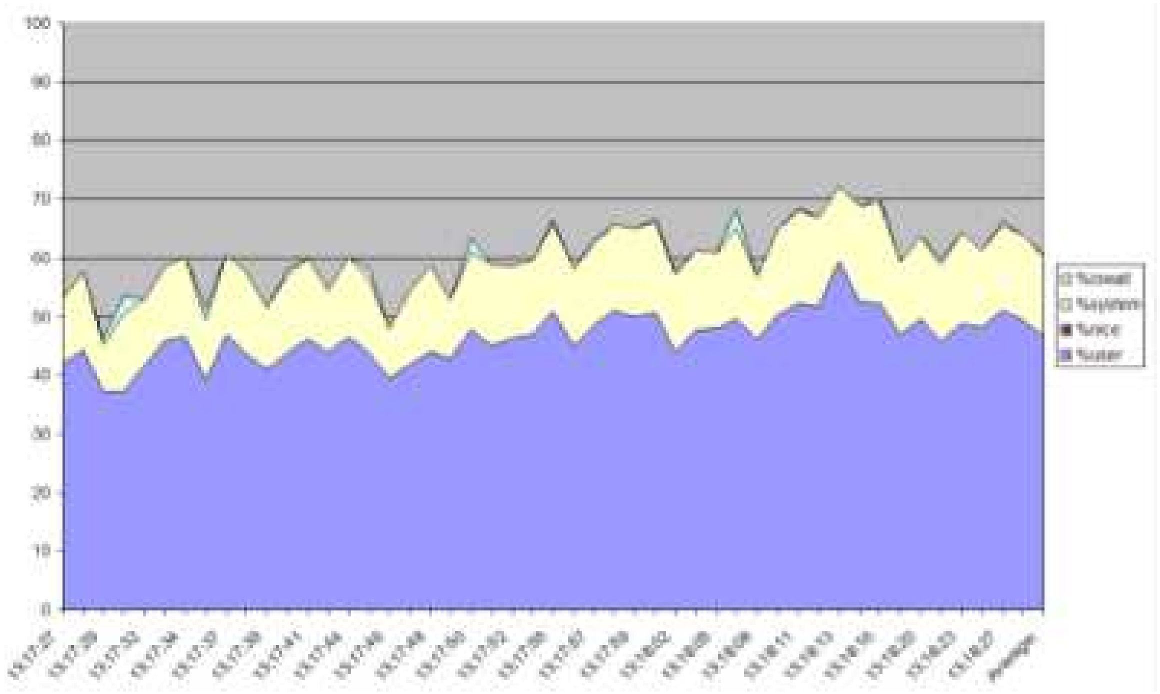


Figure G.8: Load statistics on webNode2 during the M4N Sequoia view benchmark configured with 500 views/second.

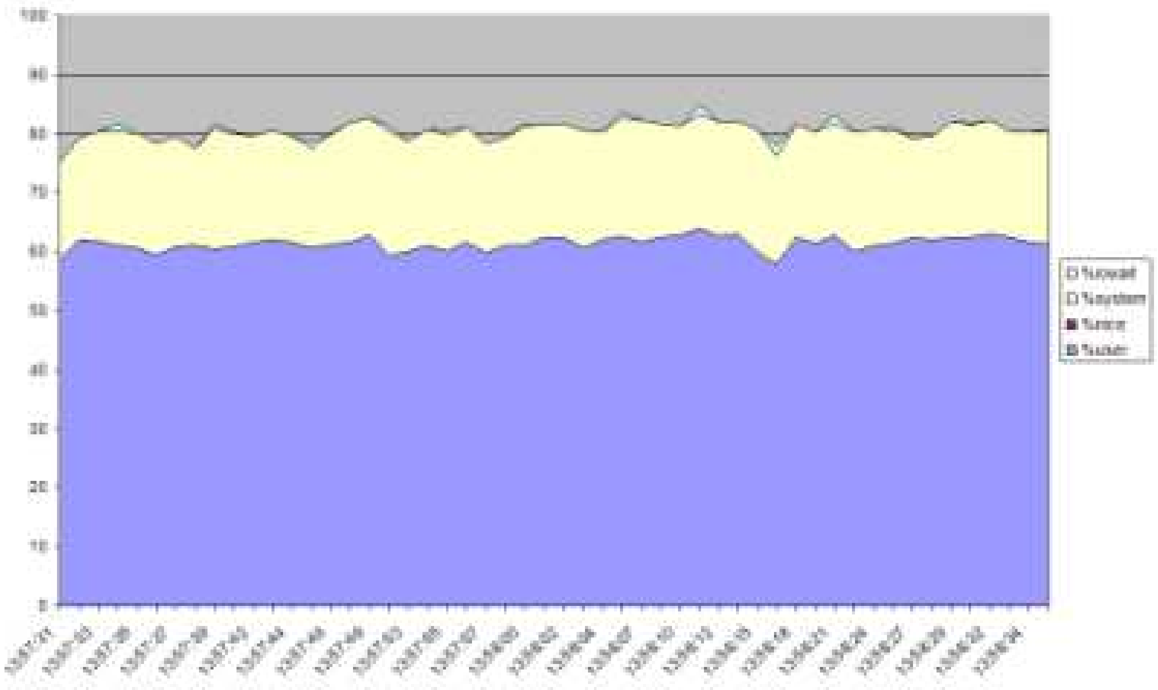


Figure G.9: Load statistics on webNode2 during the M4N Sequoia view benchmark configured with 900 views/second.

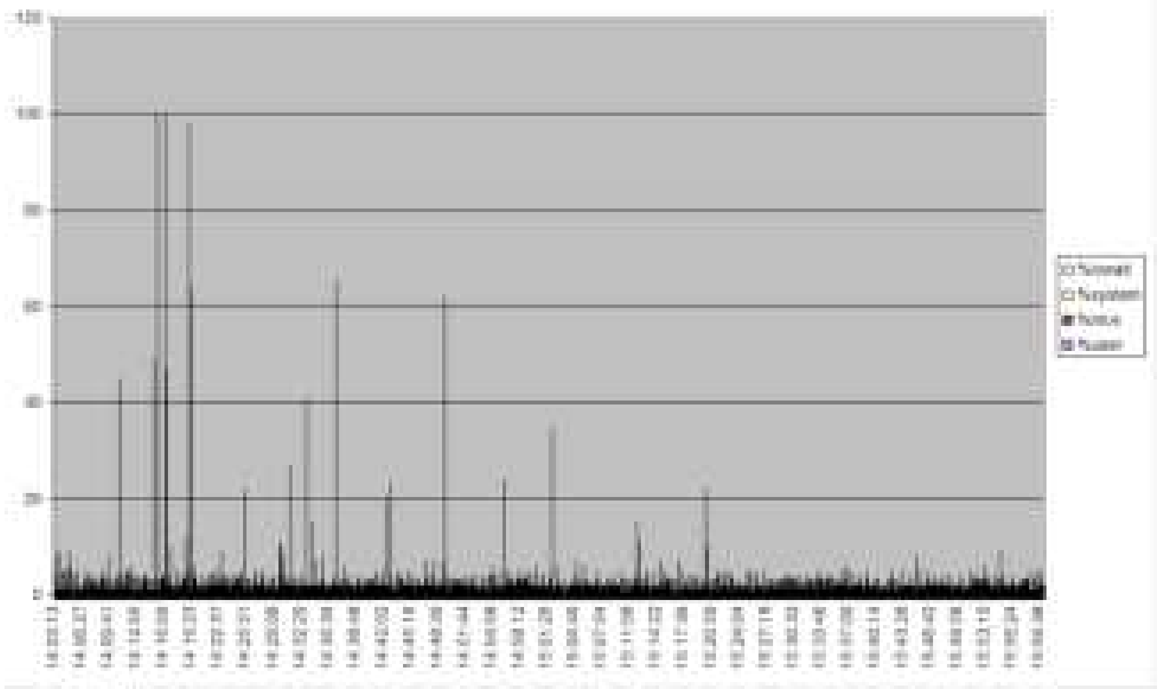


Figure G.10: Load statistics for dbaseNode1 (database server) during the entire period of the M4N Sequoia view benchmark.

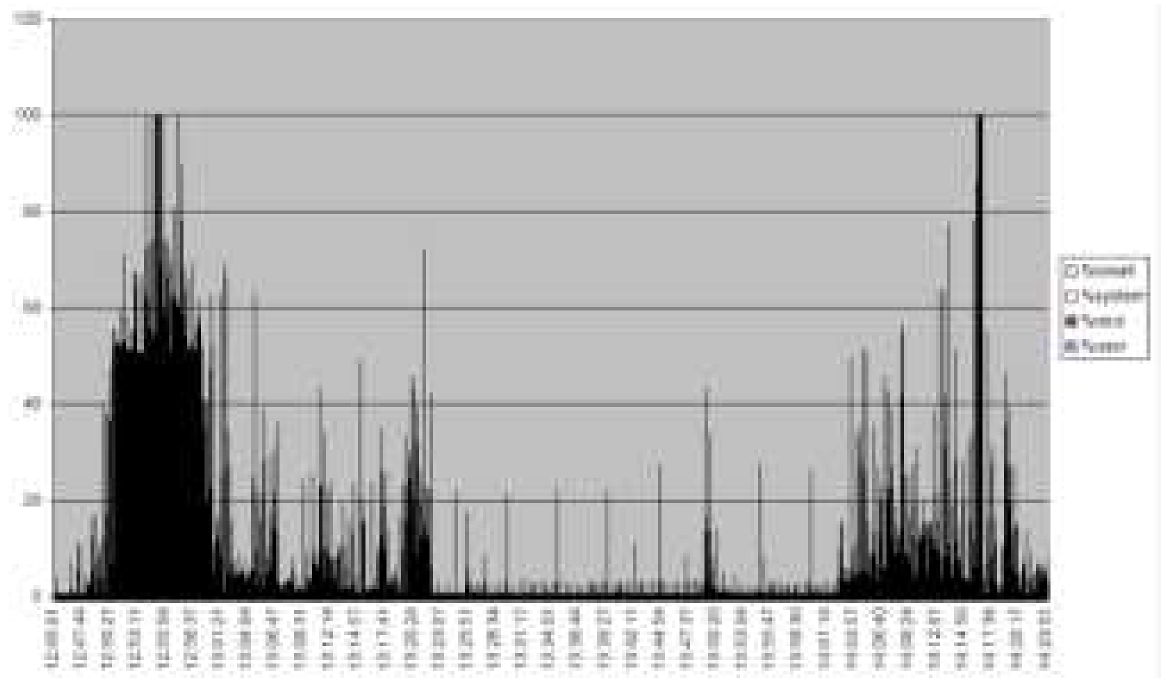


Figure G.11: Load statistics for dbaseNode2 (database server) during the entire period of the M4N Sequoia view benchmark.

# Appendix H

# Click Benchmark Load Graphs

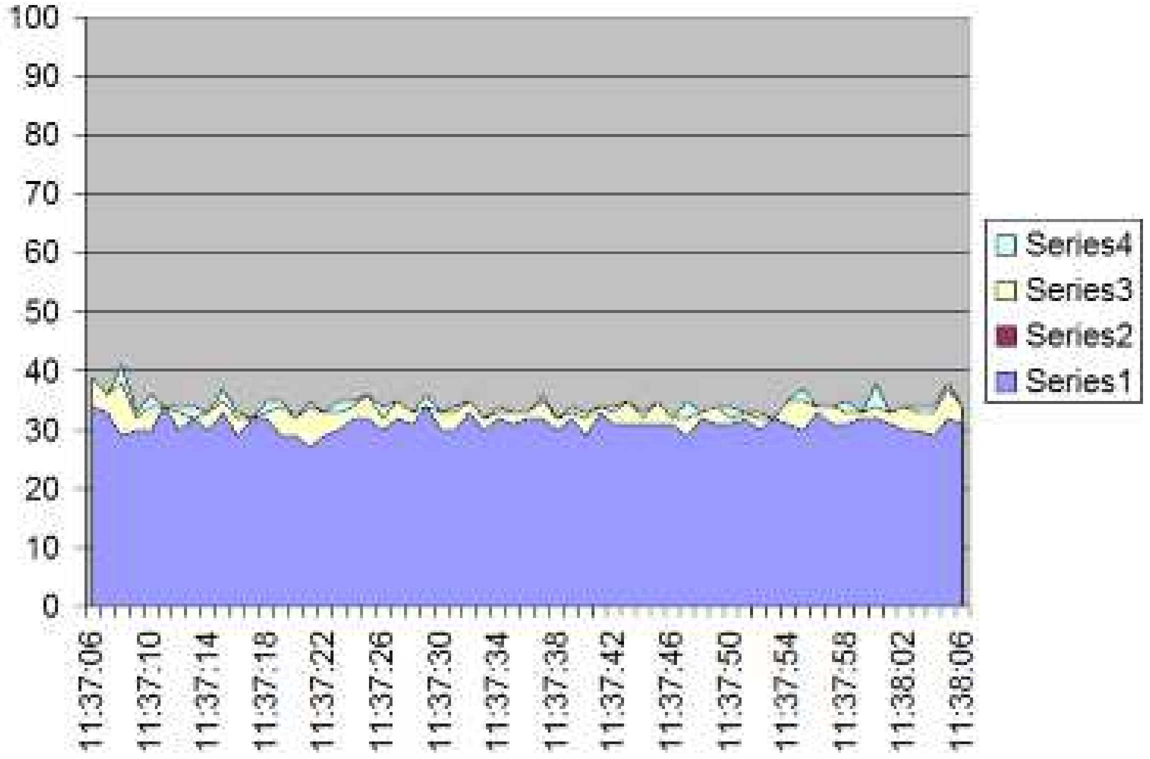


Figure H.1: Load statistics on dbaseNode1 during the M4N Default click benchmark: 1 click/second

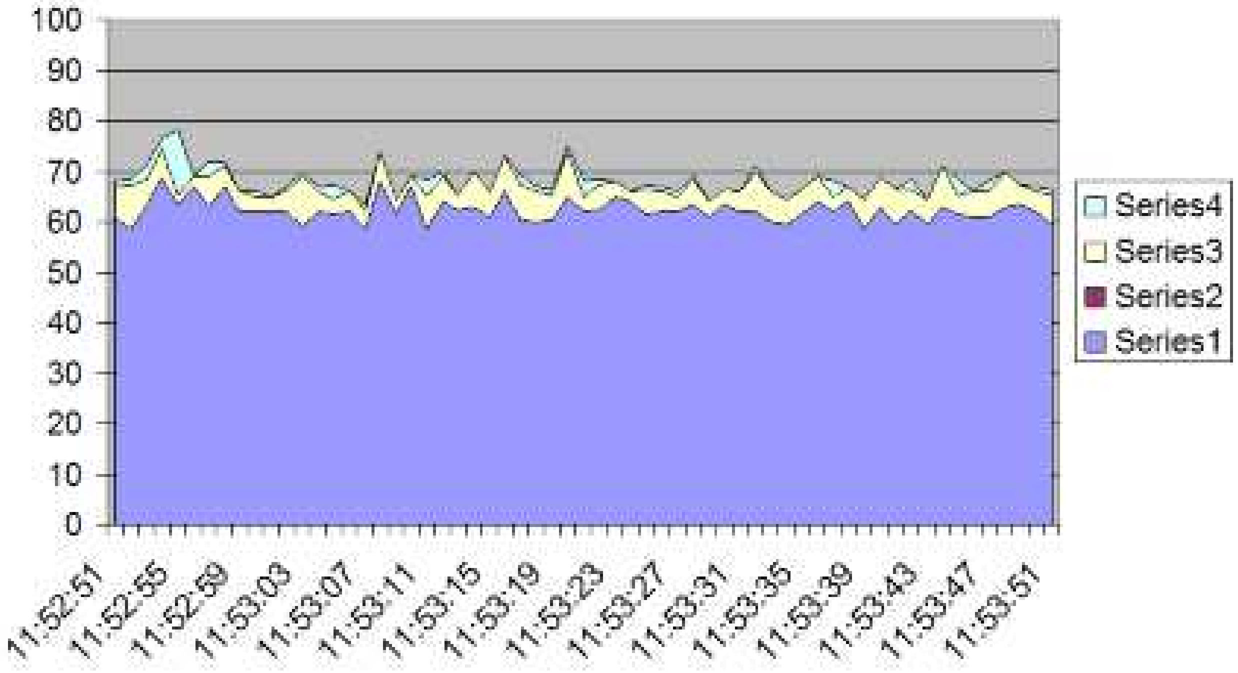


Figure H.2: Load statistics on dbaseNode1 during the M4N Default click benchmark: 2 clicks/second

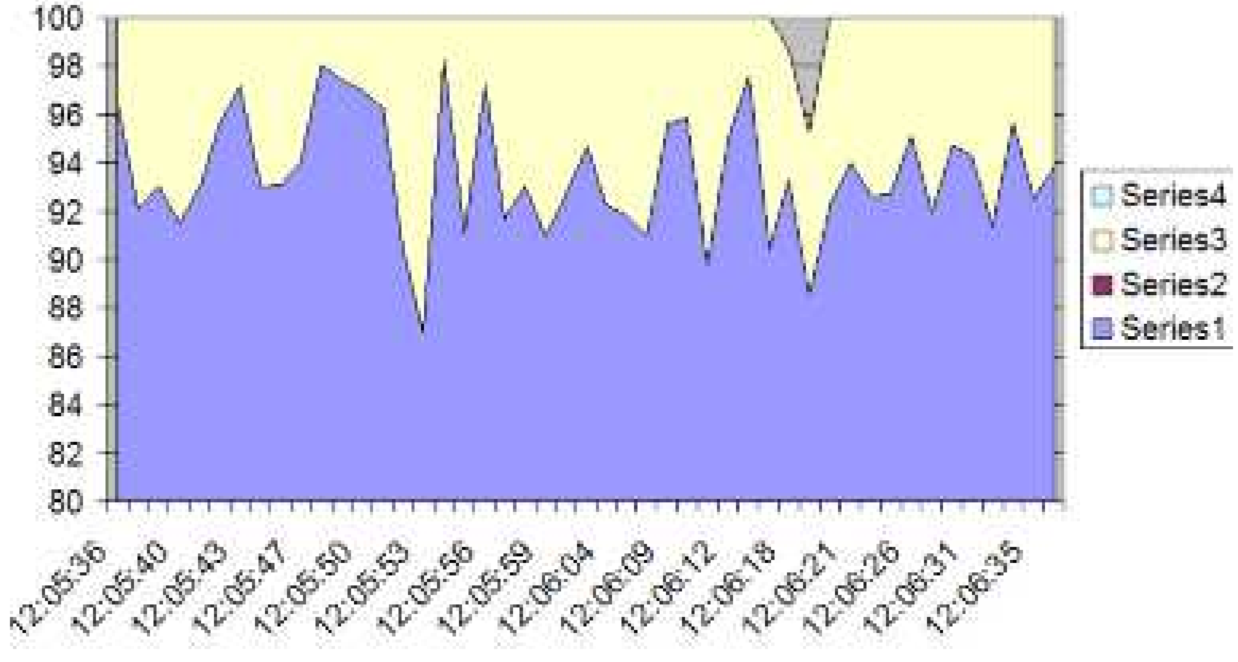


Figure H.3: Load statistics on dbaseNode1 during the M4N Default click benchmark: 4 clicks/second

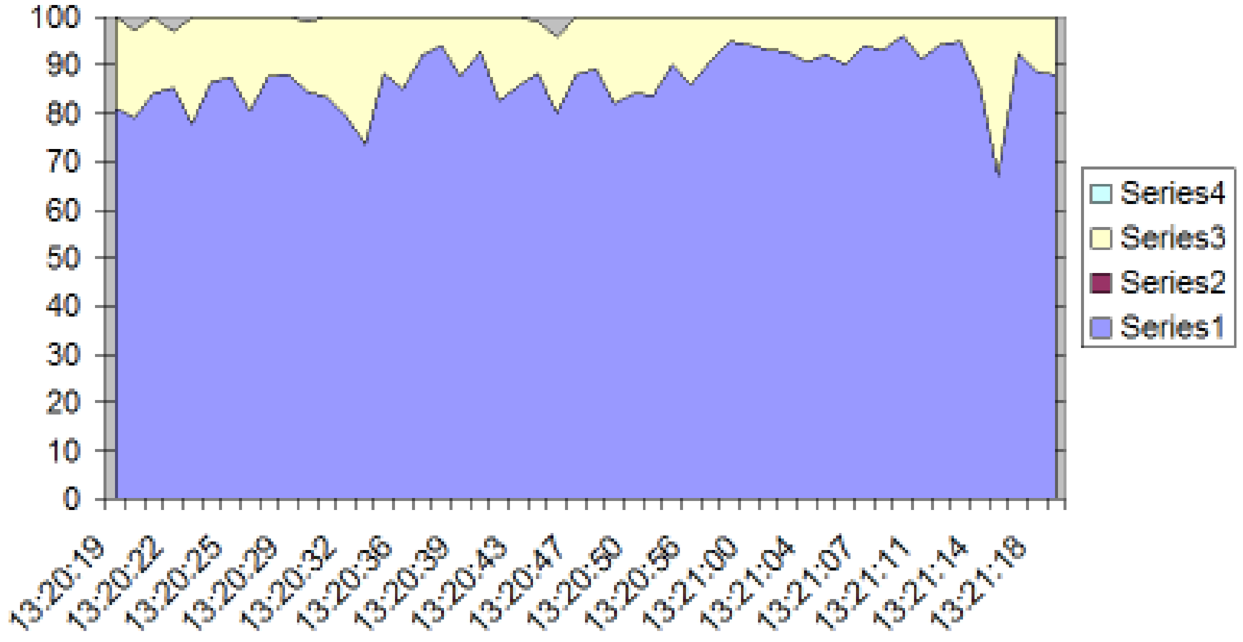


Figure H.4: Load statistics on dbaseNode1 during the M4N Default click benchmark: 40 clicks/second

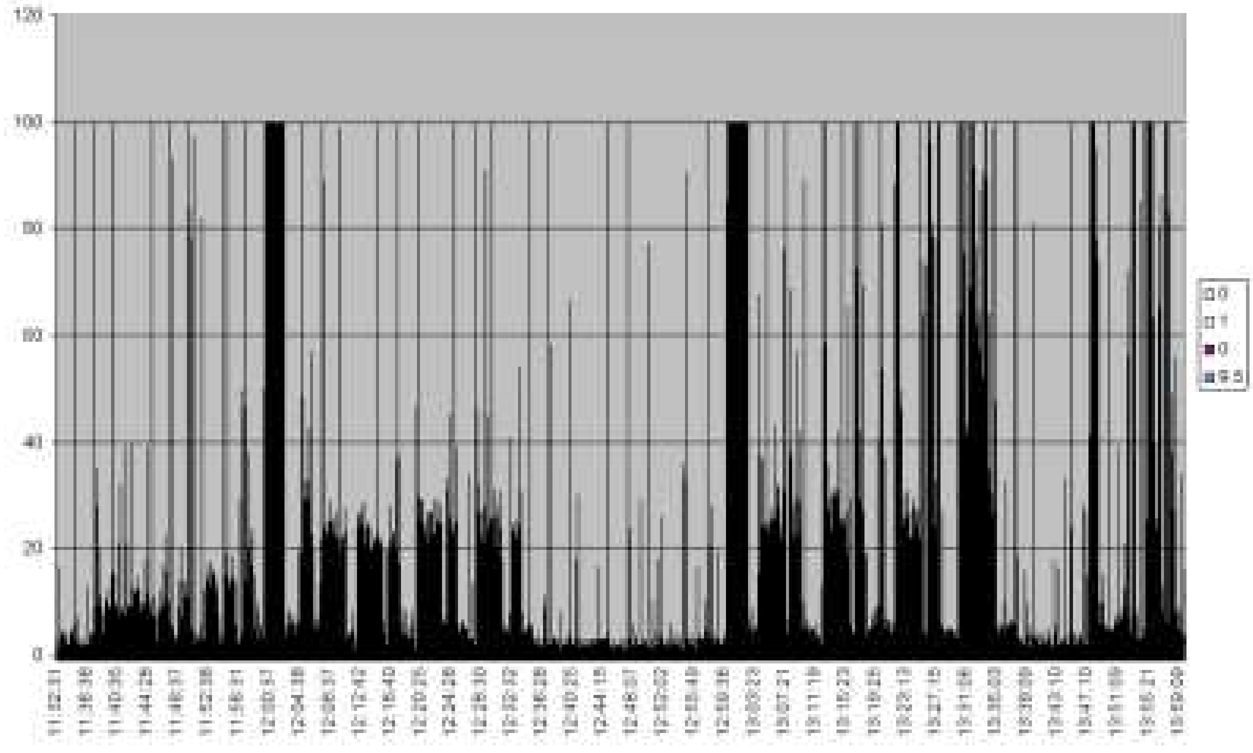


Figure H.5: Load statistics on webNode1 for the entire testing period of the M4N Default click benchmarks.

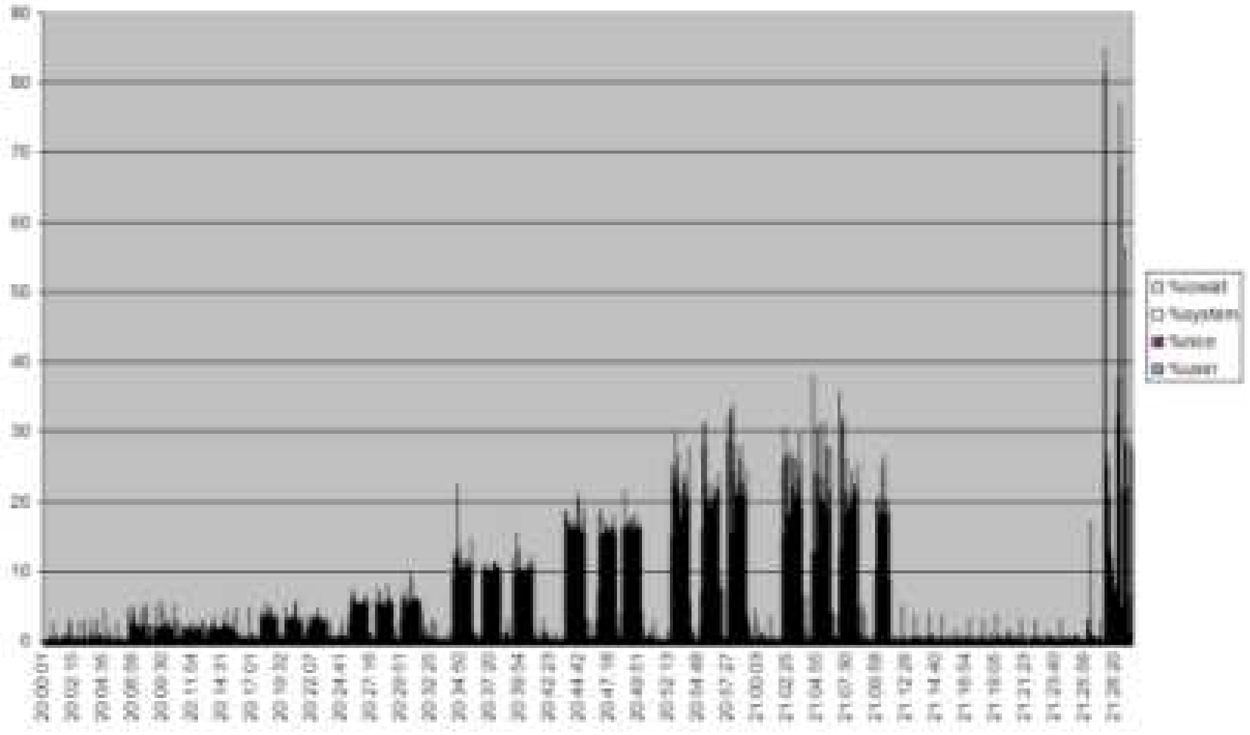


Figure H.6: Load statistics on webNode2 for the entire testing period of the M4N Sequoia click benchmarks.

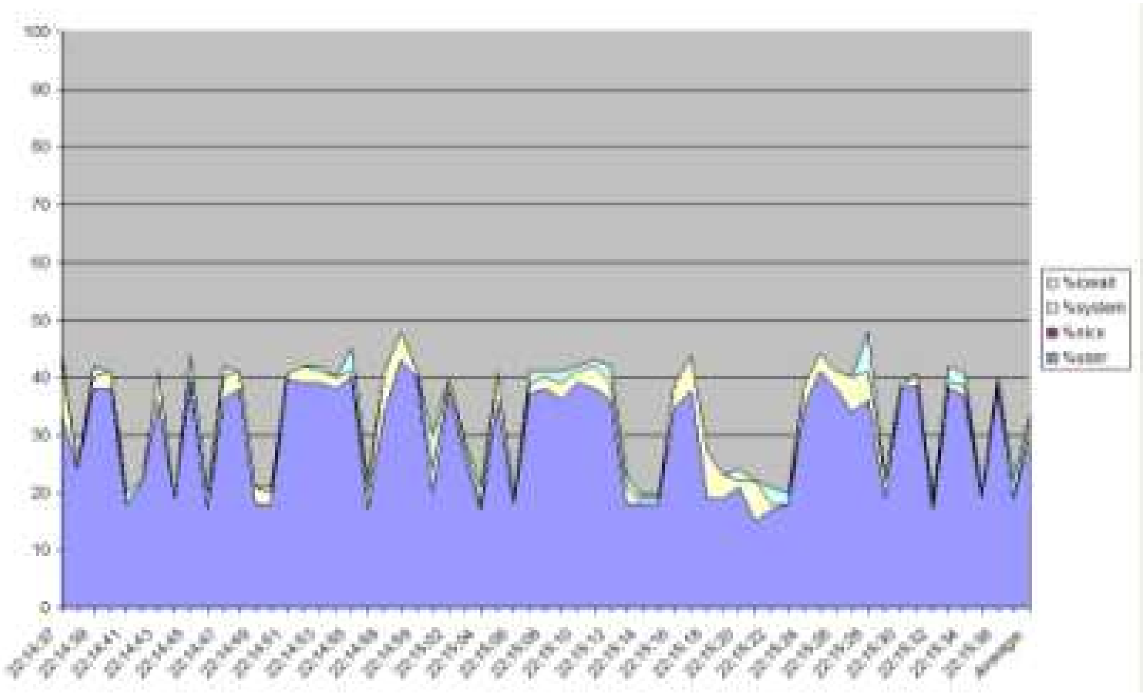


Figure H.7: Load statistics on dbaseNode1 during the M4N Sequoia click benchmark: 1 click/second.

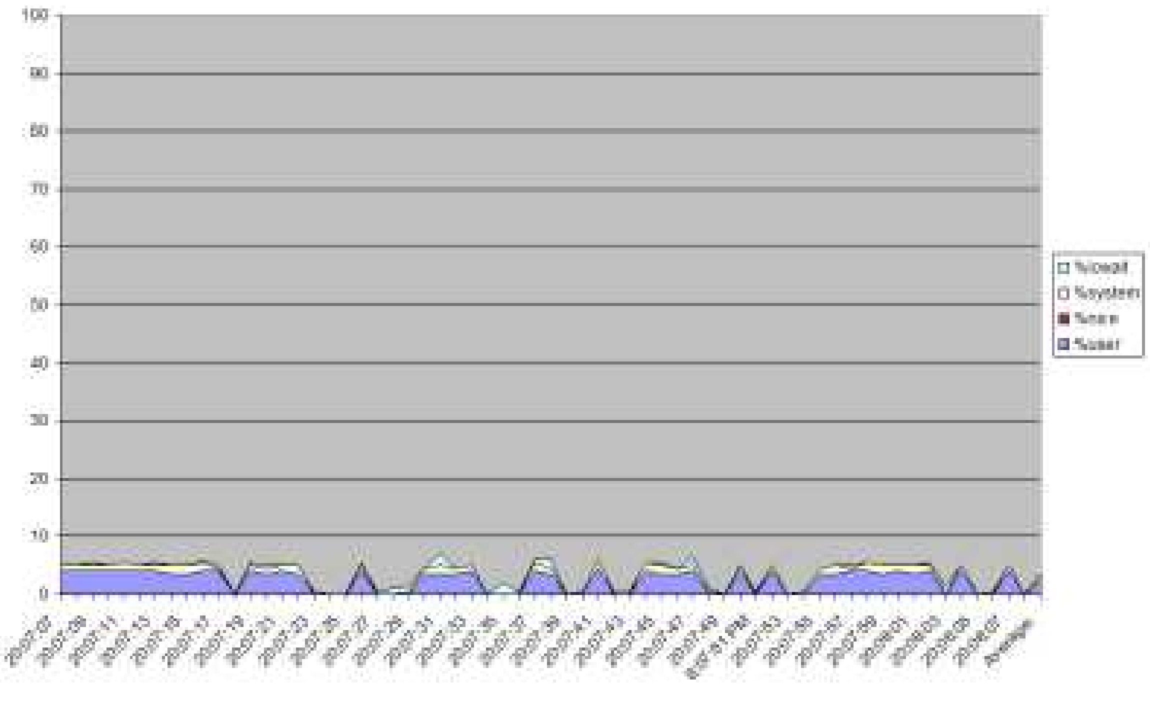


Figure H.8: Load statistics on dbaseNode2 during the M4N Sequoia click benchmark: 1 click/second.

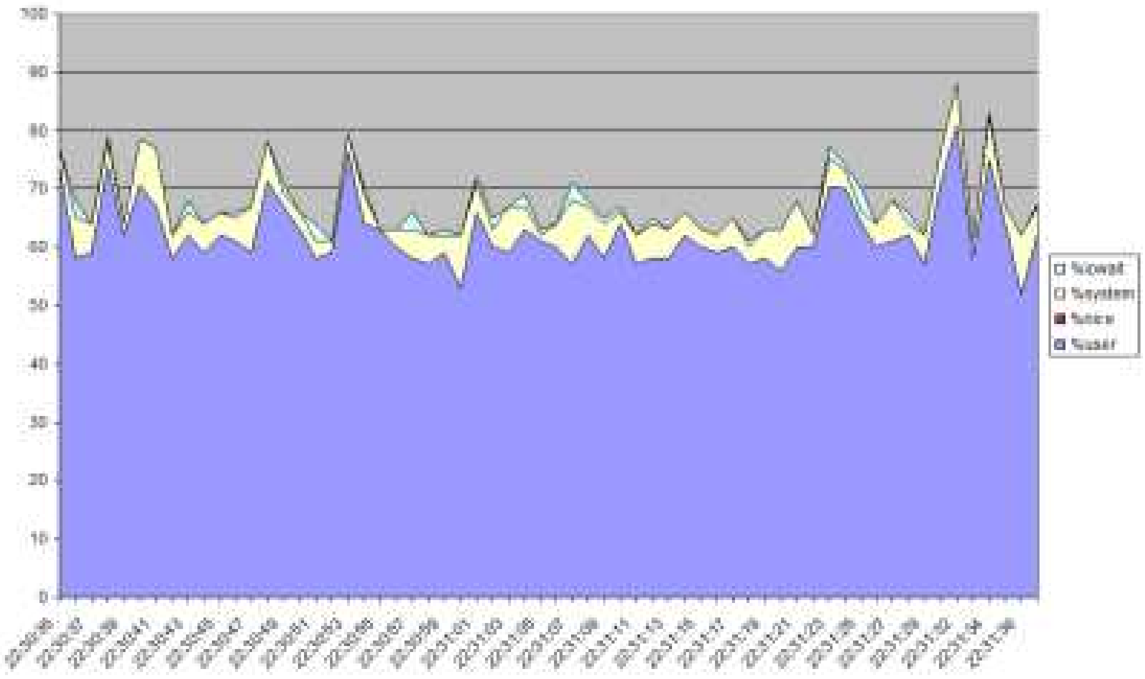


Figure H.9: Load statistics on dbaseNode1 during the M4N Sequoia click benchmark: 4 click/second.

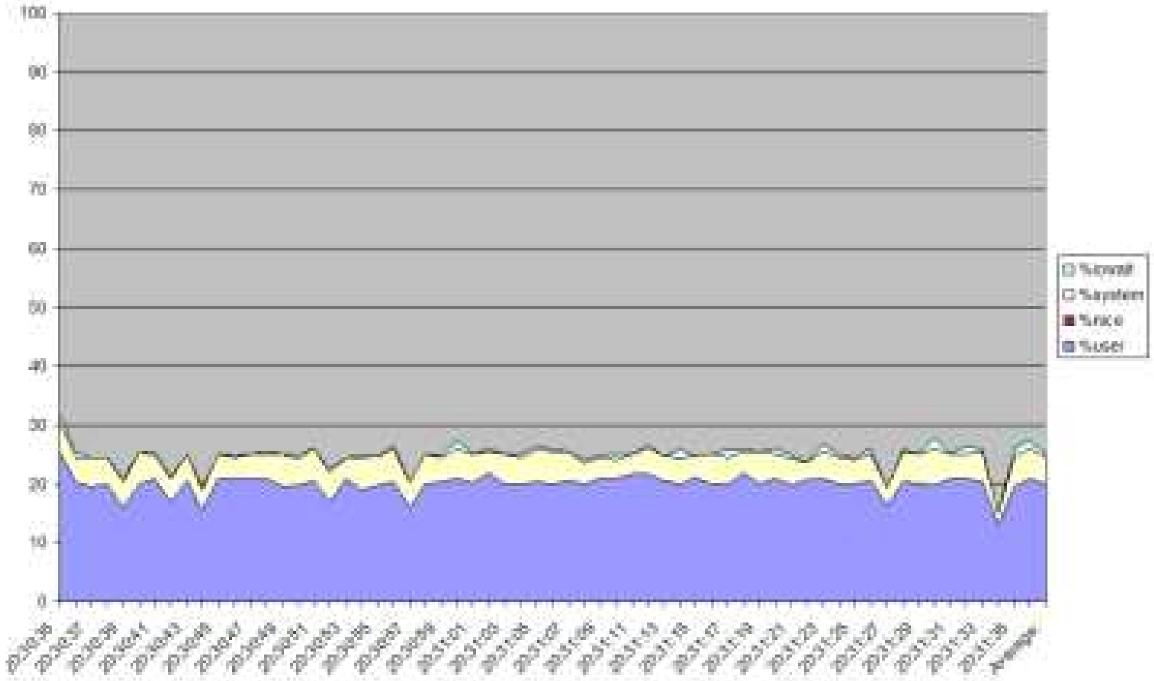


Figure H.10: Load statistics on dbaseNode2 during the M4N Sequoia click benchmark: 4 click/second.

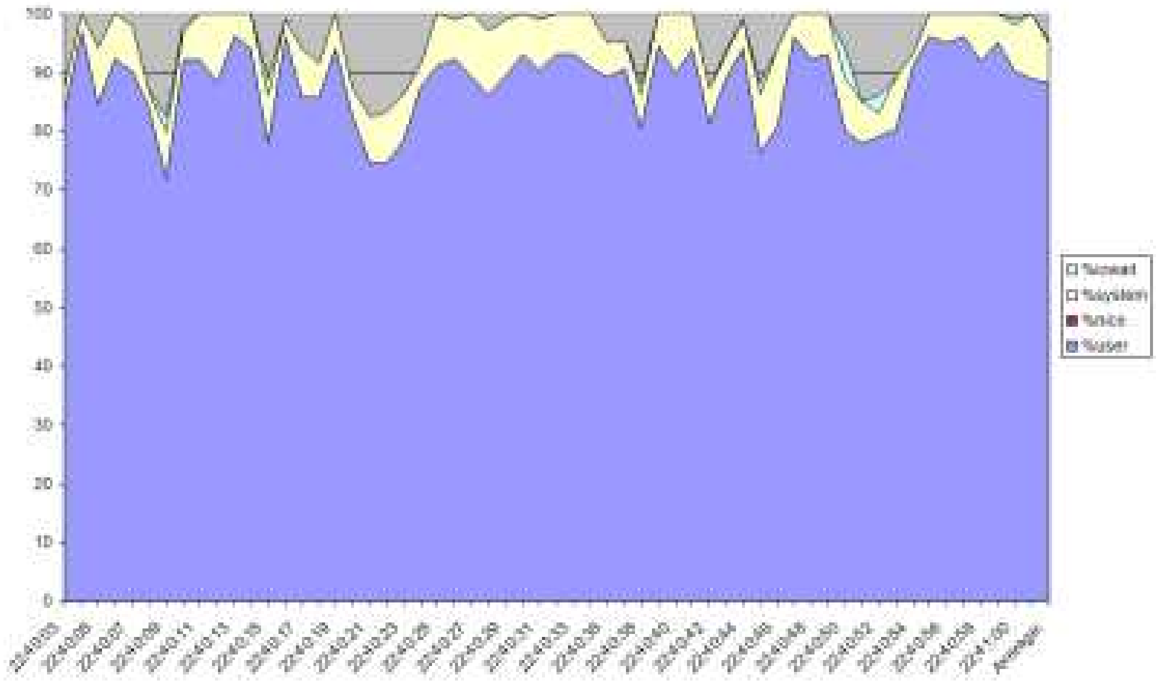


Figure H.11: Load statistics on dbaseNode1 during the M4N Sequoia click benchmark: 8 click/second.

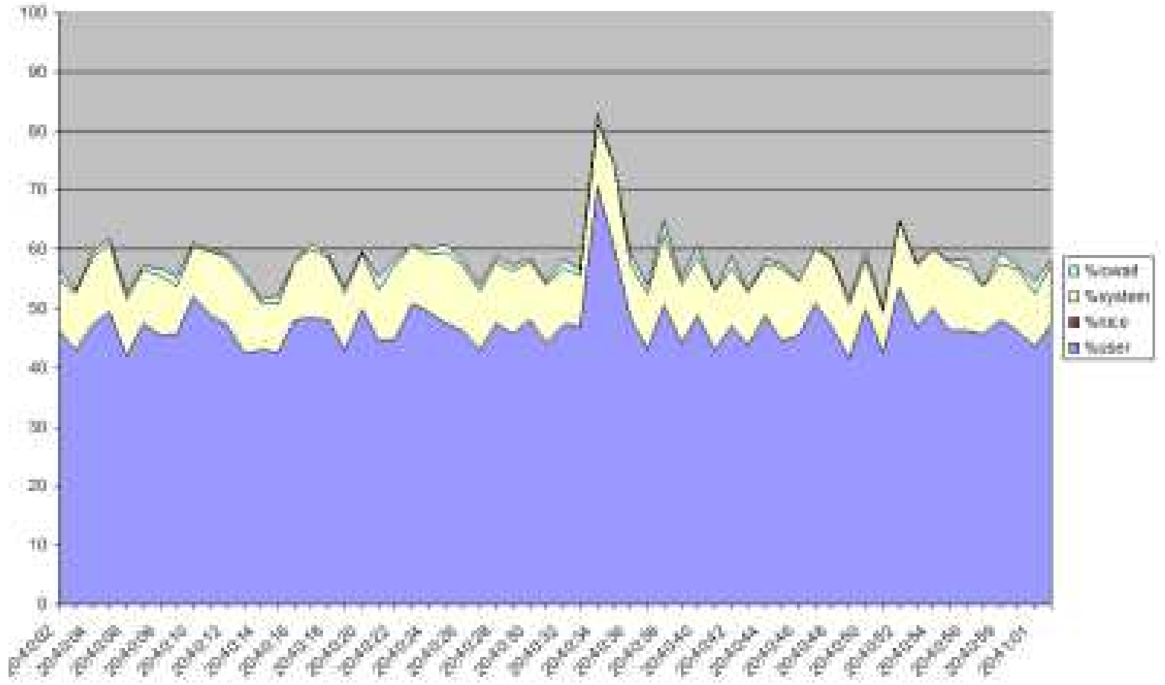


Figure H.12: Load statistics on dbaseNode2 during the M4N Sequoia click benchmark: 8 click/second.

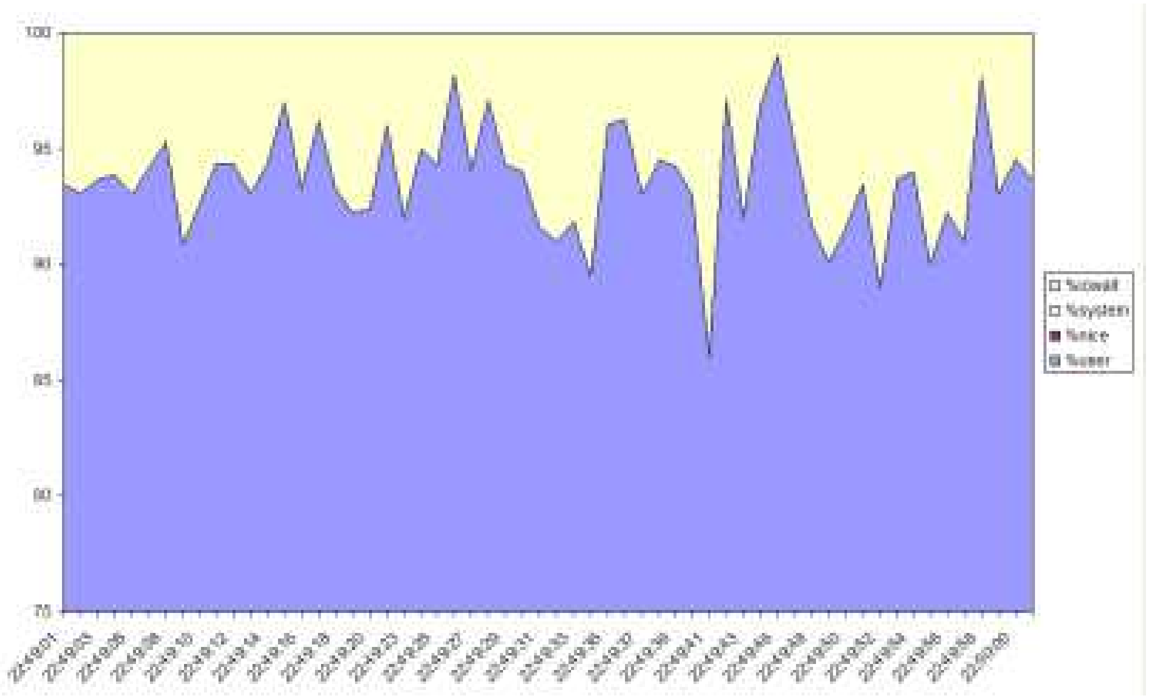


Figure H.13: Load statistics on dbaseNode1 during the M4N Sequoia click benchmark: 12 click/second.

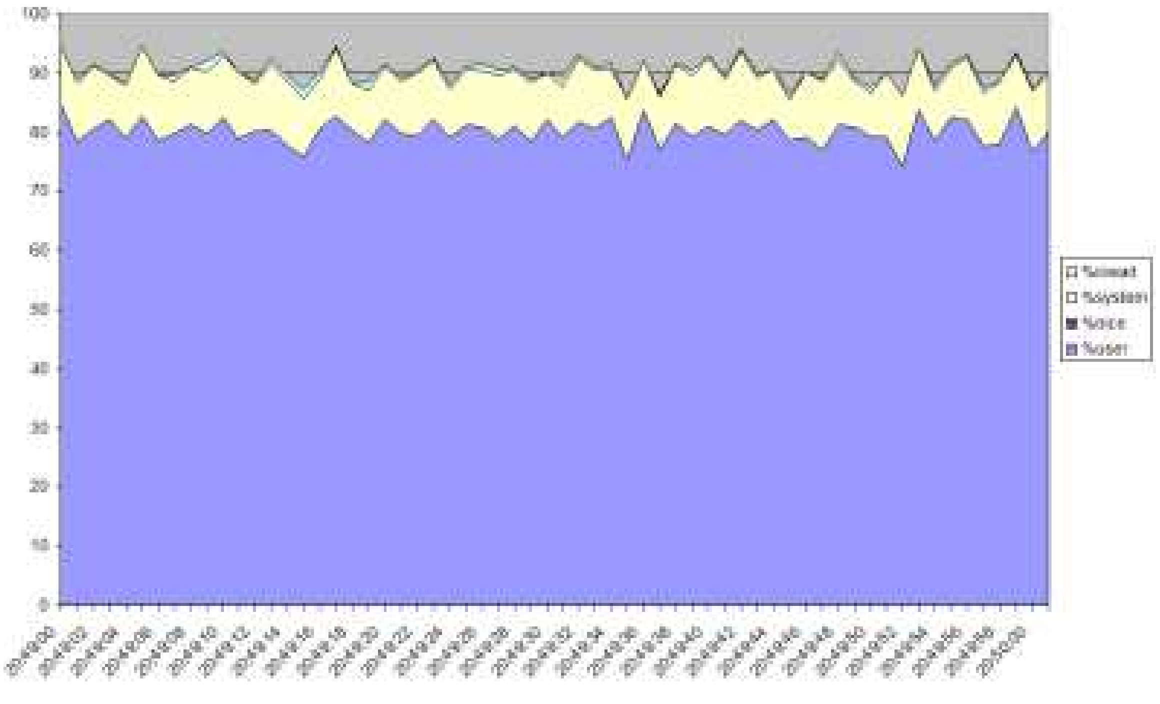


Figure H.14: Load statistics on dbaseNode2 during the M4N Sequoia click benchmark: 12 click/second.

Appendix I

# Full Simulation Benchmark Load Graphs

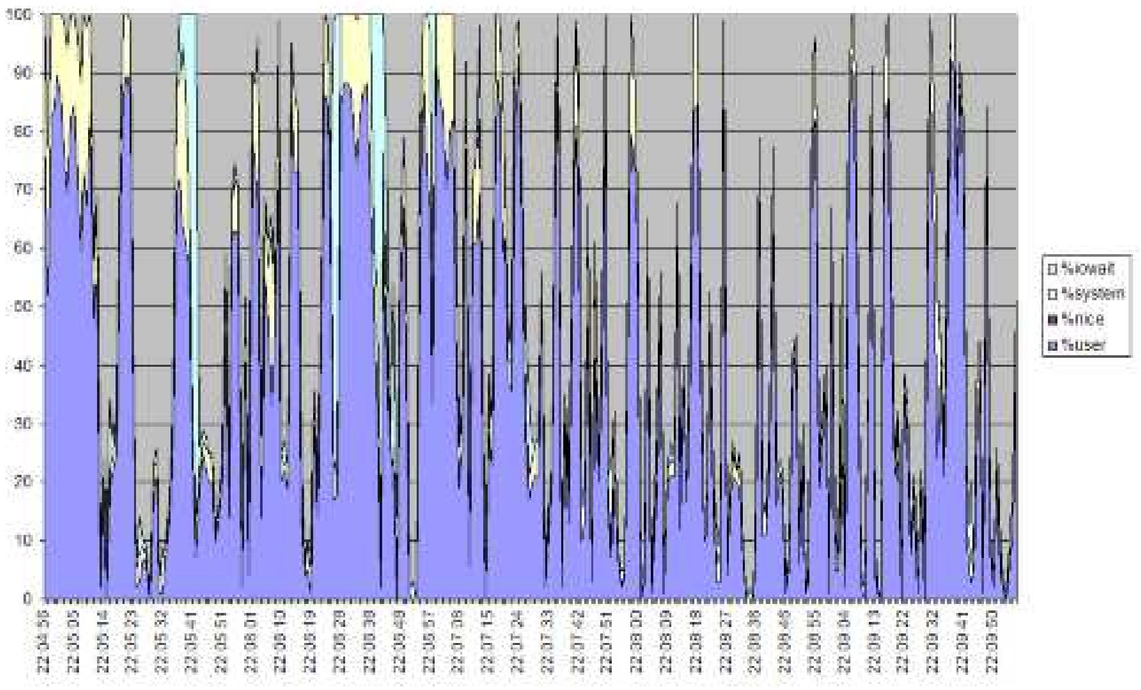


Figure I.1: Load statistics on dbaseNode1 during the M4N Default full simulation benchmark: default intensity

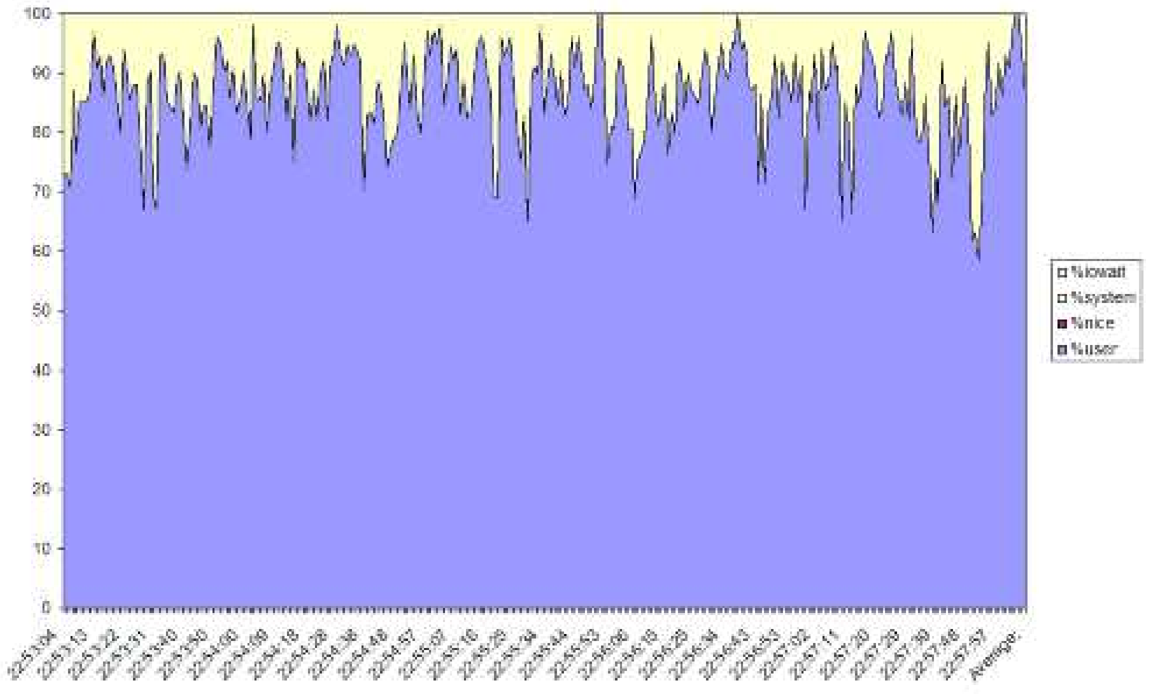


Figure I.2: Load statistics on dbaseNode1 during the M4N Default full simulation benchmark: double intensity

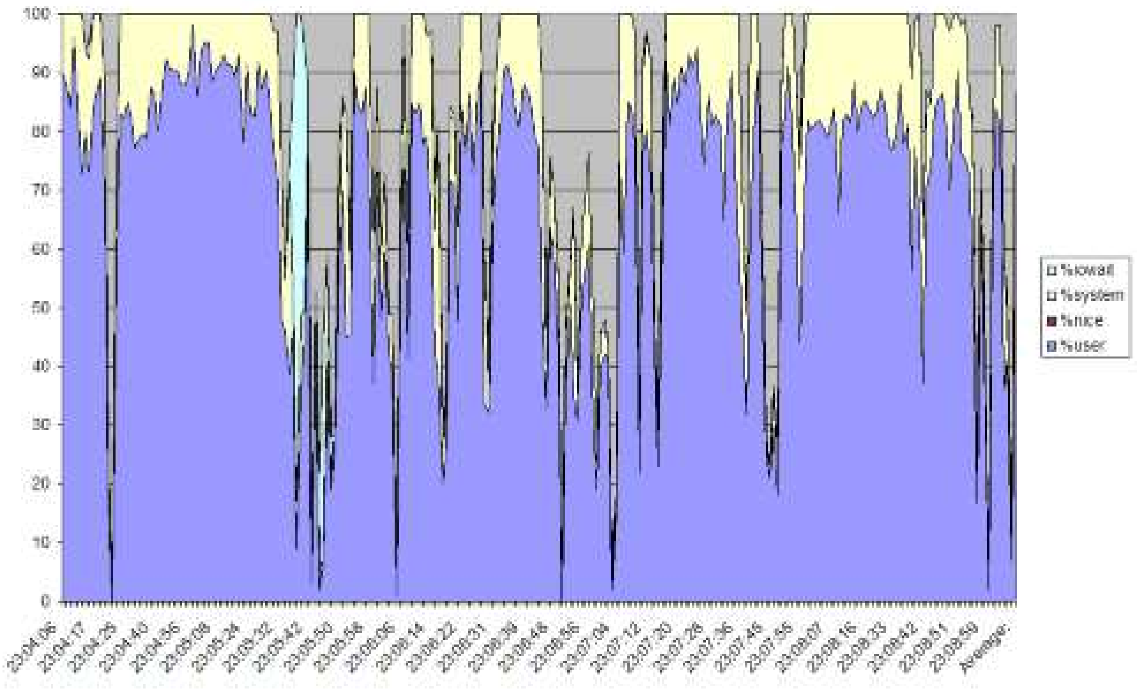


Figure I.3: Load statistics on dbaseNode1 during the M4N Default full simulation benchmark: quadruple intensity

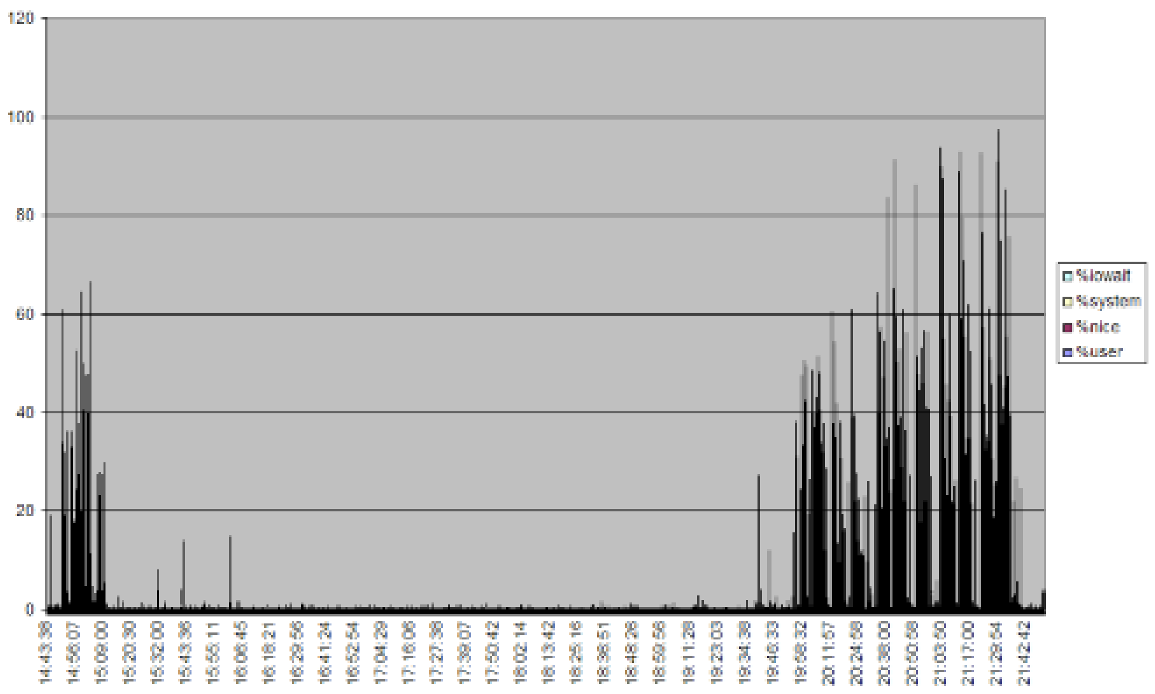


Figure I.4: Total load statistics on webNode2 during entire period of the M4N Default full simulation benchmark.

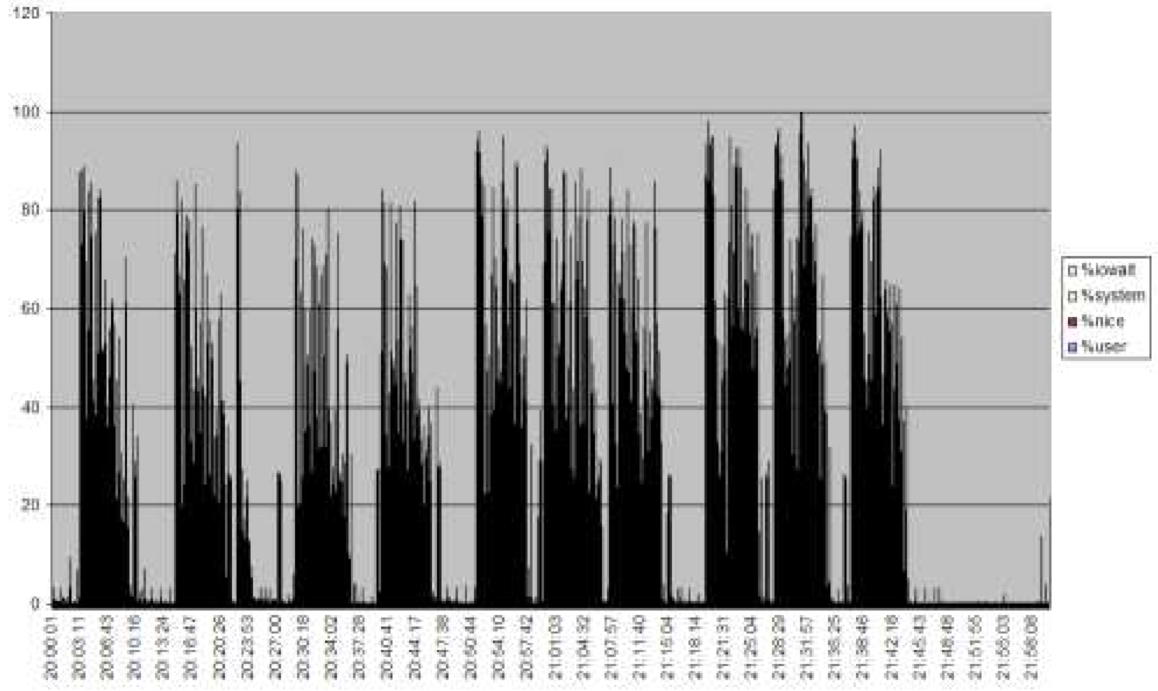


Figure I.5: Total load statistics on webNode2 during entire period of the M4N Sequoia full simulation benchmark.

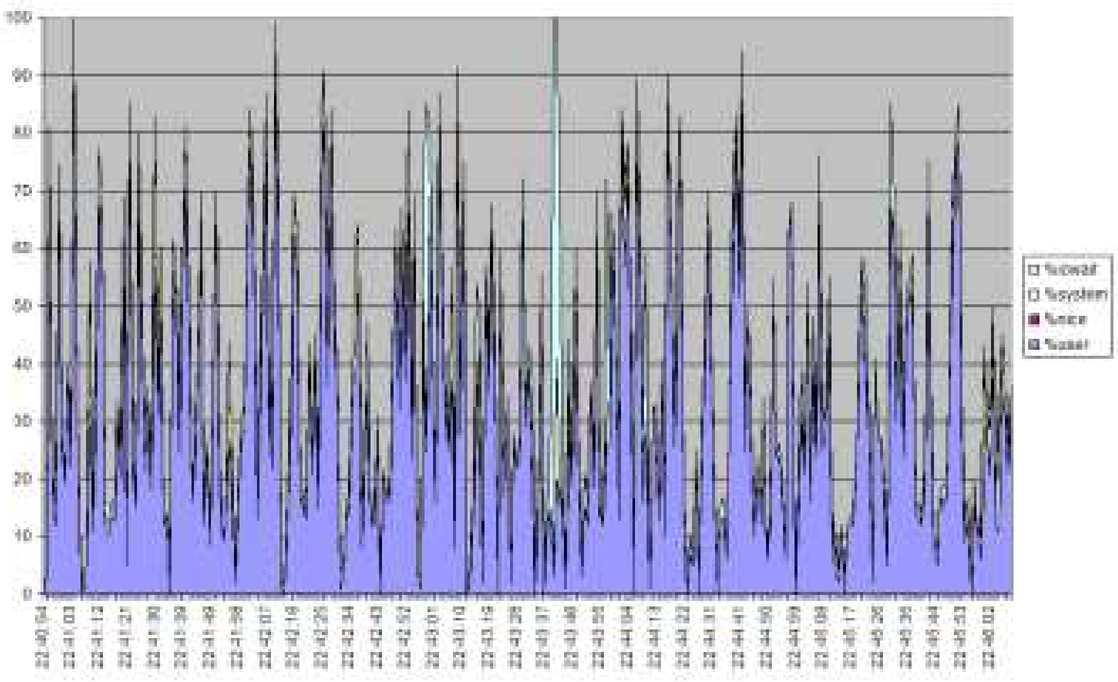


Figure I.6: Load statistics on dbaseNode1 during the M4N Sequoia full simulation benchmark: default intensity

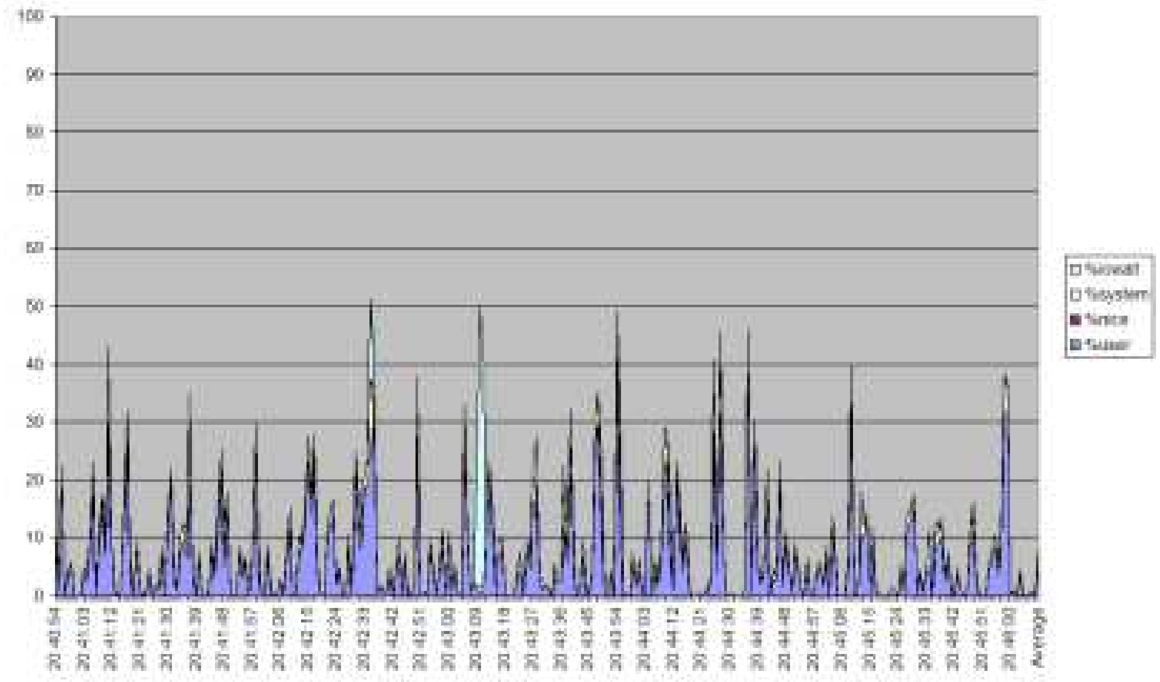


Figure I.7: Load statistics on dbaseNode2 during the M4N Sequoia full simulation benchmark: default intensity

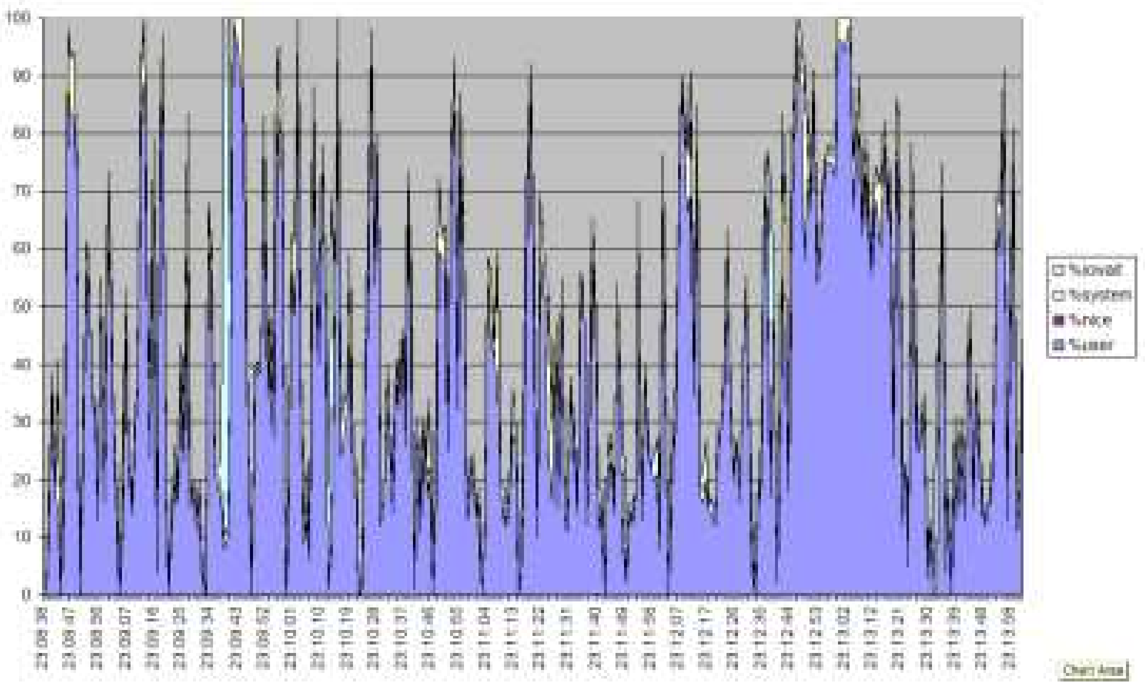


Figure I.8: Load statistics on dbaseNode1 during the M4N Sequoia full simulation benchmark: double intensity

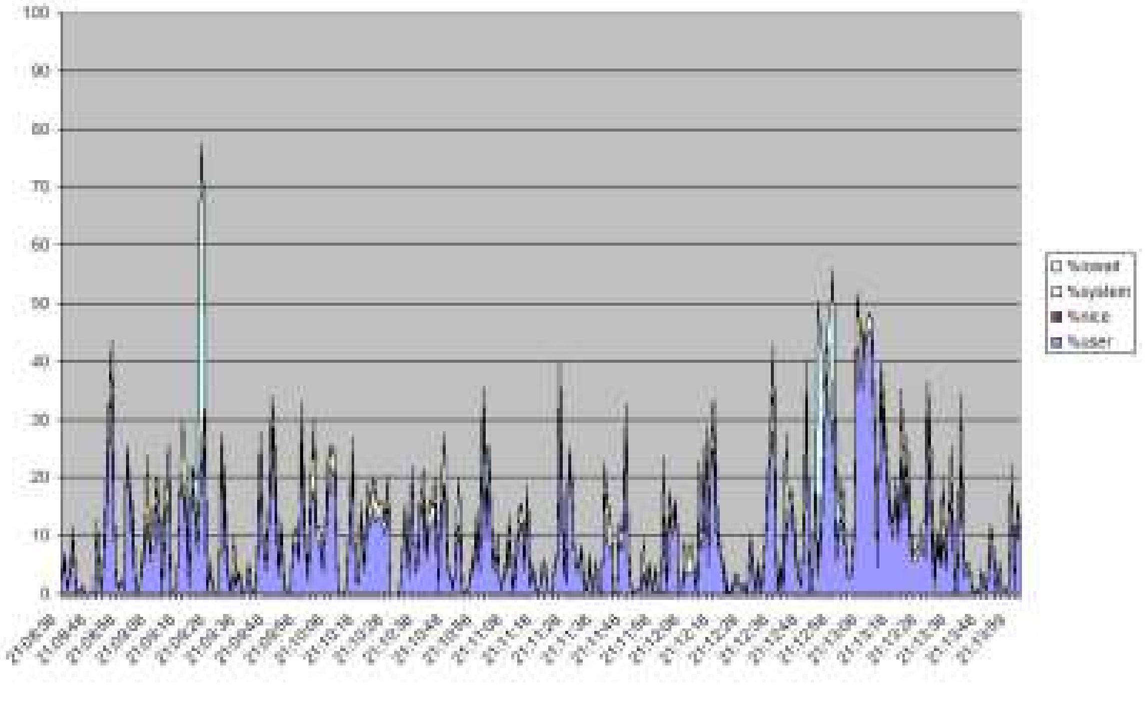


Figure I.9: Load statistics on dbaseNode2 during the M4N Sequoia full simulation benchmark: double intensity

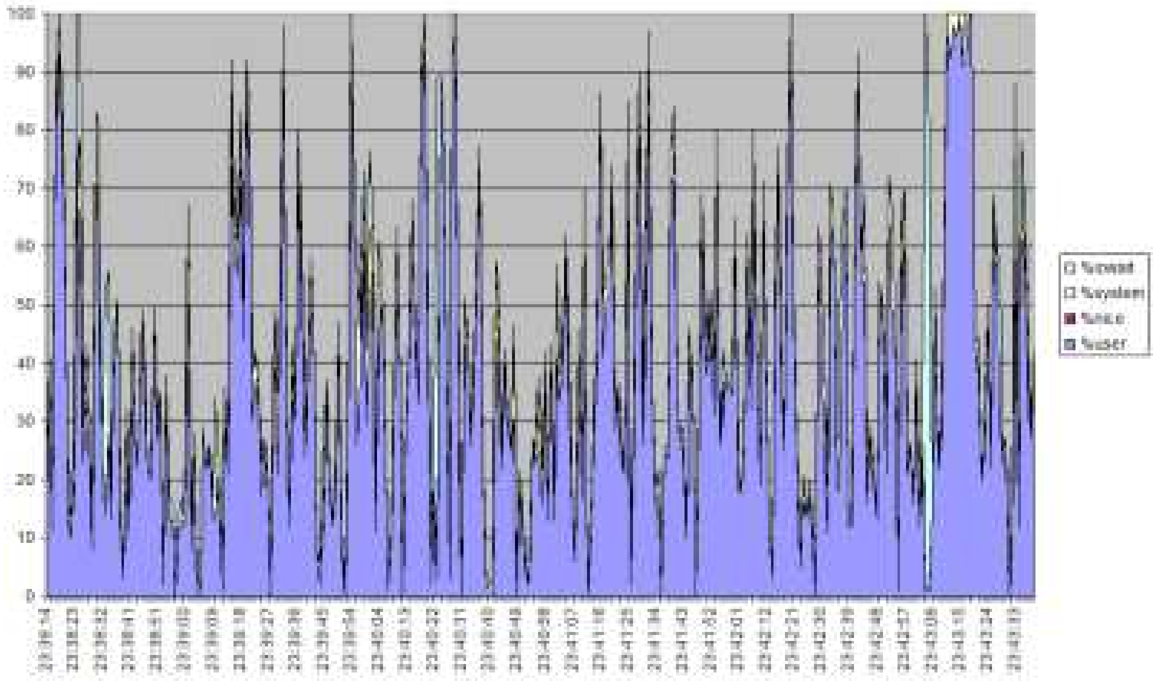


Figure I.10: Load statistics on dbaseNode1 during the M4N Sequoia full simulation benchmark: quadruple intensity

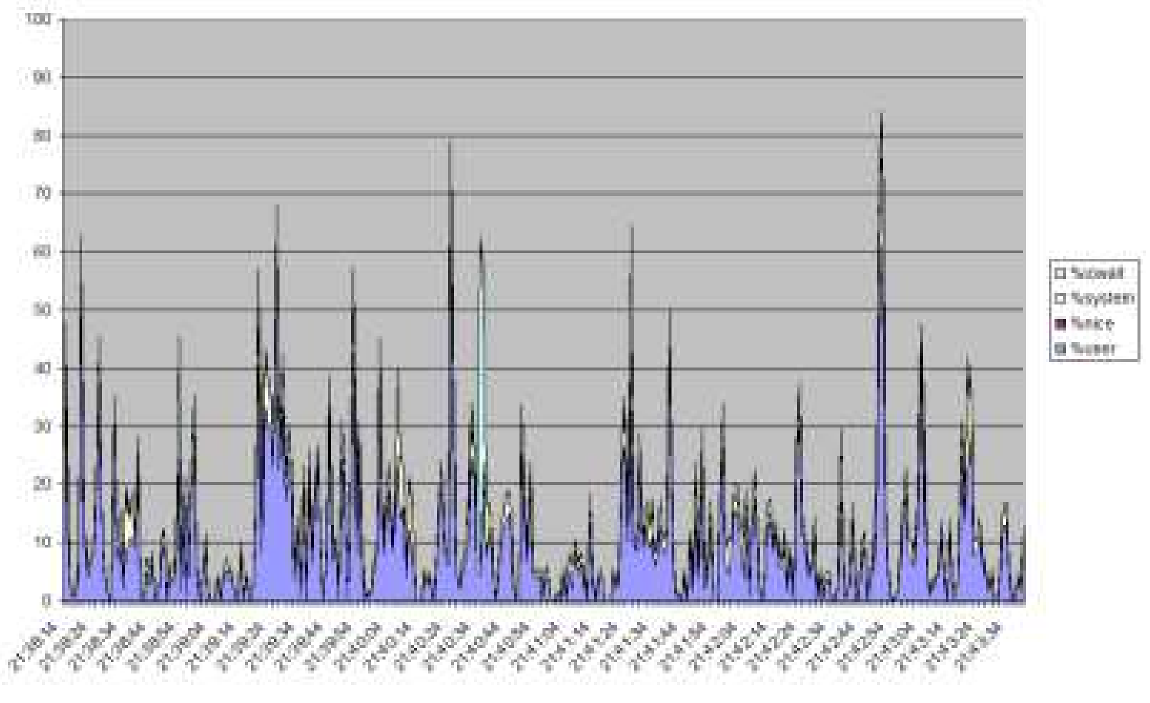


Figure I.11: Load statistics on dbaseNode2 during the M4N Sequoia full simulation benchmark: quadruple intensity