Brassersplein 2
P.O. Box 5050
2600 GB Delft
The Netherlands

T +31 15 285 70 00
F +31 15 285 70 57
info-ict@tno.nl

TNO report

Performance Analysis of Online Services

Date	4 May 2009
Author(s)	Wouter Pepping
Assignor	
Project number	
Classification report	
Title	Public
Abstract	Public
Report text	Public
Appendices	Public
Number of pages	53 (incl. appendices)
Number of appendices	1

Contents

Summary

Online services and ICT-infrastructure and its availability are becoming an important part of a business success. However, measurement data shows that performance of a large fraction of websites leaves a lot to be desired. Often loss of revenue is a direct consequence of the lack of performance. In an attempt to deal with these problems, we try to make a step in answering the following question.

What is the underlying cause of performance degradation in a web service? What ICT technologies are available to prevent this? And what is the best way to deploy these technologies to offer a solution for these problems?

In order to answer this question we divide this project in two parts. The first part consists of developing a tool to measure web service performance, called the Web Performance Monitor (WPM). This tool will be able to measure several technical aspects of the performance of websites in regular intervals and report on statistics that give an indication of the performance of these websites. Furthermore, the measurements of the WPM will serve as input for the analysis in the second part of the project.

During the second part we analyze how combinations of existing performance improving technologies can be used effectively to maximize performance improvement. We construct a model of a server farm, which allows us to evaluate combinations of several variations of Web Admission Control and Load Balancing. By means of simulation we compare the performance of Load Balancing schemes Least Connection, Round Robin and Random Assignment combined with global WAC, or WAC per server.

The results show that Least Connection load balancing greatly outperforms the other two schemes. Apparently, there is a large benefit in state dependent load balancing. However, when the offered load increases above 100%, the positive effect of LC decreases as queues fill up and the choice of which server should handle the next request becomes less interesting.

Web Admission Control was shown to have a negative effect for scenarios with an offered load below 100%. A scenario with 200% offered load showed impressive benefits. In combination with LC load balancing, we saw an increase in session success rate of more than 100%. WAC per server did not show any clear benefits and showed negative effects when LC load balancing was used. Using WAC per server does not appear to be an interesting option for real life systems.

The benefit of WAC is easily cancelled out when the wrong parameter is set. Our system’s capacity was just over 37 requests per second and the optimal WAC parameter was between 36 and 40 requests per second in every scenario. Our recommendation for WAC is having it limit the number of requests to the capacity of the system and only activating it during overload.

1 Introduction

This thesis is the result of a seven month internship at TNO Information and Communication Technology (TNO ICT), for my Master in Business Informatics & Mathematics. TNO ICT has started a knowledge project called “QoS Predict”. Part of this project is investigating performance of websites and Service Based Architectures. The goal of the project is to acquire knowledge that will help TNO provide solutions for future customers that do not have this knowledge available.

Online services are becoming more and more important in present-day society. Online banking, sales and numerous types of online reservation systems are only a few examples of services that we almost cannot do without anymore. Websites and ICT-infrastructure and its availability are becoming an important part of a business success.

Even now measurement data shows that performance of a large fraction of websites leaves a lot to be desired. The biggest problems arise at the most crucial moments. When the popularity of a website suddenly increases, servers often are not able to cope with the demand, resulting in performance drops and unavailability. For the company involved, this means bad publicity at best. Often loss of revenue is a direct consequence of the lack of performance.

The internship is a part of the “QoS Predict” project that deals with causes of performance degradation in online services, and technologies that provide means to prevent this. The internship should give TNO an insight into the available technologies for web server performance optimization and which combinations of these technologies should be used in specific situations.

1.1 Problem definition

In an attempt to deal with these problems, we try to make a step in answering the following question.

What is the underlying cause of performance degradation in a web service? What ICT technologies are available to prevent this? And what is the best way to deploy these technologies to offer a solution for these problems?

1.2 Assignment description and outline

The assignment for this internship consists of two parts. The first part is a very practical approach to enable us to evaluate the current performance of websites. The second part is a more theoretical approach, where we research causes of and solutions for performance degradation.

For the first part, a tool called the “Web Performance Monitor”, WPM for short, will be developed. This tool will be able to measure several technical aspects of the performance of websites in regular intervals. It will use these values to report a number of statistics which give an indication of the performance of these websites. Not only will this tool allow us to see current performance of online services, but the statistics reported by the WPM will also serve as input for the second part of the assignment.

The second part of the assignment starts with a study of literature, revealing common causes of performance degradation in online services and existing technologies to improve this performance.

Next, a model of a server farm will be constructed, which includes these existing performance improving technologies. This model will allow us to evaluate combinations of these technologies in a broad range of scenarios. Because the mathematical analysis of the model becomes complicated and is only applicable to a limited number of cases, we will use simulation for the analysis. Eventually, this should give us a better understanding how technologies can be combined to optimize performance.

Because the assignment consists of two parts, this document will be divided in two separate parts as well. The first part describes the functionality of the WPM. Also the measurements will be analyzed to determine realistic values for the parameters in our model.

The second part starts with a study of existing literature in chapter three. In chapter four, we define the model. We discuss our simulation and attempt to validate it by comparing results of simple scenarios with known theoretical results or results obtained through mathematics in chapter five. Chapter six gives the results of the simulations. Finally in chapter seven we present our conclusions based on these results and we end with some suggestions on future work in chapter eight.

Part I: The Web Performance Monitor

2 A description of the Web Performance Monitor

[Confidential]

Part II: Performance improvements

3 Literature

One of the largest challenges in controlling performance of internet services is the burstiness of the experienced load. As stated by Welsh in [11], peak demand for a website may be up to one hundred times as much as the average demand.

A good example for this kind of scenario is the website crisis.nl. This website provides important information in case of a disasters etc. However, the first time the website could have been of use, during a heavy storm on January 18th 2007, the demand was so high that the servers could not cope with the load and the website was unreachable.

Further research into causes of performance degradation, including a visit to Denit Internet Services B.V., a major Dutch hosting company, reveals two other common causes. One of them is Denial of Service attacks; a deliberate action from someone with malicious intent causes websites to be overloaded. These attacks can be very sophisticated, originating from dozens of locations. Often these origins are computers owned by innocent, ignorant users, infected by a computer virus, making them hard to trace.

A third cause is bad programming in the website's design. This could make a low demand on a website result in an extremely high server load, causing performance to drop.

A search for existing literature reveals four main technologies for improving performance of websites. Each of these technologies can be divided in a large number of algorithms. In the next four paragraphs these technologies will be discussed, and a number of algorithms will be explained. Because of the large amount of existing algorithms, we will only explain a selection of more popular ones.

3.1 Caching

The goal of caching is to make popular documents available so that they can be accessed by the user faster than on the original location. This can be faster because data is stored in a faster type of memory, in a more accessible position in the network or in a more favorable geographical location. Take as an example the case where a document is cached in a faster type of memory. This way, the next time the document is requested, it can be accessed faster, decreasing its processing time. Another example is caching a document that is stored on a server in the US, on a server in the EU. This way, the next request from a client in the EU can be forwarded to the cache server, which is likely to decrease its download time.

As requests arrive at a web server, there are two possibilities. The item can already be in cache, in which case the item is retrieved from the cache and the caching algorithm does nothing. Alternatively, the item could be not in the cache yet. In this case the caching algorithm has two decisions to make. One is whether to store the item in the cache or not. The other is, if the decision is made to store the item in the cache and there is no more space available, what item(s) to remove to free space for the new item.

There are lots of caching algorithms available. Some of the more popular ones are described in [1]. We will describe two popular algorithms below.

- LFU (Least Frequently Used) [2] keeps track of the popularity of items and stores them in the server-side cache. Once the cache memory is full it discards the least frequently used ones.

- LRU (Least Recently Used) [3] evicts objects from the cache that have not been used for the longest period of time. Evidently every requested object that is not in the cache yet will be added.

3.2 Load balancing

The goal of load balancing is to distribute a website's load equally over multiple servers, which contain copies of the website, to optimize the website's performance. Load balancing can only be applied if a website is served by more than one server. Take as an example a website served by two web servers. A request arrives at a load balancer. The load balancer decides to which server the request should be sent, distributing the load over the two servers.

There are two ways for a load balancing algorithm to make this decision. It can be made either by using information about the state of the servers, or in some random or deterministic way, without the need for this information. Evidently, algorithms that have more information should be able to outperform the others.

Seven different algorithms for load balancing are described in [4]. Of these seven, three popular ones are described below. The first three algorithms are dependant on information about the state of the servers, the last two are not.

- LC (Least Connection) assigns a request to the server that has the lowest number of connections at that time.

- WLC (Weighted Least Connection) is similar to LC. A request is assigned to the server for which the number of connections divided by the weight for that server is the lowest. So if N is the number of servers, c_i is the number of connections to server i and w_i is the weight for server i then the an incoming request is assigned to server n where

$$n = \arg\min_i \left\{ \frac{c_i}{w_i} \right\}$$

- LBA (Load Based Assignment) assigns a request to the server that has the lowest load, for example CPU load.

- RR (Round Robin) assigns a request to server n where

$$n = (m \bmod N) + 1$$

  With N the number of servers, and m the server the last request was sent to.

- RA (Random Assignment) assigns requests randomly to any server.

3.3 Scheduling

Scheduling is the process of determining the order in which requests are processed, in order to achieve better performance or give priority to certain types of requests.

When a request arrives the scheduling algorithm decides, based on information contained in the request, with what priority the request should be handled. This decision can be based on for example the requested URL, the size of the requested file, the originating IP-address etc.

Scheduling can be performed in a strict order scenario, where the requests with the highest priority are processed first. Scheduling can also be performed in a weighted processor sharing scenario, where requests with higher priority get a larger share of processing time than requests with lower priority.

Below we will describe two “strict order” and two “processor sharing” scheduling algorithms.

- FIFO (First In First Out) processes requests in the order in which they arrive. As the name indicates, if a request arrives first, it is the first to leave after processing as well.

- SRPT (Shortest Remaining Processing Time) [5] processes requests with shorter remaining processing times first. This means shorter requests are given priority over longer requests.

- PS (Processor Sharing) gives each request an equal part of the processing-time.

- DWFS (Dynamic Weighted Fair Sharing) [6] gives every request (or session) a fraction of the processing time proportional to the weight assigned to it. The weights are updated at a set interval, maximizing a productivity function.

3.4 Admission control

Admission control tries to regulate the arrival process in order to prevent performance degradation during server overload as much as possible.

Admission control can be divided into three steps, explained in [7]. These steps are *congestion detection, admission decision* and *admission enforcement*. In the congestion detection step there is a check whether or not one or more measurements exceed a specified threshold.

The admission decision step decides, based on the result of the congestion detection, what to do with requests. If congestion is detected, the admission decision step has to reject certain requests. There are several options for doing this, some of which, described in [7], are:

- Reject all new requests
- Reject only requests of a certain type
- Reject only newly arriving user sessions
- Reject only newly arriving user sessions of a certain type

The admission enforcement step decides what to do with rejected requests. Some of the possible actions are listed below.

- Completely ignoring the arriving packets
- Rejecting the connection
- Sending a “server too busy” response
- Returning a low resource page that explains the server is overloaded

Another option is to send a less resource intensive version of the requested data. This last option allows an admission control algorithm to perform content adaptation. A website using a content adaptation algorithm has two or more versions, which differ in resource requirement. The algorithm monitors one or more metrics. Each of these metrics has one or more set thresholds. Based on if a threshold is crossed and how many thresholds are crossed, the algorithm selects a quality level for incoming requests.

Abdelzaher and Bhatti propose an algorithm in [8]. As a metric for the system utilization they take $U = aR + bW$, where R is the observed request rate, W is the aggregate delivered bandwidth and a and b are server specific constants.

A target utilization U^* is set and a measure G is determined based on the error in utilization $E=U^*-U$ as follows: $G=G+E$ for each sampling time. Based on G a choice for a certain content quality level is made, where the resource consumption depends on the quality level. If E goes to 0, G stabilizes and with that the content quality.

3.5 Other types of performance increasing technology

Although we have discussed the most important performance increasing technologies, there are some that we have left out. In the case of bandwidth shortage and a surplus of CPU capacity, one could choose to apply HTTP compression. This will compress data on the server before sending them to the client. The compressing takes up some CPU capacity, but the files will be smaller and therefore bandwidth will be spared.

Another influence on a website’s performance is the web server architecture. As Menascé describes in [12], a web server can process a number of requests at the same time, either process-based, thread-based, or a hybrid of the two. A process-based architecture brings more stability, while a thread-based architecture is much more efficient. The number of processed or threads of-course highly depends on the available memory.

Another paper by Menascé, [13], describes automatic QoS parameter control. In this paper he gives an algorithm to update parameters for several performance influencing schemes at once, based on several performance metrics, on a regular interval. This way, changes in workload that effect the optimal choice of parameters could be taken into account and performance was increased.

4 The model

To limit the scope of the problem, we only look at the combination of two of the listed control mechanisms: admission control and load balancing. Scheduling is not used in most web servers and therefore of less practical interest at the moment of this writing. Caching is partially implemented by separating processing times for static and dynamic requests. Furthermore, it has already been taken into account in the measurements of the processing time, assuming the measured websites have implemented caching.

In figure 4 the model is shown graphically. The details of the model are described in four sections, first the arrival process, then the servers, the admission control algorithm and finally the load balancing algorithm.

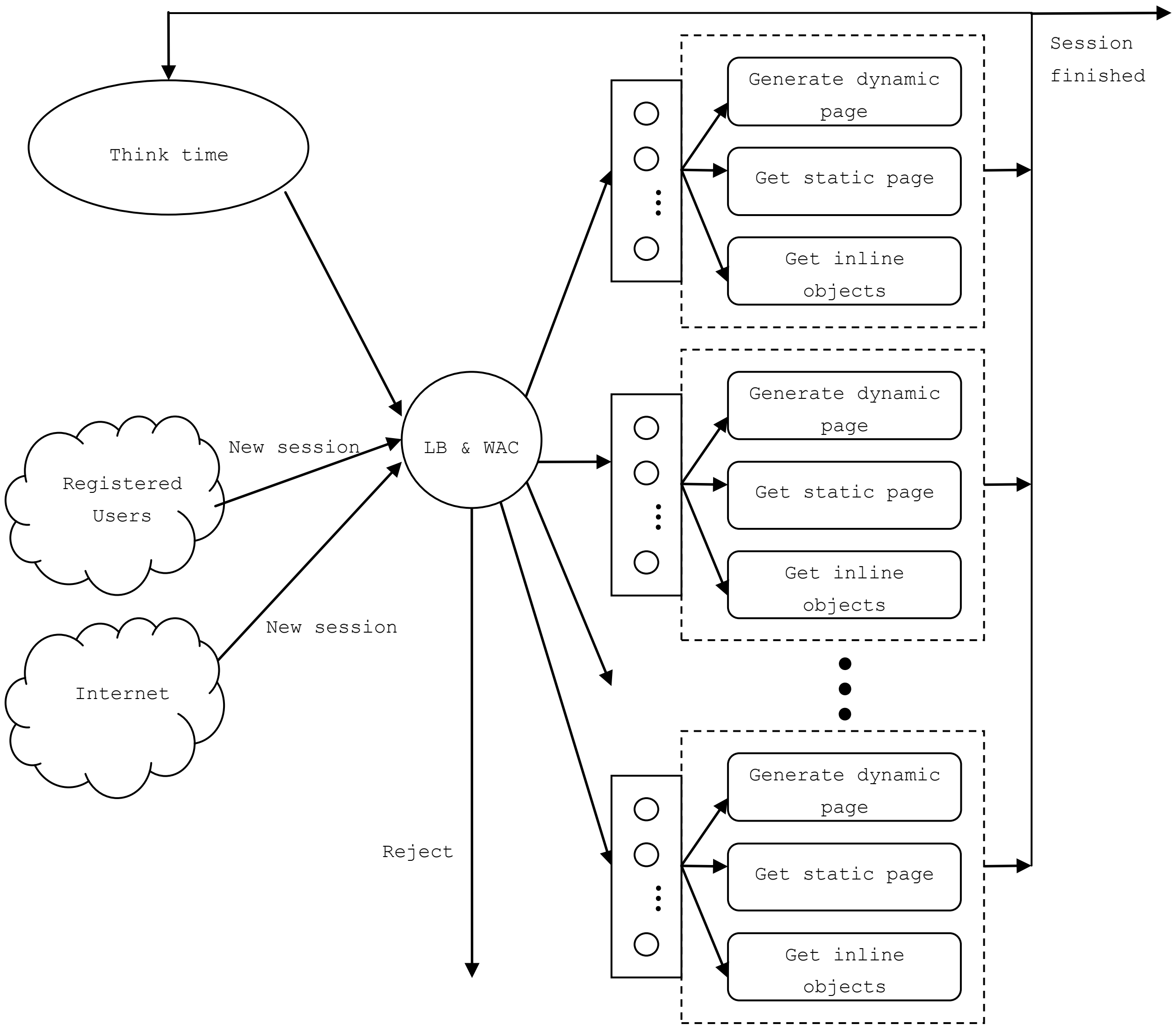


Figure 4: Graphical view of the model

4.1 The arrival process

As Paxson and Floyd explain in [14], request arrivals at a web page cannot be modelled accurately by a Poisson process. However, session arrivals can be. Therefore, our arrival process is based on sessions. These sessions arrive according to a Poisson process. We will refer to these sessions as user sessions. A user session can be one of two types, either a registered session, from a user that has logged in to a website, or a regular internet session, from an anonymous user. Registered sessions arrive with rate λ_{sr} and internet sessions arrive with rate λ_{si}.

The same deterministic number of page views *r* originates from each user session. These page views consist of two requests, a HTML code request and an inline objects request. We will refer to this pair of requests as a page session. A HTML code request can be either dynamic or static. The request type will be drawn from a Bernoulli distribution where a dynamic request is drawn with probability *d* and a static request is drawn with probability (1-*d*).

In reality, a page session will have more than one inline object request, even up to hundreds. However, these requests arrive approximately at the same time and can almost be seen as one request. Because they arrive at the same time, they see the system in the same state, causing them to have approximately the same waiting time and sojourn time. Moreover, the control mechanisms we are evaluating will treat them the same as well.

We have however separated the HTML code and inline objects. The reason for this, is that while downloads of inline objects are independent of each other, they are not independent of the download of the HTML code. This means that any of the inline object requests is rejected, all others will still be downloaded. However, if the HTML code request is rejected, the client will not know what inline objects to request, so these will not be requested. Following these arguments, we suggest it is realistic to treat the inline objects as one request to simplify the model and speed up the simulation

Page sessions are separated by a think time, which is exponentially distributed with average *T*.

4.2 The servers

The system has *S* identical servers. These servers can be switched on or off in order to scale the system and in order to analyze the impact of (temporary) web server defects. Each server is a processor sharing server with a limited number of threads, denoted by *n*. When are threads all occupied, arriving jobs will be queued until a thread is freed. The queue size is infinite.

The processing time for each request is drawn from an exponential distribution with different means for each of the request types. The static, dynamic and inline objects requests respectively have mean processing times β_s, β_d and β_o. If the total processing time for a page exceeds a certain limit, *r,* the session will be aborted.

4.3 Admission control

WAC is performed based on the number of arrivals per second. It can be performed globally or for each server separately, determined by the parameter WAC_{serv}.

WAC can be set to be page session aware, meaning it will never reject an inline objects request when the HTML code request was admitted. This is determined by the parameter $WAC_{session}$. If WAC is set to be session aware, load balancing automatically is as well.

Finally, WAC can differentiate between registered users and regular internet users. It will accept requests from any origin up to WAC_i requests in a second. From WAC_i to WAC_r it will only accept requests from registered users. Above WAC_r it will reject all requests.

Clearly, there are a lot of parameters to control the admission control mechanism. Therefore, WAC is best explained in a diagram, as shown in diagram 1.

When a request is rejected, it will be retried for a fixed number of times, denoted by q. If the request is still rejected after q retries, the session will be aborted. After the reject, first there is one think time period. After the think time period, the entire page session is retried, so every retry starts again with a HTML code request.

4.4 Load balancing

Load balancing can be performed using one out of the following three algorithms: Least Connection (LC), Round Robin (RR) or Random Assignment (RA), denoted by LB_{arg}. Least Connection is really based on the number of jobs being processed or in the queue of a server. The other two algorithms work as described in chapter four.

Load balancing can be set to be page session aware. If so, the inline objects request will always be assigned to the same server as the HTML code request. This would be the case if session information is stored on the server and cannot be shared with the other servers in the system. This parameter is denoted by $LB_{session}$.

If WAC is performed per server, it is assumed the load balancer is aware what servers still accept requests and never assigns a request to a server that will reject it if there are others available.

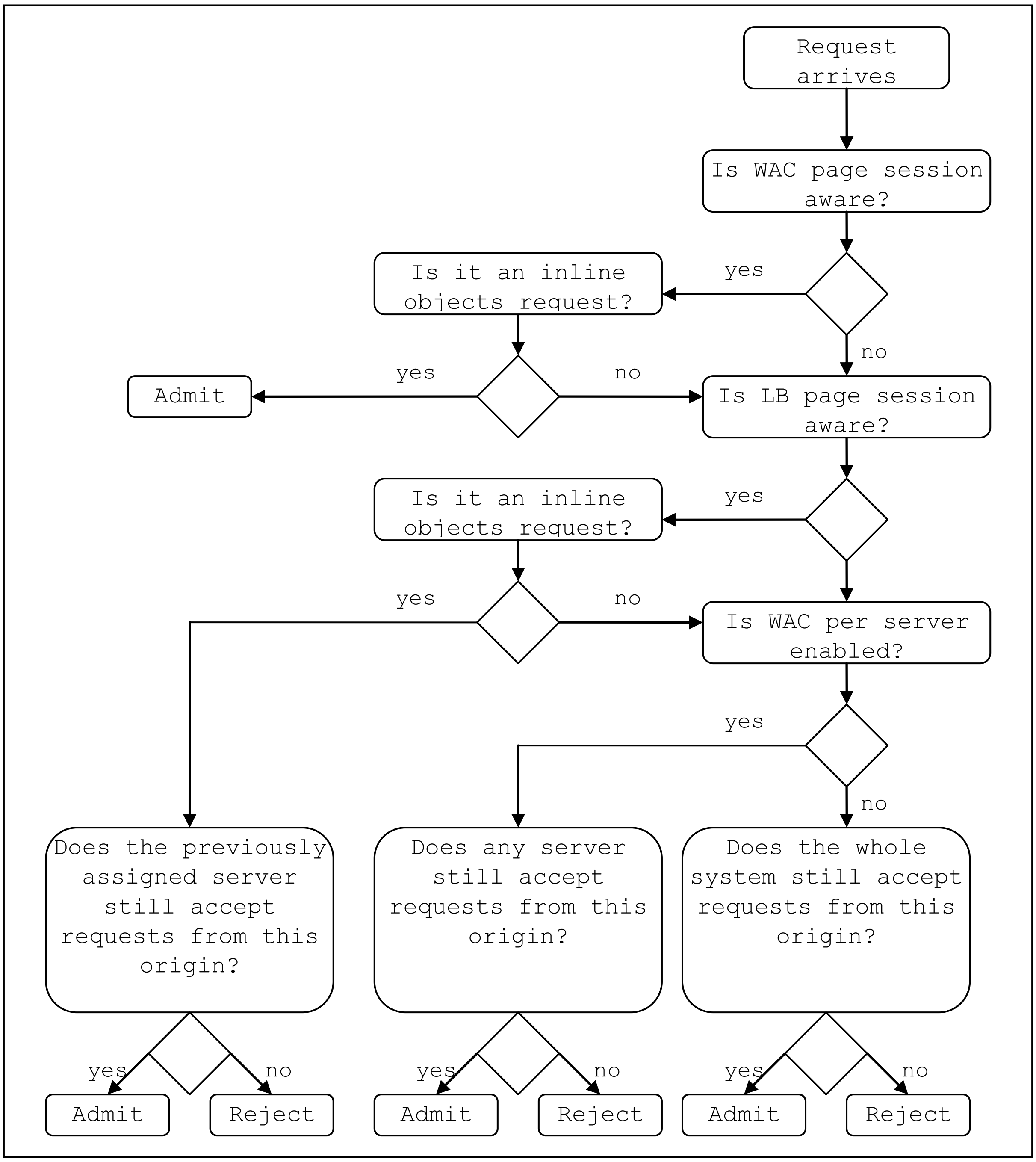

Request arrives
Is WAC page session aware?
yes
no
Is it an inline objects request?
yes
no
Admit
Is LB page session aware?
yes
Is it an inline objects request?
yes
no
Is WAC per server enabled?
yes
no
Does the previously assigned server still accept requests from this origin?
Does any server still accept requests from this origin?
Does the whole system still accept requests from this origin?
yes
no
Admit
Reject
yes
no
Admit
Reject
yes
no
Admit
Reject

Diagram 1: Web Admission Control decision

5 Approach

For analysing a model like this we can distinguish two general approaches. We can either use a mathematical approach or use simulation to produce the desired insights. Both methods have their own advantages and disadvantages. It is often critical to choose the right approach, as this can have a large influence on the end results of a research project.

5.1 Mathematical analysis

Mathematical analysis has as an advantage that it will often produce exact results. Of course these results are only exact for the model, which itself is an approximation of reality. Another advantage of mathematical analysis is the fact that it is relatively fast compared to simulation. In the best case mathematics will give an answer by simple calculation, in the worst case it will take some iterations. Either way, it generally produces results faster than simulation.

The benefit of mathematics giving a result faster than simulation is that one can get results for slightly different scenarios, where only a number of values differ from the old one, quite quickly. However, if the scenario differs slightly more, for example if one would want to analyze the influence of a change in probability distribution, the solution might not be valid anymore. In this case, the model would have to be reanalyzed, which would take time and could even prove to be impossible.

Another disadvantage is the analysis can easily become very complicated. If a model is too detailed, one might not be able to find an answer with mathematics. In such a case either the model will have to be simplified, or one can try to find upper and lower boundaries. Both of these solutions will lead to a less accurate result.

5.2 Simulation

The most important disadvantage of simulation is that is takes time. Simulation is based on the law of large numbers. The consequence is that something has to be done a large number of times, before any useful result can be achieved. There are many types of simulation, but none of them escape this fact.

Another disadvantage is simulation will never give exact answers. However, by simulation long enough, one can achieve any required accuracy. This brings us back to the fact that simulation takes a lot of time. Any accuracy can be achieved if the necessary time is available.

The most important advantage of simulation is its flexibility. Simulation can analyze practically any model, even the most complex ones. Moreover, small changes in a scenario can be implemented easily. For example, changing a probability distribution is a matter of minutes. However, after implementing the change, the simulation will have to be run all over again, which takes time.

5.3 Our approach

In our case, we want to be able to analyze a large number of different, complex scenarios. For this analysis, time is not a noteworthy bottleneck. Considering these constraints, simulation seems the appropriate choice.

Another motivation for choosing simulation over mathematics is the fact that mathematics might be unable to copy with the scenario where admission control is enabled and load balancing set to least connection. This type of state dependent open queuing network can not be expressed in a product form solution and any insights in this scenario would be very hard to attain using mathematics. Considering this is one of the most intelligent scenarios, we are expecting good results from it and leaving it out is definitely not an option.

Now that the choice for simulation is made, the question of how to implement the simulation model remains. Again, we have two choices. Either we use a graphical tool to create the simulation model in, or we program the simulation model from scratch, in any available programming language.

Both options have advantages and disadvantages. Programming from scratch gives more freedom, as you can do anything you like, rather than being bound to the options given by a simulation tool. Also, it can be significantly faster, depending on the language used and the quality of the programming.

Using a graphical tool is often more flexible, as in most tools one can change a large number of parameters for your model very easily. For example one can usually set an arrival process to use a specific distribution for the inter-arrival time with only a few mouse clicks. Maybe the most important benefit of a graphical tool is the graphical representation that it uses. Not only does this give a clear overview of what happens in the model, it makes it very easy to explain and discuss a model between any amounts of people. Furthermore, it facilitates validation of certain parts of the model when a graphical presentation and preferably animations while running a simulation are available.

We decided to make use of a graphical tool that offers the ability to write your own code, to extend the options that are given by default. The tool we are using is called *Extend*. The ability to write your own code gives the user nearly as much freedom as using a programming language. Besides that, simulation time was not a bottleneck in our case. Finally, the ability to discuss a graphical representation of the model facilitated the collaboration during our project a lot.

We have translated our model into an Extend simulation. The simulation supports up to five separate servers and matches the model given in chapter 4. For detailed information about the implementation in Extend, see appendix A of this document.

5.4 Validation

In order to validate the results of the simulation, we have run some very simple scenarios and compared the results to known theoretical results. For the first part of our simulation, we are simulating a simple M/M/1 processor sharing queue. To achieve this, we will have to disable all but one server. Also, we have to remove the inline objects requests from our simulation and set the processing time for static and dynamic requests to the same value. Finally, we will disable admission control.

Load balancing has no effect, as there is only one server. Neither has the fraction of dynamic requests, as the delay time for static and dynamic requests is the same. Also it does not matter whether we have registered or regular internet arrivals, because admission control is disabled. Therefore we will refer to the total arrival rate as λ and the delay time as β. We will refer to the load as ρ.

We will look at three different cases for the M/M/1 processor sharing queue.

1. $\lambda = 4 \quad \beta = 0.1 \quad \rightarrow \quad \rho = 0.4$
2. $\lambda = 12 \quad \beta = 0.05 \quad \rightarrow \quad \rho = 0.6$
3. $\lambda = 4 \quad \beta = 0.2 \quad \rightarrow \quad \rho = 0.8$

The theoretical result for this queue gives for the mean sojourn time

$$E[T] = \frac{\beta}{1-\rho}$$

For the three cases this gives the respective values for the mean waiting time of 1/6, 1/8 and 1. The results of the simulation are listed in table 2.

Scenario	**Mean**	**Lower-bound 95%-confidence interval**	**Upper-bound 95%-confidence interval**
1	0,167671	0,166529	0,168813
2	0,124514	0,123719	0,125309
3	0,990324	0,967057	1,01359

Table 1: Results validation HTML code only

We did 20 runs for each of the scenarios. As we can approximate the distribution of the means of these runs as with a normal distribution, we can give an approximate confidence interval. For every scenario, the theoretical result is inside the 95%-confidence interval. We conclude the simulation produces accurate results for the M/M/1 processor sharing queue.

With inline objects enabled, the server will continue processing the inline objects immediately after the HTML code, because the queue is full processor sharing with an unlimited number of threads. Therefore we can treat this as one job, enabling us to establish theoretical results for the mean sojourn time using the formula specified before with $\beta = \beta_{html} + \beta_o$. Validation for this scenario was done in the same way as the validation for HTML code only.

As the theoretical results, if any, for the model with load balancing and/or admission control enabled are complicated, we will not do any validation based on this. Fortunately, Extend enables us to animate the simulation. This way, we can see exactly what happens during a simulation run. We used these animations to validate the decisions made by load balancing and admission control.

For load balancing we had Extend show us the number of jobs in each server. Using this information, we could verify if the decisions made by the load balancing algorithm were correct. Similarly, we had Extend show us the number of jobs admitted into the system so far. With this information, we were able to verify the decisions made by the admission control algorithm. We conclude load balancing and admission control in our simulation functions as expected.

6 Results

In the experiments we focus on a server farm consisting of 5 servers. In order to produce realistic results, we need realistic parameters for our model. Some of these parameters were already determined in chapter 2. For clarity, they are listed once more below.

• Processing time static HTML code:	0.01s
• Processing time dynamic HTML code:	0.03s
• Processing time inline objects:	0.25s
• Fraction of requests that is for dynamic content:	0.5

In other words, for the processing times $\beta_s = 0.01$, $\beta_d = 0.03$ and $\beta_o = 0.25$ and for the fraction of dynamic requests $d = 0.5$.

This leaves us with five parameters to be determined:

- The number of threads for a server (n)
- The think time (T)
- The number of retries after a reject (q)
- The number of page views per session (r)
- The maximum download time for a page before the session is aborted.

We set the number of threads to the apache default, which is 256. For the think time and the number of retries after a reject, we consulted several papers. We found the choices in [15] to be quite realistic, giving us a think time of 5 seconds and a 1 retry after a reject. The maximum download time was already discussed in part I of this paper, the part about the WPM. There we defined user availability as the fraction of times the download time was under 8 seconds. Here we again take 8 seconds as the maximum.

Because we did not succeed in finding a value for the number of page views per session in any paper, we had to take another approach. We have analyzed statistics for a large number of websites. This analysis revealed most sites have an average number of page views per session around 5. This is the value we will use in our experiments.

This gives us $n = 256$, $T = 5$, $q = 1$ and $r = 5$. The parameters that we do not yet have a value for are the arrival rates and the WAC and LB settings. These will be determined by our scenarios.

6.1 Scenarios

For some initial results, we look at the influence of different load balancing algorithms, combined with admission control either global or per server. This gives us six scenarios, as shown in diagram 2.

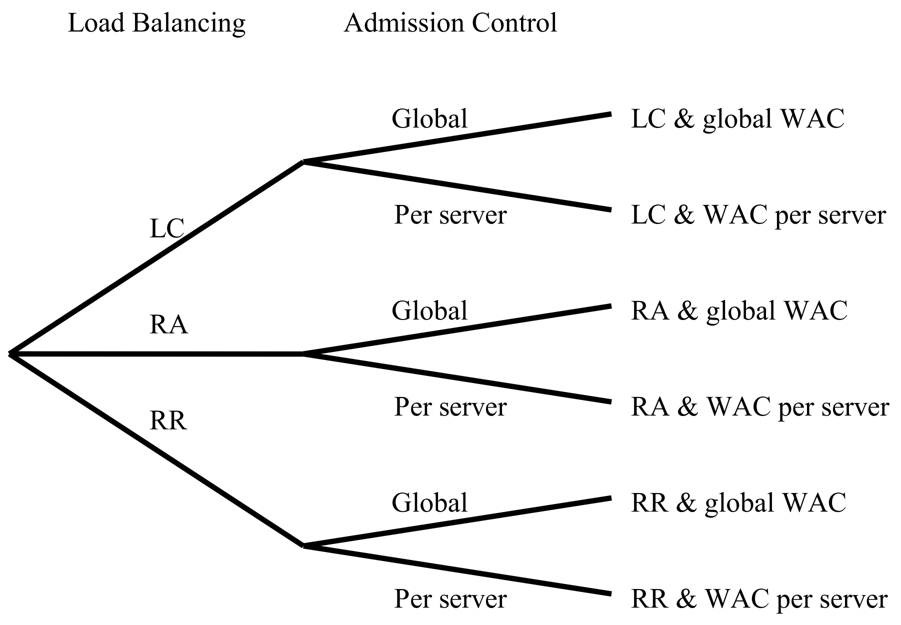


Diagram 2: six different scenarios

To make sure WAC has some effect, we will have to choose a high load for the system. For our first runs we use an offered load of 99%, such that WAC will be active during high peaks. While 99% is the offered load, the actual load on the servers will be lower, due to aborts.

First we will have to find the average arrival time corresponding to this load, as this is the parameter we need as input for our model. We decided not to differentiate between registered and internet users yet, so all load will be due to regular internet users.

We are looking for an arrival rate λ_{si} corresponding to a load of ρ=0.99. If we define the number of pages requested per second as λ_{r} and the total processing time for a page as β_r, we can calculate the load as:

$$\rho = \frac{\lambda_r \beta_r}{S}.$$

The total processing time for a page as β_r can be calculated as

$$\beta_r = d\beta_d + (1-d)\beta_s + \beta_o.$$

Given that the system is stable, the number of pages requested per second is equal to the number of page views per session times the session arrival rate, so

$$\lambda_r = r\lambda_s.$$

By using the values for our model parameters determined earlier, we can now express the arrival rate in terms of the load.

$$\lambda_s = \frac{S}{r(d\beta_d + (1-d)\beta_s + \beta_o)}\rho ,$$

where

$$\frac{S}{r(d\beta_d + (1-d)\beta_s + \beta_o)} = \frac{5}{5 \cdot 0.27} = \frac{100}{27} .$$

For a load of 0.99, this gives a session arrival rate of 99/27 sessions per second.

6.2 Optimal WAC parameter

This leaves one parameter to be set. This parameter is the number of requests per second WAC will allow. We will refer to this as the WAC parameter from now on. It is interesting to optimize the WAC parameter for the scenario’s we are looking at.

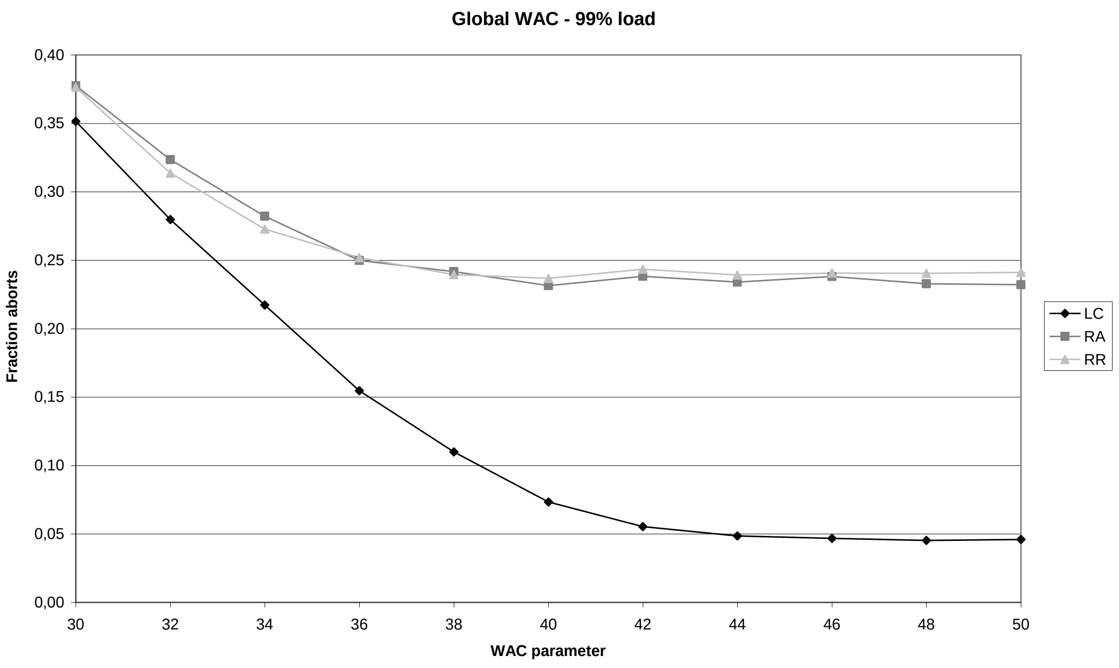
Before we do this, we have to define optimal. As mentioned in [15], revenue from a website is usually only generated if a session is completed. For example, ordering a product from a website goes through several stages, like selecting a product, entering address and billing information and finally a confirmation. The order is placed if the whole session was processed successfully. Therefore, a good measure for the performance of a website is the fraction of sessions that is completed successfully, or similarly, the fraction of aborted sessions.

We will try to optimize the performance of a website in terms of short term revenue by minimizing the fraction of sessions that ended in an abort. We say short term, because long term revenue might be related to other measures, like response and download times or website design. If a customer successfully places an order, but sees lower download times or better design at another website, he might order future products elsewhere. However, we will leave this effect out of consideration.

6.3 Simulation results

Because we are looking at the combination of load balancing and web admission control, we have to look at scenarios where both are active. If the load is too low, WAC will not be activated, therefore we have chosen to look at high load scenarios. The first we will look at is at 99% load.

In graph 1, the fraction of aborts is plotted against the WAC parameter for all three load balancing algorithms at 99% load.



Graph 1: Fraction of sessions that are aborted for global WAC at 99% load.

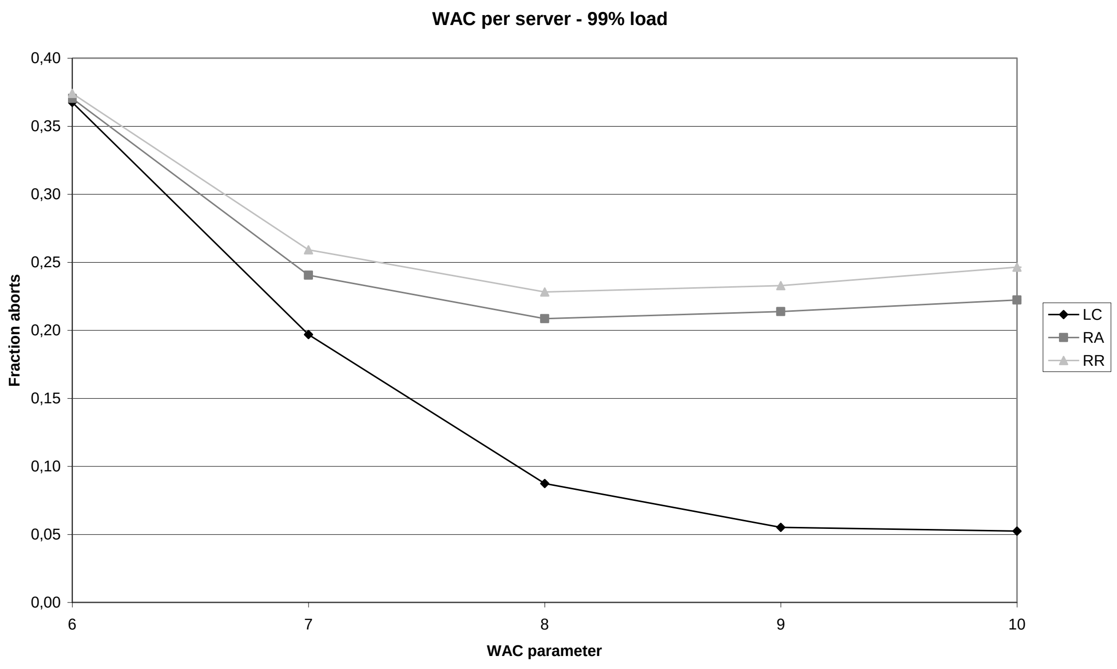
This graph shows some interesting results about WAC and LB. First of all, we can observe the performance increase of LC over RR or RA is impressive. It decreases the abort rate with around 80%, from 0.25 to 0.05. The difference between RA and RR however, is hardly worth mentioning, if it is there at all.

The fact that LC beats the other two algorithms is not surprising, because it is the only algorithm that is state dependent. This means it needs more information than RR and RA. Of course this information comes at a cost, the cost of communicating with each server and storing information about them. However, the benefit of LC is so large, that it would probably be worth the costs.

As mentioned, the difference between RA and RR is negligible. This is surprising, considering the RR algorithm uses more intelligence than RA. What this algorithm does is sending the next request to the server that has not been sent a request the longest. This should make it more likely that the request arrives at a shorter queue. However, the results do not reflect this fact. It might be interesting to compare RA and RR in a simpler scenario, to see if our choices or assumptions influenced this behaviour.

For WAC we can really conclude only one thing. In this scenario, at 99% load, WAC should be turned off, independent of the load balancing algorithm. If WAC would have a positive effect at some setting, we would have seen a minimum for a particular WAC parameter value. However, the graphs are monotonically descending, therefore WAC should be turned off or set to such a tolerant value that it will not cause any aborts.

In graph 2 we have the same graph, only for WAC per server.



Graph 2: Fraction of sessions that are aborted for WAC per server at 99% load.

The graphs look quite similar. However, there is one interesting observation to be made. WAC per server does not have a clear influence with the LC or RR load balancing algorithms, but clearly improves the performance for the RA load balancing algorithm. This could be explained by the fact that WAC per server adds some intelligence to the load balancing algorithm, by rejecting requests if the algorithm sends too many requests to the same server.

For LC, this will only have a negative effect, because LC takes into account how many requests were sent to the server *and* how many requests were processed by the server. The WAC per server algorithm might cause requests to be sent to a server that has more requests in processing and in queue at that moment, which, at least in the case of exponential processing times, is less efficient.

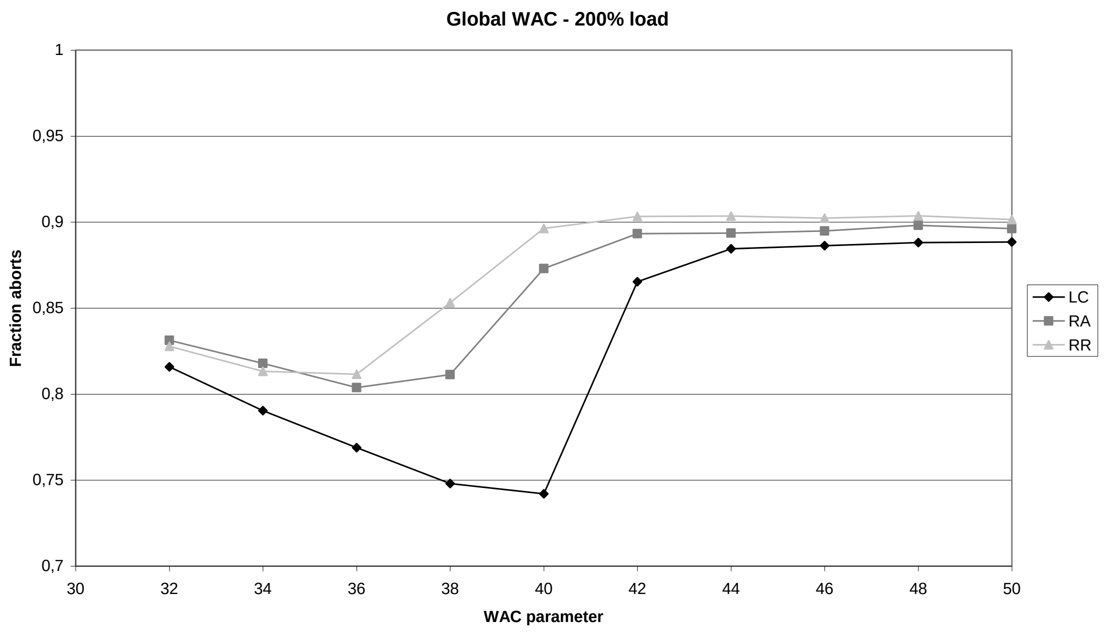
For RR, global WAC or WAC per server will not make any difference at all. RR fills the WAC slots one by one WAC per server will never influence the load balancing decision.

For RA, WAC per server could have a positive effect. Because the RA algorithm is completely random, it could, by chance, send a number of requests in a row to the same server. At some point the WAC per server algorithm will let this server reject requests and let them be sent to another server. This benefits the performance.

However, not only the load balancing is affected by the switch from global WAC to WAC per server. In the case of 5 servers, we might, for a certain scenario, find an optimum WAC parameter for global WAC of N requests per second. Because the servers are identical, we would set the WAC parameter for WAC per server to *N/5*. Because only natural numbers make sense for this parameter, we would have to round it. This could cause the number of requests per second that is let into the system as a whole to be sub-optimal.

6.4 200% Load

Of course we have only been looking at scenarios where WAC normally would not be enabled up to now. Because it is interesting to look at scenario's where WAC is actually effective, we will look at scenarios with 200% load next. In graph 3 the fraction of sessions that is aborted is shown for all three load balancing algorithms with global WAC for the 200% load scenario.



Graph 3: Fraction of sessions that are aborted for global WAC at 200% load.

For this scenario, a clear optimum is visible for the WAC parameter for all three load balancing algorithms. Again, several interesting observations can be made. First of all, it is obvious WAC is essential during overload. Combined with LC, session success rate is more than doubled.

However, this success is strongly dependent on the choice of the WAC parameter. It is crucial that this parameter is not chosen too high. If we look at the graph for LC load balancing, we see that the optimum is at 40 requests per second. If we increase this to 42, we loose almost all benefit from WAC.

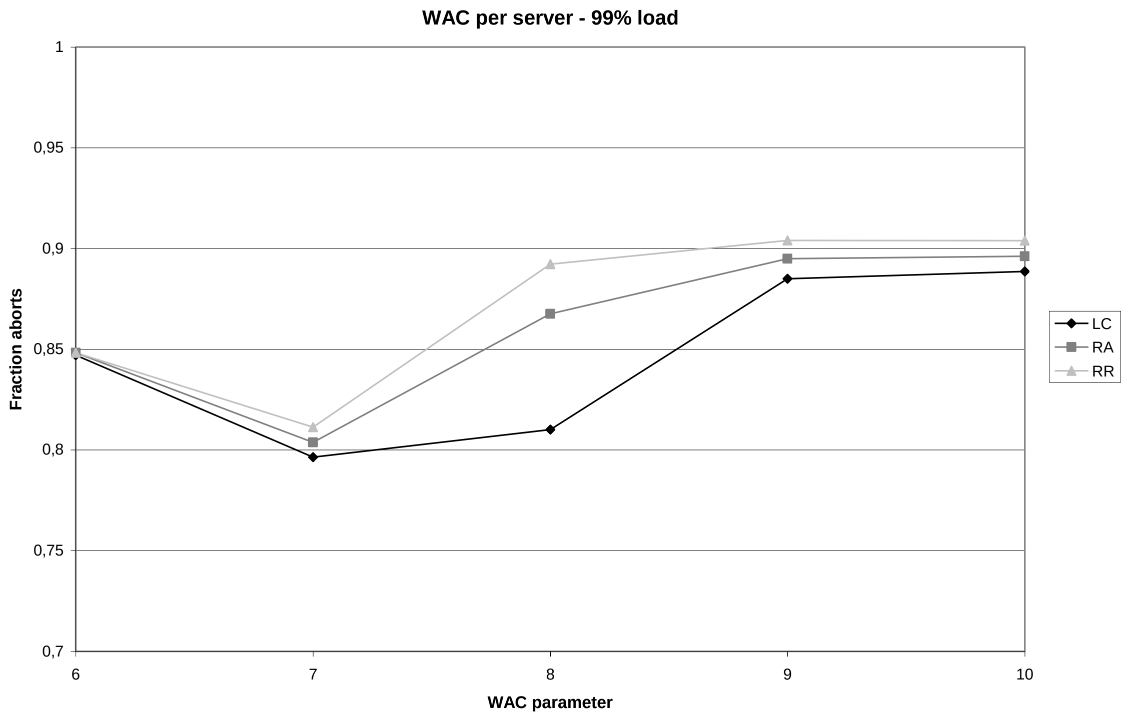
To be on the save side, we could take a value of 38, which has hardly any effect on the fraction aborts and with that the session success rate. If the load is increased though, the value might again be too high, causing the same effect as the step from 40 to 42 in this scenario. In order to keep session success rate near the optimum, it would be advisable to have the WAC parameter update automatically with load changes.

Another fact that is of interest is the effect of the choice of load balancing algorithm. Contrary to the 99% load scenario, when WAC is (almost) disabled, the difference in performance between the various load balancing algorithms is negligible. Where the fraction aborts dropped from nearly 0.25 to under 0.05 before, it just stays at 0.9 at 200% load.

If we look at the optima for each graph, there is a clear difference between LC and the state independent algorithms. However, it is not as impressive as in the 99% load scenario. LC decreases the fraction aborts from just over 0.8 to just under 0.75.

It can be concluded that the choice of load balancing algorithm is not as essential at higher load, while WAC clearly is. It seems that the servers are so full most of the time, the effect of the distribution of requests between servers only has a minimal effect. Of course this only holds as long as the servers are identical and the requests are distributed evenly in the long run, as is the case here.

On a side note, if we look at the session success rate, we see an increase of 25%, which is close to the increase seen at 99% load. Therefore, this conclusion is mainly valid from the users point of view, looking at aborts instead of the server throughput in successful sessions.



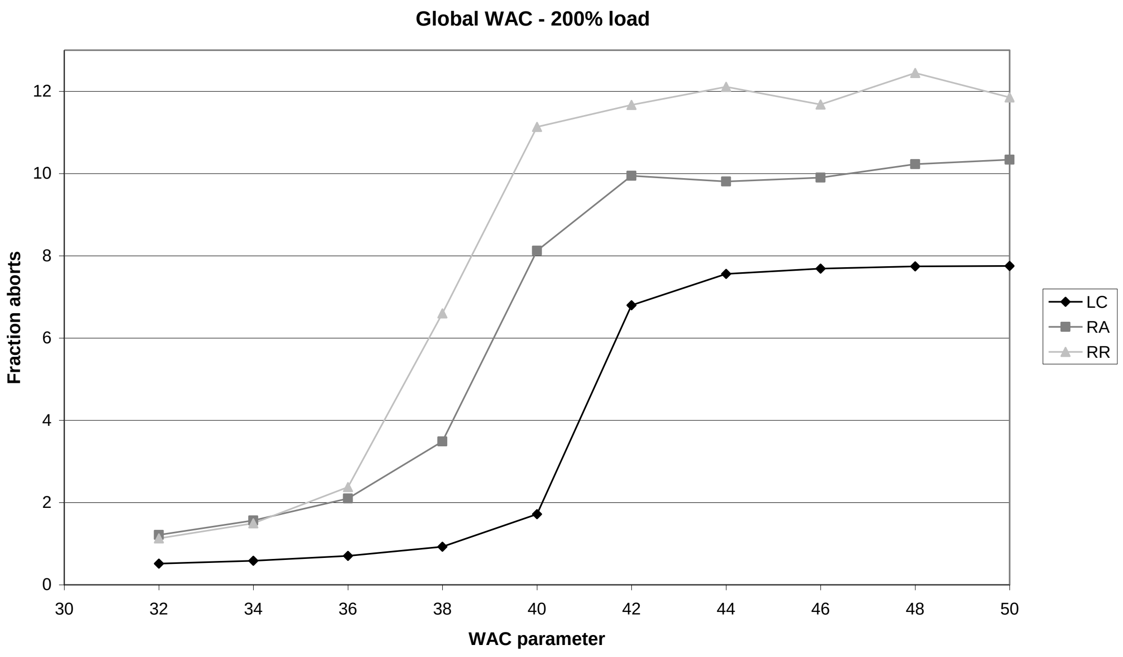
Graph 4: Fraction of sessions that are aborted for WAC per server at 200% load.

In graph 4 the results for the 200% load scenario with WAC per server are shown. The first thing we see is that in this scenario, the influence of the choice of load balancing algorithm is negligible. The negative effect of the high load, combined with the effect of WAC per server has limited the difference between the algorithms extremely.

We are also unable to observe the positive effect of WAC per server on RA load balancing anymore. As said before, with the servers being as full as they are, the distribution of requests only has a minimal effect and the already small contribution of WAC per server was nullified.

6.5 Processing times

Another measure of interest is the processing time. We will look at the average processing time for one of the scenarios discussed earlier. The graph below shows the average processing time for global WAC at 200% load.



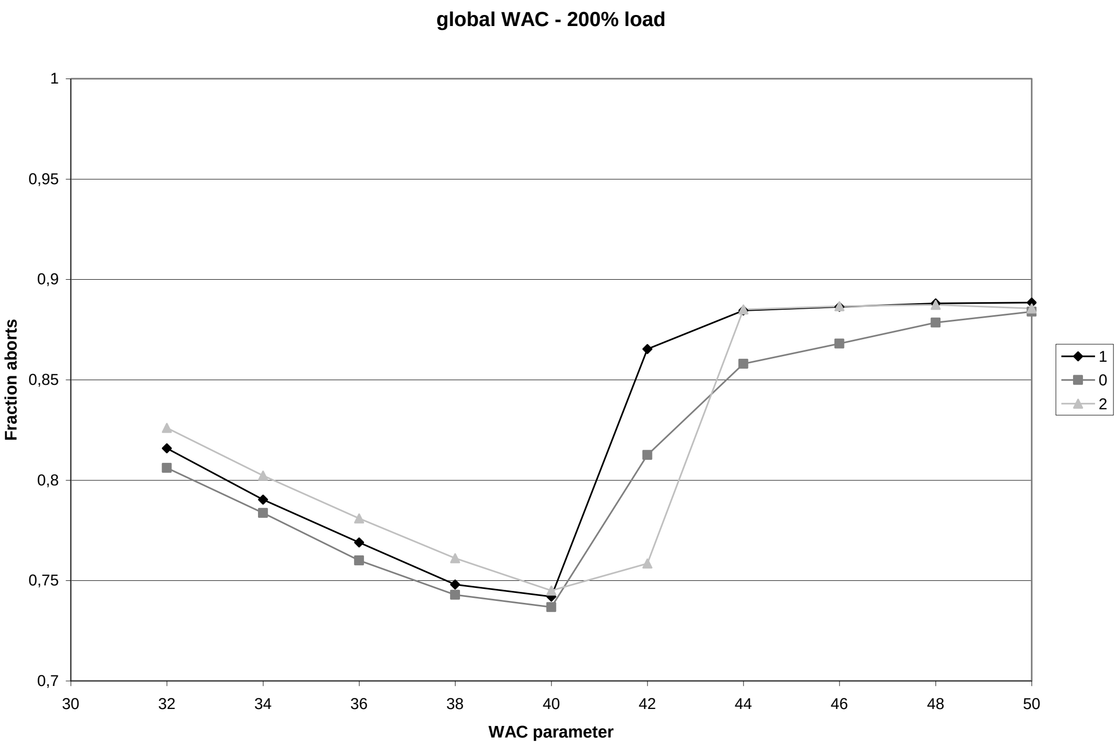
Graph 5: Average processing time for global WAC at 200% load.

The optimal WAC parameter determined by the method used above was 40 for LC and 35 for RA and RR. If we look at these points in graph 5, we see that processing times increases very slowly up to this point, but increase very fast beyond it. This means that not only for the session success rate, but also for the average processing time it is crucial that the WAC parameter is not chosen too high.

We also observe that the average processing time is around 2 seconds for the optimal WAC parameter, which is very acceptable. Once this point is passed, it increases by a factor of 4 or more, to the unacceptable level of nearly 8 seconds for LC and over 10 seconds for RA and RR.

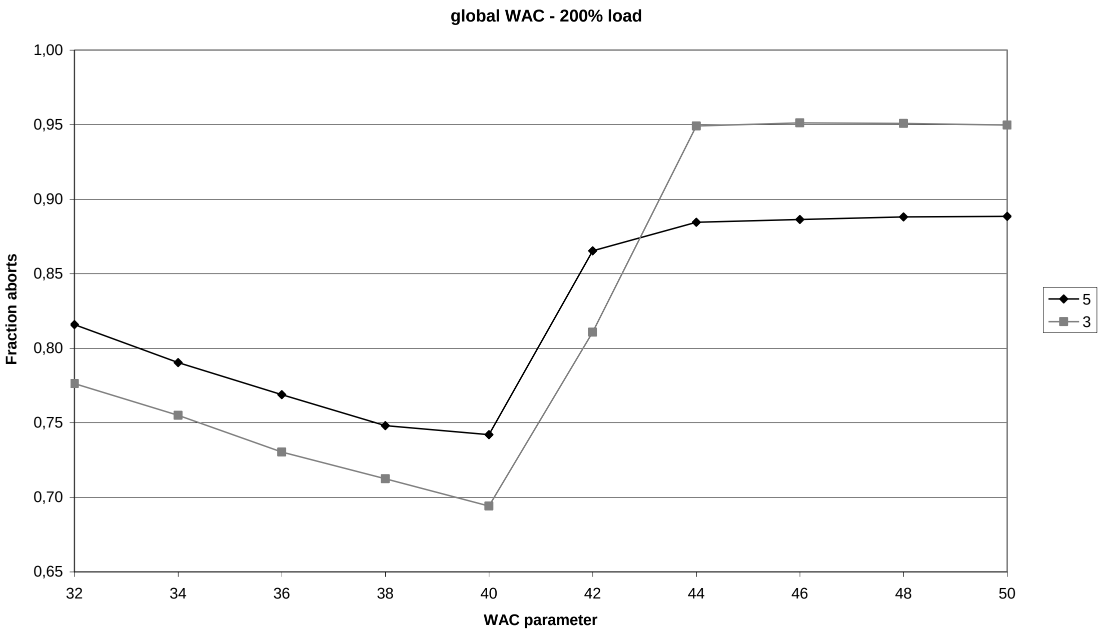
6.6 Varying parameters

To evaluate how a number of assumptions we have made influence the results, we look at some alternative values for the number of retries after a reject and the number of page views per session. In graph 6 we compare our previous scenario of 1 retries at 200% load with global WAC and LC load balancing with scenarios where the number of retries is 0 or 2.



Graph 6: Varying the number of retries at 200% load with global WAC and LC load balancing.

The lines differ slightly at a WAC parameter between 40 and 44. Outside of that interval, the difference is minimal. The optimum is the same for all three scenarios and behaviour when varying the WAC parameter hardly deviates from the original scenario. This leads us to the conclusion that the choice of number of retries does not affect our results, as long as a realistic parameter is chosen.



Graph 7: Varying the number of page views per session at 200% load with global WAC and LC load balancing

In graph 7, we evaluate the influence of the assumption of having 5 page views per session, by comparing the results to the scenario with 3 page views per session. Again, we do this at 200% with global WAC and LC load balancing.

What we see is that the shape of the graph is the same, only stretched vertically. The explanation consists of two parts. First we explain why the line for 3 requests per second lies under the line for the original scenario. If the number of page views per second is lower, WAC will be more effective in preventing useless work being done. Assuming the probability for a reject is the same for every request in a session, the requests that have already been processed when a session is aborted, *u*, can be calculated using the following equation.

$$u = \frac{1}{v}\sum_{0}^{n-1} v = \frac{1}{2}(v-1)$$

This function is gives higher values for higher *v*, meaning the work that has already been done on a session that is aborted gets increases with the number of page views per session, causing WAC to be less efficient.

Of course, this goes for the aborts due to large processing time as well. Although the function is slightly different, it increases with the number of page views. However, while WAC contains some intelligence to regulate the arrival process, these aborts do not. Because there are less page views in a session, the average abort will cause less relieve on the system. This hurts the regulating effect of the aborts, causing the fraction of sessions that are aborted to increase.

While the number of page views per session affects the value of the fraction of aborts, the behaviour and optima are exactly the same. For this reason we believe our results are valid for scenarios where this parameter is different and even in reality, where the number of page views per session varies.

7 Conclusion & future work

7.1 Conclusion

In the previous chapter we looked at three load balancing algorithms, combined with two types of web admission control. We evaluated these six strategies at two load levels and found some interesting results.

First of all, we have seen that the Least Connection load balancing algorithm clearly outperforms Round Robin and Random Assignment. Apparently, there is a large benefit in state dependent load balancing. However, this effect was far less under a higher offered load. Because queues fill up very fast under high load, the choice of which server should handle the next request is less interesting. We conclude that the choice of load balancing algorithm is of higher importance in cases where the offered load is less than 100%.

For web admission control, we had to conclude the best strategy at an offered load of 99% was to turn it off. Obviously, this goes for any lower offered load as well. We can conclude WAC should only be enabled in case of overload. At an offered load of 200%, we saw clear benefits from web admission control. When using LC load balancing, we saw an increase in session success rate of more than 100%. We conclude that web admission control, as implemented in the simulation, is only beneficial in case of overload, but can produce impressive performance increase.

However, the benefit of WAC is easily cancelled out when the wrong parameter is set. Our system's capacity was just over 37 requests per second. The optimal WAC parameter was always between 36 and 40 requests per second. We recommend having WAC limit the number of requests to the capacity of the system, but only activating it during overload. Because we saw no clear benefit in performing WAC per server and negative effects on LC load balancing were very clear, we see no reason to implement this in any system.

Finally, we would like to emphasize that we haven't nearly used all possibilities of our simulation tool. While the results so far were interesting, the constructed tool has enabled TNO to analyze a large number of different scenarios. Moreover, this tool can be used to analyze specific cases for clients, where a website might just have 2 or maybe 10 requests per session, more or less processing intensive pages, all of which can be easily entered into the simulation tool as parameters.

7.2 Future work

One of the most interesting concepts that we have not taken into account is session awareness in web admission control. Session awareness means WAC does not make an admission decision for each separate request, but only once for each session. This will eliminate the useless work performed on sessions that end in an abort due to WAC. The reason for this is quite obvious, WAC will never reject in the middle of a session. Either no work is performed on the session, or the session will be admitted completely.

The only useless work that is being done is on sessions that abort due to long processing times. Our results indicate this part of the aborts is very small as long as the WAC parameter is chosen at the optimum or lower. Because of the added intelligence to WAC it might even be optimal to choose the parameter even lower than we do now, decreasing the aborts due to long processing times even more. Results from other authors show almost 100% of the capacity can be used effectively.

A feature we did implement in our model, but did not consider in our analysis yet, is prioritizing registered users. One can imagine managers of websites wanting to give priority to users that have logged in to a website. For example, a bank might want to give online banking priority over pages with general information.

This system could very easily be adapted to prioritize certain URLs over others. A web store, for example, might want to prioritize the actual ordering of products over the browsing of catalogues. Even without requiring users to log in, this could be realized by prioritizing specific URLs.

Finally, as mentioned in the conclusion, the simulation tool we constructed can be used to analyze specific cases for clients that do not match the average website. A number of parameters can be easily adapted to conform to various scenarios, making this tool an excellent aid in helping clients with configuration choices.

8 References

[1] Javaranch FAQ. http://faq.javaranch.com/view?CachingStrategies

[2] D. A. Menascé. 2003. Scaling Web Sites Through Caching. *IEEE Internet Computing* 7, 4 (Jul. 2003), 86-89.

[3] Y. Lu, A. Saxena, and T. Abdelzaher. Differentiated caching services; A control-theoretical approach. In Proceedings of the 21st International Conference on Distributed Computing Systems, 615-624, Phoenix, AZ, April 2001.

[4] D. Roubos, S. Bhulai and R.D. van der Mei. Performance analysis of session-level load-balancing algorithms. 2007.

[5] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. Technical Report CMU-CS-02-143, Carnegie-Mellon University, June 2002.

[6] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware Web servers. In Proceedings of IEEE INFOCOM 2002.

[7] B.M.M. Gijsen, P.J. Meulenhoff, M.A. Blom, R.D. van der Mei and B.D. van der Waaij. 2004. Web Admission Control: improving performance of Web-based services. Proceedings Computer Measurements Group international conference, CMG

[8] T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In the International Workshop on Quality of Service, London, UK, June 1999.

[9] ITU-T Rec. G.1030, Estimating end-to-end performance in IP networks for data applications, International Telecommunication Union, Telecommunication standardization sector, November 2005

[10] A. Bouch, A. Kuchinsky and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In Proceedings of CHI2000 Conference on Human Factors in Computing Systems (ACM Press, 2000), 297-304

[11] M.D. Welsh. An Architecture for Highly Concurrent, Well-Conditioned Internet Services, Ph.D. Thesis, University of California, Berkeley, August 2002.

[12] D. A. Menasce. Web Server Software Architectures. IEEE Internet Computing 7, 6 (Nov. 2003), 78-81.

[13] D. A. Menascé, Automatic QoS Control, IEEE Internet Computing, v.7 n.1, p.92-95, January 2003

[14] V. Paxson and S. Floyd, "Wide-area Traffic: The Failure of Poisson Modeling," IEEE/ACM Transactions on Networking, pp.226-244, June 1995.

[15] L. Cherkasova, P. Phaal: Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites. In Proceedings of Seventh International Workshop on Quality of Service, IEEE/IFIP event, London, May 31-June 4, 1999.

A Technical description of the simulation model

The simulation model is based on a group five identical web servers that offer identical content. A combination of admission control and load balancing can be applied in an attempt to optimize performance. Requests arrive at the web servers according to a session based arrival process. Each web server can be turned on or off and several parameters can be used to tune the admission control and load balancing schemes.

The simulation is performed in a tool called Extend. Because Extend is an illustration based tool, this allows us to describe the model by showing and describing several screenshots. We will describe the simulation in the following paragraphs.

A.1 The main screen

In figure 1 the main screen of the simulation model is shown. This screen consists of a number of white blocks, which contain detailed underlying structures that will be discussed later. Besides that, the main screen contains blocks for the simulation input and output.

On the bottom left of the main screen, there is a section with simulation parameters. Instead of describing them all at once, we will describe the parameters as a part of the simulation blocks they affect. On the bottom right, there is a section with server switches. These switches can turn servers on or off. In the current state only server 1 is turned on. We will see the impact of these switches when discussing the “WAC & LB” block.

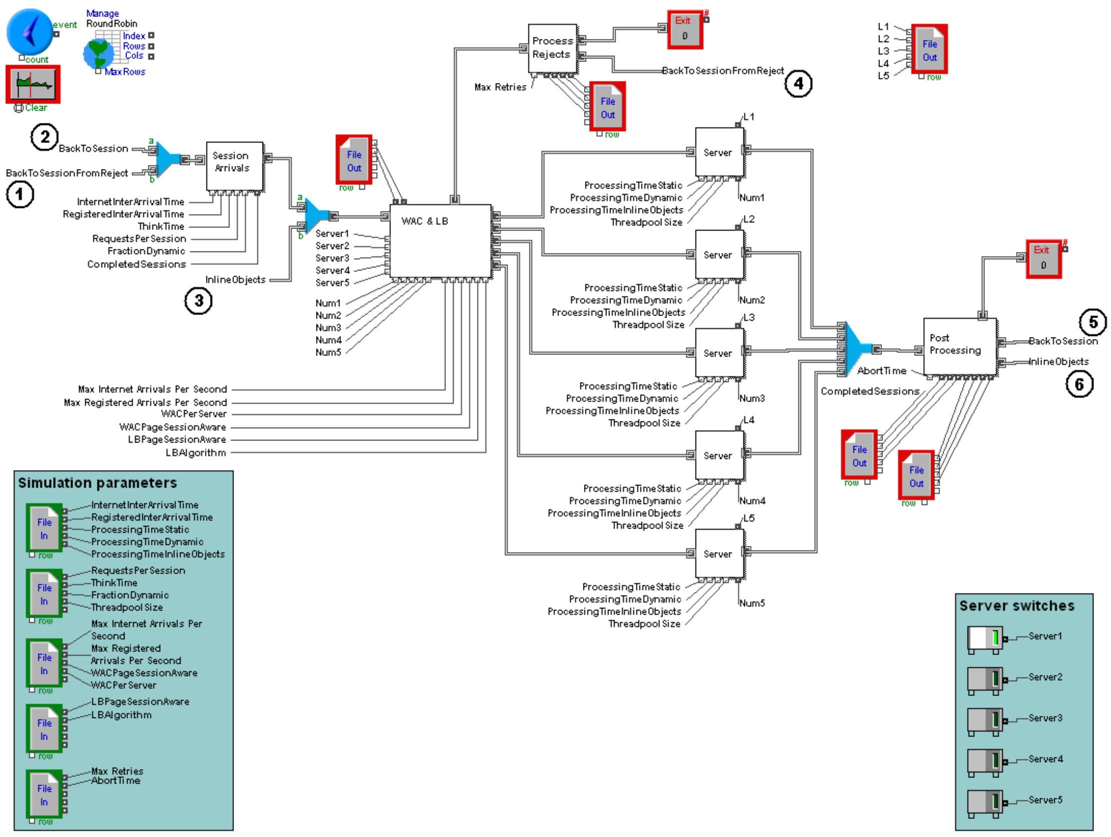


Figure 1: Main screen.

The “Session Arrivals” block controls the entire arrival process. Sessions are created and destroyed in this block and requests inside a session are generated. Each generated page-request consists of two separate requests to the server. It is initiated with a HTML-code request. Once this has been processed, an inline-objects request is sent to the server. We have grouped all inline-objects into one request, instead of a separate request for each object, to keep the scale of the simulation limited, to improve performance in terms of speed. It is a reasonable assumption, considering all requests for inline-objects are sent almost simultaneously, causing it to behave almost as one request.

Requests coming from the “Session Arrivals” block are first sent to the “WAC & LB” block. Admission control and load balancing is performed here. If a request is rejected, it is sent to the top output, on to the “Process Rejects” block. If a request is accepted it will be assigned a server and sent on to the appropriate server.

The “Process Rejects” block decides whether a request will be retried after a reject, or the session is aborted because of it. If a session is aborted, it is sent to the “Exit” connected to the “Process Rejects” block. If the request is retried, it is sent through the bottom connection to “BackToSessionFromReject” at (4), as shown in figure 1. (4) Is connected to (1), causing the request to be sent back to the “Session Arrivals” block.

Each “Server” block contains a queue and a processor sharing node. As long as there is no thread available, arriving requests are queued. As soon as there is a thread is available, a request will be accepted for processing. The server stores the waiting time and sojourn time for each request.

Once the request has been processed by the server, it is sent to the “Route Request” block. This block reads the waiting time and sojourn time and keeps statistics for these values. Where the request is sent next depends on the type of request. If the request was a HTML-code request, an inline-objects request will follow. The request is sent on through the bottom connection to “InlineObjects” at (6). (6) Is connected to (3), causing the request to be sent directly to the “WAC & LB” as an inline-object request.

If the request was an inline-objects request, the page request is finished and the request can be sent back to the “Session Arrivals” block. It is sent through the top connection to “BackToSession” at (5). (5) Is connected to (2), sending the request back to the “Session Arrivals” block, where either another request is generated after a certain think time, or the session ends.

Now that we have given a general description of the simulation as a whole, we will give a more detailed description of the blocks that were discussed.

A.2 The Session Arrivals block

As mentioned earlier, the “Session Arrivals” block controls the entire arrival process. Sessions are created and destroyed in this block and requests inside a session are generated.

Sessions can be of two types, *registered* sessions or regular *internet* sessions. Registered sessions are distinguished to enable analysis of quality differentiation for users that logged in to a subscribers section of a webpage. Once a session is created, it is assigned a fixed number of requests, determined by an input parameter.

When the server completes an inline objects request, the session is returned to the “Session Arrivals” block. After a certain “think time”, another page-request will be sent. When all these requests have been processed, the session ends.

The “Session Arrivals” block has five input parameter and one output parameter. They are described in table 1 and 2.

A.2.1 Inputs

Parameter	**Description**
InternetInterArrivalTime	The average time between internet session arrivals. If this is -1, there will be no internet arrivals.
RegisteredInterArrivalTime	The average time between registered session arrivals. If this is -1, there will be no registered arrivals.
ThinkTime	The average time between the end of one page-request and sending the next one.
RequestsPerSession	The number of requests in each session.
FractionDynamic	The fraction of the page requests that is a dynamic page. The fraction that is a static page will be one minus this number.

Table 1: Inputs for the session arrivals block.

A.2.2 Outputs

Parameter	**Description**
CompletedSessions	The number of successfully completed sessions so far.

Table 2: Outputs for the session arrivals block.

If we open the “Session Arrivals” block, a new screen is opened. This screen is shown in figure 2.

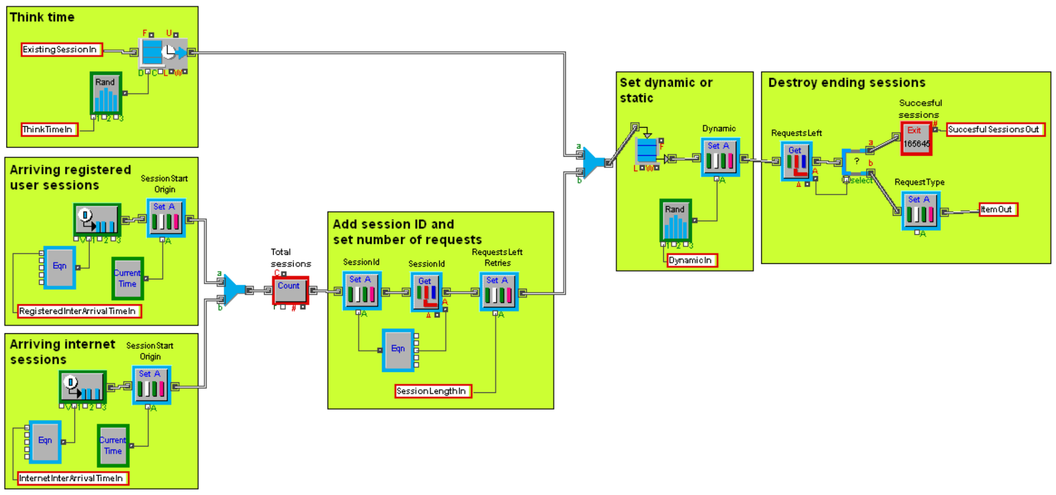


Figure 2: Session Arrivals block.

On the bottom left, two sections can be found, called “Arriving registered sessions” and “Arriving internet sessions”. This is where sessions are generated. The inter arrival time is drawn from an exponential distributions with the `InternetInterArrivalTime` and `RegisteredInterArrivalTime` inputs mentioned earlier as a mean. As soon as a session is generated, the session start time and the origin are stored. The origin is either a registered user or a regular internet user.

After leaving either one of these sections, a session enters a count block where all created sessions are counted. Next, it enters the “Add session ID and set number of requests" section, where, evidently, the session ID and the number of requests in a session are set.

Because a session always starts with a request to the server, the start of the session coincides with the sending of the first request. First there is a section to determine whether the request is for a static of a dynamic page, based on the `FractionDynamic` input. This is drawn from a binomial distribution.

Finally there is a section that checks the remaining number of requests of a session. If this is zero, the session is destroyed. If not, the request type is set to HTML-code request and a request is sent on to the "WAC & LB" block.

In front of the "Set dynamic or static" section, the stream of requests is joined with a stream coming from the "Think time" section. The "Think time" section is where requests that have just been processed and have been sent back to the "Session Arrivals" block arrive. In this block a think time delay is drawn from an exponential distribution with the `ThinkTime` input as mean. After this delay a new request is sent.

A.3 The WAC & LB block

The "WAC & LB" block is the most complicated block. This block models the web admission control and load balancing techniques. It determines if a request should be admitted into the system and if so to which server it should be sent. The block has 10 input parameters and 5 input variables which are used by the simulation to communicate between blocks. The input parameters are described in table 3. The "WAC & LB" block has two outputs; they are described in table 4.

A.3.1 Inputs

Parameter	**Description**
Max Internet Arrivals Per Second	The number of arrivals per second that will be shared between internet and registered users. If this is set to -1, WAC is disabled.
Max Registered Arrivals Per Second	The number of arrivals per second that is reserved for registered users. If this is set to -1, WAC for registered users is disabled.
WACPerServer	0: Off, perform WAC for all servers as a group. 1: On, perform WAC for each server separately.
WACPageSessionAware	0: Off, perform WAC for each request separately. 1: On, perform page session aware WAC, if a HTML code request is accepted, the matching inline objects request will never be rejected.
LBPageSessionAware	0: Off, perform LB for each request separately. 1: On, the HTML and inline objects request will always go to the same server.
LBAlgorithm	0: Load balancing is performed following the LC algorithm. 1: Load balancing is performed by assigning each request to a random server. 2: Load balancing is performed following the RR algorithm.
Server1 – Server5	These inputs indicate whether a server is turned on or off. They can be set using the server switches on the bottom right of the main screen.

Table 3: Inputs for the WAC & LB block.

A.3.2 Outputs

Parameter	**Description**
ArrivalsOut	The total number of arriving requests at “WAC & LB”.
AdmissionsOut	The number of admitted requests.

Table 4: Outputs for the WAC & LB block.

It should be noted that when WAC page session awareness is enabled, the variable “LBPageSessionAware” is ignored and LB page session awareness is always enabled. If this were not the case, some server A could be forced to accept an inline objects request following a HTML request that was accepted by server B, even though it would not accept any new requests. This behavior is undesirable.

Since many scenarios can be distinguished in making this decision, WAC is best explained with a diagram, as shown in diagram 1.

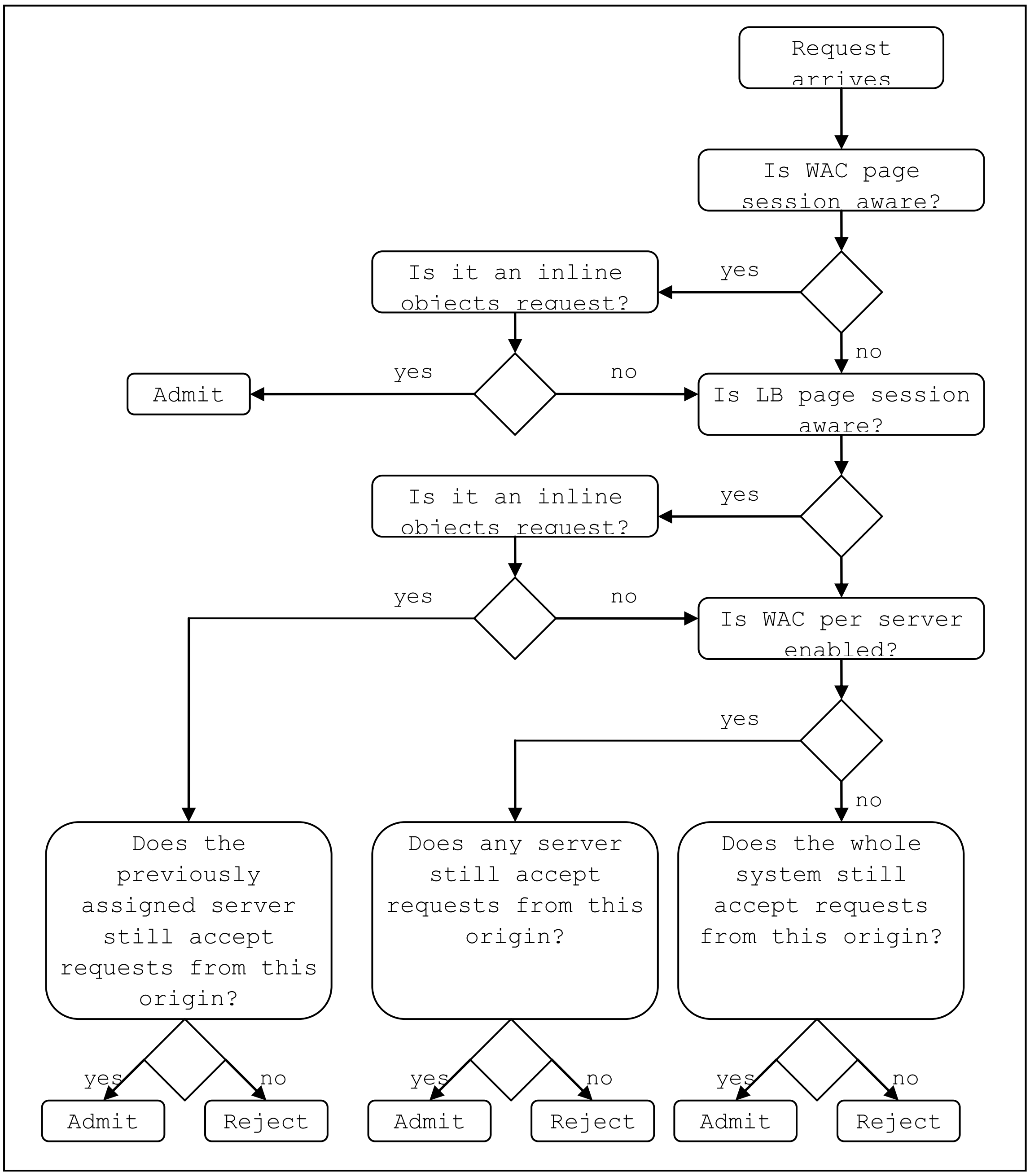

Request arrives
Is WAC page session aware?
yes
no
Is it an inline objects request?
yes
no
Admit
Is LB page session aware?
yes
Is it an inline objects request?
yes
no
Is WAC per server enabled?
yes
no
Does the previously assigned server still accept requests from this origin?
Does any server still accept requests from this origin?
Does the whole system still accept requests from this origin?
yes
no
Admit
Reject
yes
no
Admit
Reject
yes
no
Admit
Reject

Diagram 1: WAC decision diagram.

When a request enters the “WAC & LB” block, it arrives at the “Check origin and server number” section, as shown in figure 3. Apart from checking whether the origin is a registered or internet session and checking if a server number has already been assigned, it also counts the total number of arrivals at this block. This number is equal to the number of requests generated by the “Session Arrivals” block. Finally this section checks whether the request is for HTML code or inline objects.

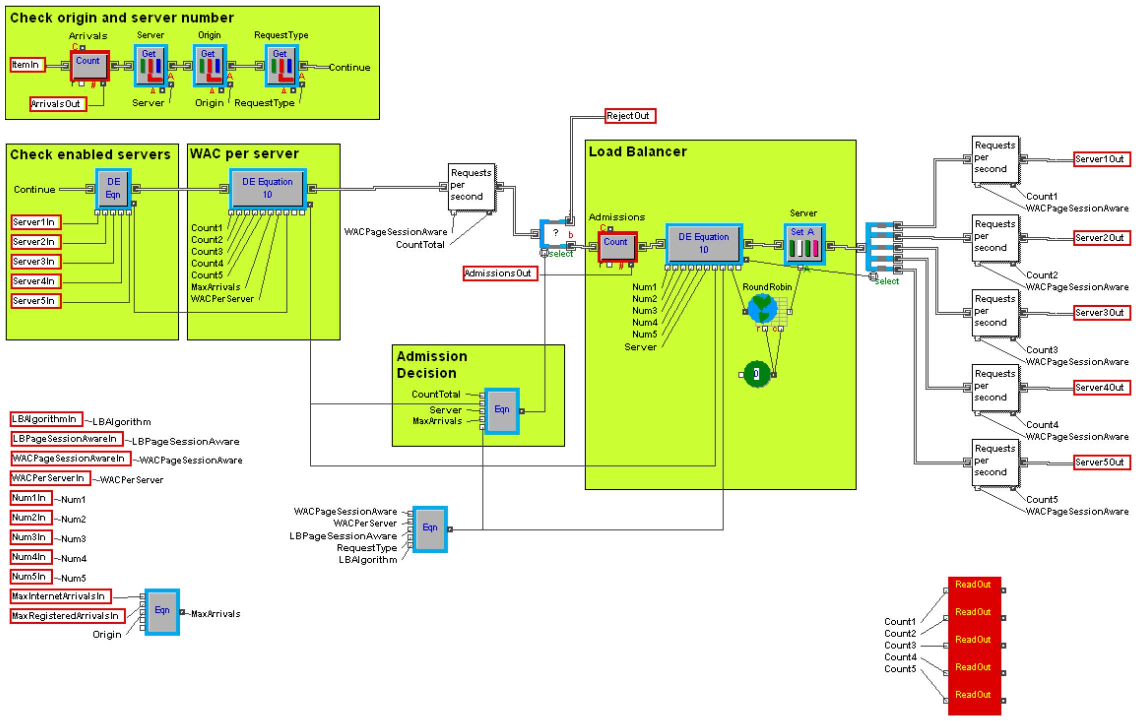


Figure 3: The WAC & LB block.

The origin has to be known to be able to distinguish internet and registered sessions. This is relevant when the number of arrivals per second reserved for both internet and registered sessions is higher than zero. The server number will be stored when a request is assigned to a server. If page session awareness for load balancing is turned on an inline objects request will be assigned to the same server as the HTML code request it follows.

From here, the request is sent to the “Check enabled servers” section. This section checks which server switches are enabled in the main screen and passes this on to the “WAC per server” section.

In the “WAC per server” section, the servers for which the WAC per server algorithm will not admit any more requests, are removed from the list of enabled servers. The result is then passed on to the “Admission decision” and “Load Balancer” sections. If WAC per server is not enabled, the list of enabled servers passes unaltered.

If the request is from a registered session, the `MaxArrivals` input will contain the sum of the input parameters `Max Internet Arrivals Per Second` and `Max Registered Arrivals Per Second`. Each server for which the number of requests that arrived in the current second is smaller than `MaxArrivals` still accepts new requests.

If the request is from a regular internet session, the `MaxArrivals` input will contain the value of the `Max Internet Arrivals Per Second` input parameter. Each server for which the number of requests from internet sessions that arrived in the current second for each server is smaller than `MaxArrivals` still accepts new requests.

This approach ensures that the `Max Internet Arrivals Per Second` slots are shared between registered and internet sessions and that the `Max Registered Arrivals Per Second` slots are reserved for registered sessions.

From the “WAC per server” section, the request passes through a counter to a select block, like the one shown in figure 4. The select block receives an input from the “Admission decision” section. This input decides whether the request should be rejected and sent to the top output (a), or accepted and sent to the bottom output (b). If the request is rejected, it immediately exits the “WAC & LB” block, on to the “Process Rejects” block in the main screen. If the request is accepted it is forwarded to the ”Load Balancer” section.

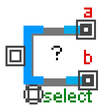


Figure 4.

The “Admission decision” section determines if a request should be accepted based on the process in diagram 1. If WAC per server is disabled, the same approach that was taken to determine if a server will accept a request is now applied to the whole in this section.

Before a request arrives at the ”Load Balancer” section, it first passes a counter. This counter keeps track of the number of request the system has admitted. Next, the ”Load Balancer” section decides what server should be selected to process the request.

If `LBAlgorithm` is set to 0, load balancing is performed using the Least Connection algorithm. This means requests will be sent to the server that has the least number of jobs in queue plus processing at that time. If `LBAlgorithm` is set to 1, load balancing is performed by assigning each request to a random server. If `LBAlgorithm` is set to 2, load balancing is performed using the Round Robing algorithm. If LB session awareness is enabled, the inline objects request will always be processed by the same server as the HTML code request. If not, an inline objects request will be handled in the same way as a HTML code request.

When a request leaves the ”Load Balancer” section, the server number is stored in the request. Next, it passes through a counter for the assigned server and exits the “WAC & LB” block, on to the “Server” block.

A.4 The Server block

The server block takes care of the queuing and processing of requests. The server block has four inputs and only one output. However, waiting times and sojourn times are stored in the requests, so they can be processed in the “Route Request” block later. The inputs are described in table 5, the output in table 6.

A.4.1 Inputs

Parameter	**Description**
ProcessingTimeStatic	The expected time it would take the server to process a static HTML code request if there was only one job present.
ProcessingTimeDynamic	The expected time it would take the server to process a dynamic HTML code request if there was only one job present.
ProcessingTimeInlineObjects	The expected time it would take the server to process an inline objects request if there was only one job present.
ThreadpoolSize	The number of jobs the server can process at once. If more than this number of jobs is present in the server, they will be queued. If this is -1, the number of threads will be infinite.

Table 5: Inputs for the Server block.

A.4.2 Outputs

Parameter	**Description**
Num1 – Num5	The number of jobs in the server. This number is sent to the “WAC & LB” block for load balancing.
L1 – L5	The load for each server

Table 6: Outputs for the server block.

Because the processing time of the inline objects request is largely independent of whether the page is static or dynamic, there is only one input for the processing time the inline objects request.

When a request enters the server block, it arrives at the “Get request information” section. In this section, it is checked whether the request is dynamic or static and whether we are dealing with a HTML code request or an inline objects request. This information is sent to the “Determine processing time” section. Using this information and the earlier described inputs, the expected processing time for the request is determined. This will be used as the mean for an exponential distribution. The final processing time for the request is drawn from this distribution.

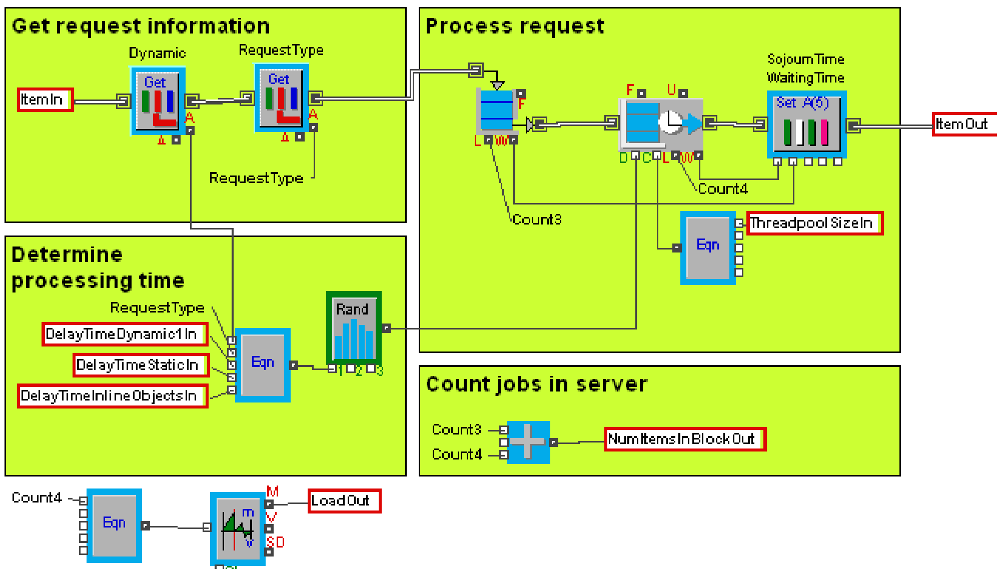


Figure 5: The Server block.

From the “Get request information” section, the request is sent on to the “Process request” section. This section contains a queue and a processor sharing server. As mentioned before, the server has a limited thread pool. When are threads all occupied, arriving jobs will be queued until a thread is freed. The queue size is infinite.

When the server has finished processing a request, the waiting time and sojourn time are stored in the request and it is sent on to the “Route Request” block. The fourth section in the server block, the “Count jobs in server” section, sums the number of objects in the queue and in the server and outputs the total number of jobs in the server block.

A.5 The Route Request block

The “Route Request” block has two functions. First, it sends the request on to the right place. If it receives a processed HTML code request, it is sent directly to the “WAC & LB” block as an inline objects request. If it receives a processed inline objects request, the request is sent back to the “Session Arrivals” block.

Second, it keeps track of the waiting time and sojourn time statistics and separates them according to request type. The “Route Request” block has no inputs and six outputs. The outputs are listed in table 7.

A.5.1 Inputs

Parameter	**Description**
AbortTime	This parameter represents the patience of the user. If the total sojourn time for a page session exceeds this value, the user will abort the session.

Table 7: Inputs for the Route Request block.

A.5.2 Outputs

Parameter	**Description**
PageRequestsComplete	The number of page requests (HTML code and inline objects) that were completed successfully.
HtmlCodeRequestsComplete	The number of HTML code requests that were completed successfully.
SojournTimeStatic	The average sojourn time for static HTML code requests.
SojournTimeDynamic	The average sojourn time for dynamic HTML code requests.
SojournTimeInlineObjects	The average sojourn time for inline objects requests.
WaitingTime	The average waiting time.

Table 8: Outputs for the Route Request block.

When a request arrives at the "Route Request", first the waiting time is stored. Next the requests are divided in HTML code requests and inline objects requests, so they can be processed separately.

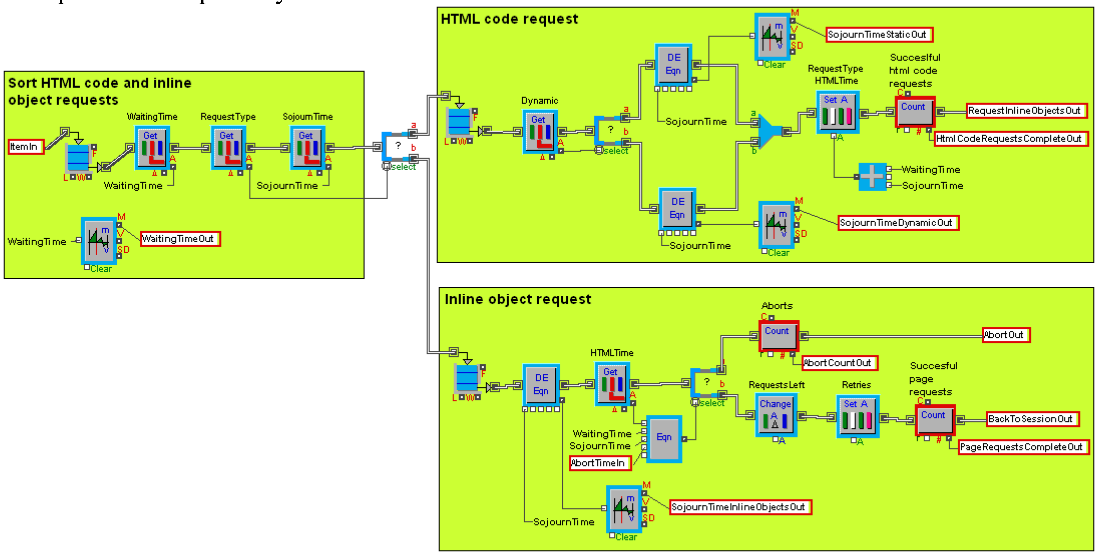


Figure 6: The Route Request block.

The HTML code requests, after the sojourn time has been read, are divided in static and dynamic requests. For each, the sojourn time is stored separately. Next the request type is set to inline objects request, the request passes through a counter to count the successful HTML code requests and an inline object request is sent to the "WAC & LB" block.

For the inline objects requests, there is no difference between static and dynamic pages. After the sojourn time is read, the block will check if the total sojourn time for the request exceeds the `AbortTime` the session will be aborted. If not, the sojourn time is stored and the session will be continued. Since one page request has now successfully been completed, the number of requests left in the session is decreased by one. Also, the number of retries is set to zero. A definition of this value will be gives in the description of the "Process Rejects" block.

Because the request type is set to HTML code request when a request leaves the "Session Arrivals" block, we do not have to set it here. The request passes through a counter and is forwarded to the "Session Arrivals" block.

A.6 The Process Rejects block

The process rejects block decides what happens with rejected requests. This block has one input and four outputs, described in table 8 and 9.

A.6.1 Inputs

Parameter	**Description**
Max Retries	The number of times one request will be retried.

Table 9: Inputs for the Process Rejects block.

A.6.2 Outputs

Parameter	**Description**
RegisteredAborts	The number times a registered user aborted a session after a reject.
InternetAborts	The number times a regular internet user aborted a session after a reject.
RegisteredRetries	The number of times a registered user retries a request.
InternetRetries	The number of times a regular internet user retries a request.

Table 10: Outputs for the Process Rejects block.

When a request enters the "Process Rejects" block, first it is checked if the number of retries so far is smaller than the `Max Retries` parameter. If so, the request should be retried, if not, the reject leads to an abort. Next, the number of retries is increased.

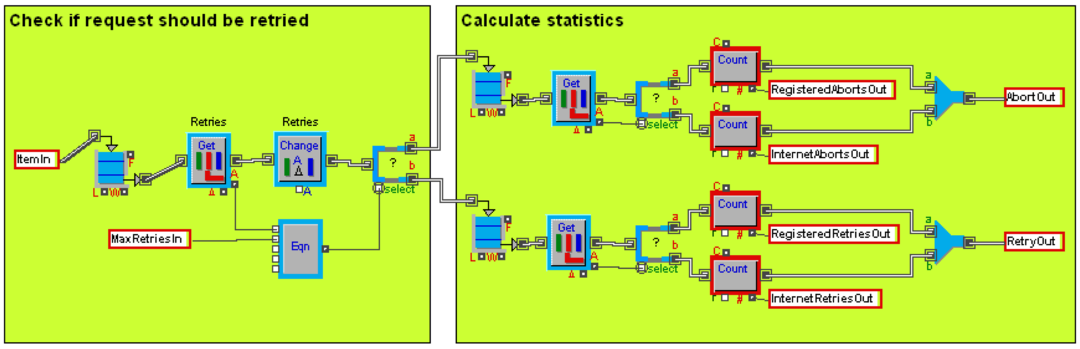


Figure 7: The process Rejects block.

After that, the request arrives at a select block, as we encountered before. If the request should be retried, the bottom output is chosen. If the request leads to an abort, the top output is chosen. Next, the requests are divided in registered and regular internet requests, so the statistics can be kept separated. They pass through a counter, and are sent on to either an exit block for the aborts, or the "Session Arrivals" block for retries. Every retry is sent after a think time