department of computer science
faculty of sciences

vrije Universiteit amsterdam

*Thesis:* Master Of Science
*Specialization:* Internet and Web Technology

# FlashCBC

flash crowd mitigation for dynamic web sites

*Author:*
Robert Maarten Been
1164848

*Supervisor:*
Dr. Guillaume Pierre

December 2, 2007

**Abstract**

Web sites are constantly threatened by sudden surges in traffic, also known as flash crowds. Current systems deal with flash crowds by utilizing replication or edge computing to distribute the load between multiple servers. However, when dealing with dynamic websites, the database part of the system quickly becomes the bottleneck. As a result, current systems are unsuitable when applied to dynamic websites. In this thesis, we introduce flashCBC, a content-blind query result caching middleware designed to unburden the database server during a flash crowd.

# Contents

# Chapter 1

# Introduction

With the enormous amount of internet users, websites are constantly threatened by the possibility of a sudden overload. Such overloads are usually caused by large numbers of users suddenly being interested in the website. This interest can be triggered by several events. For example, the website could be brought to people's attention by large news sites like Slashdot [21] or search engines like Google [11]. However, a website might also be extensively covered by other media. For example, people can be drawn to websites showing the current polls during presidential elections. These overloads are usually referred to as flash crowds, Slashdot effects or hot spots.

During a flash crowd the server will try to service as many requests as it possibly can, dropping all the requests that exceed its limit. Naturally this is an unwanted situation for the owners of such a server.

Non-commercial owners can run into problems with their providers because of bandwidth constraints or other websites hosted on the same server becoming unreachable as well. Commercial owners however can face severe financial losses. When a server is unreachable, customers will often look for another location to buy the product. Therefore, it is of crucial importance that during a flash crowd as many requests as possible are serviced within reasonable time.

Many research efforts have been conducted over the years for addressing flash crowds. Previous research efforts suggested to replicate the website using Content Delivery Networks like Akamai [4] or Globule [9]. The replicas are placed on edge servers that are part of the Content Delivery Network. After the replication clients are redirected to the replicas, reducing the load on the origin server. This approach works well for static websites, where all documents remain the same for all clients. However, when applied to dynamic websites problems arise.

Dynamic websites can display personalized websites to every client. For example, the online e-commerce website Amazon [5] displays a personal web page with recommendations based on previous purchases, recently written reviews and more. As shown in Figure 1.1 a dynamic website usually consists of a web application which uses a database to generate the pages for the client. When a client issues a request, the web application examines it, fetches the appropriate data from the database and generates the correct response for that client.

The application part of a dynamic website can usually be distributed pretty easily using edge computing. One just replicates the application code to the different edge servers and start the application. Equal to static websites, the clients can now be redirected to the

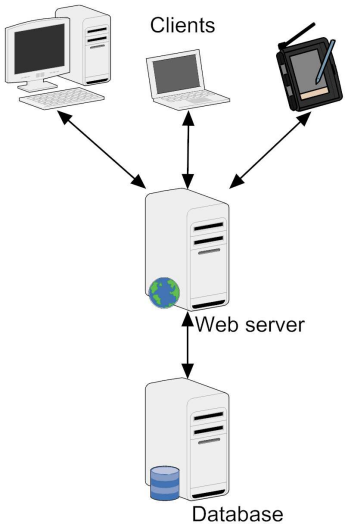



**Figure 1.1:** Example of a dynamic website configuration

different edge servers, increasing the capacity of the website.

Clearly we can gain a similar capacity increase for the database part of the system if we replicate it across multiple edge servers. However, when we do this problems quickly arise. The first problem is one of size. A database can easily be several hundreds of megabytes large. Replicating such an amount during a flash crowd would be too costly for the infrastructure of the Content Delivery Network. However, the main problem is not the size of the database, the main problem is consistency.

When a database is replicated across several edge servers, all copies of the database need to remain in a consistent state at all times. Even though several solutions to solve this problem exist, these solutions require quite a lot of resources to provide this consistency. However, during a flash crowd we need all the resources we can spare to service the incoming requests.

Since replicating the database is not feasible to replicate the database part of the dynamic website, we are faced with a problem. If we replicate only the application part to the edge servers, we reduce the load on the application server. However, all the applications on the edge servers need access to the database. Therefore the database quickly becomes a bottleneck.

Fortunately we have several tools at our disposal to help us deal with flash crowds. First of all, we have a flash crowd prediction system [12]. This prediction system continuously monitors the system load and tries to issue warnings several minutes before a flash crowd actually reaches its full size. This gives us a little time to prepare for the incoming flood of requests.

Second, previous research observed that during a flash crowd the locality of the requests increases a lot [14]. This means that most of the requests made by clients during a flash crowd target the same data. In the case of a flash crowd triggered by a product, most requests will simply be for the details of the product in question. With a news article, most requests will be for the contents of that article.

In another research the authors observed that clients of a weakly consistent distributed system find it confusing when updates they issued earlier seem to be missing at a later time. This can happen when the client gets directed to a new server which did not receive that

update yet. However, updates by other clients do not need to be visible to all clients at all times. As long as a client is displayed with a consistent view of the system for his own writes, the client will be content using the system. This mechanism is called Read Your Writes [23].

Read Your Writes can also be used for dynamic websites. For example, if a client purchases a certain product, the purchase should immediately be visible to the client. However, to the other clients it does not really matter that that particular purchase has been made. A similar observation can be made for news websites. Usually, users can post comments to the articles published on the website. While it would be nice that all users would see all comments made at all times, it is not crucial for the experience of visiting the website. It does not really matter that some comments show up a bit later than usual, as long as the users sees his own comments immediately after they have been made.

Finally, we have GlobeCBC [20]. GlobeCBC uses content blind query caching to reduce the load on a database server. GlobeCBC allows us to store query results at cache servers located between the edge servers and the database. However, since it is a caching mechanism, all updates are propagated directly to the database. This means that the database will always remain in a consistent state. Once an update has been executed at the database, all caches are informed of the update and all related query results are invalidated.

Although GlobeCBC provides a good starting point for dealing with flash crowds, we run into problems with the invalidations sent to the caches. During a flash crowd we will typically deploy as many edge servers as possible, each maintaining it's own query cache. All updates executed at the caches will immediately be forwarded to all other caches. Therefore, the system will eventually be crippled with invalidations coming in from all the edge servers. Effectively the efficiency of the cache will be greatly reduced by the constant stream of invalidations coming in.

This thesis first presents extensive evaluations on the effect of flash crowds on query locality. Using real world examples of flash crowds, I will show that locality increases significantly during a flash crowd. Using this observation, I propose to use an adaptation of GlobeCBC to reduce the load on the database during a flash crowd; FlashCBC. FlashCBC assumes each client is assigned to the same edge server during its whole session with the website. As in GlobeCBC query results are stored at the caches. However, here we relax the consistency scheme. Once an update has been issued at a cache, the other caches are not invalidated immediately. Instead, the invalidation is delayed by several tens of seconds.

By delaying the invalidations, we can greatly improve the performance of the caches. First, cached items remain valid for longer periods in time, improving the cache hit rate. Second, because of the delay multiple updates can occur before the invalidation is finally sent. This way, with one invalidation we capture multiple updates at the same time. With these two improvements the amount of SELECT queries directed at the database server is greatly reduced. Since usually the majority of requests to a database server are SELECT queries, FlashCBC greatly improves the performance of the system.

Since we assume a client will always be directed to the same edge server, it is only necessary to keep the edge server consistent with its own updates. Updates originating from the other servers can even be delayed by as much as 1 minute, while the users do not notice any degradation in the experience of their visits. However, the query caches will not be crippled by invalidations anymore.

All update queries will still be directly executed at the database server. However, the server will still be able to handle the load. Since the majority of requests are SELECT queries, the load on the database will be reduced enough by FlashCBC. Therefore all remaining queries

will be executed normally.

The remainder of this thesis is structured as follows. Chapter 2 introduces the prediction system as well as several solutions dealing with static websites, followed by an explanation of GlobeCBC. Chapter 3 shows the results of my research into flash crowd locality. Chapter 4 introduces our adaptation of GlobeCBC; FlashCBC. Chapter 5 describes the setup used to test the performance of our system and evaluates the results of our experiments. Finally, Chapter 6 proposes future work and concludes.

# Chapter 2

# Related Work

Many research efforts have been conducted to address the problem of flash crowds. With these research efforts many sub problems have been addressed. They can however be classified into related topics as follows.

Section 2.1 starts with a general characterization of flash crowds. After this general characterization, Section 2.2 will discuss some characteristics typical to flash crowds.

Characterizing flash crowds is of course necessary as a first step toward a solution to handle them, however in order to perform any meaningful research a formal definition is needed. Section 2.3 presents such a definition. This formal definition can not only be utilized for research purposes, it can also serve as a framework for flash crowd detection. Besides detection, prediction of a flash crowd is also of vital importance to be able to deal with it appropriately. Indeed, if a flash crowd can be predicted before it actually happens, appropriate countermeasures can be taken in due. Therefore section 2.4 discusses a system that detects and predicts flash crowds.

After a flash crowd has been detected or predicted some form of action needs to be taken to handle the incoming flood of requests. Several solutions have been proposed to solve this problem, Section 2.5 discusses some of these solutions in detail. However, when dealing with dynamic websites a lot of different difficulties are introduced which will have to be dealt with accordingly. In order to successfully distribute the load on a dynamic website we will use a content-blind query caching system based on Globe-CBC [20]. Therefore section 2.6 will introduce the concept of content-blind query caching and globeCBC, along with another content-blind query caching example: Qcache.

## 2.1 Flash crowd examples

A flash crowd is a sudden increase in request rate to a web server. This increase can be caused by various factors. For example, some flash crowds are triggered by large public events like the World cup soccer matches. Other flash crowds can be triggered by publications on large news websites like Slashdot [21]. When such an event takes place or an announcement is made, a large number of people is encouraged to visit the same server. This literally causes a flood of requests toward the website. This flood can last for hours, after which it decreases gradually until traffic levels are back to normal.

Examples of flash crowds caused by large news websites can be found in [3]. Here the

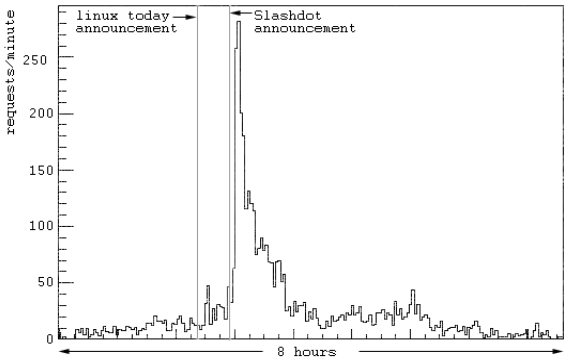



**Figure 2.1:** Adler flash crowd load over time (adapted from [3])

author analyzes three flash crowds he encountered after placing three articles. After publishing each article, large news websites like Slashdot, Linux Today and Freshmeat announced the articles on their front pages. Each of these announcements triggered a flash crowd toward the author's web server. After these events, the author placed an analysis of these events in a new article online [3].

Figure 2.1 shows the traffic pattern during the flash crowd caused by the third article, entitled "An Ode to Richard Stallman". This article had been announced to members of the LXNY group and the general red hat mailing list before it was placed online, therefore we already see a slight traffic increase before the big news site announced the article. The first line indicates when Linux Today announced the article on their website. Interestingly we do not see a great surge in traffic yet. The main surge took place when Slashdot announced the article. Then we see a rapid increase in request rate from about 30 requests per minute to over 250 requests per minute in a time period of about 15 minutes.

Clearly the increase in request rate after the Slashdot announcement is impressive. What is more interesting though is the reason why the increase in requests did not occur after the announcement on Linux Today. This can be explained by the form of the announcement. Where Slashdot simply announced the article and linked users toward the original document, Linux Today published a text-only variant of the article along with the announcement. Apparently less people needed to visit the original document, since the contents could already be read on Linux Today, effectively causing Linux Today to share some of the load of the origin server.

Another type of flash crowds can be found when studying the access logs of the 1998 FIFA World cup. The access patterns of the server are very stable throughout the entire event. Every time there is a soccer match, the server gets flooded with a flash crowd, which leads to about two flash crowds a day. Clearly these flash crowds are not caused by large news sites, but by the sporting event itself. During a match thousands of peoples worldwide accessed the website to remain updated on the match's progress.

In Figure 2.2 we show a typical trace of a day with two matches. The first large peak is caused by the first match, while the second match causes the smaller second peak. Although it is unclear as to why the second peak is smaller than the first, a possible explanation could be that during the second match more people had time to watch the match then during the first. The first match was at the end of the afternoon, therefore a lot of people would still be

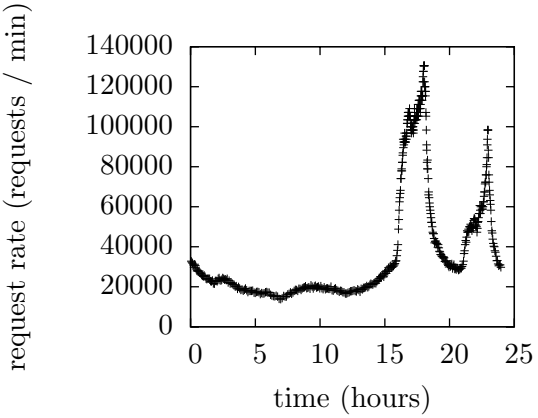



**Figure 2.2:** WorldCup98 flash crowd load over time

working, while during the second match viewers could simply watch the match on television from home.

While all servers should be able to handle a higher load than normal, the load introduced during a flash crowd is of such a magnitude that no server can be expected to handle the load on its own; statically dimensioning every server in preparation of potential future flash crowds would not be economically feasible.

## 2.2 Flash crowd characterization

As explained in Section 2.1 during a flash crowd a server gets flooded with requests. Another similar event where a server gets flooded with request is a denial of service attack. In [14] the author discusses the differences between flash crowds and denial of service attacks, and characterizes several interesting properties of flash crowds.

When we monitor client access before the flash crowd it is possible to partition the clients into clusters. These clusters can be formed by grouping the clients together based on their location. When we have created such a partitioning of clusters we can observe that during a flash crowd most clients originate from clusters which have already been identified before the flash crowd arrived. This means that most of the clients contributing to the flash crowd originate from more or less the same regions as the regular clients.

A very important property of flash crowds is related to the file access patterns. Over 60% of the documents accessed during a flash crowd are not accessed before or after the event. This leads to a problem when activating the content delivery network because even if we were to copy existing caches to the edge servers these servers will be faced with a high number of cache misses at the start of the flash crowd. Another problem we have to face is that the requests for the most popular documents originate from a large number of clusters. Because different clusters access different edge servers all these servers can have many cache misses, which in turn leads to a large load on the origin server to serve the caches at the edge servers.

Although the previous properties are not very promising to apply caching to handle flash crowds, the next observation is. During a flash crowd less then 10% of the requested documents generate more than 90% of the requests. This is clearly in favor of some sort of caching solution. While during the startup phase of the content delivery network the origin server

could still be flooded with requests, after the startup phase almost all load should be handled by the edge servers. If the load on the origin server is still too great at startup some form of cooperative caching could be used. Although the overall time to service requests can increase, the load on the origin server would be reduced.

## 2.3 Formal definition

A relatively simple description of flash crowds such as presented in the previous sections is of course useful to understand the phenomenon. However to be able to conduct measurable experiments which provide us with meaningful results a more formal definition is needed. In [6] the authors provided a first attempt at such a definition. This definition has been improved in [12].

We base the definition of a flash crowd on Figure 2.3. This hypothetical trace contains two flash crowds and one burst. A burst is a short increase in request rate, however the increase does not last long enough to justify the cost of activating the content delivery network. In order to correctly identify the flash crowds several parameters are required. The authors of [6] define $r_t$ as the number of page requests in a time slot. The time slot is defined as the time unit, which can be varied according to the data structure used and desired reaction time. As second parameter they define $H$ as the maximum capacity of the server. Then they define a flash crowd at time $t$ as the event during which $r_t > H$.

Unfortunately the samples taken at each time slot $r_t$ show a large variation. Samples oscillate around $H$ causing the server to constantly switch states between flash crowd and normal state. Because the system would be too unstable in this manner, the authors propose to aggregate the samples into windows of size $W_d$, allowing for reliable detection. Their definition of a flash crowd is therefore:

**Definition 2.1** *We say that traffic is experiencing a flash crowd at time $t$ if the average request rate over $[t - W_d, t]$ is at least $H$.*

Although this is a good starting point for a formal definition, Definition 2.1 is still too limited to be used correctly. We will show this using the hypothetical flash crowd depicted in Figure 2.3. The graph represents the average request rate over $[t - W_d, t]$ for each $t$. In this graph we should identify two separate flash crowds and one temporary burst. The author of [12] improved upon Definition 2.1 by introducing some new parameters to be considered.

First to be able to deal with temporary drops in request rate which last longer than $W_d$ they used the classical watermarking technique. This technique is commonly used by other systems to handle oscillations in the request rate. They define a flash crowd level $H_d$ (low watermark) at a certain fraction of the $H$ threshold (high watermark). They also define three load zones:

- red: $r_t \geq H$,
- grey: $H_d \leq r_t \leq H$,
- white: $r_t < H_d$.

The addition of these new parameters help to stabilize the system even further, but at the large drop in request rate indicated by 6 the flash crowd is still incorrectly assumed to

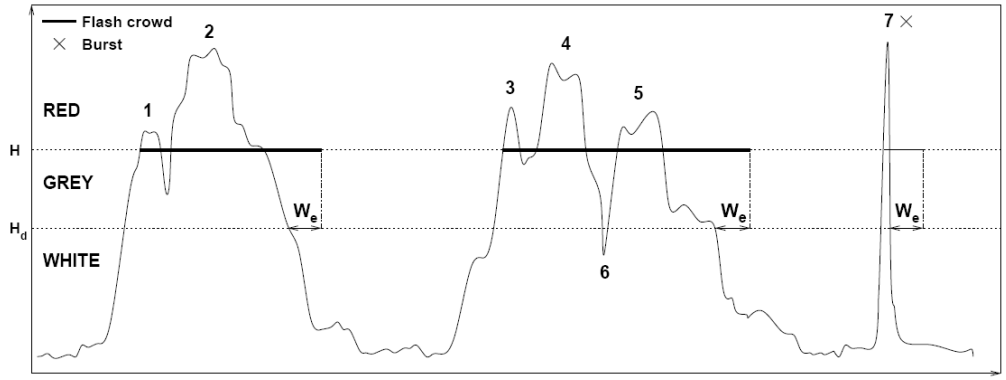



**Figure 2.3:** Hypothetical flash crowd (adapted from [12])

be finished. The drop indicated by 6 is referred to by the authors as a "negative burst" and should not cause the system to decide the flash crowd is finished, since it continues at 5. To remedy this error a new parameter $W_e$ is introduced, which denotes the longest time for which the request rate can remain in the white zone during a flash crowd. Thus a flash crowd should only be considered finished when it remains in the white zone for more then $W_d$ time slots.

Now the two flash crowds of the hypothetical trace are correctly detected, however the burst indicated by 7 is also considered a flash crowd. This is because so far the duration of an overload is not taken into account. Therefore the authors introduce a final parameter $\tau_{fc}$, which denotes the minimal time for which a flash crowd should last. The final formal flash crowd definition is therefore as follows:

**Definition 2.2** *The flash crowd is said to start at time $t_{start}$ when the traffic enters the red zone, i.e. the average request rate over $[t_{start} - W_d, t_{start}]$ is at least $H$. It is said to stop at time $t_{end}$ when it enters and remains in the white zone for more then $W_e$ time units, i.e. $\forall_{s \in [t_{end} - W_e, t_{end}]}$ the average request rate over $[s - W_d, s]$ is less than $H_d$. The difference $t_{end} - t_{start} - W_e$ should be greater than $\tau_{fc}$. All other cases, where for time $t$ the average request rate over $[t - W_d, t]$ is at least $H$ are defined as bursts.*

## 2.4 Detection and prediction

The previous sections introduced and formalized the phenomenon of flash crowds. However, in order to take appropriate countermeasures, an accurate prediction system is required. This system should detect the same flash crowds as those identified based on the formal flash crowd definition given in Definition 2.2. However it should also issue warnings before a flash crowd actually begins, allowing for the origin server to take action in order to preserve high client-handling performance. This section describes such a prediction system, which was introduced in [12].

The prediction system uses the prediction algorithm which [6] introduced. This algorithm uses linear regression to predict future request rates based on samples collected in the past. The authors discuss several prediction algorithms, and demonstrate that linear regression offers an excellent balance between accuracy and simplicity.

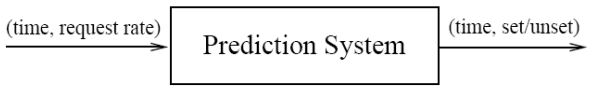



**Figure 2.4:** Prediction system black box (adapted from [12])

### 2.4.1 Linear regression

Linear regression is a mathematical method to predict the value of some independent variable at some future time $t$ given a number of variable samples measured in the past. The authors of [6] show that, When applied to observed request rates at the origin server, we can detect a potential flash crowd a short time before it actually starts using linear regression.

The authors apply linear regression at a time $t$ over samples collected between time $t - W_p$ and $t$. $W_p \geq 0$ is the so called prediction window parameter. This parameter can be used to adapt the amount of samples collected, allowing one to adapt the prediction algorithm to different circumstances. The samples can be used to predict the request rate at time $t + \tau$, where $\tau \geq 0$ is the advance notice parameter. This parameter can be changed to determine how far into the future the prediction should be. If $\tau$ is chosen too large, a lot of false predictions could be given, whereas if $\tau$ is chosen too small, the prediction should be accurate, however it will be too late to take any appropriate actions.

Using $t$, $W_p$ and $\tau$, the authors define a mapping from observations in $[t - W_p, t]$ to a number $P_t \equiv P_t(\tau) \geq 0$ of predicted load value in the interval: $[t + \tau, t + \tau + 1]$, starting $t$ time units in the future. Linear regression uses extrapolation of least-squares linear fit to define $P_t = f(t+\tau)$, where the coefficients of $f(s) = a_t s + b_t$ are chosen to minimize the mean quadratic deviation over the window $[t - W_p, t]$, that is $\sum_{i=t-W_p}^{t} [f_t(i) - r_i]^2$.

### 2.4.2 Prediction system

The prediction system introduced in [12] can be seen as the black box shown in Figure 2.4, which receives a stream of $(time, request\ rate)$ pairs. Each pair denotes the observed request rate for the given time, all pairs are sorted ascendingly according to the time. Upon receiving a new pair from the input stream the prediction system outputs it's decision as a $(time, alarm)$ pair. The *time* field denotes the same time as received from the input pair. The *alarm* field can have two values; *set* if the system should take countermeasures or remain adapted, or *unset* which means the origin server can switch to or remain in a normal state, because no flash crowds are expected in the near future.

In order to make accurate decisions, the prediction system is organized using a modular design. This allows for easy adaptation of the different modules, without affecting the rest of the modules. The four modules used by the system are the aggregator, the detector, the predictor and finally the merger.

The $(time, request\ rate)$ pairs are first preprocessed by the aggregator module. Usually the samples presented to the prediction system are very fine grained, such as one sample per second. However, as discussed in Section 2.3, a larger time interval is needed to provide accurate detection of flash crowds. For the prediction algorithm, usually an even larger interval is needed. Therefore the aggregator first aggregates the pairs into samples of $p_d$ seconds, which can be used by the detector. Then it further aggregates the pairs into samples

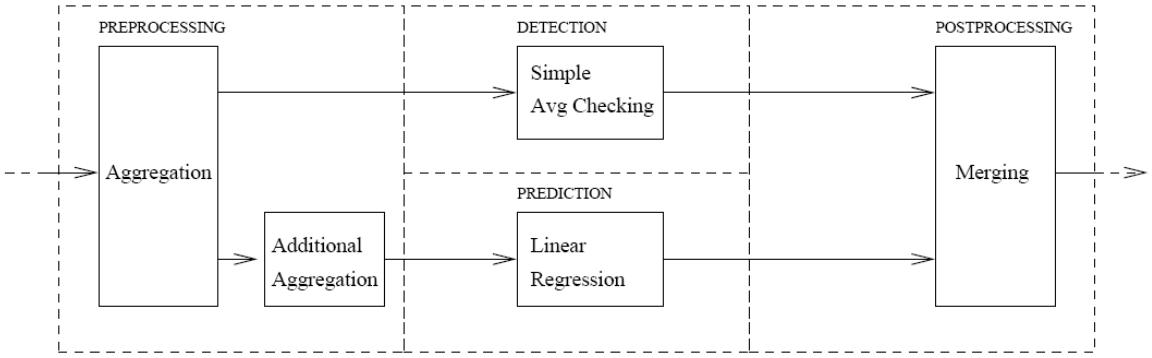



**Figure 2.5:** Detailed architecture of the prediction system (adapted from [12])

of $p_p$ seconds for the predictor.

The detector module implements the formal definition given by Definition 2.2. The detector outputs for each aggregated sample a boolean value indicating whether the system is currently in a flash crowd according to the formal flash crowd definition. To adjust its operation three parameters can be modified.

The first parameter is the window size $W_d$, which denotes the size of a window over which the average request rate is computed. This value should be chosen as a multiple of $p_d$, such that all averages are calculated over the same number of samples. $W_d$ should not be chosen too large though, since it takes on average $\frac{W_d}{2}$ seconds to detect a flash crowd.

The second parameter is the low watermark $H_d$. The low watermark causes the detector to unify closely situated peaks, reducing the number of unnecessary adaptations. Choosing this value too low will result in an ever lasting flash crowd, while choosing this value too high will reduce the detectors capability to connect peaks. Note that the high watermark $H$ can not be changed, since that value is determined by the maximum capacity of the origin server.

The final parameter is $W_e$ which is the interval during which the request rate should remain in the white zone before the flash crowd ends. Using this parameter the system can ignore negative bursts. In practice this is the time during which the content delivery network should remain activated after the flash crowd is finished, and typically depends on the cost of utilizing the network.

The predictor module uses linear regression to predict the formation of a flash crowd. The actual prediction takes place in two steps. First, the predictor uses linear regression to produce a sequence of future loads. Second, it verifies whether and when the load is expected to exceed $H$.

Reliable prediction requires analyzing general trends observed from groups of samples, rather than produce predictions based on few individual points. Therefore the linear regression window $W_p$ should be chosen sufficiently large, as only then it is possible to accurately detect trends in request rates. The window size also determines the maximum value for the advance notice $\tau_{max}$. If a $\tau$ beyond $\tau_{max}$ is chosen, predictions are too unreliable to be made.

However the predictor must not only predict future loads, it also determines whether the current load approaches a dangerously high level. Using this capacity the predictor can inform the merger module when using trend analysis makes no sense, since the current request rate is

already very close to $H$ and therefore $H$ can be exceeded at any moment. The merger can use this information to decide which action to take. A possible action could be to preventingly activate the content delivery network.

Oscillations around $H$ are dealt with in a similar fashion as the detector does. Again three zones are defined: white, grey and red. However the grey zone is chosen much smaller than is the case with the detector. The low watermark $H_p$ is typically chosen at about 90% of $H$. Another difference is that de predictor uses linear regression instead of load averaging to determine the current load, as it is interested in the current situation, not the average situation of the past.

The prediction module interface can therefore be defined as follows:

**Definition 2.3** *We denote the result of the prediction algorithm as a triplet* $(t_1, zone, t_2)$. $t_1$ *is the time at which the prediction was given. Zone corresponds to the load zone to which the request rate belongs at time* $t_1$. *This is the result of linear regression with* $\tau_{max} = 0$. $t_2$ *is the time for which the prediction was issued,* $t_2 \in [0, \tau_{max}] \cup \infty$. *In case the flash crowd is predicted to occur too far away in the future, then* $t_2 = \infty$. *If the flash crowd has already started or has just been detected, then* $t_2 = 0$. *Otherwise* $t_2$ *is the time at which the linear regression line crosses the flash crowd level* $H$.

To clarify the possible outputs of the prediction algorithm, we will discuss the actual semantics of the returned triplets. Being in the white zone $(t_1, white, \infty)$ is the normal situation. In this situation the request rate is well below the capacity threshold, no flash crowd is expected in the near future. However, when the request rate begins to rise and it is expected that $H$ will be exceeded in the near future, the triplet $(t_1, white, t_2)$ is presented. This triplet indicates that although the system is currently safe, a flash crowd is expected in $t_2$ seconds.

When the system is in the grey zone, this is an indicator of a dangerously high request rate. The possible outputs are $(t_1, grey, \infty)$ and $(t_1, grey, t_2)$. Although $t_2$ can be provided, it should only be used as an additional piece of information. The value of $t_2$ could easily be wrong in this situation, since the sample window used by linear regression is relatively long when compared to the small size of the grey zone.

The final situation is the red zone, in this zone no prediction needs to be done, since the flash crowd is already in full effect. The predictor therefore only returns information about the current zone: $(t_1, red, 0)$.

As with the predictor, we can adjust several parameters to fine-tune the behavior of the prediction system. How $W_p$ and $\tau_{max}$ influence the performance of the prediction system has already been discussed at the beginning of this section. However, n additional observation is that $W_p$ should be chosen several times larger than $W_d$. Because the detector needs to be highly responsive to current load values, $W_d$ is typically chosen as small as possible. However the predictor analyzes longer lasting trends in request rates, which can be reliably identified when $W_p$ is chosen large enough.

The general parameters defining a flash crowd apply to the detector as well, including the watermarks and the $W_e$ interval. As is the case with the detector $H$ and $W_e$ define the capacity and cost of the system. The low watermark $H_p$, however, defines the beginning of the predictors grey zone and should therefore be carefully chosen. It should be much higher than the detectors low watermark $H_d$. Typically $H_p$ should be set at about 90% of $H$.

The merger module finally combines the outputs of the predictor and detector module to decide if the content delivery network should be activated or not. In order to make this

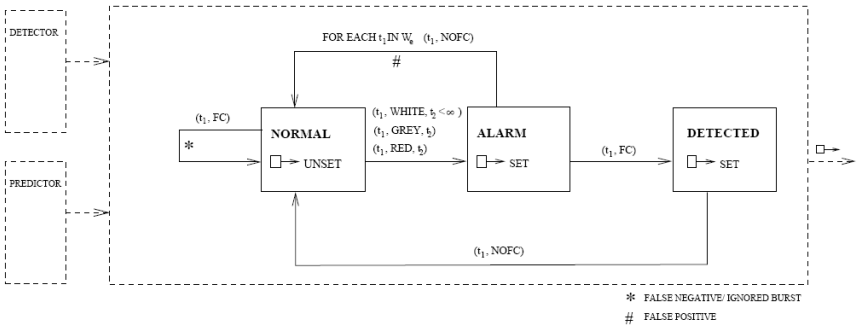



**Figure 2.6:** Merger state machine (adapted from [12])

decision the merger implements the state machine depicted in Figure 2.6. When there is no flash crowd detected the system is in the **NORMAL** state. In this state the merger relies more on the predictor, since the high sensitivity of the detector can cause false alarms.

The predictor module detects trends in request rates and therefore detects when the request rates grows significantly. The predictor can issue two types of warnings. First, it could detect an upcoming flash crowd when still in the white zone. Second, it might detect that the current request rate has entered the grey (or even red) zone. Both occurrences cause the merger to switch to the **ALARM** state. During this state the merger recommends to activate the content delivery network.

While the merger is in the **ALARM** state it waits for the detector to confirm the flash crowd. Once a confirmation from the detector has been received, the merger switches to the **DETECTED** state. However, if no confirmation has been received within $W_e$ seconds the merger returns to the **NORMAL** state, recommending to disable the content delivery network.

While the merger is in the **DETECTED** state, it maintains the recommendation of activating the content delivery network. During this state the merger simply waits until the detector reports the flash crowd is finished. After this the merger switches back to the **NORMAL** state, recommending to disable the content delivery network.

As can be easily seen, the prediction system will also mark the beginning of bursts as flash crowds and therefore raise the alarm. Although this behavior can be easily adjusted by using the $\tau_{fc}$ parameter to distinguish between flash crowds and bursts, this is not recommended. When using this parameter, the system would wait for $\tau_{fc}$ seconds before an alarm would be raised. If a burst is detected, no alarm will be raised, unfortunately this also causes all flash crowds to be detected $\tau_{fc}$ seconds too late, since it would have to wait for confirmation. Effectively this would destroy the purpose of the prediction system. Therefore since in the early phases no practical distinction between flash crowds and bursts can be made, this prediction system will cause the content delivery network to be activated on both occurrences. Note that when the network is activated during a burst, the cost will be minimized, since after $\tau_{fc}$ seconds the flash crowd will not be confirmed and the system switches back to normal operation.

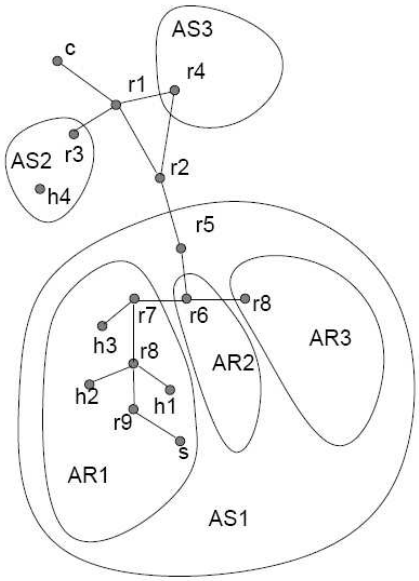



**Figure 2.7:** An example of an internetwork (adapted from [16])

## 2.5 Handling a flash crowd

Different solutions for the problem: Section 2.4 introduced a system to detect and predict flash crowds. However, once a flash crowd has been detected, appropriate action needs to be taken. Several solutions to handle large loads on a website have been introduced, most of these solutions focus on static documents and implement some kind of replication and/or caching scheme using multiple edge servers to spread the load.

When the internet started all sites were static sites, meaning the contents of a site did not change when users accessed it. The only people who could change the contents were the website administrators. This meant that usually the contents of a website remained the same for long periods of time. Therefore caching and replication works very well to reduce the load on such websites.

### 2.5.1 RaDaR

RaDaR is a system designed to allow a hosting service to handle large loads on static websites [16]. It has been designed to scale without bottlenecks and to allow different kinds of replication and request distribution algorithms. Not only that, the replication is done completely dynamically, which simplifies the administration of the system. The entire replication should be transparent to the user, meaning that the user thinks he/she is simply interacting with a single website instead of one of the replicas.

The architecture is depicted in Figure 2.7. It assumes the Internet environment, where individual destinations are organized into OSPF areas (areas for short). These areas are organized into autonomous systems, ASs. A hosting service is then assumed to be distributed globally across several ASs. Messages between two nodes of the system can travel through third-party nodes to reach their destination. As an example, if we take a hosting service constituted of AS1, AS2 and AS3, as depicted in Figure 2.7, a message traveling from a node

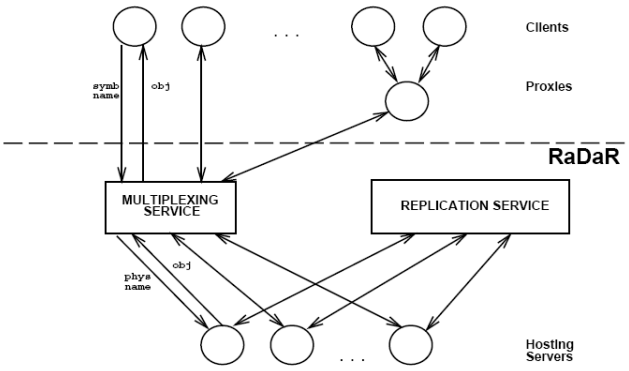



**Figure 2.8:** RaDaR high level view (adapted from [16])

in AS1 to a node in AS2 must travel through external routers r1 and r2. Finally nodes that are within an AS or AR are called internal nodes, all other nodes are called external nodes.

Requests from clients to nodes within the RaDaR system are first handled by a multiplexing service. Such a request is made to a symbolic link representing the object. The multiplexing service then decides which host containing a physical copy of the object will be used to handle the request based on the current load and client location. It then fetches the copy and returns it to the client, see Figure 2.8. Although the multiplexing service could become a bottleneck it has been organized in such a way that this is prevented.

Each host within the system decides on its own to drop, migrate or replicate objects. The target host for migration or replication is chosen in cooperation with the replication service. This service keeps track of all hosts and their loads respectively. The replication service is organized in an hierarchical fashion. Each AR has its own replicator, each AS has its own replicator as well. Finally each RaDaR system has one root replicator.

A host knows about several entities. First, it knows about all the hosts within its AR. Second it knows about all the ARs within its AS. Finally, it knows about all the other ASs within the system. When the host decides to migrate or replicate an object, it selects a suitable candidate from all the entities it knows about. If an AR or AS is chosen, the replicator of that entity decides on which host within its region to place the objects.

The decision on a suitable candidate is based on preference paths, which are periodically obtained from the systems routers. A preference path is a route the object often follows within the system to service clients. Entities that appear frequently on the preference paths of the object are good candidates, since the object often passes near these entities when traveling to the clients.

Each replicator $r$ keeps track of the following parameters. First it keeps the identity $h_{min}$ and load estimate $load_{min}$ of the least-loaded host among the hosts in its region. It also keeps track of the average load $av_{load}$ of all the hosts in its region, this is called $r$'s average load. Finally it keeps track of the average load of every child replicator $r_i$ within it's region,

When a server gets overloaded it sends an offload request to the replicator of its AR. The replicator in turn then tries to find an underloaded host within its own region. If this fails, it forwards the request to its parent replicator.

A replicator decides where to place an object when a migration, replication or offload requests arrives. It does this by first examining every $av_{load}(r_i)$ of its child replicators. If some replicator $r_j$'s average load $av_{load}(r_j)$ is below the threshold the request is forwarded to

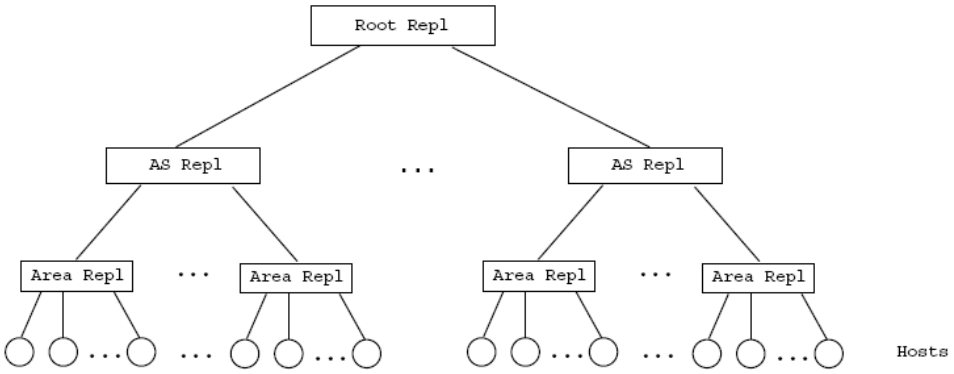



**Figure 2.9:** RaDaR replicator hierarchy (adapted from [16])

$r_j$. If no such node can be found it checks if $load_{min}$ is below the threshold, if so the request is forwarded to $h_{min}$. When that fails as well, the request is simply forwarded to its parent replicator.

The threshold has to be defined by the systems administrator and simply indicates the maximum load a host may have to become a candidate for offloading. Since the decision making on the placement is solely based on the clients and parent of a replicator, addition and removal of replicators can be done very easily.

RADAR is an excellent system for managing websites consisting of static objects. However, when faced with dynamic objects, a problem arises. Where a static website consists solely of static objects which can be moved around relatively easy, a dynamic website is usually structured in such a way that migration becomes difficult.

A typical dynamic configuration would have a web server servicing requests of clients. In order to generate a requested page the web server makes requests to an application server running on either the same machine, or a separate one. The application server in turn sends queries a database server to obtain the data it needs to service the requests made by the web server.

The web server and application server could be easily replicated or migrated using the RADAR system. However replicating or migrating an entire database could be very troublesome. RADAR is based on the assumption that the cost of migration or replication of an object is relatively low to the amount of requests being directed to it. However, since a typical database can easily become very large, the cost of transferring a database to another host will be too high compared to the performance gained by placing a database at the new location.

Another problem is database consistency. When a database is replicated across several servers, measures have to be taken to ensure all copies of the database are consistent. RaDaR does not provide any mechanisms for database consistency. Therefore replication of a database would quickly lead to inconsistency throughout the system.

Using RADAR we could reduce the load on the web server and application server by replicating it to different hosts. However, since replicating the database server is infeasible, the load on the database server would remain equally high, or it would increase even further since it gets queried by multiple application servers at once. Therefore the database server would rapidly become the bottleneck of the system.

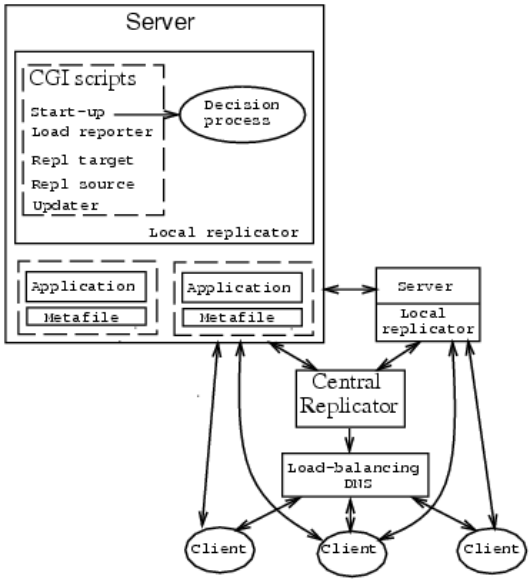



**Figure 2.10:** ACDN prototype architecture (adapted from [17])

### 2.5.2 ACDN

ACDN is a system designed for Edge Computing [17]. It allows for dynamic replication, migration and deletion of applications to the edge servers of a content delivery network. These actions will be performed asynchronously to the handling of incoming requests. By not allowing an application to replicate or migrate while handling a request, the amount of state that needs to be transferred is reduced to a minimum.

Further, incoming requests will only be redirected to servers already running the required applications. Since starting an application is a costly procedure, starting the application when a request comes in would take to much time. If an application is migrated or replicated to a server, requests for that application will not be redirected to the new server until that application is fully started.

The general architecture is depicted in Figure 2.10. ACDN has been designed to rely completely on the HTTP protocol and web servers. Therefore each ACDN server is a standard web server. However, each server also contains a replicator, which implements the ACDN specific functionality. The replicators are implemented as CGI scripts, which can easily be added to any standard web server. There is also a global replicator which mainly keeps track of the locations of all the replicas within the system. Although in theory the global replicator could become a bottleneck, it requires so little computation that in practice it can even be co-located with the DNS server. It is not needed for the actual handling of requests, so even if the central replicator were to fail, the system would still work. The failure would only stop the migration and replication process, until the central replicator is restarted.

The replicator on a ACDN server consists of the following CGI scripts: the startup-script, the load reporter, the replica target script, the replica source script and the updater script. The start-up script should be invoked once a new ACDN server is added to the system. This scripts forks a decision process which periodically decides if any of the applications on the server needs to be replicated or deleted. The load reporter is periodically invoked by the

central replicator and returns the current load of the server. The other scripts are used to implement the application distribution framework and will be discussed below.

An ACDN application consists of two parts, the first part is the application itself. The second part is the meta file. This meta file is a file describing the application. It consists of two parts: the list of all the files of the application including their last modified dates; and the initialization script that a server has to run first before accepting any requests to the application. The meta file is a simple XML formatted file, which can be fetched using a standard HTTP request.

The replica target and source script are located within the replicator of an ACDN server. The decision process decides if a replica needs to be replicated from the current server (source server). In order to create a replica of an application the source server first needs to find a suitable target server. In case of an overload the source server queries the central replicator to find the least loaded server in the system. When the reason is improving proximity of demand, the server determines the target server by itself, based on its replica usage.

Once the target server has been identified, the source server invokes the replica target script on the target server passing the URL to the application's meta file as a parameter. The target server in turn invokes the replica source script on the source server providing the same meta file URL. The source server responds by sending a tarrball containing all the files of the application. Once received, the target server unpacks the tarr file and runs the initialization script found in the meta file. Once the initialization script has been run, the replica target script informs the central replicator of the new replica. The central replicator in turn updates the DNS such that requests will also be redirected to the new replica.

The decision process also initiates the deletion of a replica. In order to delete a replica the server first sends the central replica a request for permission to delete the replica. If that replica is not the latest replica of the application, the central replicator sends an update to the DNS server. The DNS server recomputes its request distribution, such that requests will not be directed to that replica anymore.

Once the new policy has been confirmed by the DNS server, the central replicator sends a permission to delete the replica to the ACDN server. The permission contains the DNS time-to-live (TTL) associated with the domain name of the application. The ACDN server marks the replica as "to be deleted" and finally deletes the replica after the TTL provided by the central replicator.

The consistency of replicas is maintained using the meta file. Once the primary replica is updated at the origin server, its meta file is updated as well to reflect the changes. The other replicas can use standard cache invalidation techniques to check if the current meta file is still up to date. If it is outdated, the server downloads the new meta file and copies all the modified objects together as described in the meta file.

ACDN is a nice system we can use to replicate applications. However, there is one serious drawback that makes this system unsuitable in its current form to deal with flash crowds. The system does not provide any mechanisms to maintain consistency for changes made by normal users. Only changes made by the administrator are forwarded to all the replicas. It assumes that if users need to update information a shared data source, like a database, is used by all replicas. Again that shared data source would quickly become the bottleneck of the system. However, if we were to find a mechanism to remove this bottleneck, ACDN could prove to be an elegant solution for the replication of the application part of the system.

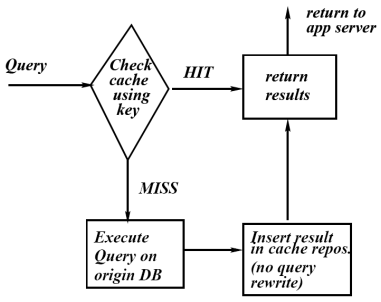



**Figure 2.11:** Content-blind Cache (adapted from [20])

## 2.6 Content-blind query caching

Content-blind query caching is a caching mechanism that stores query results, without knowing what is inside the result. Figure 2.11 shows how this caching mechanism works. Once a query comes in from the application server, the cache first checks to see if the corresponding result is already present in the cache repository. This check is done using some kind of key to identify the different query results. If the result is present, it is immediately returned and no query to the database needs to be made. If the result is not present, the query is forwarded to the database server. The database server executes the query and returns the result to the cache. The result is added to the cache repository, after which it is returned to the application server for further processing. An important property to conten-blind query caching is that the cache does no processing on the query result, it simply stores the result structure for later use.

### 2.6.1 GlobeCBC

GlobeCBC is a content-blind query result caching system designed for use in web applications [20]. It is designed as a middleware situated between the application server and the database server. All communication between the application server and the database server takes place using the middleware.

Since globeCBC uses content-blind query caching some identification mechanism is needed to distinguish the different results. Not only do we need to determine whether or not a certain query result is already present, we also need to invalidate results once an update has been made. GlobeCBC uses query templates for this. These query templates are defined by the system designer.

The system designer determines which SELECT queries are used by the application server. For each SELECT query a separate query template is defined, where each template receives an unique template-id. After all query templates have been defined, the system designer assigns each update query to an update query template and determines which query templates are affected by the update. Once an update is executed, all corresponding query templates are invalidated.

The globeCBC System Architecture consists of two key components, the caching module and the invalidator (see Figure 2.12). The caching module is the middleware which runs on each edge server and intercepts all calls made to the database server. The invalidator runs on the origin server and monitors all update queries.

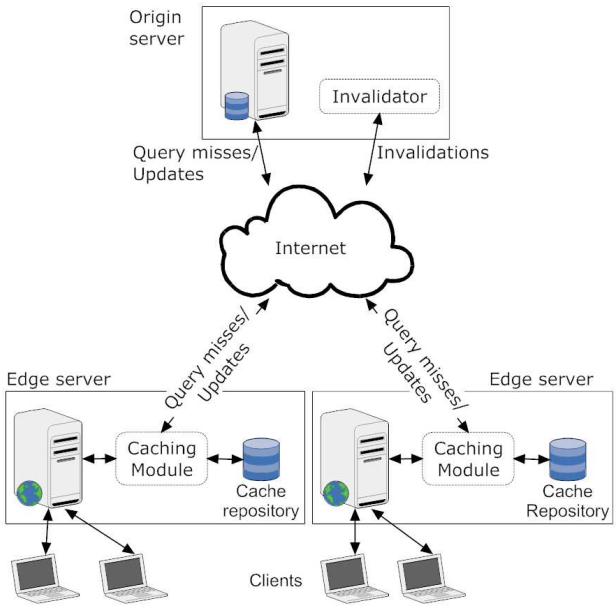



**Figure 2.12:** GlobeCBC Architecture

When a SELECT query arrives at the caching module, it first determines the query template id and the query parameters. These values uniquely identify a query result. Using these values the cache repository is searched for the corresponding result. If it is found, it is returned to the application server. If it is not found, the query is executed at the database server. After the database server returns the result, globeCBC stores the query result in the repository, using the template id and the query parameters as the identifier of the result. Note that globeCBC does not use a database management system to store the results. Instead, each result is stored either as a file on the local file system, or in main memory to improve performance. The caching module then subscribes at the invalidator to the invalidation channels corresponding to the cached template-id. By using a publish-subscribe mechanism, each caching module only needs to be informed of changes that can possibly affect its stored results.

When the caching module receives an update query, it is directly executed at the database. However, the caching module also sends a message tot the invalidator with the template identifier of the update query. The invalidator in turn informs all caching modules affected of the update, using the publish-subscribe channels. All invalidations are sent asynchronously, thus globeCBC does not provide strong consistency. However, if a certain template can not tolerate any staleness, caching of that template can be disabled by the system designer.

As explained earlier, the system designer determines which update query templates affect which query templates. This is translated into a conflict map for all the update query templates, which is stored at the invalidator module. The invalidator maintains a publish-subscribe channel for each update query template in the map. The edge servers can subscribe to the channels they need. Once an update query is performed at the database, the invalidator is informed by the edge server's caching module of the update. Upon receiving the message, the invalidator sends the update query key to all the subscribers on the channel corresponding to the write identifier. Upon receiving such an update query key, the caching

module invalidates all results with a query template that conflicts with the write query.

Since not all updates affect all SELECT results, using template based invalidation helps reducing the amount of invalidations. For example, if we have a bookstore application, the cached result of a query to find the best selling books need not be affected by an update query changing the price of a book. However, if a book is purchased, the cached result should be invalidated.

Sometimes using templates alone results in too conservative invalidations, if all query templates matching an update query template are invalidated. Consider for example the following template: QT1: "SELECT price, stock, details from book where id=?" and update query template: UT1: "UPDATE price=price+1" from book where id=?". Using the normal template based invalidation scheme, making an update to a single book (e.g. $\langle UT1, 100\rangle$) will invalidate all cached results of QT1.

To improve this behavior, globeCBC needs to take into account the parameter of the updated item and invalidate only those cached queries that are affected by the updated item. However, since globeCBC uses content-blind caching, this approach can only work for simple queries that rely on the primary key for their selection. So in the above example, when the system receives an update query with key: $\langle UT1, 100\rangle$, the invalidator will only invalidate the cached item whose key is $\langle QT1, 100\rangle$.

If an application has a large number of read/write conflicts, even the addition of the extra identifier could prove insufficient to allow for good scalability. Therefore, globeCBC allows for tunable consistency. The system designer can tune two parameters, $TTI_i$ and $Max\_Upds_i$, for each update template $U_i$ in order to fine tune the system for scalability. $TTI_i$ is a time-based mechanism to delay the invalidation messages sent by the invalidator. For each update template $U_i$ the system designer can define $TTI_i$. Upon receiving an update $U_i$, the invalidator starts a timer for $TTI_i$ seconds (unless it has already been started). After the timer expires, all queued invalidation messages for conflicting read templates are sent.

$Max\_Upds_i$ works in a similar fashion. However, this parameter describes the maximum number of updates the invalidator can tolerate from update template $U_i$. For each $U_i$ the invalidator keeps track of the amount of updates executed using a counter ($Num\_Upds_i$). When $Num\_Upds_i > Max\_upds_i$, the invalidator sends out the invalidation messages and resets $Num\_Upds_i$ to 0.

Using these parameters the system can capture multiple updates with one invalidation, this reducing the amount of invalidations and increasing the performance of globeCBC. However, this performance increase comes at the cost of reducing the overall consistency of the system.

While globeCBC is an excellent system for reducing the load on a database, during a flash crowd it will run into problems. With the enormous amount of users during a flash crowd, the caching modules will also receive a large amount of update queries. These queries will cause a large number of invalidations to be sent to all the caches. All these invalidations severely cripple globeCBC.

Unfortunately the $Max\_Upds_i$ and $TTI_i$ parameters will not help a lot, since first of all these values are typically chosen as small as possible. They should be chosen as small as possible, because all invalidations will be delayed. Thus if a user performs an update, this update will not be visible until either $Max\_Upds_i$ or $TTI_i$ has been reached. Thus the system does not even provide read your writes consistency when the tunable consistency parameters are used.

### 2.6.2 DotSlash Qcache

DotSlash Qcache is a content-blind query result caching system that is part of the DotSlash automated web hotspot rescue system [27]. The DotSlash system can be seen as a community of cooperating servers [25], [26]. Each server can remain in three mutually exclusive states: *SOS state* if it gets rescue services from other, *rescue state* if it provides rescue services to others, and in *normal state* if it does neither.

If a server gets overloaded it tries to find rescue servers. Once suitable servers are found, the server switches to *SOS state* and the rescue servers to *rescue state*. The rescue servers act as reverse caches and cache all requested static objects locally. They also replicate applications dynamically and run them locally. If the application requires database access, the database at the origin server is used. Once the load returns to normal, the rescue servers are released and the origin server switches back to *normal state*.

As already explained, the database at the origin server quickly becomes a bottleneck. Therefore, DotSlash was expanded with the Qcache query result caching system. Qcache works in a similar fashion as globeCBC. However, instead of using query templates for identification, Qcache simply uses the entire query string as an identifier for a stored result. Along with the identifier, Qcache also stores the Time-To-Live ($TTL$) value with the query result. This $TTL$ is used to invalidate the query result after a certain time.

Qcache does not allow edge servers to perform update queries. Only the origin server is allowed to perform write queries. The main reason for this is security, because it is unlikely that a DotSlash rescue server would be allowed to perform updates on the origin server. Thus all requests involving update queries need to be directed directly toward the origin server.

When experiencing heavy load the caching system has two levels of data consistency. The first consistency level is for site administrators. They can always perform read and update queries and will always be presented with an up-to-date view of the system. The second level of consistency is meant for regular users. Regular users can only perform read queries, and will be shown a delayed view of the system by means of the query result caching. The $TTL$ parameter is used to invalidate cached results after a specific time, to allow updates to the database to be visible to the users.

Using Qcache the system will probably not receive many updates at all. The Qcache system does not allow regular users to execute update queries, only site administrators. However, the chance that a site administrator will start making changes in the database when he knows the website is heavily loaded are slim. Therefore, the database will most likely stay the same at all time, which is very good for the performance of the cache. As seen in the previous section, the main problem for a viable query caching solution are the invalidations.

Obviously the Qcache system performs very well, it can handle much higher loads than a system without caching would. Unfortunately, this performance comes at a high price. Because the system does not allow normal users to perform updates, the use of a dynamic website basically disappears. For example, if we have a news site like slashdot, the whole idea is that users are involved in the stories by allowing them to comment on them. This would not be possible using Qcache.

The situation is even worse for online businesses. When an e-commerce website is hit by a flash crowd, it receives a massive amount of possible customers. If these customers were faced with a message explaining to them they can not buy the product they came for at this time, they would most likely try to find the product at another website. Thus a lot of possible revenue is lost.

Because of its limitations, the Qcache system is not a viable solution for dynamic websites to handle a flash crowd. Although the performance of the Qcache system is very good, the cost of this performance is too high to be used in any dynamic website which requires user interaction. Qcache can only be used for dynamic sites that only use the database to store the content of the website, without users actively interacting with the system.

## 2.7 Discussion

The discussed research efforts could aid us dealing with flash crowds. The flash crowd prediction system allows us to take necessary precautions. Several systems designed to handle large loads have been proposed. However, these systems are unsuited in their current form to be used by dynamic websites to handle the load found during a flash crowd. The main problem is the bottleneck introduced by the database. GlobeCBC can be used to reduce that bottleneck. However, during a flash crowd globeCBC will be severely crippled by the amount of invalidations coming from all the edge servers. Qcache tried to solve this problem by disallowing clients to perform updates. Although performance increases greatly, it is at too high a cost, since users will be faced with an incomplete system and will probably not return very soon.

Fortunately, globeCBC can be extended to be able to handle flash crowd circumstances. Using the high locality found during a flash crowd, I propose flashCBC, a globeCBC adaptation designed to handle flash crowds. The following chapters will introduce flashCBC.

# Chapter 3

# Flash Crowd Locality

Using a caching mechanism to reduce the load on a server is only truly beneficial if the locality of the incoming requests is sufficiently high. That is, if requests come in for random documents or database queries, the use of a cache is limited. In such a situation a replication or partitioning of data might prove better results. However, during a flash crowd, the locality properties of the requests seem in favor of using some form of caching to reduce that load [14].

This chapter will provide a closer look into the locality properties found during a flash crowd. This research has been performed using a caching perspective. I analyzed several real world access logs, measuring how a cache added to the system would have performed given the request distribution found in each access log. The first part of this chapter will describe how to simulate a cache and the measurement metric used. This will be followed by a discussion of the access logs and the results found simulating each log.

## 3.1 Simulation setup

In order to study the locality properties of website access logs we can choose among different approaches. One of the approaches could be to count the requests for each object and analyze the resulting request distribution. However, we are interested in ways to reduce the load on a database server, therefore we can also analyze the logs using a virtual cache.

A virtual cache simulates how a real cache would have behaved based on the access pattern found in the access logs. However, instead of actually storing the objects requested, it only maintains a list of the identifiers of the objects. For each request in the access log the virtual cache determines if the identifier of the object requested is present in the list. If the identifier is present, a cache hit is produced, if not a cache miss. For every request, the virtual cache outputs a $(time, cache\ hit)$ pair, where *cache hit* is a value of 1 for a hit, or 0 for a miss.

Since each $(time, cache\ hit)$ pair by itself is not really meaningful, post processing is required. First of all the pairs need to be aggregated into windows such that we can calculate averages. Thus for each window the average cache hit rate can be calculated, providing us with an exact measure of how the cache hit ratio changes over time.

However, since we are interested in the effect of a cache on the database server load it is better to show how the cache miss ratio changes over time. Because each cache miss means a hit on the server, this value demonstrates more clearly the effect of adding a cache to the system. For example, if the cache miss ratio changes from 20% to 15%, it means that the database receives 25% less requests than before.

When we look at the averages found in each window, we see that these value fluctuate quite a bit. Therefore, we need to stabilize the values in order to perform a meaningful analysis of the access logs. To stabilize the averages, it is best to use an exponentially weighted moving average. Using such a moving average, the fluctuations found between windows are reduced by the influence of the previous averages. Therefore, we get a stabilized view of the cache miss ratio over time.

In order to calculate the exponentially weighted moving average of the cache miss ratio we first aggregate the $(time, cache\ hit)$ pairs into windows of $N$ requests. In every window $w_i$ we keep track of $cacheMisses$ which simply counts the number of cache misses within that window. The cache miss ratio $cmr_{w_i}$ for each window $w_i$ is calculated by $cmr_{w_i} = 100 * cacheMisses/N$.

The exponentially weighted moving average of the cache miss ratio $avg_{w_i}$ for each window $w_i$ is then calculated as $avg_{w_i} = \alpha * avg_{(w_{i-1})} + (1 - \alpha) * cmr_{w_i}$. For the first window $w_0$ we simply take $avg_{w_i} = cmr_{w_i}$. The $\alpha \in [0, 1]$ value is used to control how much $cmr_{w_i}$ influences the weighed average $avg_{w_i}$. As can be easily seen from the formula, as $\alpha$ increases, the influence of $cmr_{w_i}$ decreases.

As the cache replacement algorithm we used a simple Least Recently Used (LRU) caching policy [22]. This algorithm was was chosen for its simplicity. if we gain a significant performance increase using this simple algorithm, we could probably gain even more using a more sophisticated algorithm. With LRU caching we simply replace the cached item which has been accessed the furthest in the past. Thus recently accessed objects will remain in the cache, while objects that have not been accessed recently will be removed when a new object is added to the cache.

Using the averages, we can show how the cache miss ratio is influenced by the amount of requests arriving at the server. As can be seen below, all access logs show that as traffic increases to flash crowd levels, the cache miss ratio decreases accordingly.

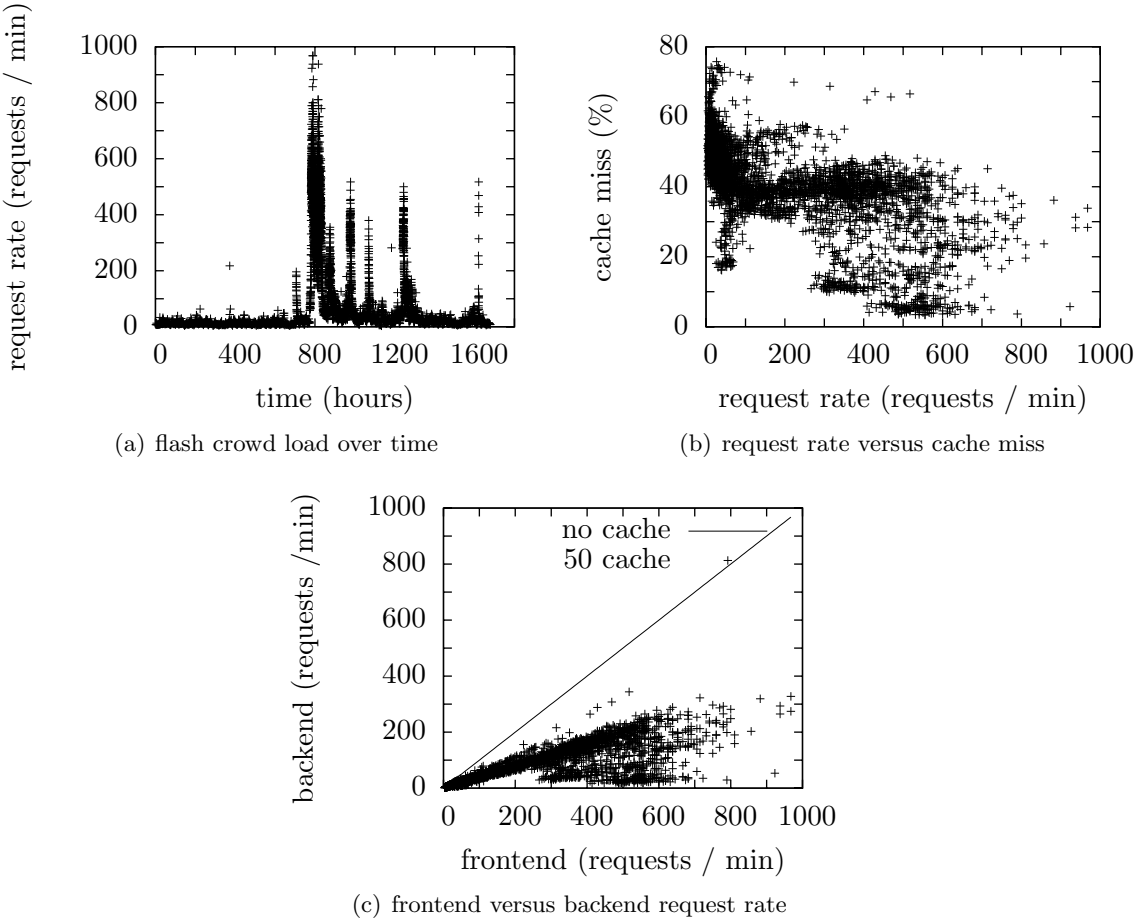
## 3.2 The access logs

This section will introduce the flash crowd access logs used for my analysis. This analysis has been done on access logs taken from websites consisting of static pages. Therefore, at first glance these results need not apply to dynamic websites as well. Fortunately, during a flash crowd, people are usually drawn to a particular story or item on the website. It is therefore safe to assume that we will see a similar increase in locality when addressing dynamic websites.

### 3.2.1 Tanenbaum access log

The flash crowd depicted in Figure 3.1(a) was caused by the public statement Andrew S. Tanenbaum placed on his website concerning a book from the Alexis de Tocqueville Institution. In this book the professor was severely misquoted, so he responded by a public statement on his views on the entire situation. Several prominent news websites placed stories on this, with a link to the statement and thus the server was quickly swamped in requests.

We can see this is a classical flash crowd example. First we see the normal usage of the website, which remains stable over a longer period. Then, after the article is placed, we see a dramatic increase in request rate. This increase lasts several days after which the request rate begins to return to normal levels. However, after the first peak, several smaller flash crowds can be identified. These could be caused by the follow up articles placed by Andrew




(a) flash crowd load over time

(b) request rate versus cache miss

(c) frontend versus backend request rate

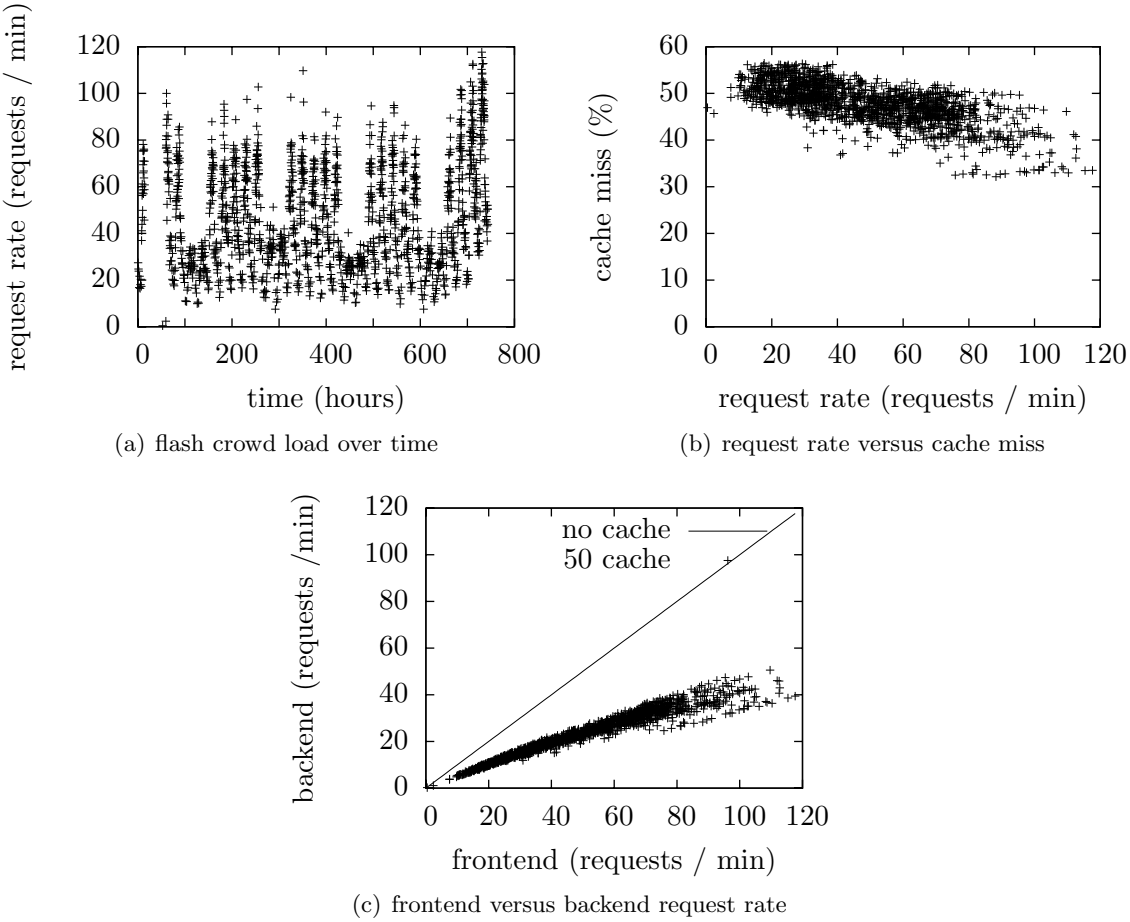
**Figure 3.1:** Tanenbaum virtual cache analysis

Tanenbaum, another reason could be that smaller news sites picked up on the article and created the smaller flash crowds at a later time.

In order to simulate the behavior of a cache for this access log we used a virtual cache size of 50 objects and an $\alpha$ value of 0.95. When we look at the cache miss ratio of the virtual cache, we see a decrease of the cache miss ratio as the load on the server increases. In Figure 3.1(b) we can see the correlation between the server load, expressed as requests per minute, and the cache miss ratio, in percent. As can be clearly seen, the average cache miss ratio decreases significantly as load increases. Where the cache miss ratio ranges from around 35% to 75% during normal load, the cache miss ratio decreases dramatically during higher loads, sometimes the cache miss ratio even drops below 5%.

If we translate the cache miss ratio into actual requests being forwarded to the origin server. We can clearly see the benefits caching can provide when faced with a website. In Figure 3.1(c) we can see the effect of using a cache on the requests being made at the origin server. On the horizontal axis we show the request rate on the front-end, e.g. the request rate generated by all the clients accessing the website. The vertical axis shows the requests that were forwarded to the back-end, e.g. the requested document was not found in the cache, thus needed to be fetched from the origin server.

The straight line running from the bottom left to the upper right depicts the situation




(a) flash crowd load over time

(b) request rate versus cache miss

(c) frontend versus backend request rate

**Figure 3.2:** NASA virtual cache analysis

without caching. Clearly, in that case, every request received is forwarded to the origin server. This is the least favorable solution. Beneath the line we see the situation as we add a cache. Clearly the cache unburdens the origin server significantly. Without caching the load on the back-end linearly increases with the request rate at the front-end. When we use a simple cache of as little as 50 objects, however, we see that as the load on the front-end increases, the load on the back-end increases much less. The load on the back-end does not even exceed 450 requests per minute, while the load on the front-end almost reaches 1000 requests per minute.

### 3.2.2 NASA access log

The flash crowd depicted in Figure 3.2(a) was taken from the NASA website [15]. This log shows the visits to the site in August 1995. In this month a mission of the Space Shuttle Endeavour was planned. However, it ran into some problems causing the flight to be delayed until eventually, on the 7th of September, the Shuttle took of on its mission.

An interesting part is at the beginning of the graph, where we see a period where no requests to the server have been recorded. A possible explanation for this absence of requests could be that the server was down in that period.

During the entire month of August the website got hit by multiple flash crowds. Each flash crowd can be easily matched with the publication of a new status report on the mission [18]. After each report the site got hit with a new flash crowd. Every flash crowd is about the same size. However, as the final launch date approaches, we see an increase in the magnitude of the flash crowds. The final three flash crowds are clearly much larger than the others, indicating that more people got interested as the launch date drew nearer.

This access log was simulated using a virtual cache size of 50 objects and an $\alpha$ value of 0.90. As we take a look at the correlation between request rate and cache miss ratio of the NASA flash crowds, we see less of a difference between lower loads and higher loads. However, we can still see a decrease of over 10% between the lower request rates and the higher request rates.

Again, when we compare the load observed at the front-end to the load observed at the back-end we see a clear improvement when we use a cache. As the request rate at the front-end increases and almost reaches 120 requests/minute, the load on the back-end increases at a much lower rate. The observed load at the back-end does not even exceed 60 requests/minute, all with just the addition of a cache with 50 objects in it.

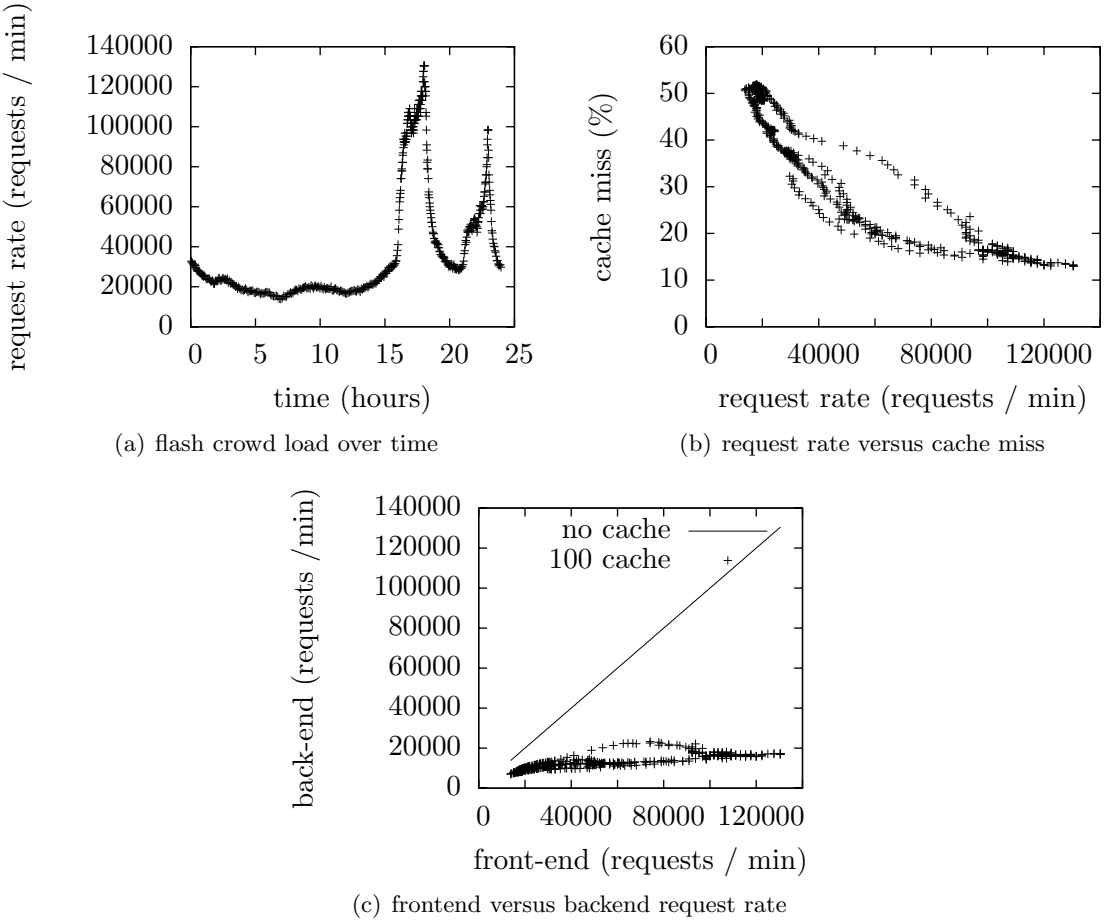
### 3.2.3 Worldcup 1998 access log

The flash crowd shown in Figure 3.3(a) was taken from the 1998 FIFA World Cup. This worldwide soccer event took several weeks, during which the website of the event got hit by a flash crowd on a regular basis. The access logs of multiple days of the event showed a similar access pattern. Therefore it is only necessary to discuss one day of the event.

The website has fairly stable access patterns throughout the day. However once the first match begins at 16:30 CEST the site gets hit by the first flash crowd. After the match is finished traffic returns to normal, until the second match of the day begins at 21:00 after which a second smaller flash crowd can be identified. As explained in Section 2.1 it is unclear as to why the second peak is smaller than the first. However, a possible explanation could be that during the second match more people had time to watch the match then during the first. The first match was at the end of the afternoon, therefore a lot of people would still be working, while during the second match viewers could simply watch the match on television from home.

This access log was simulated using a virtual cache size of 100 objects and an $\alpha$ value of 0.95. Again we can see a clear correlation between the increase in load and the decrease in cache miss ratio. As we see in Figure 3.3(b) the load on the server is very high all the time, ranging between 10000 and 20000 requests per minute. However, as the load increases to even higher values we see the cache miss ratio drop from around 50% at the normal load levels, to below 15% at the peak of the flash crowd (more than 130000 requests per minute).

When we compare the load observed at the front-end to the load at the back-end we see a huge decrease of the back-end by the addition of the cache. With the addition of the cache, the load on the origin server hardly increases as the load on the front-end increases. Clearly the locality of this flash crowd is very high and by far the most people who visit the site request the same documents. Using a cache of just 100 objects we could unburden the origin server significantly, allowing it to run as if no flash crowd is happening, while the caches handle most of the requests originating from the clients.




(a) flash crowd load over time

(b) request rate versus cache miss

(c) frontend versus backend request rate

**Figure 3.3:** Worldcup 1998 virtual cache analysis

## 3.3 Discussion

With each example we see a decrease in cache miss ratio as the request rate increases. This means that more users request the same documents and locality increases. This confirms the research discussed in Section 2.2. Using this observation one can see that caching might prove a viable solution for dealing with flash crowds. Using caching techniques one can clearly unburden the origin server and spread the load to edge servers.

One of the main problems we run into with a caching solution for database queries are the invalidations we receive after a user has done an update. For example, once a user places a comment, or purchases an item, the caches need to be informed about that change. Especially during a flash crowd the invalidations can cripple the caches. Therefore, I will introduce flashCBC in Chapter 4. FlashCBC is a caching mechanism designed with flash crowds in mind, aimed at solving the invalidation problem found in dynamic websites.

# Chapter 4

# FlashCBC

As seen in the previous chapters, caching could provide a viable solution for dealing with flash crowds. However, the large number of invalidations can severely reduce the effectiveness of caches. Therefore this chapter introduces flashCBC. FlashCBC is a content-blind query result caching system based on globeCBC [20].

FlashCBC has been designed to reduce the negative effect of invalidations on the caches, such that the cache miss rate remains as low as possible while maintaining a consistent view of the system for all the users. This has been achieved by adapting the consistency scheme used by globeCBC to a more suitable scheme. GlobeCBC uses a system wide consistency model, where delays in invalidations will be visible to all users. FlashCBC, instead, provides a session-based level of consistency where each user is presented with Read Your Writes consistency [23].

Read Your Writes consistency guarantees that the effects of every update made within a session are visible to reads within that session. This means that if a user, for example, places a comment, that comment will be immediately visible to the user. However, that comment need not be visible to all other users immediately. Thus, a user will always be presented with his or her own updates, while updates made by others can be propagated at a later time.

Because of the central invalidator module used by globeCBC, all invalidations take place asynchronously to the updates. This means that globeCBC does not provide Read Your Writes consistency. From a user perspective this means that it is possible for a user to execute an update and not be presented with the effects of that update immediately. Even in its strongest form, where all invalidations are forwarded immediately to the caches, it is possible users will see a delay in the effects of their update. In practice this delay is within acceptable bounds, such that most of the time users will not notice this delay.

However, when faced with flash crowd request levels, the performance of globeCBC is severely crippled by all the invalidations. Because of all the invalidations, the results stored in the cache need to be refreshed constantly, severely decreasing the hit rate of the cache. We can reduce the invalidations using the $TTI$ and $Max\_Upds_i$ parameters explained in Section 2.6.1. Unfortunately, these parameters affect the consistency of the entire system. Therefore, all invalidations will be delayed, and users will be presented with a system where every update will only be visible after several tens of seconds or even minutes. Such delays are unacceptable if we want to provide a good user experience.

Table 4.1 compares the possible approaches. GlobeCBC in its "strong" form, where all invalidations are forwarded immediately, provides reasonable consistency. However, its

| | performance | consistency |
|---|---|---|
| GlobeCBC "strong" | low | pretty good |
| GlobeCBC "weak" | high | unacceptable |
| FlashCBC | high | good |

**Table 4.1:** comparison between globeCBC and flashCBC

performance is too low to be able to handle flash crowd loads. When we use globeCBC in its "weak" form, where we delay the invalidations, the performance increases. However, the consistency drops to unacceptable levels and users will not be satisfied with the system performance. The goal of flashCBC is to solve both issues.

In flashCBC the central invalidator module has been removed. Instead, each edge server has its own invalidator module. This invalidator module keeps track of the updates performed at its own edge server. When an update is performed, all corresponding query results are invalidated immediately in the local cache. This guarantees Read Your Writes consistency. However, the invalidator module does not forward the invalidations to the other edge servers immediately. Instead it delays the invalidations such that multiple updates can be caught with a single invalidation.

Although the invalidations are delayed, these delays often will not significantly degrade the user experience of the website. For example, in a bulletin board application, if a comment made by another user shows up a little bit later, it is not really a big problem. Even under normal usage things like that happen when multiple users comment at the same time. Also if the stock indication of an item in a webshop is a little bit out of date it does not really matter, as long as the order arrives in a timely fashion.

Note that Read Your Writes consistency may not be acceptable to all applications. For example an auction site like ebay requires that all bids performed by users are immediately visible to all the other users. Ready Your Writes can not guarantee this, thus will not be suitable for these kind of applications. We however assume that such applications are rare.

By providing read your writes consistency and delaying invalidations sent to other caches, we can severely improve the performance of the caches, while providing a consistent view of the system to each user. However, this functionality depends strongly on clients using the same edge server on each access within a session. For example, if a client accesses a website from his home computer and is redirected to edge server $x$, the client should be redirected to edge server $x$ for all following requests to the website. This is a task for the redirection mechanism used by the content delivery system and outside of the scope of this research.

This chapter describes the flashCBC system. First we will discuss the system architecture, which introduces the key components and functionality of flashCBC. This is followed by an in depth look into the flashCBC modules.

## 4.1 System architecture

FlashCBC, like globeCBC, is designed as a middleware situated between the application and the database. Thus all communication between an application server and the database server takes place using the middleware.

FlashCBC uses the same kind of content-blind query caching as globeCBC does. Once a query comes in, the cache first checks to see if the corresponding result is already present in

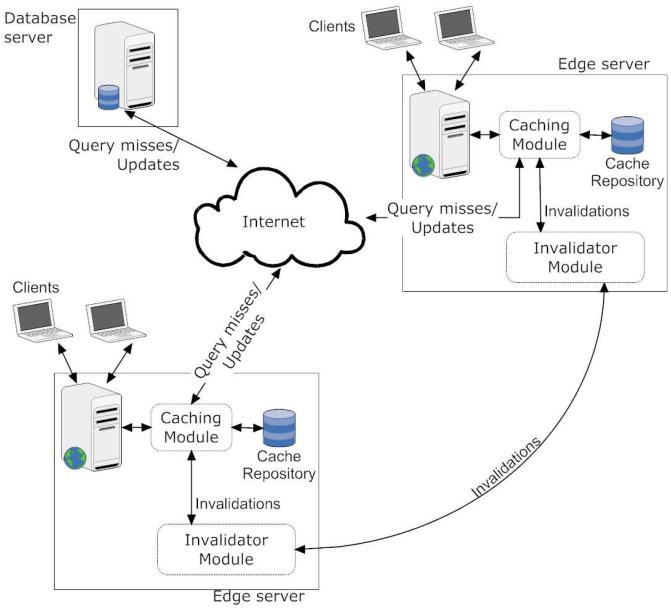



**Figure 4.1:** FlashCBC architecture

the cache repository. If the query result is found, it is immediately returned to the application server. If not, the cache queries the database for the result and adds it to the cache.

In order to determine whether a query result matches a query, the same query template mechanism is used as in globeCBC. The system designer first determines which SELECT queries are used by the application server. For each SELECT query a separate query template is defined with a unique template-id. After all query templates have been defined, the system designer assigns each update query to an update query template and determines which query templates are affected by the update. When an update is executed, all corresponding query templates are invalidated.

The flashCBC system architecture consists of two key components, the caching module and the invalidator module. The caching module is the middleware that runs on each edge server and intercepts all calls made to the database server. The invalidator module monitors all updates. However, unlike globeCBC, the invalidator module is not located at the database server. Instead each edge server contains its own invalidator module. This module immediately invalidates query templates in the local cache corresponding to an incoming update. After the local invalidation, the other edge servers are informed of the update.

Having a separate invalidator module on each edge server makes the management of the edge servers a bit more complicated, since all edge servers need to know of each other at all times. However, removing the invalidator from the origin server serves a dual purpose. First of all, the database server no longer needs to keep track of the invalidations, allowing it to focus on its main task: serving incoming queries. The second reason is that the invalidator module will not be a single point of failure anymore, thus all caches will always continue to function as long as the database server is running. If one of the invalidation modules fails, only the edge server it belongs to will not be able to use its cache, a simple restart of its invalidator module will restore full functionality of flashCBC on that edge server again.

```
loop
  wait for incoming message
  if message_type is SELECT then
    if result_query not in cache then
      execute query on database
      add result_query to cache
      set TOR_query to current_time
    end if
    return result_query
  end if
  if message_type is UPDATE then
    execute query on database
    send invalidation to local invalidator
  end if
  if message_type is INVALIDATION then
    for all cached_results do
      if cached_result_i conflicts with invalidation_key and TOR_i < TOU_invalidation then
        invalidate cached_result_i
      end if
    end for
  end if
end loop
```

**Figure 4.2:** flashCBC caching module functionality

### 4.1.1 Caching module

The caching module is the middleware which runs on each edge server and intercepts all calls made to the database. Figure 4.2 shows the functionality of the cache module. Once a SELECT query arrives, the cache first checks to see if the corresponding result is already present in the cache repository. If it is present, the result is immediately returned. If not, the query is forwarded to the database server. The database server executes the query and returns the result. This result in turn is added to the cache repository, after which it is returned to the application. Along with the query result the caching module also stores a Time-Of-Request ($TOR_i$) parameter.

The $TOR_i$ parameter denotes when the result has been requested and can be used when an invalidation arrives to determine if the stored result is already up-to-date. Since invalidations sent from remote edge servers will always be delayed, we can prevent unnecessary invalidations if we know a cached item has been requested after the latest update. The invalidation message contains a parameter denoting when the latest update has been executed. If a cached item has been requested after that time, we can safely assume the cached item is up-to-date for now. However, if the cached item has been requested before the latest update, we know the item is outdated and should be invalidated.

Queries are matched to query results using the query template id and the identifier of the item requested. If we have for example the query template QT1: "SELECT price, stock, details from book where id=?" and we request the information of the book with $id = 100$, the cached item can be uniquely identified by the two parameters $\langle QT1, 100 \rangle$.

The cache repository can be implemented using the local file system as storage, or the main memory of the server, or a combination of both. The usage of one over the other mainly depends on the resources available on the server. Using only the main memory for storage will naturally provide the fastest response from the cache and is therefore recommended. However, if the server has low memory, or memory usage is restricted, using the file system will prove beneficial as well.

When an update query arrives, it is executed directly at the database server. Along with the update, the caching module informs its local invalidator module of the update by sending the $\langle UT_i, ID \rangle$ pair associated with the update. The invalidator immediately returns an invalidation message to its local caching module, informing it to invalidate all conflicting query results. After the local invalidations, the invalidator informs the other edge servers of the update in due time.

An invalidation message is constructed by an invalidator module. The message always consists of the following members: The update query template identifier ($UT$) and Time-Of-Update ($TOU$) which denotes the last time this update was executed. Along with the two mandatory members a third optional member can be added: The item identifier associated with the update ($I\_ID$), which is used for fine grained invalidations where possible.

As is the case with globeCBC, stored results are matched using their template id and item identifier. However, in addition to matching the query templates and item identifiers, two extra parameters are used to determine if a result needs to be invalidated. A result only needs to be invalidated if its $TOR_i$ is smaller than the $TOU$ parameter of the invalidation message. Using these two parameters, we prevent unnecessary invalidations. If a SELECT query has been executed after an update has been executed from a remote server (e.g. $TOR_i > TOU$, the current result is already up to date, so it does not need to be invalidated.

As an example, suppose we have the following update query template $UT1$: "UPDATE price=price+1 from book where id=?". This update query template conflicts with the query templates $Q1$, which is defined as before, and $Q2$: "SELECT price, details FROM book WHERE price > ?". The caching module receives an invalidation message with the following parameters set: $UT = UT1$, $TOU = time$ and $I\_ID = 100$. The caching module will traverse its cached items and invalidate every stored result that conflicts with these parameters.

Suppose we have an item in cache with the following identifier: $\langle QT1, 20 \rangle$, which contains the information of the book with $id = 20$. This item conflicts with $UT1$, thus the caching module examines the item identifier, this clearly does not match 100, thus the result is not invalidated.

Another item in the cache is identified by: $\langle QT2, 10 \rangle$, which contains all the books with a price larger than 10. Its query template identifier conflicts with $UT1$, thus the caching module examines the item further. Clearly the item identifier should not be used for this match, since we can not know for sure if the book with $id = 100$ was present in the list, or will be after the update. Therefore the item identifier is not used in the matching. So, now the caching module takes a look at the $TOR_i$ value of the cached item. This query result was created after the last update executed, thus $TOR_i > TOU$. Therefore, this item is up-to-date for now and will not be invalidated.

Next, the caching module examines the stored query result $\langle QT1, 100 \rangle$. This result obviously conflicts with the update query identifier. The item identifier also matches, thus the caching module finally examines the $TOR_i$ value of the item. This item was created before the last update was made to the item, therefore $TOR_i < TOU$ and the item needs to be invalidated.

**if** $timeout_i$ **then**
  write $invalidation\_key_i$ to $channel_i$
  unset $timer_i$ and set $Num\_Upds_i$ to 0
**end if**

**loop**
  wait for incoming *invalidation*
  **if** unset $timer_{invalidation}$ **then**
    set $timer_{invalidation} = current\_time + TTI_{invalidation}$
  **end if**
  increase $Num\_Upds_{invalidation}$
  **if** $Num\_Upds_{invalidation} \geq Max\_Upds_{invalidation}$ **then**
    write $invalidation\_key$ to $channel_{invalidation}$
    unset $timer_{invalidation}$ and set $Num\_Upds_{invalidation}$ to 0
  **end if**
**end loop**

**Figure 4.3:** globeCBC invalidator module functionality

### 4.1.2 Invalidator module

The invalidator module keeps track of the updates performed and sends invalidation messages to the caching modules when needed. These invalidations are needed when an update is executed and the corresponding query results need to be refreshed. Once the caching module of an edge server receives an update query, the update query is forwarded to the database server and executed there. Once the query is executed, the caching module sends the $\langle UT_i, ID \rangle$ pair associated with the update to its local invalidator module.

Figure 4.4 shows the basic functionality of the invalidator module. We compare this to the functionality of the globeCBC invalidator module (Figure 4.3). We show only the "weak" version of the module, which uses $TTI$ and $Max\_Upds$ to delay invalidations. However, if we set both $TTI$ and $Max\_Upds_i$ to 0 we get the behavior we want for "strong" globeCBC.

When we compare flashCBC to globeCBC, we see several differences. First of all, the publish-subscribe channels are no longer present in flashCBC. The invalidator modules are now distributed across all edge servers. Therefore, it would require a lot of management to keep track of all the subscriptions. Since we need all the resources available to handle the load, the publish-subscribe channels have been removed.

The $Max\_Upds_i$ parameter has also been removed. Since during a flash crowd many updates can be expected, this parameter could simply increase too fast. Therefore, this parameter has been removed, thus we will only delay invalidations using the $TTI$ parameter. The $TTI$ parameter in turn has been changed to a global parameter. In globeCBC each update query template can have its own $TTI_i$ parameter. In flashCBC the $TTI$ parameter is used to delay all remote invalidations by the same amount. This makes management of flashCBC easier, especially for a system with many update query templates.

Once the invalidator receives the update pair it first sets the $TOU_i$ parameter for this update pair to the current time. Then it constructs an invalidation message to be sent to its local caching module. In this message the $UT$ parameter is set to $UT_i$ defined in the update pair. Its $TOU$ parameter is set to $TOU_i$ and if present the $I\_ID$ is set to the $ID$ found in

**if** $timeout_i$ **then**
  send $invalidation\_message_i$ to other edge servers
  unset $timer_i$
**end if**

**loop**
  wait for incoming *invalidation*
  set $TOU_{invalidation} = current\_time$
  send $invalidation\_message_{invalidation}$ to local caching module
  **if** unset $timer_{invalidation}$ **then**
    set $timer_{invalidation} = current\_time + TTI$
  **end if**
**end loop**

**Figure 4.4:** flashCBC invalidator module functionality

the update pair. The message is sent to the local caching module, which in turn invalidates all conflicting query results as described in the previous section.

If the $\langle UT_i, ID \rangle$ update pair has not been stored yet, the invalidator module stores the update pair to be used for a delayed invalidation of the remaining caches and starts a timer, which expires after $TTI$ seconds. Once the timer for an update pair expires, the invalidator module sends an invalidation message to all the other edge servers in the system. This message contains the parameters of the update pair and the $TOU_i$ parameter. The receiving edge servers will invalidate all cached results that match the received invalidation message.

The $TTI$ parameter can be used to tune the trade off between performance and consistency of flashCBC during a flash crowd. As the $TTI$ parameter increases, the amount of invalidations sent to edge servers reduce. This is caused by the fact that updates for a same object can be grouped into one invalidation. For example, if an update query $UT_i, ID$ is executed, the $TTI$ timer is started for that update. If more updates for $UT_i, ID$ arrive while $TTI$ has not been expired yet, the updates are executed and the locally cached items invalidated as needed. After the local invalidations, the invalidator module sets $TOU_i$ to the current time, thus we now now the last update performed at this cache for $UT_i$ has been at time $TOU_i$. However, no invalidation messages have been sent for $TU_i$ yet. As soon as $TTI$ expires, the invalidator module sends invalidation messages to the other edge servers, containing the update pair and $TOU_i$. In this way we can group $N$ updates into one invalidation, instead of $N$ invalidations.

The system administrator can determine the value $TTI$ needed by the system. However, the experiments presented in Chapter 5 show that a $TTI$ value of 60 seconds is a good trade-off between performance and consistency requirements.

Since the invalidator module immediately sends an invalidation message to its local caching module, the end user will be presented with the updated situation immediately after he or she made a change. For example, if the user places a comment, it will be immediately visible to that user. Since we require that a user is redirected to the same edge server for each request, the system naturally provides Read Your Writes consistency [23].

The only possible scenario whereby a user will not be presented with Read Your Writes consistency is if the user would move to another location and by that movement is redirected

to another edge server. However, we assume that the redirection policy is such, that a user would have to relocate to at least another building to be redirected to another edge server. This relocation will certainly take longer than the $TTI$ parameter, thus Read Your Writes is guaranteed, as long as the redirection policy works as required.

# Chapter 5

# Performance

In order to test the performance of a system, some form of benchmarking is required. Several benchmarks exist, modeling different kinds of applications. We decided to test flashCBC using a benchmark modeling an e-commerce application: TPC-W [24]. This benchmark is an industry standard e-commerce benchmark that models an online bookstore.

Since TPC-W considers all books to be equally popular, the benchmark randomly determines which books an emulated browser will select. Therefore, the benchmark displays pretty low query locality, which is unfavorable for a caching solution. However, I adapted the benchmark in such a way, that its access patterns reflect the properties found during a flash crowd. Using these adaptations, the query locality of the benchmark increases, making caching a more suitable candidate for dealing with the load. The changes made to the benchmark are explained in full in Section 5.1.

The experiments were performed on the Distributed ASCI Supercomputer 3 (DAS-3) [1]. Using this supercomputer, the experiments could be run using a various number of edge servers and load conditions. Section 5.1 explains more about the DAS-3 supercomputer and how the experiments were configured.

In order to show how flashCBC improves query latency we compare the system using several different configurations, ranging from no caching at all, to full flashCBC where we delay remote invalidations as long as possible. Section 5.2 shows the results of these comparisons.

## 5.1 Experiment Setup

All experiments were run on the DAS-3 supercomputer [1]. This is a five-cluster wide-area distributed system shared by several Dutch universities. All the experiments were run on the VU cluster, consisting of 85 nodes. Each node has dual-core dual-CPU AMD Opteron processors running at 2.4 GHZ and 4 Gigabytes of RAM. Because of the large amount of RAM available on each node, all experiments could be run with the cached items placed in main memory.

The benchmarks were executed using a JAVA implementation of TPC-W [13]. We used tomcat 5.5.23 webservers to run the benchmark, and postgresql 8.1.9 as the database. The database was populated with 100,000 books, 25,000 authors, 144,000 customers and 129,600 orders.

Since TPC-W in its normal form is not designed to simulate flash crowd traffic, I adapted the benchmark to more accurately model an e-commerce site flooded by a flash crowd. TPC-W

uses emulated browsers to simulate the traffic directed toward a web server. These emulated browsers communicate with the web server using HTTP requests, and follow a pseudo random access pattern across the site. However, access patterns are normally not pseudo random. Instead, traffic patterns follow a Zipf-like distribution [14] [2]. Therefore, I adapted the emulated browsers so they follow such a distribution. Using an $\alpha$ parameter, we can control the locality of the generated traffic. If the $\alpha$ value is increased, the requests focus on a smaller subset of the available items.

We can distinguish between two types of traffic when we analyze a flash crowd. We have the normal traffic, which consists of users that normally visit the website. However, a large portion of the requests will be made by a second group, consisting of users belonging to the flash crowd. Therefore, I created two types of emulated browsers: normal and flash. The normal browsers access the website using a Zipf-like distribution with an $\alpha$ value of 1, while the flash browsers use an $\alpha$ value of 4.

A typical browsing page of the TPC-W benchmark contains 5 recommended books on top of the page. These books are selected from the database using a single query. In the original version of the benchmark, this query selects 5 random books from all the books in the database. However, this does not reflect how a normal e-commerce website behaves. Normally, the owner of a website would want to have a little control on which books are presented to the user, possibly favoring new books over older books. Therefore I changed the recommended books behavior. Instead of selecting random books, the benchmark selects recommendations from a subset of 100 books.

These changes increase the locality of the TPC-W benchmark. By implementing the Zipf-like distribution, more emulated browsers will request the same items. The main improvement, however, was created by adapting the recommended books behavior. Because the books were normally selected randomly from all the books available, each page visit would most of the time generate a new set of books. Because of these new sets, caching would not work well, since these books would flush the cache every time. By introducing the subset of books, we still show a fair amount of recommended books, while the cache does not get flushed anymore.

Because we do not need all the statistics TPC-W normally gathers for our purposes, we decided to only measure database query latency and cache metrics. The query latency is used to measure how long each database query takes to be executed. This can be either the time it takes to execute the query on the database server, or the time it takes to fetch the result from the query cache.

We will show the cumulative distribution function (CDF) of the observed latencies. The CDF displays which percentages of the requests were handled within a certain time. For example, if we see a CDF value of 40% at 120 milliseconds, this means that 40% of the requests have been handled in 120 milliseconds or less. Thus using the CDF we get a clear picture of the performance gained using a particular solution.

The cache metrics are used to determine how the cache itself behaved during an experiment. We can determine which percentage of query results was fetched from the cache, which percentage from the database and which percentage of the requests was an update query.

Along with these basic measurements, we can also measure if a cached item was out-of-date. A cached result is out-of-date when an update conflicting with that result has been executed and the result has not been invalidated yet. However, since the caching system in itself is weakly consistent, we can only measure this value relative to immediate invalidations where $TTI$ is set to 0s.

Each run of an experiment is performed using the same basic setup. The origin node

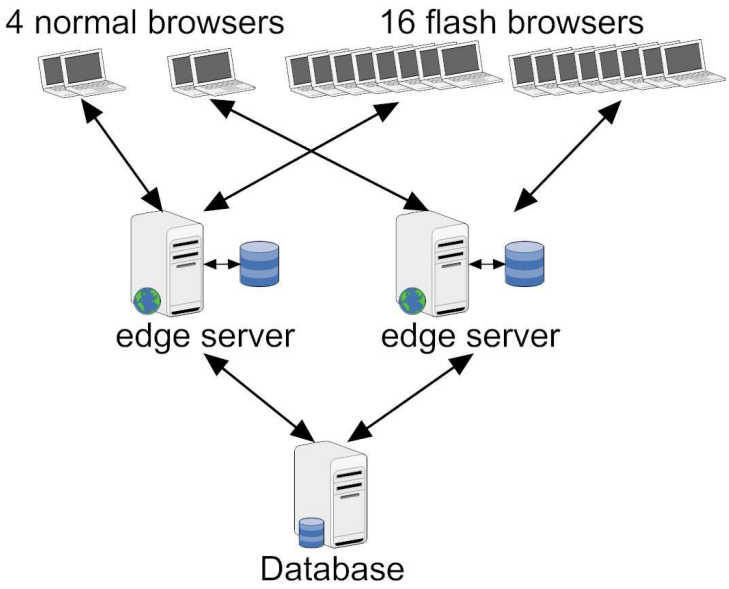


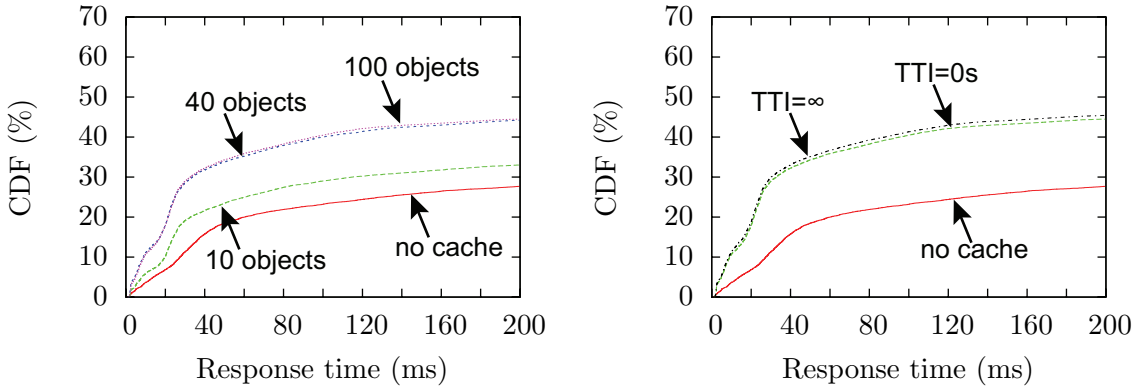

**Figure 5.1:** Example experiment setup using 4 normal emulated browsers, 16 flash emulated browsers, 2 edge servers and one database server.

contains the database server. Every edge server has its own separate node and runs a web server, which connects to the database at the origin server using a single static database connection. Using a single database connection simplifies cache management significantly. Because only one query can be executed at a time, we can easily manage the cache on a per query basis. For each edge sever another separate node is reserved for the emulated browsers directed to that node. By reserving an extra separate node for the emulated browsers, we reduce the chances of the nodes running emulated browsers becoming a bottleneck themselves.

The emulated browsers will always be uniformly distributed among the edge servers. Figure 5.1 shows an example distribution of 4 normal browsers and 16 emulated browsers across 2 edge servers. Each edge servers gets assigned an equal amount of browsers, 2 normal browsers and 8 emulated browsers each. If there are less browsers than edge servers, some edge servers will not be assigned a browser. Suppose we have $N$ browsers and $M$ edge servers, $N < M$. Then $N$ edge servers will be assigned a browser, while the remaining $M - N$ browsers will not.

## 5.2 Results

This section discusses the results of the experiments performed using various amounts of edge servers. These are used to demonstrate the influence of the size of the cache and the $TTI$ parameter on the performance of flashCBC. Since flashCBC has been based on globeCBC, this section will also compare flashCBC to globeCBC. This can be done pretty easily, since the behaviour of flashCBC with a $TTI$ of 0s is pretty much the same as globeCBC in its "strong" form. However, we will also show how flashCBCs performance compares to "weak" globeCBC, where globeCBC delays all invalidations.




(a) The effect of cache size ($TTI = 0s$)

(b) The effect of $TTI$ (cache size = 100 objects)

**Figure 5.2:** Performance analysis using 1 edge server, 1 cache, 8 normal emulated browsers and 56 flash emulated browsers

### 5.2.1 Single server analysis

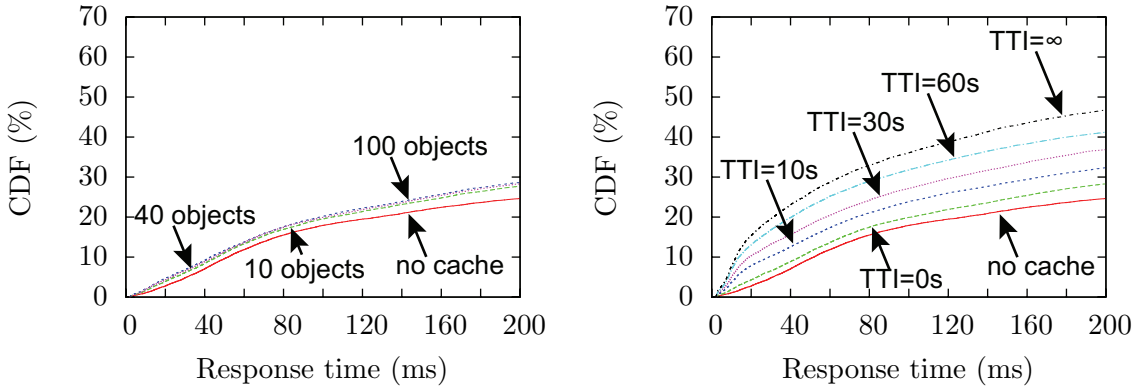
This analysis shows how query caching can improve the performance of the database when there is just one server handling the load. Using this analysis we show that we can improve database performance by adding a cache. However, since $TTI$ reduces the amount of remote invalidations, the introduction of this parameter does not improve performance in a single edge sever setup.

Figure 5.2 shows the analysis of the experiments using just one edge server, one cache, 4 normal emulated browsers and 16 flash emulated browsers. This could be the situation where we only have one server and would only add a query cache between the application server and the database running at that origin server. Clearly just adding a cache increases the server performance. Where without caching not even 40% of the requests are handled within 200 milliseconds, using a cache of just 40 objects we can reduce the query latency to about 30 milliseconds or less for 40% of the requests. We also see a gap between a cache of 10 objects an a cache of 40 objects. However, a cache of 100 objects does not improve the query latency much further.

Without any remote invalidations, changing the $TTI$ parameter clearly does not influence the results, as can be seen in Figure 5.2(b). Changing the $TTI$ parameter does not increase the performance of the system at all. Since the $TTI$ parameter is of no use in this situation, the consistency evaluation has been omitted. It is clear, that with just one server, flashCBC does not perform different than globeCBC.

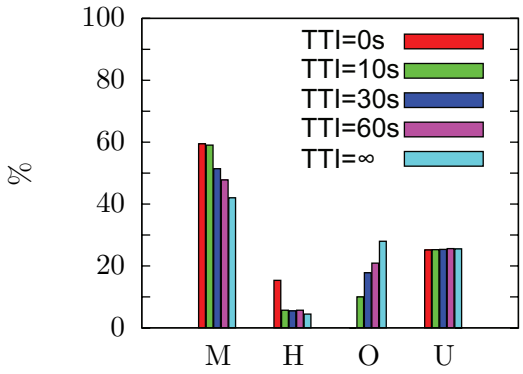
### 5.2.2 $TTI$ analysis

The main difference between flashCBC and globeCBC is the use of the $TTI$ parameter which delays remote invalidations. Remember that items in the local cache are always invalidated as needed immediately. To clearly demonstrate the effect $TTI$ has on the performance of the system, we need an experiment setup which features a lot of remote invalidations. In order to achieve this, many edge servers are required. Therefore this section discusses the analysis of our experiments using 8 edge servers. Each edge server has its own cache. For these experiments we use a load of 8 normal and 56 flash emulated browsers. This leads to




(a) The effect of cache size ($TTI = 0s$)

(b) The effect of $TTI$ (cache size = 100 objects)




(c) The effect of $TTI$ on cache metrics. M = cache miss, H = cache hit, O = outdated hit, U = update.

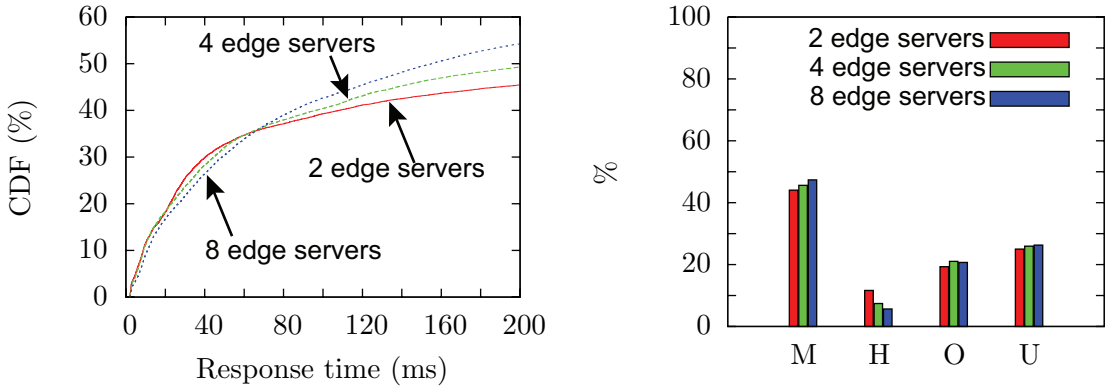
**Figure 5.3:** Performance analysis using 8 edge servers, 8 caches, 8 normal emulated browsers and 56 flash emulated browsers

one normal emulated browsers and 7 flash emulated browsers per edge server.

As we can see in Figure 5.3, adding a query cache to each edge server helps decreasing the response time of the database a little. Using globeCBC in its "strong" form is limited in its performance increase. However, Figure 5.3(b) shows that, as $TTI$ increases, the response time decreases significantly using flashCBC. Without delaying the invalidations not even 20% of all the requests are handled within 200 milliseconds. If we disable remote invalidations the performance increases such that 40% of all requests are handled within 130 milliseconds. However, by delaying the remote invalidations by 60 seconds, we already gain enough performance such that 40% of the requests are handled within 200 milliseconds.

Figure 5.3(c) shows the effect of $TTI$ on the cache metrics. The histogram shows for each setting of the $TTI$ parameter the percentage of requests resulting in a cache miss, indicated by the M column. Next to that we see the percentage of cache hits that are up-to-date, indicated by the H column. In the O column we see the amount of cache hits that are out-of-date. Finally, the U column shows the percentage of queries that are update queries. Naturally, the percentage of updates does not vary a lot between runs.

We can clearly see that the increase in performance observed in Figure 5.3(b) is caused




(a) Response time comparison

(b) Cache metrics comparison. M = cache miss, H = cache hit, O = outdated hit, U = update.

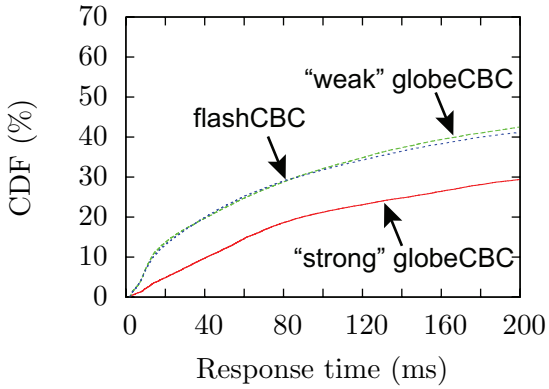
**Figure 5.4:** The effect of the amount of edge servers on the performance of flashCBC using a cache size of 100 objects, a $TTI$ of 60 seconds, 4 normal emulated browsers and 32 flash emulated browsers.

by a decrease in the cache miss rate. As the cache miss rate decreases, we see the cache hit rate remains fairly stable, decreasing slightly as we delay the remote invalidations longer. However, the outdated hit rate increases significantly as $TTI$ increases. This clearly causes the reduced cache miss rate. Looking at the cache metrics where $TTI = \infty$ we see the maximum decrease in cache miss rate we can achieve in this setup is about 18%. However, by delaying invalidations by only 60 seconds we already decrease the miss rate by 12%.

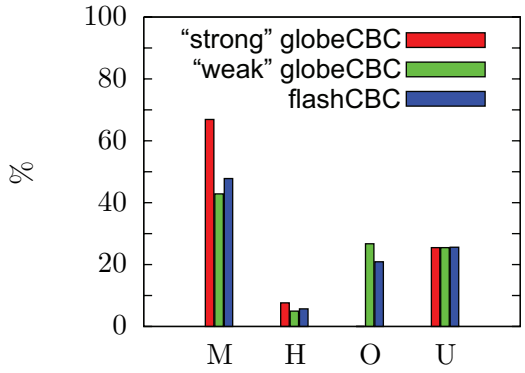
### 5.2.3 Edge server analysis

FlashCBC has been designed to unburden the database server during a flash crowd. As a flash crowd grows, more edge servers will be needed to handle the load. Normally this causes the database to quickly become the bottleneck of the system. Therefore, it is important to understand how flashCBC helps to remove this bottleneck. This sections shows how flashCBC unburdens the database server as we increase the amount of edge servers. This analysis was again performed using a load of 4 normal and 32 flash emulated browsers. Each edge server has its own cache of 100 objects and $TTI$ is set to 60 seconds.

Figure 5.4 shows that as the amount of edge servers increase, the response time of the database decreases. As we increase the amount of edge servers from 2 to 8, we reduce the response time from 111 to 85 milliseconds. This shows that, as we add edge servers to the system, the bottleneck of the database server can be reduced using flashCBC. However, we also see that flashCBC performs slightly better using less edge server looking at response times below 60ms. Figure 5.4(b) shows that this is caused by the fact that the cache miss rate increases slightly as the number of edge servers is increased. This can be explained by the amount of clients directed to the caches. Where the caches serve 18 clients each in the case of 2 edge servers, the caches serve a maximum of 5 clients using 8 edge servers. As a cache serves less clients, the effectiveness of the cache reduces. Therefore, the amount of edge servers should always be fine-tuned as much as possible to the observed load, to maximize the cache's performance.




(a) Response time comparison




(b) Cache metrics comparison. M = cache miss, H = cache hit, O = outdated hit, U = update.

**Figure 5.5:** Comparison between "weak" globeCBC and flashCBC using 8 edge servers, 8 caches, a cache size of 100 objects, a $TTI$ of 60 seconds, 8 normal emulated browsers and 56 flash emulated browsers.

### 5.2.4 "Weak" globeCBC analysis

So far we only compared flashCBC to the "strong" version of globeCBC. However, the performance of "weak" globeCBC is better than the "strong" version. Although the consistency of "weak" globeCBC is unacceptable, the performance increase is undeniable. Therefore this analysis shows how flashCBC compares to "weak" globeCBC. This analysis was performed using 8 edge servers each with their own cache again. The cache size is set to 100 objects for both globeCBC and flashCBC. $TTI_i$ is set to 60 seconds for all query templates for globeCBC, $TTI$ has also been set to 60 seconds for flashCBC.

Figure 5.5 shows that flashCBC performs slightly worse than "weak" globeCBC. However, although "weak" globeCBC performs slightly better, the increase in performance is not as much that it justifies the cost in consistency. Remember that globeCBC does not provide Read Your Writes consistency, thus every update will only show after one minute in this setup, naturally this is unacceptable. FlashCBC performs slightly less than "weak" globeCBC, however, Read Your Writes is guaranteed, allowing users to remain satisfied with their experience of the website. Therefore, flashCBC is the better choice.

## 5.3 Discussion

During a flash crowd, the query latency can be improved significantly by using flashCBC. Even without distributing users to edge servers, we can gain performance by using flashCBC. However, the main performance increase is achieved when multiple edge servers are used. As the amount of edge servers increases, the performance of each individual cache increases. This is caused by the load being distributed among the edge servers.

However, as we increase the amount of edge servers and load, we also increase the amount of remote invalidations. These invalidations cause the cache miss rate to increase. This can be clearly seen in Figure 5.3(c). However, the analysis presented there also clearly demonstrates how $TTI$ can be used to decrease the negative effects of the remote invalidations. By delaying

remote invalidations 60 seconds, we can decrease the cache miss rate by as much as 20%. As seen in Figure 5.3(b), this leads to a significant decrease in query latency.

Careful planning of the amount of edge servers is required to maximize the efficiency of the cache. Caches perform less when they server less clients, therefore the amount of edge servers should always be fine-tuned to the actual load to maximize the efficiency of the caches.

# Chapter 6

# Conclusions and Future Work

In this paper, we presented flashCBC, a content-blind query caching middleware for hosting web applications in an edge computing infrastructure based on globeCBC. FlashCBC has been designed to help reduce the load on the database server during a flash crowd. Chapter 3 showed that, during a flash crowd, the locality increases significantly. As query caching works better when applications exhibit high locality, it is a viable strategy to unburden the database.

However as the number of edge servers increases, the amount of remote invalidations caused by updates increase as well. These invalidations normally severely reduce the effectiveness of a caching system. FlashCBC has been designed to solve that problem. It has been shown that the amount of invalidations can be reduced by delaying the invalidations sent to remote servers. By delaying the invalidations by 60 seconds, we can gain significant increases in performance. However, because flashCBC invalidates the local cache immediately, flashCBC still provides Read Your Writes consistency. This ensures users will always immediately see the results of their own actions.

This approach works well for applications that do not require extensive interaction between its users, like e-commerce stores or bulletin boards. However, when real time user interaction is required, for example on an auction site like ebay, Read Your Writes consistency is not enough to ensure user satisfaction. Therefore, flashCBC is unsuitable for that kind of applications. In order to reduce the load on the database for that kind of systems it could be a better course of action to partition the database across multiple servers, each sharing a part of the load.

FlashCBC could be used as a part of a flash crowd rescue system used by content delivery networks like Akamai or Globule. Such a framework would ideally use an implementation of the flash crowd prediction system introduced in Section 2.4. Once a flash crowd is detected, the system can immediately configure edge servers with the application, using flashCBC as the middleware between the application servers running on the edge servers and the database server. As it is difficult to predict how far a flash crowd will continue to grow, the system should constantly monitor the trend of the request rate to assign more edge servers when needed. Once the flash crowd is over, the system should switch back to normal usage, releasing the edge servers and disabling flashCBC. If a caching scheme is required during normal usage, the system designer could choose to switch to globeCBC, or continue to use flashCBC with a $TTI$ value of 0 seconds when Read Your Writes consistency is needed.

# Bibliography


[1] Distributed ASCI Supercomputer 3. *http://www.cs.vu.nl/das3*.

[2] ACM. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology (TOIT), v.1 n.1, p.44-69, August 2001.*

[3] S. Adler. The slashdot effect: An analysis of three internet publications. *http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html*, 1999.

[4] Akamai. *http://www.akamai.com*.

[5] Amazon. *http://www.amazon.com*.

[6] Y. Baryshnikov, E. G. Coffman, G. Pierre, D. Rubenstein, and M Squillante. Predictability of web-server traffic congestion. *Proceedings of the Tenth IEEE International Workshop on Web Content Caching and Distribution*, pages 97–130, 2005.

[7] Ebay. *http://www.ebay.com*.

[8] Quaternion Julia Fractals. *http://local.wasp.uwa.edu.au/~pbourke/fractals/quatjulia*.

[9] Globule. *http://www.globule.org*.

[10] Gnuplot. *http://www.gnuplot.info*.

[11] Google. *http://www.google.com*.

[12] Herman-Izycka. Flash crowd prediction. Master's thesis, Vrije Universiteit, 2006.

[13] TPC-W implemented in JAVA. *http://www.ece.wisc.edu/~pharm/tpcw.shtml*.

[14] J. Jung, B Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. *Proceedings of the International World Wide Web Conference*, pages 252–262, 2002.

[15] NASA. *http://www.nasa.gov*.

[16] M. Rabinovich and A Aggarwal. Radar: A scalable architecture for a global web hosting service. *Computer Networks (Amsterdam, The Netherlands: 1999 31, 11-16*, pages 1545–1561, 1999.

[17] M. Rabinovich, Z Xiao, and A Aggarwal. Computing on the edge: A platform for replicating internet applications. *International Workshop on Web Caching and Content Distribution, Norwell, MA, USA*, pages 57–77, 2004.

[18] NASA August 1995 Space Shuttle Status Reports. *http://www-pao.ksc.nasa.gov/kscpao/status/stsstat/1995/aug/aug95.htm*.

[19] S. Sivasubramanian. *Scalable Hosting Of Web Applications*. PhD thesis, Vrije Universiteit, 2007.

[20] S Sivasubramanian, G Pierre, M Van Steen, and G Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, June 2006. `http://www.globule.org/publi/GCBRCDWA_ircs022.html`.

[21] Slashdot. *http://www.slashdot.org*.

[22] A.J Smith. Cache memories. *ACM Computing Surveys (CSUR), Volume 14 , Issue 3 (September 1982)*, pages 473 – 530, 1982.

[23] D. G. Terry, A. J. Demers, K. Petersen, M. J. Preitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, September 1994*, pages 140–149, 1994.

[24] TPC-W. *http://www.tpc.org/tpcw/*.

[25] W Zhao and H Schulzrinne. Dotslash: A self-configuring and scalable rescue system for handling web hotspots effectively. *International Workshop on Web Caching and Content Distribution (WCW), Beijing, China, October 2004*.

[26] W Zhao and H Schulzrinne. Dotslash: Handling web hotspots at dynamic content web sites. *IEEE Global Internet Symposium, Miami, Florida, March 2005*.

[27] W Zhao and H Schulzrinne. Enabling on-demand query result caching in dotslash for handling web hotspots effectively. *IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb'06), Boston, Massachusetts, November 2006*.