Performance Analysis of a Flash-Crowd Management System

Reinoud Esser (1142909)
Faculty of Sciences
Vrije Universiteit, Amsterdam, The Netherlands

August 2006
Master thesis, Computer Science

Supervisor:
Dr. Guillaume Pierre

vrije Universiteit amsterdam

Flash-crowds are a growing obstacle to the further expansion of the Internet. One of the solutions to this problem is to replicate the most popular documents to different web servers and to redirect client requests to these replicas. In this thesis we present a performance analysis of a flash-crowd management system based on RaDaR. We adjust the architecture of RaDaR to focus more on adaptability rather than scalability, to give the system a better chance against a flash-crowd by using algorithms from another system by the same authors, ACDN. Because existing benchmarks do not show realistic behavior, we first propose our own synthetic benchmark. Finally we use the benchmark tool to replay requests from a trace of an actual flash-crowd.

Our results are three-fold: first we show how to dimension a RaDaR-like system. Second, we demonstrate based on a synthetic benchmark as well as a trace-based benchmark that the RaDaR-like system adjusts to a flash-crowd in a timely and efficient fashion. Finally, we identify an inherent instability in the replica placement and request distribution algorithms that handle offloading.

Keywords: flash-crowd, performance analysis, benchmark, RaDaR

Contents

1. Introduction

The term “flash-crowd” was first used in a short science-fiction story by Larry Niven [12]. The story talks about the invention of a transfer booth that could take one anywhere on earth almost instantly. The designers of this booth did however not anticipate on thousands of people simultaneously visiting the scene of any event reported in the news, resulting in chaos and confusion. Something similar can happen on the Web, when the news about an event can result in a large number of users suddenly trying to access a (newly) popular web site. Some of these events are predictable, like specific competitions at the Olympic games or the US Presidential Elections; others like epidemics or terrorist attacks are not. When a web site is not prepared for the enormous growth in demand it might lead to significant delays, requests not reaching it and eventually complete failure. Even when the event is predictable it is difficult to predict the exact magnitude of the extra demand and to react in time.

One of the solutions to the flash-crowd problem is to throttle the demand at an early stage of each request, hereby servicing only part of the requests and rejecting the rest as early as possible; this obviously does not give satisfying results to every user. A better (and more commonly used) solution is to replicate the popular web documents on different web servers and redirect users to these servers. In the past, the decision on the amount and placement of the replicas was entirely done by system administrators, which is an immensely difficult and tiresome job. Nowadays systems, called *replica hosting systems*, are used to create, migrate and remove replicas dynamically.

A promising system to handle flash-crowds is RaDaR [2]. It uses multiple levels of machines to redirect users, and can create replicas on the fly when necessary. However, RaDaR was designed with scalability in mind and therefore a trade-off between scalability and adaptability is made. For instance, no live information about the load of the web servers hosting the replicas is used to make decisions on where to redirect users. This approach minimizes the internal communication of the system, but makes it react slower to any sudden changes in load.

This thesis presents a benchmark that is representative for real world use of a web server. The benchmark is then used to measure the performance of RaDaR under normal load as well as in a flash crowd. This goes much further than the original RaDaR measurements. To give RaDaR a better fighting chance against a flash-crowd we altered the design by replacing algorithms with newer ones from another system, ACDN, by the same authors as RaDaR [14].

Our results are three-fold: first we show how to dimension a RaDaR-like system. Second, we demonstrate based on a synthetic benchmark as well as a trace-based benchmark that the RaDaR-like system adjusts to a flash-crowd in a timely and

efficient fashion. Finally, we identify an inherent instability in the replica placement and request distribution algorithms that handle offloading.

This thesis is organized as follows. In Section 2 we discuss a selection of replica hosting systems and different approaches to decide where to place documents, where to redirect users and how to detect flash-crowds. In Section 3 we describe the qualities a good web server benchmark should possess. We present four different synthetic methods of benchmarking web servers, each method being an improvement over its predecessor. Finally we explain why the final method gives reliable and representative results. In Section 4 we describe how we implemented a RaDaR-like system and what changes were made to make RaDaR a more adaptable system in the context of a flash-crowd. The rest of the chapter contains details about the experiment and the benchmarking results from the synthetic benchmark as well as from replaying the trace of an actual flash-crowd. In Section 5 we draw our conclusions.

2. Related work

Earlier research has resulted in a lot of different systems, such as RaDaR, DotSlash and CoralCDN, to handle the distribution of web documents across multiple servers. We present the basics of a selection of these systems and their different approach to decide where to place documents, where to redirect clients and how to detect flash-crowds. First an overview of all the systems is given in Section 2.1; this includes the philosophy behind the systems, the layout and a short description of the different sub-systems. Section 2.2 and Section 2.3 describe how the different systems handle replication with respect to replica placement and redirection. Finally Section 2.4 discusses the way these systems detect and react to a flash-crowd.

2.1. Replication systems overview

In this section, we present a general description of the most important content delivery networks. More details about specific aspects of their technical characteristics are discussed in the remaining of this chapter.

2.1.1. RaDaR

RaDaR (Replicator and Distributor and Redirector) is an architecture for information hosting systems that can be deployed on a global scale [2]. It places and moves documents dynamically. The architecture tries to balance the load among the different servers used, places replicas of documents in the proximity of clients from which most of the requests originate and scales without creating bottlenecks.
Figure 2.1 shows a high-level view of RaDaR. Every document has a physical and a symbolic name. A *multiplexing service* maintains the mapping between the two names. The physical name is the name by which the objects are accessed internally by the system (for example the URL of a replica). The symbolic name is the name that is used by external web clients to access the object. A symbolic name can be mapped onto multiple physical names when the document is replicated.
When an object is created, it is placed on one of the *hosting servers* and then registered with the multiplexing service. Registering involves picking a symbolic name for the object and telling the multiplexing service where the object was placed.
The host name part of a symbolic URL always resolves into the multiplexor's identity. This way all requests are sent to the multiplexing service. The multiplexing service will then pick a physical location of the object based on client location and host load, fetch the object and return it to the client.
Decisions on the number and location of object replicas are taken by the hosting servers in a distributed manner. Every hosting server collects access statistics about each of its objects and periodically decides whether to drop, migrate or replicate objects. When a host decides to migrate or replicate an object, it sends a request to the *replication service* to find a new location for the object. The replication service keeps

track of the load of all hosts in the system, which allows it to place replicas according to their loads.

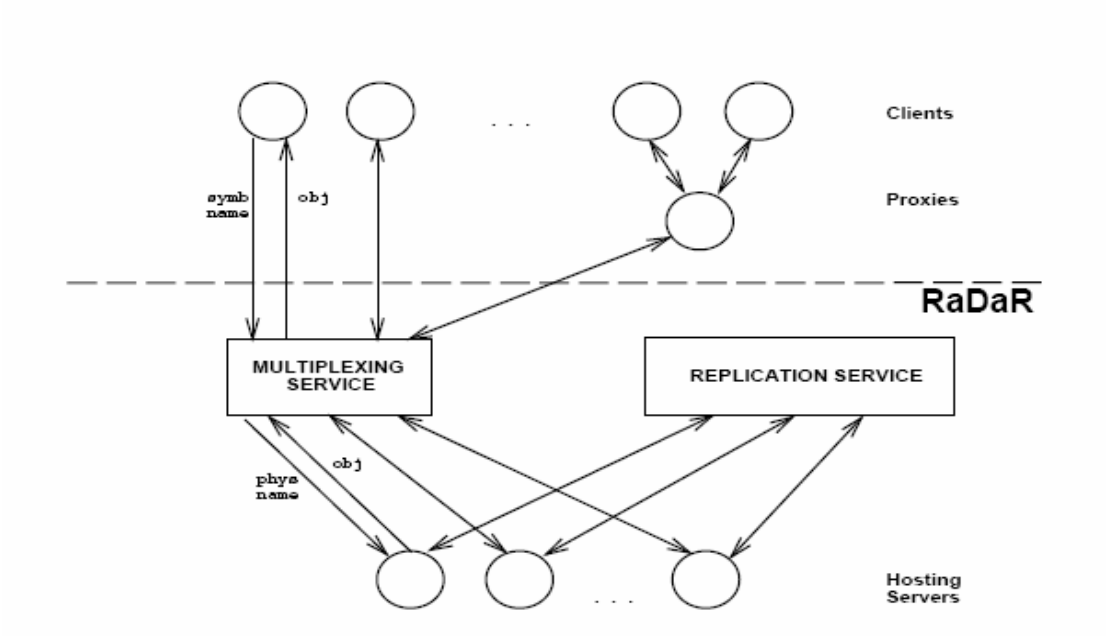

Figure 2.1: High-level view of RaDaR

2.1.2. ACDN

ACDN is a content delivery network for applications [14]. It improves performance of accesses to dynamic content and other computer applications. It can be seen as the successor of RaDaR. Its architecture is very similar to that of RaDaR but includes algorithms for automatic redeployment of applications on networked server and for distributing client requests among application replicas based on their load and proximity. The system also contains a mechanism that keeps application replicas consistent in the presence of developer update to the application data.
Each ACDN server is a standard web server that also contains a *replicator,* which implements ACDN-specific functionality. There is also a global *central replicator* that mainly keeps track of application replicas in the system. The server replicator contains scripts to decide if any application on the server should be replicated or deleted and a script to report the load of the server.

2.1.3. Akamai

Akamai is a commercial content delivery system that serves requests from a variable number of surrogate origin servers at the network edge [5]. Caching proxies are used to replicate documents, while a hierarchy of DNS servers redirects client requests to these proxies. When choosing a server to handle a client request, a trade-off will be made between a low round-trip-time and low packet-loss. To gather information

about server resources, Akamai places a monitor in every replica server. Clients are simulated to measure overall system performance and to communicate with border routers to get an estimation about distance-related metrics.

2.1.4. DotSlash

DotSlash allows websites to form a community and use spare capacity in the community to relieve flash-crowds experienced by any individual site [18]. A web server can join the community by registering it selve with a DotSlash service registry. In case of being overloaded, participating servers will discover and use their spare capacity to take over some load. A web server is in one of the following states at all times: SOS state when its receiving rescue, rescue state when its providing rescue or normal state otherwise. Figure 2.2 shows an example of a possible rescue relationship in DotSlash. S_1 and S_2 are origin servers in this example, while S_3, S_4, S_5 and S_6 are providing rescue. S_7 and S_8 are not involved in rescue services at all.

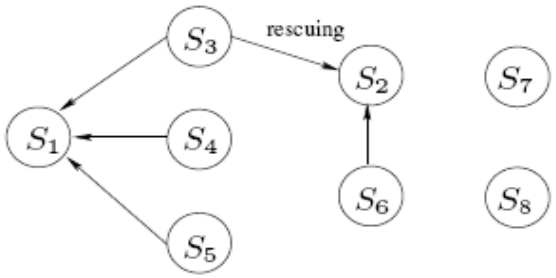

Figure 2.2: example of a possible rescue relationship in DotSlash

2.1.5. CoralCDN

CoralCDN is a peer-to-peer content distribution network that replicates content in proportion to the contents popularity, regardless of the publisher's resources [9]. One of the systems goals is to avoid flash-crowds. CoralCDN is composed of three main parts:

1. a network of cooperative HTTP proxies that handle user requests
2. a network of DNS servers that map clients to nearby HTTP proxies
3. an underlying indexing infrastructure (Coral) on which the first two applications are built

Every node in the network runs the indexing infrastructure. This infrastructure does not implement a DHT but a so-called 'distributed sloppy hash table' (DSHT). DSHTs provide applications with the means to store key/value pairs, where multiple values may be stored under the same key. The difference between normal distributed hash tables and DSHTs is that DSHTs use a sloppy storage technique that not only stores

key/value pairs on the node closest to the key, but also caches the pairs at nodes whose IDs are close to the key. This technique greatly reduces the creation of hot-spots in the overlay. CoralCDN uses this service to map a variety of types of keys onto addresses of CoralCDN nodes. This way CoralCDN is able to find nameservers, HTTP proxies caching particular web objects and nearby Coral nodes for the purpose of minimizing internal request latency.
The nodes are grouped together in so-called clusters characterized by a maximum desired network round-trip-time (the diameter of a cluster). A fixed hierarchy of diameters known as levels parameterizes the system. Each node is therefore a member of one cluster at each level. When searching for information, Coral first uses the smallest and fastest cluster it belongs to, and queries higher-level clusters only if necessary. Besides running Coral, all the nodes also have a DNS server and a HTTP proxy running.

2.2. Replica placement

Replica placement algorithms are used to determine when and where a copy of a document must be created or removed. These algorithms can be triggered either periodically, aperiodically or in a hybrid fashion [15]. When triggered periodically, a number of input variables are analyzed at fixed time intervals. Such an evaluation scheme can be effective for systems that are relatively stable. With less stable systems it might be difficult to determine the right interval to evaluate: a too short period will lead to unnecessary overhead, while a too long period might cause the system to react too slowly to changes. Aperiodic triggers will monitor the variables continually (usually using metrics like client latency or server load). This has the advantage of allowing the system to react quickly to the current situation. This does however require continuous evaluation of the variables. Hybrid triggers combine the best of both worlds. They combine periodic and aperiodic triggers to be able to perform global optimizations as well as reacting fast enough to emergency situations.

In RaDaR, replica placement is done in cooperation between the replica servers and the replication service. To make this service scalable it is implemented as a hierarchy (see Figure 2.3).

When a host is overloaded it can send an offload request to its own *replicator* to find under-loaded hosts where to migrate documents. Besides using replicating for load balancing, every host also periodically runs the replica placement algorithm to improve proximity between clients and hosts. If a document replica receives fewer requests than the deletion threshold U (and this is not the sole replica of the document), it will be deleted. When the load of the document exceeds U, it will be migrated to a host that occurs on the majority of preference paths ($MIGR\_RATIO > 0.5$); these paths are based on information periodically extracted from routers. In case migration fails (when for instance there is no such host) an attempt is made to replicate the document to a host that occurs on at least *REPL_RATIO* of the router paths. The condition for replication *REPL_RATIO* is a lot

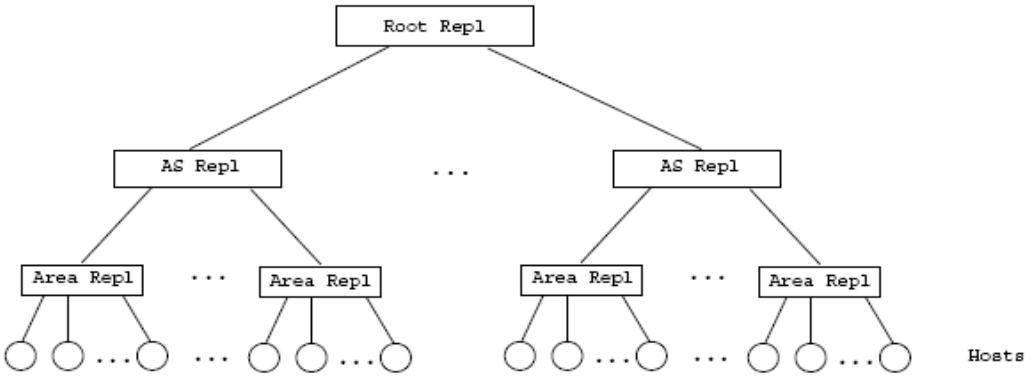

Figure 2.3: the replicator hierarchy in RaDaR

weaker than for migration (*MIGR_RATIO*). To make replication worthwhile, the load should always be above the replication threshold *M* (minimum load where replication outweighs the cost of creating a new replica).

In ACDN, replica creation is also initiated by the server hosting an existing replica. Similarly to RaDaR, when the server is overloaded, it will query the central replicator for a low-loaded server in the system, which will then be asked to create a new replica. When the reason for replication is improving proximity, the target server will be identified locally from its replica usage. Finally the target server informs the central replicator about the new replica. The central replicator sends this update to the DNS server that does the redirecting, which can now recompute its request distribution policy.
The decision process on a server with the application replica initiates replica deletion as well. The server sends the central replicator a request to delete the replica. When the server's replica is not the last one in the system, the replicator sends the deletion update to the DNS server, which can now recompute its request distribution policy. Once the DNS server has updated its policy, the server gets permission to delete the replica after the DNS time-to-live (TTL) has expired, to avoid running into problems with requests that arrive at the server due to earlier DNS responses.
The replica placement algorithm uses three parameters: the deletion threshold *D*, the redeployment threshold *R* and the migration threshold *M*. The deletion threshold characterizes the lowest demand that still justifies having a replica. The redeployment threshold reflects the amount of demand from clients in the vicinity of another server (*i*) to justify an application replica at that server. Let *Bi* be the amount of data served to clients close to this server. Now when *Bi* is larger than the total cost of replicating, the application will be replicated. Sometimes it is even beneficial to replicate even if Bi is smaller than the total cost, just to improve the latency. Also *Bi* should be larger than the deletion threshold to avoid creating a replica, which will then be deleted shortly after because of lack of demand.

The migration threshold *M* is used to make the migration decision. When *Bi/Btotal* > *M*, and if the bandwidth benefit would be sufficiently high relative to the overhead, the application will be migrated.
When a server is overloaded it will also replicate the application. The system has two load watermarks, high watermark *HW* and low watermark *LW*. A server considers itself overloaded when the load reaches *HW*, and will continue considering itself overloaded until the load drops beneath the low watermark.

In Akamai documents are replicated by caching them. When a replica server receives a request for a document it does not own, it treats it as a miss and fetches the document from the origin server. This means that the creation of a replica is delayed until the first request. The decision to place a replica in a certain server is taken when redirecting client requests to this server. To improve the performance of uncacheable documents, the documents are divided into cacheable and uncacheable portions.

DotSlash also uses a form of caching. The rescue servers generate a virtual host name for all the origin servers it is willing to rescue. It keeps a table mapping these virtual host names to the origin servers. The origin servers will redirect to these virtual host names and the rescue server can look up in the table which content to return. Whenever a rescue server has a cache miss for one of the origin servers it will issue a reverse proxy request to the origin server.

In CoralCDN each client keeps a local cache from which it can immediately fulfill requests. When a client requests a non-resident URL, CoralProxy first attempts to locate a cached copy of the referenced resource using Coral. If CoralProxy discovers that one or more other proxies have the data, it attempts to fetch the data from the proxy to which it firsts connects. If Coral provides no referrals or if no referrals return the data, CoralProxy must fetch the resource directly from the origin.

2.3. Request routing

Whenever a request for a certain document is received from a client, the system needs to decide which replica server shall best service this request and how to actually redirect the client to this server. This problem is called request routing [15]. The entire request routing problem can be divided into two sub-problems: coming up with a redirection policy and selecting a redirection mechanism.
A redirection policy is an algorithm that defines how to select a replica server in response to a given client request. A redirection policy can be either adaptive or non-adaptive. Adaptive policies take the current system conditions into consideration when making their decision, while non-adaptive policies do not. The information that adaptive policies may use, for example, is the load of replica servers or congestion of network links. Some request-related information like the object requested or client location might also be useful.
The redirection mechanism informs the client about the decision made by the redirection policy. Redirection mechanisms can be classified into transparent, non-

transparent and combined mechanisms. Transparent mechanisms hide the redirection from the client, while in non-transparent mechanisms this is visible to the client. Combined mechanisms take the best from the two previous types and eliminate their disadvantages.

RaDaR uses a non-adaptive redirection policy to select a replica server for a given request. All replica servers are ranked based on their predicted load, which is derived from the number of requests each has received so far. So instead of looking at the current load of the replica servers, as an adaptive policy would do, it uses the load information it has gathered in the past. Clients are then redirected to a replica server close to them, while the following invariant has to be maintained: the access count of the most heavily loaded replica is at most a constant times the count of the most lightly loaded replica.
The redirection mechanism is implemented in the multiplexing service. To help make the system scalable the multiplexing service is split into two types of components: *redirectors* and *distributors.* The redirector maintains the mapping from symbolic to physical names. The total symbolic namespace is divided over the different redirectors based on some hash function. The amount of redirectors is kept low because of the high frequency of updates to the symbolic-name/physical-name mapping. The second type of component, the distributor, stores the hash function mapping the symbolic names to the different redirectors. Because the function is fairly static, distributors can be highly replicated. Figure 2.4 shows the multiplexing service.

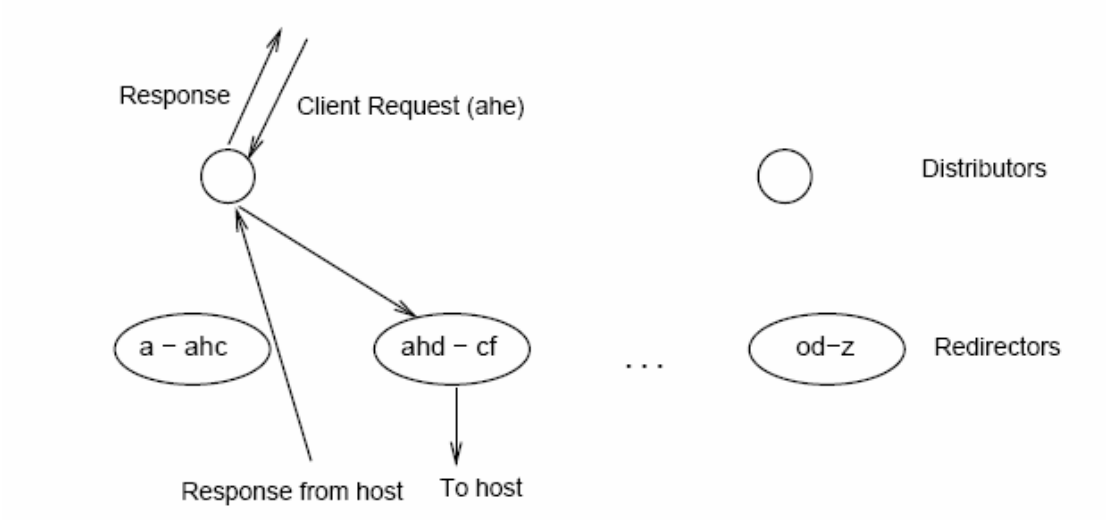

Figure 2.4: the multiplexing service in RaDaR

A client request first arrives at a distributor close to the client. RaDaR uses DNS servers (which makes this totally transparent to the client) to map the domain name of the requested website to an IP address of a distributor. The distributor will forward the request to the appropriate redirector using the hash function. The redirector will

then send select the best host for the client and will forward the request to it. The host sends the object directly to the distributor, which forwards it back to the client.

In ACDN every DNS server stores tuples of the form (*R*, *Prob(1), ..*, *Prob(N)*), where *R* is a region (could be geographical regions or network regions) and *Prob(i)* is the probability of selecting server *i* for a request from this region. The central replicator computes these tuples and sends them to the DNS servers whenever there is an update.
The load watermarks *LW* and *HW* seen earlier in this ACDN when replica creation was concerned are also used in the request distribution algorithm. The algorithm operates in three passes over the server. The first pass assigns a probability to each server based on its load. Servers with a load above *HW* get zero weight, servers with a load beneath *LW* get unity weight and servers with a load between *LW* and *HW* get a weight depending on where the load falls between *LW* and *HW*. If all servers are overloaded, the algorithm assigns the load evenly among them. In the second pass the algorithm will compute probabilities depending on distance from the region. In the third pass the probabilities will be normalized so that they will sum up to 1.

The important difference to notice between RaDaR and ACDN is that ACDN uses load feedback to take request distribution decisions while RaDaR does not. The central replicator in ACDN uses the load information that is periodically being send to it by all the hosting servers to calculate the distribution tuples for the DNS server. In RaDaR however, the redirectors keep track of the number of accesses each replica receives themselves. A drawback of the latter method is that redirectors have no information about how much load other redirectors might put on a given replica server. The reason for using this algorithm though is that RaDaR was designed to be deployed on a global scale, using load feedback would mean that every redirector needs information about the load of all the replica servers; this could mean an enormous amount of overhead when a large system is concerned.

Akamai also uses a dynamic DNS system for its redirection. The DNS system has a 2 level hierarchy. When a resolver queries an Akamai top level DNS server, it will return a domain delegation to a low level DNS server (with a TTL of about an hour) that is in the same location as the client. Then the resolver will query this DNS server, which will return the IP addresses of servers that are available to satisfy the request. This resolution has a short TTL to allow Akamai to redirect requests to other locations or servers as conditions change.

DotSlash has two configurable parameters: the lower load threshold and the upper load threshold. These two parameters define three regions: lightly loaded region [0, lower load threshold), desired load region [lower load threshold, upper load threshold] and the heavily loaded region (upper load threshold, 100]. Now when the load of an origin server gets into the lightly loaded region it will decrease the probability of redirecting requests, where it will increase the probability of redirecting requests when the load gets into the heavily loaded region. When a rescue server gets into the lightly loaded region it will increase the bandwidth available for taking the

requests of origin servers and it will decrease the bandwidth available for taking requests when it gets into the heavily loaded region. The amount by which the probability is changed depends on the difference between the load and the reference load (the average of the lower threshold and the upper threshold).
To offload client requests to its rescue servers DotSlash uses both HTTP redirecting and DNS round robin.

A good example of a system that uses a non-transparent redirection mechanism is CoralCDN. To use CoralCDN, a content publisher appends ".nyud.net:8090" to the hostname in a URL. Now when the client accesses this Coralized URL, the local DNS resolver of the client will query a Coral DNS server. This DNS server will probe the client to determine its round-trip-time and last few network hops. Based on the probe results, the DNS server checks Coral to see if there are any known nameservers and/or HTTP proxies near the client's resolver. If none were found, it returns a random set of nameservers and proxies. The client will now send the request to the specified proxy. If the proxy is caching the file locally, it returns the file and stops. Otherwise, the proxy looks up the web object's URL in Coral. If Coral returns the address of a node caching the object, the proxy fetches the object from this node. In case no other nodes are currently caching the object, the proxy will download the object from the origin server. The object is then returned to the client, and a reference will be stored in Coral, recording the fact that this proxy is now caching the URL.

2.4. Detecting flash-crowds

In a flash-crowd, before doing any replication or adapting of the redirection policy, the problem is to first actually detect the flash-crowd. The most systems use aperiodic triggers to do this (see Section 2.2): one or more variables are monitored continuously to look for any irregular behavior. One of the most used and simplest metrics to monitor is the request rate. When the request rate exceeds a certain threshold, the system starts adapting to the new situation. The idea behind this is that a high request rate might be a sign of an upcoming flash-crowd [11]. Using a single threshold however, is in practice not always enough; the oscillating behavior of the request rate could result in a lot of unnecessary system adaptations. To address this problem, systems like DotSlash and ACDN use a watermarking technique with two thresholds [14][18]. The system defines two watermarks: the low watermark and the high watermark. After exceeding the high watermark, replica servers consider themselves overloaded, which will remain unchanged until the request rate reaches the low watermark.

Initiating document replication after one of the servers is considered overloaded might however prove to be too late when a flash-crowd occurs. It might be advantageous to try to avoid flash-crowds more proactively; Felber et al. describe an algorithm that introduces a third watermark to detect flash crowds more gradually and to adjust the system according to the current phase of the flash crowd [7]. The extra

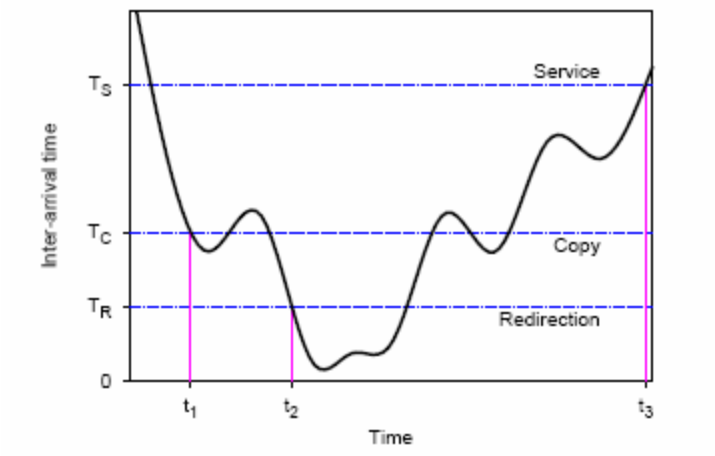

Figure 2.5: adding a third watermark

watermark is positioned between the low watermark and the high watermark, and is treated as an early warning about an upcoming flash-crowd. Figure 2.5 illustrates how this algorithm works.

Inter-arrival time is computed as an exponential weighted moving average of the difference between the arrival time of a hit and the arrival time of the previous hit. In other words: the lower the inter-arrival time, the higher the load on the server. When the inter-arrival time reaches the "copy threshold" T_C (at time t_1) the system suspects a flash-crowd and will take early measures by starting to replicate documents. However, because CPU and network have not yet reached saturation during this point in time, requests will not be redirected to these new replicas until the inter-arrival time reaches the "redirection threshold" T_R (at time t_2). After the flash-crowd has passed, the system has to return to normal operation. Again, to avoid that small oscillations around the average inter-arrival time repeatedly active and deactivate request redirection, the system does not deactivate request redirection until the inter-arrival time reaches the "service threshold" T_S (at time t_3). The service threshold is typically a few times bigger than T_R.

3. Benchmarking

To study the performance of flash-crowd management systems in a reliable and systematic fashion we need a good benchmark. This chapter describes the characteristics of such a benchmark, and explains why the benchmark that we propose here meets all these demands.

Using the definition of a flash-crowd as a sudden growth in the request rate, a flash-crowd management system needs to detect this peak and then try to balance the increased load between the available web servers. Given this fact, the performance of a flash-crowd management system depends on

- how fast it detects the flash-crowd. The faster it detects it, the faster it will be able to react to it.
- how long it takes before the load is (almost) evenly balanced between the servers.
- how evenly the load is being balanced between the servers.

This means that we need a benchmarking program that is able to change the request rate directly to simulate the flash-crowd and to stress the web server enough to get it overloaded. However, most existing benchmarking programs allow one to control the concurrency degree (amount of threads sending requests concurrently), but not the request rate. For instance Flood and HammerHead are scenario-based benchmarks for web servers that will hit URLs from a list in sequential order [8][10]. Although one can change the load on the web server by altering the number of used threads, this will not get the server into an overloaded state, because the request rate will effectively be adjusted to the capacity of the server; only a slowdown will be noticed. On the other hand, certain programs (like the Apache benchmark tool ab) do give the operator the option to set an even request rate. However, they will send the same requests over and over again which does not produce realistic results.

Besides the above requirements, to make a benchmark useful to the user in general, and to create credible and meaningful results, the benchmark has to provide the following non-functional properties:

- be reproducible. If the results of the benchmark are not reproducible the results will have no meaning.
- be representative. If the benchmark is not representative for a real system, the results will have no meaning for a real situation.
- be scalable. This will make the benchmark useful to a whole array of differently sized systems.

Clearly, a good benchmarking tool should not issue the same single query again and again. Section 4.2 therefore contains details about how to create a representative list of requests to be addressed to the benchmarked system. Then, Sections 4.3 to 4.6 describe various methods to issue requests, from a simple but naïve implementation to the actual tool, which was used in the remaining of this thesis.

3.1. Creating a representative list of requests

To ensure that benchmarking gives realistic results, the benchmarking program needs to issue requests for different documents. In actual real world deployment the systems would be getting requests for a whole array of different files as well.

Earlier research has shown that the file size of requested documents follows a Pareto distribution (see Figure 3.1 and Figure 3.2) with a median average file size of 2000 bytes, and a mean average file size of 4000-6000 bytes [3][4][13]. A Pareto distribution is a heavily tailed probability distribution that is found in a large number of real-world situations [16]. If X is a random variable with a Pareto distribution, then the probability that X is greater than some number x is given by:

$$P(X > x) = \left(\frac{x}{x_{\mathrm{m}}}\right)^{-k}$$

for all $x \geq x_m$, where x_m is the (necessarily positive) minimum possible value of X, and k is a positive parameter.

The popularity of documents follows a Zipf (see Figure 3.3 and Figure 3.4) distribution. Zipf's law [17] states that

$$P_n \sim 1/n^a$$

where P_n is the frequency of item n and ^{a} is almost equal to 1.

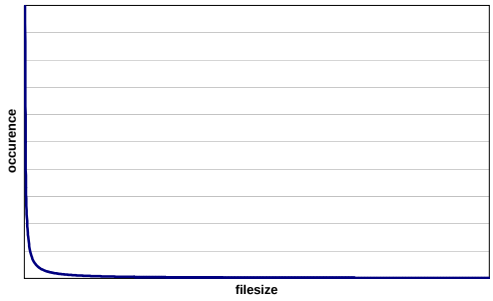

Figure 3.1: filesize versus occurrence, a Pareto distribution

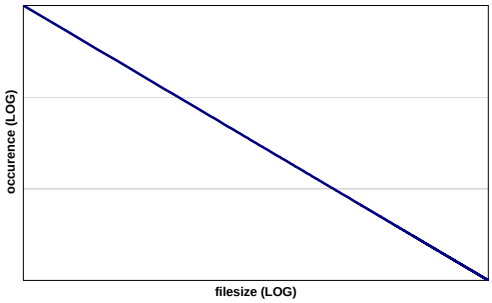

Figure 3.2: filesize versus occurrence on a LOG/LOG scale

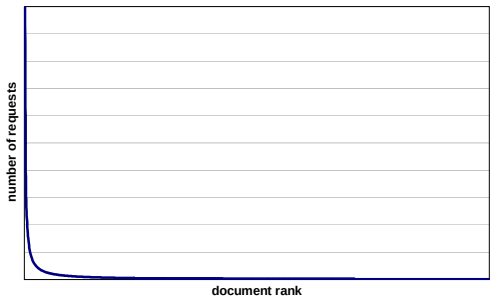

Figure 3.3: document rank versus number of requests on a normal scale

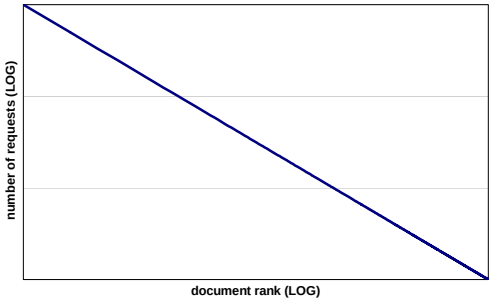

Figure 3.4: document rank versus number of requests on a LOG/LOG scale

The reason that the figures of the Pareto distribution and the figures of the Zipf distribution look so much alike is that the Zipf distribution is essentially a discrete version of the Pareto distribution [1]. The x-axis in Figure 3.3 and Figure 3.4 is continuous, while bins are used in Figure 3.1 and Figure 3.2.

A request list can be constructed in two ways. One can implement algorithms describing a Pareto and a Zipf distribution to determine the file size and popularity of documents respectively. Then these algorithms can be used to construct the request list. This method is called *synthetic benchmarking.* On the other hand one can use existing request traces. A log file is taken from a web server and used to replay the requests. This is called *trace-based benchmarking.*

This means there is a trade-off to be made between flexibility and realism. The first method gives enormous flexibility; one can decide exactly what the list will look like and how long it is going to be. This is a lot harder to do when using actual traces taken from a web server, but one can however expect the results to be more realistic. We decided to use the first method to give me the flexibility we need to simulate a flash-crowd. To make the results more realistic we generated the request list based on statistics extracted from actual web server log files.

We started by taking the log file from the web server of the Computer Science Department of the Vrije Universiteit Amsterdam (http://www.few.vu.nl). See Table 3.1 for more information about this log file. Figure 3.5 shows how document rank relates to the amount of requests on a LOG/LOG scale.

Vrije Universiteit log file	
date log file	April 23 2005
number of requests	2,863,248
number of successful requests	2,671,887
number of unique documents	320,099
percentage of requests for top 10 files	7.50%

Table 3.1: Vrije Universiteit log file properties

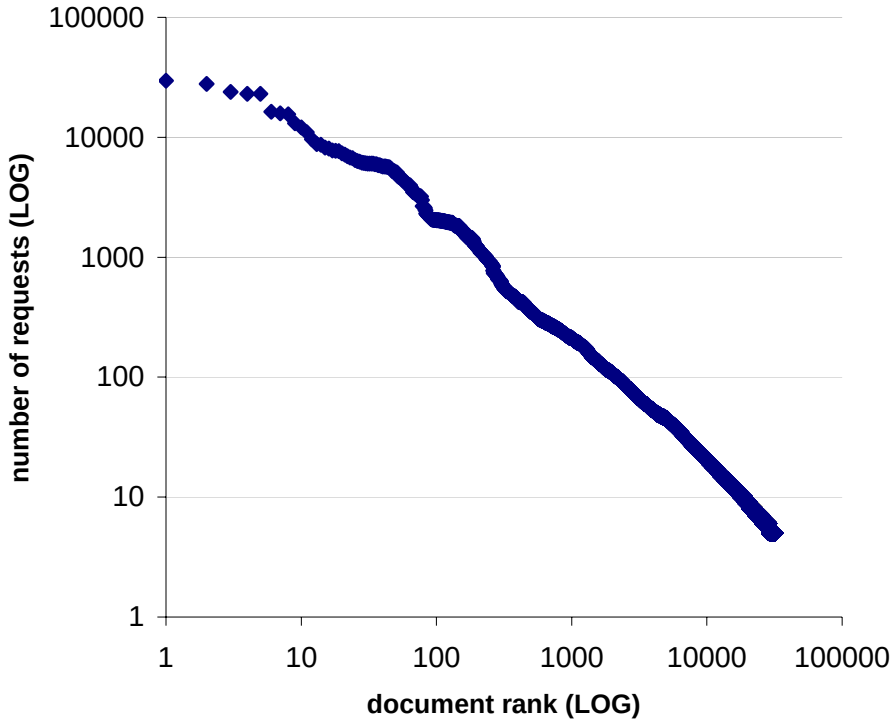

Figure 3.5: document rank versus number of requests for the Vrije Universiteit log file, on a LOG/LOG scale

If we compare this figure with Figure 3.4, we can see that this log file clearly follows a Zipf distribution. This means that this log file is representative for real world use of a web server, as far as document popularity distribution is concerned.

The same type of comparison can be made for the distribution of the file size of the requested files. Figure 3.6 shows file size versus occurrence. And again is shown that this log file shows the expected behavior if we compare Figure 3.6 with the model behavior of Figure 3.2.

We however decided not to base the benchmark on this trace. The reason is that even the most popular documents get only a relatively moderate amount of requests (as can be seen in Table 3.1). This shows that the VU website may simply not need replication of documents. Should its load exceed a single server capacity, all that would be needed is distribution of the documents over multiple web servers. However, real flash-crowds receive requests for only a few documents, which requires replication. We therefore need a log file containing fewer documents, so that each one accounts for a larger fraction of the total. This way the flash-crowd management system will be forced to replicate documents and it is possible to evaluate how well it does this.

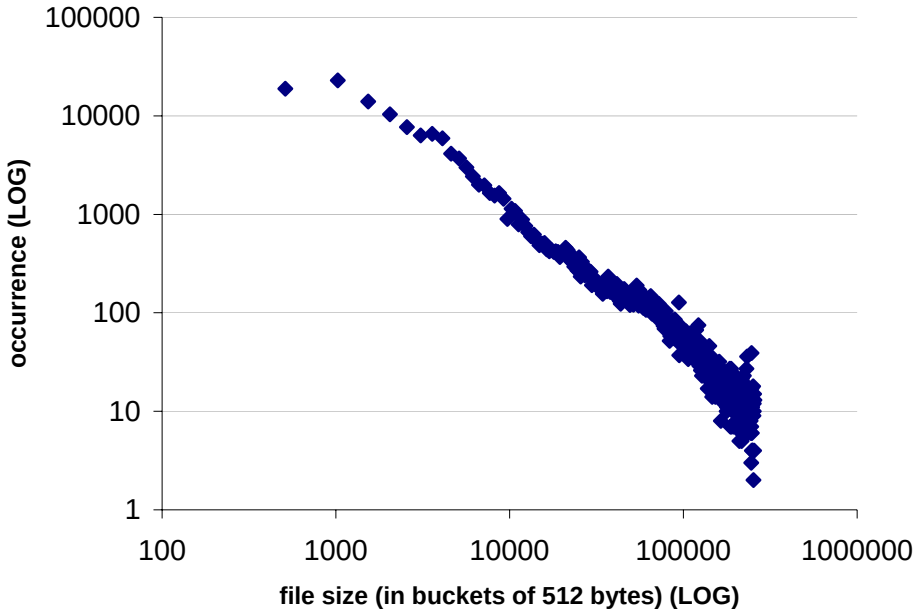

Figure 3.6: file size versus occurence of the Vrije Universiteit log file, on a LOG/LOG scale

A better log file for the purpose of testing flash-crowd management systems is the log file of the Minix3 web server. With the release of version 3 of the MINIX operating system the web server experienced a huge amount of requests for a relatively small number of documents. Table 3.2 shows more information about this log file.

The reason why the total number of requests and the total number of successful requests are equal here is that the Globule server used to host this site only logs the successful requests. This is not a problem for us because only the successful requests are used anyway.

The same procedure described earlier can be used to analyze this log file. Figure 3.7 shows document rank versus number of requests.

Minix 3 log file	
date log file	November 15 2005
number of requests	606,997
number of successful requests	606,997
number of unique documents	507
percentage of requests for top 10 files	78.8%

Table 3.2: Minix3 website log file properties

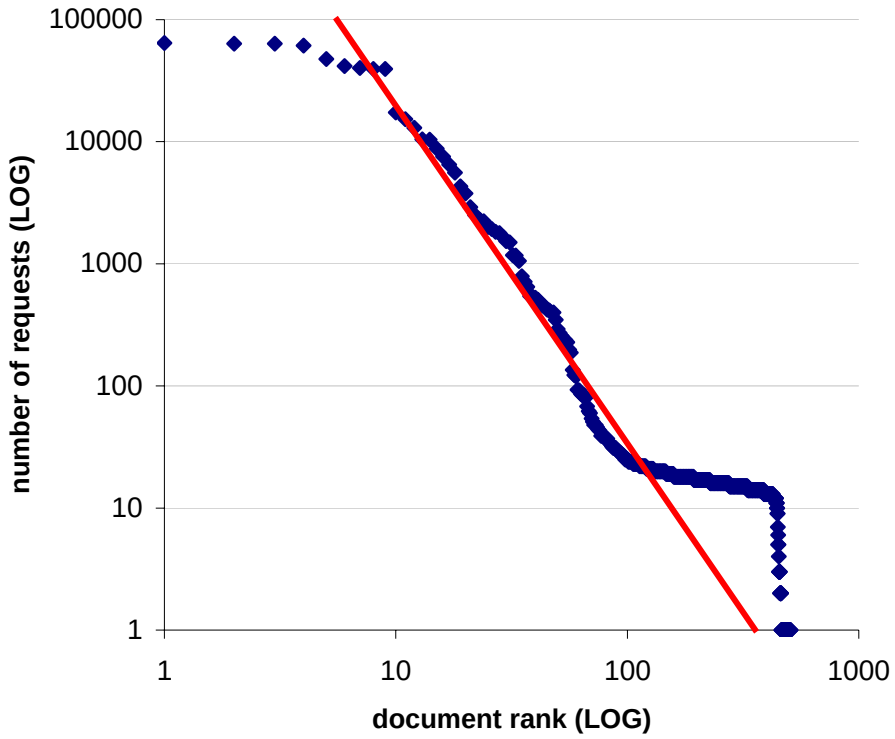

Figure 3.7: document rank versus number of requests for the Minix3 website log file, on a LOG/LOG scale

Although not as clear as with the Vrije Universiteit website log file, the beginning of this curve still follows a Zipf distribution whereas the end deviates from it a little bit. Because this only involves a few requests (on a total of 606,997) this is not of very much importance.

Figure 3.8 shows the file size distribution of this log file. As can be seen there are a lot of small files as opposed to only a few larger files. This is exactly the behavior of a Pareto distribution (see Figure 3.2).

Because replaying the entire log file for every benchmark would take an enormous amount of time, we instead used the data about document popularity to create a new shorter request list. The choice for a certain document for each entry in this list is based on the probability function:

$$Pd = \frac{\text{number of requests received by } d}{\text{total number of requests in the log file}}$$

This way it is possible to generate a request list as big, or as small, as necessary.

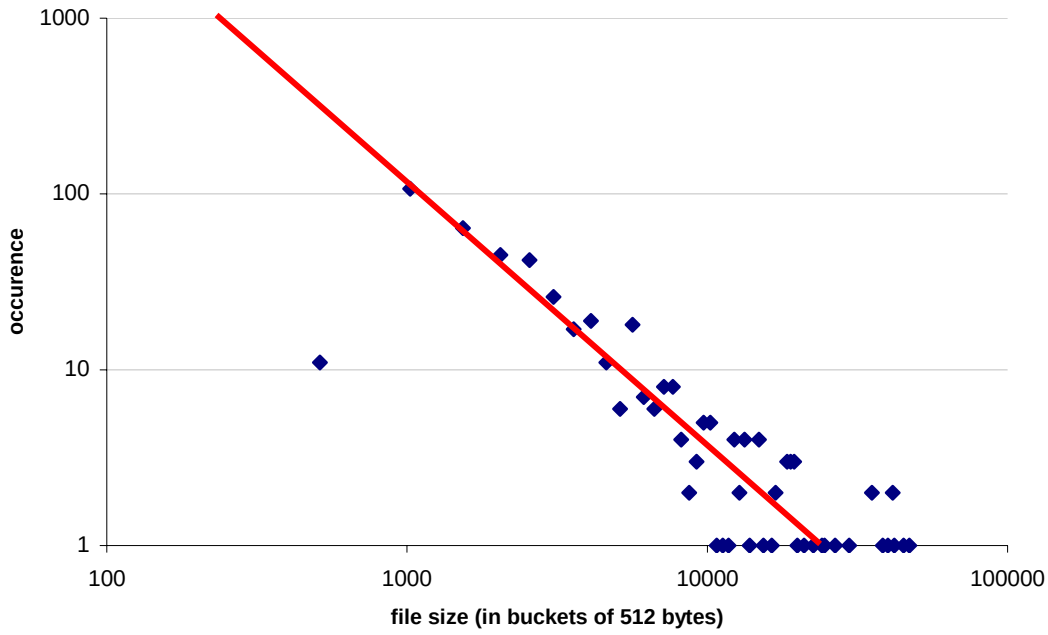

Figure 3.8: File size versus occurrence of the Minix3 website log file

The next thing after creating a request list was coming up with a method to decide how exactly the requests should be sent. Because the results of the benchmark have to be reproducible there has to be some sort of regularity in it. Also, changing the timing can be used to influence the load on a system.

3.2. Method 1: Finish request after request, as fast as possible

The first method we considered spawns a number of workers (threads), each of which starts sending requests sequentially from the request list, as fast as possible. A worker will not send a new request until its last request is completed and the answer has been received. This method is similar to the ones used in Flood and HammerHead.

In this method one can control the number of workers, thereby controlling the load on the system. This means that one can fix the concurrency degree. A metric like “number of workers” can now be used to quantify the load on the system during a certain benchmark.

It is important that every worker does not start at the top of the same request list. Otherwise, the system would receive an amount of requests for the same document (the one on top of the list) equal to the number of workers. The same thing would happen with the second document on the list, etc. Because web servers store a few recently requested disk blocks in memory, implementing method 1 like this would have the undesirable side-effect that most documents are fetched from memory

instead of disk. A solution to this is to let every worker start benchmarking at a different offset in the request list using the following algorithm:

$$O_x = x * (R / W)$$

O_x denotes the offset from the start position for worker x
R denotes the total number of requests in the request list
W denotes the total amount of workers

Figure 3.9 shows an example of determining the offset from the start position for 3 workers.

Another solution would have been to start a worker only after the worker that was started before this worker has already sent a few requests (see Figure 3.10). The only problem with this is that is really hard to determine at what moment in time all the workers are actually sending requests. Because of this problem we used the solution described earlier for this method.

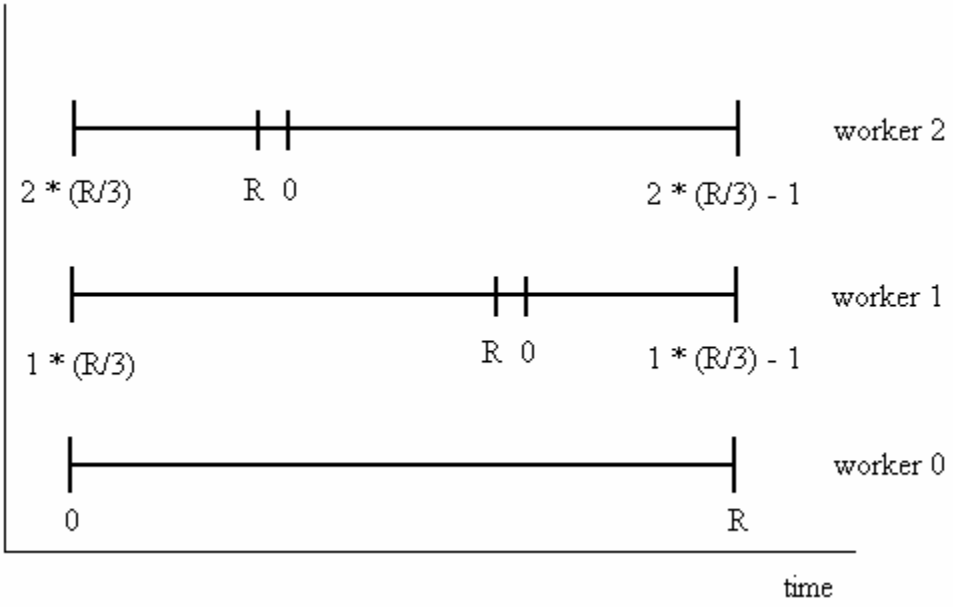

Figure 3.9: example of determining the offset for 3 workers

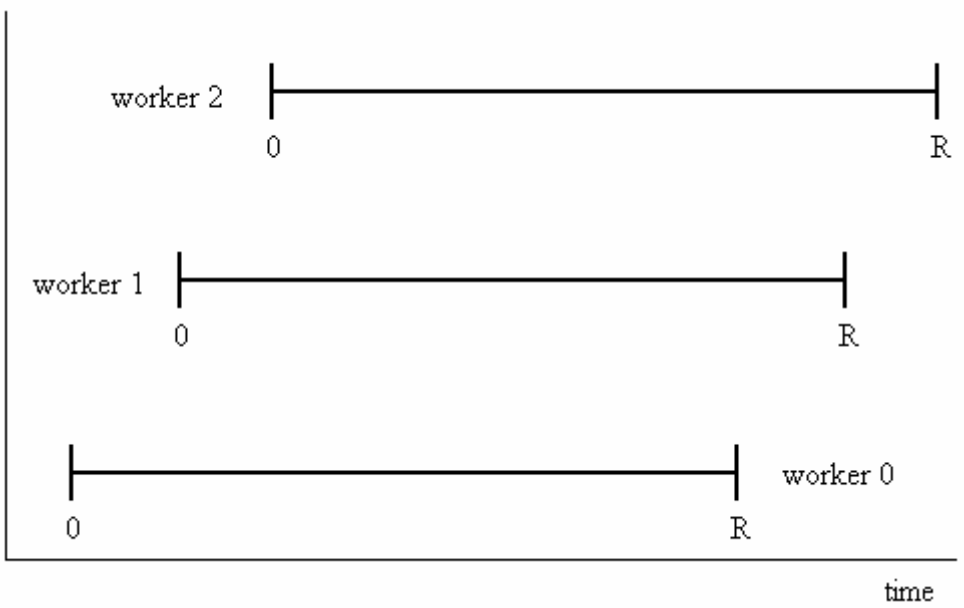

Figure 3.10: Another (bad) solution to ensure that workers do not send requests for the same file at the same time.

Figure 3.11 shows the results of this method when it is being used for benchmarking an Apache web server. We notice that the server does not show any limit in its processing capacity but simply responds slower and slower as the concurrency degree increases. The reason is that as the server slows down the workers will slow down as well, because they will wait for the answer of a request before sending a new request.

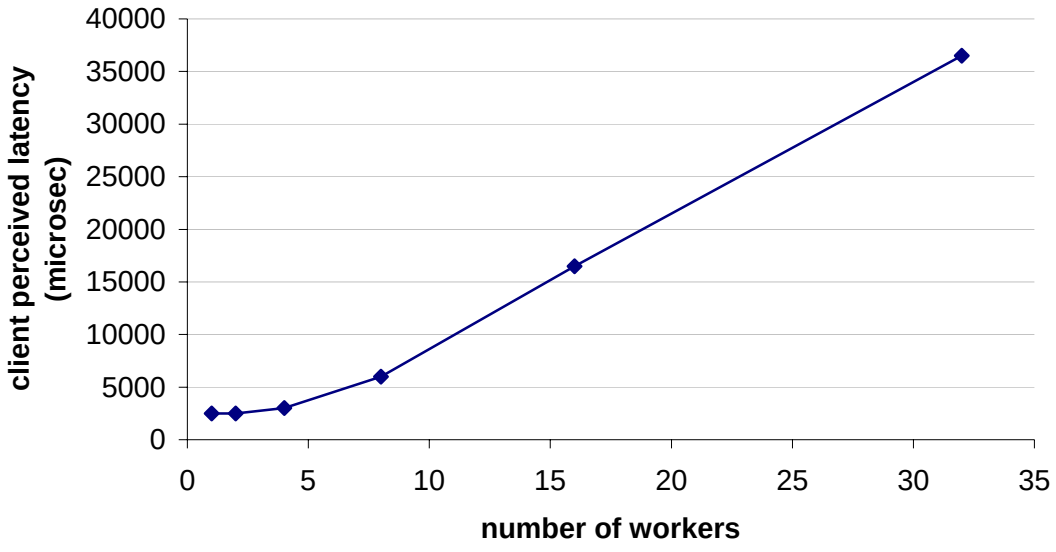

Figure 3.11: benchmarking an Apache web server using method 1

This behavior does not correspond to real situations of server overload. In a real flash-crowd, users will keep sending their requests regardless of the load on the system, which effectively can bring a server to a halt. We therefore need a better method for sending requests, in which it is possible to select the request rate rather than the concurrency degree.

3.3. Method 2: Fixed amount of requests per second, colliding requests

To address the issues of the first method, the second method tries to control the request rate rather than the concurrency degree. It spawns a number of workers, but instead of sending requests as fast as possible, each worker will send exactly 1 request per second. The same algorithm for determining the offset from the start position of the request list as was used by the first method is used here as well.

Now the number of workers can be used to specify the number of requests that should be sent each second ("number of workers" and "requests per second" essentially mean the same thing in this method).

In this method, we need to make sure that all workers will not send their requests at the same moment, leading to the server receiving a burst of requests every second. To get an even distribution of the different requests over the 1 second timeslot, the time at which a worker will send a request is determined by:

$$T_x = T_0 + (x * (1000000 / W))$$

T_x denotes the time worker x will send its request (in microseconds)
T_0 denotes the absolute start time of the time slot (in microseconds)
W denotes the total amount of workers

Figure 3.12 shows the result of this method when it is being used for benchmarking an Apache web server. As can be seen from the figure this method has accomplished exactly what it was designed for: it is able to push the web server into an overloaded state. However, problems with this method arise when several client machines are used for spawning worker threads. This may be necessary to be sure that the machines used by the benchmarking tool are not the bottleneck in the whole benchmarking setup.
When a multitude of machines are used, starting times of requests will collide, the exact same thing we tried to avoid by using different times for sending requests. This means that a little adjustment to this method is necessary.

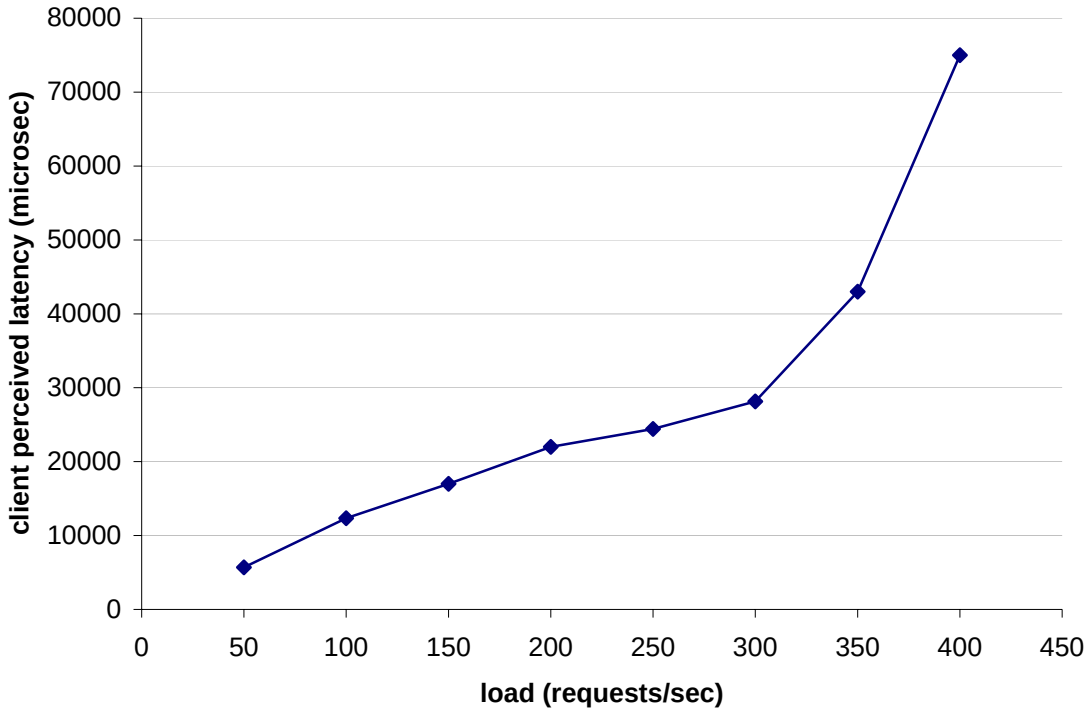

Figure 3.12: benchmarking an Apache web server using method 2

3.4. Method 3: Fixed amount of requests per second, interleaving requests

Method 3 is the same as the second method, with the small difference that when multiple machines are used by the benchmarking tool for spawning workers, they will agree on a time to start benchmarking and can now avoid sending requests at the exact same time as other machines. This requires a small adjustment to the algorithm:

$$T_x = T_0 + (x * (1000000 / W)) + (M_x * (1000000 / W / M_{tot}))$$

T_x denotes the time worker x will send its request (in microseconds)
T_0 denotes the absolute start time of the time slot (in microseconds)
W denotes the total amount of workers
M_x denotes the number of the machine worker x is on
M_{tot} denotes the total amount of client machines

This makes the request distribution extremely even, but it is actually so even that it is not representative anymore. Also, synchronizing the different machines is very hard. What we need is a new method that shows some fuzzy behavior (requests in a real situation are not perfectly distributed over time either) but does have reproducible results.

3.5. Method 4: Fixed amount of requests per second, random timing

Method 4 builds on the idea of sending a fixed amount of requests per second. This method will also spawn a number of workers, but instead of having them send exactly 1 request per second, the interval between requests will be selected randomly, with an average between 0.7s and 1.3s. After a few requests the total load per worker will average at 1 request per second, but with the added bonus that there is no need for algorithms to determine the exact time to send a request. To determine where in the request list to start, this method also uses the algorithm described earlier for method 1.

Because of the fact that a random number is picked, most requests will not collide, but sometimes they will, just as would be the case in a real situation. This makes the benchmark very representative of how a system would actually be loaded with requests. Because every worker will average 1 request per second after a while, the results are also very reproducible (as long as the same request list is used).

When multiple machines are used for the benchmarking tool, there is no need for communication between the machines; this makes the benchmark very scalable. Extra machines can be added to increase the load.

When being used on a LAN (where almost all the factors can be controlled) this method has all the properties of a good benchmark.

4. Benchmarking RaDaR

Because an implementation of RaDaR was not available at the time this thesis was written, we had to implement a RaDaR-like system first. However, rather than merely imitate an existing system, we decided to adjust its architecture to make it perform better in the context of flash-crowds. First of all, because proximity and migration are less important than creating extra replicas in a flash-crowd, we decided to focus exclusively on replication and to ignore proximity/migration. Secondly, RaDaR was designed with scalability rather than adaptability in mind; in particular the redirecting algorithm is not using direct load feedback from the replica servers, which harms adaptability. In their ACDN research where scalability was less of an issue, the RaDaR authors used another algorithm that does include load feedback. Because of the importance of adaptability when a flash-crowd occurs we decided to use the ACDN redirecting algorithm in our RaDaR system instead. Similarly, we used the ACDN replica placement algorithm. The last change we made in the RaDaR architecture that will benefit performance is to use redirecting instead of proxying when delivering web documents to clients.
This chapter is structured as follows: Section 4.1 describes our implementation in detail. Section 4.2 describes the results of benchmarking RaDaR using the synthetic benchmark. Then, Section 4.3 shows the results of replaying a trace of an actual flash-crowd.

4.1. Implementing RaDaR

While implementing the RaDaR-like system we used as many existing components as possible. Software like Apache and Squid have a solid and stable implementation that have been used in production for many years; using them in the RaDaR system will ensure a high implementation quality for the various parts.

As shown in Figure 4.1, the RaDaR system is implemented in a four level hierarchy. Client requests are sent to the top level, and then they travel down the tree.

The first level consists of the distributors. A distributor receives incoming requests and then selects the appropriate redirector for the requested web document based on a hash function. The distributors are implemented as an Apache web server with a custom module that is registered with Apache as a filter. The default configuration of Apache 2 is to use a hybrid mode that uses both multiple processes and multiple threads. To make it easier to share routing information between the different instances of Apache, we configured Apache to use only threads.

In RaDaR, after selecting the redirector, the distributor forwards the client request to this redirector. The redirector then decides on which replica server to use and finally the replica server sends the document to the distributor, which will return it to the client. The big advantage of this form of proxying is that it is totally transparent to the

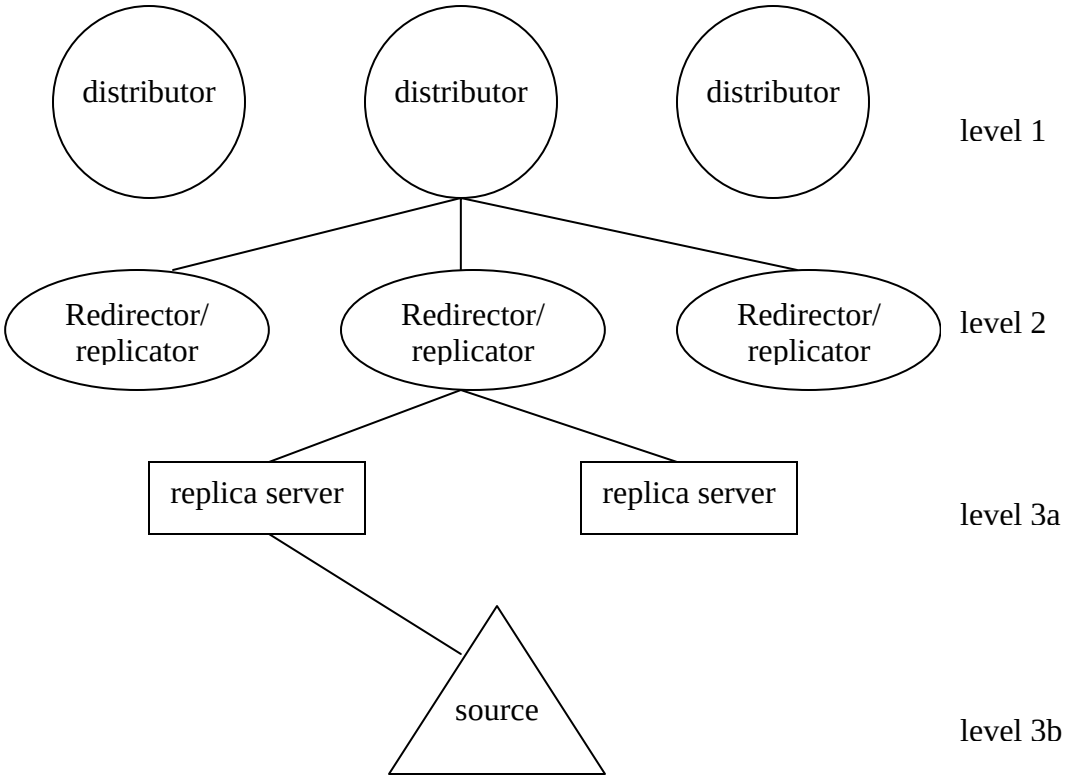

Figure 4.1: high-level view of the RaDaR-like system

user. We decided to use HTTP redirecting (using HTTP response code 302) instead though. The client is first redirected to a redirector by the distributor, which redirects the client to a replica server where the replica can be retrieved. This way we have lost transparency, but gained extra performance by relieving the distributors from the burden of proxying large amounts of data.

Redirectors and replicators make up the second level. Because the workload of a replicator is relatively low, and to simplify communication between redirectors and replicators, every machine from level 2 acts as a redirector as well as a replicator.
Each redirector/replicator-machine is in exclusive charge of a fraction of the URL space. It is responsible for monitoring the load of replica servers, and for making decisions on where to replicate documents and to redirect requests accordingly.
Similarly to the distributors, these machines are running an Apache web server with custom redirection modules. Periodically, the replica servers send load reports and offloading requests to the relevant redirector/replicator. These reports and offloading requests are implemented as simple HTTP requests for a specific URL, and are filtered out by the replicator module. The replicator then adjusts the load table that is stored in shared memory or initiates replica creation. In case of replica creation, the replicator picks the least loaded server not already hosting this replica and adjusts the redirection table also stored in shared memory.

The redirector module filters out the forwarded client requests and then picks a replica server to service the client using the ACDN redirection algorithm. When making redirecting decisions, this algorithm takes into account the load information that was stored by the replicator module. The original RaDaR research uses a non-feedback method for request redirecting that is not very suitable for the situation we are simulating here. The benefit of using the ACDN algorithm is that redirectors get exact information about the load of replica servers instead of an estimation. Because the focus of this thesis is on adaptability and not on scalability we do not have to worry about the drawbacks of the ACDN algorithm that were discussed in Section 2.3. After selecting a replica server the client will be redirected to this server.

The next level consists of machines that are running one Squid cache each, which will simulate the behavior of a replica server. The caches are configured as reverse proxies to a single Apache web server that will act as the source of all the web documents. Before benchmarking starts every document is requested once and is thus cached on a single machine; this machine will be seen as the origin server for that document. The Squid caches are large enough to hold all the documents, this way there will never be any swapping of documents. Documents are never refreshed either to minimize communication with the Apache server. After benchmarking has started, the only time the Apache server is used is when a replica has to be created. The way replica creation works is that a redirector will add the target replica server to its redirection table. The next time the redirector receives a request for this replica, it will send the client a HTTP redirect to the Squid cache where the replica has to be created, which will result in a cache miss. The cache will then reverse proxy the document from the Apache server, cache it (thus creating a replica) and then return it to the client. The first difference with RaDaR is that in RaDaR the target replica server would get the document directly from the replica server that initiated replication. The second difference is the moment of replication: in RaDaR the replica gets created immediately, where in our system this happens when the first request for the new replica is issued.
Besides running Squid, replica server machines also run a small standalone program that is used to monitor the load on the replica servers. This standalone program tails the Squid access log file and records load information per document. The average load is periodically reported to the replicators on the second level. When the standalone program notices the server gets overloaded (meaning that the load grows higher than the *HW*), it uses the ACDN Replica Placement Algorithm discussed in Section 2.2 to offload. However, instead of asking the central replicator for the least loaded server, the standalone program will use the same hash function used by the distributors to find the replicator that is co-located with the appropriate redirector for a given document. This replicator can then directly alter the redirecting table, where in RaDaR the replica server would have to tell the distributor to do this. Figure 4.2 and Figure 4.3 illustrate the difference between the two approaches.

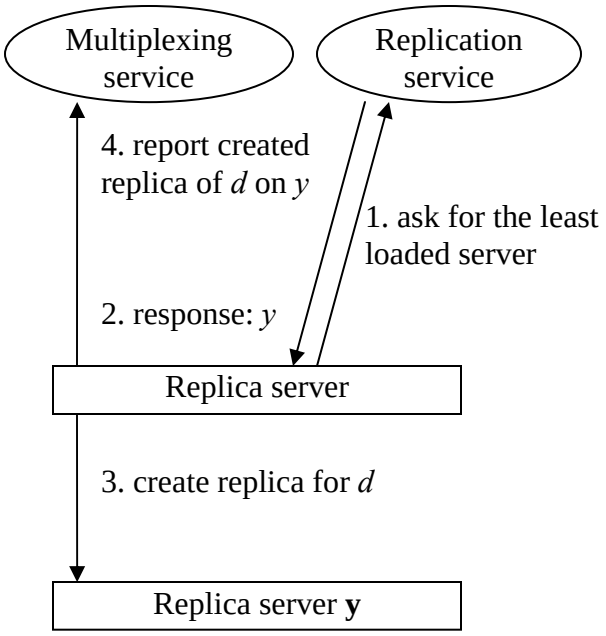

Figure 4.2: Replica creation in RaDaR

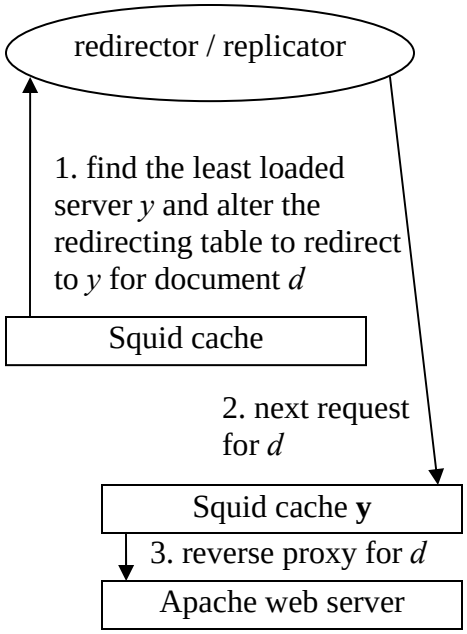

Figure 4.3: Replica creation in our implementation

4.2. Synthetic benchmarking results

For the benchmarking environment we used the Distributed ASCI Supercomputer 2 (DAS-2) [6]. The DAS-2 is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging, and was built by IBM. The DAS-2 consists of five clusters, located at five Dutch universities. The cluster we used contains 72 nodes and runs RedHat Linux. Each node contains:

- two 1-GHz Pentium III CPUs
- at least 1 GB of RAM
- a 20 GByte local IDE disk
- a Myrinet interface card
- a Fast Ethernet interface (on-board)

In Section 4.2.1 we show the results of benchmarking a single Apache web server to get some reference data. In Section 4.2.2 we describe the results of benchmarking the separate components of the RaDaR-like system. In Section 4.2.3 we study how to dimension the RaDaR-like system by putting an ever growing load on it. In Section 4.2.4 we show the results of using a step function to simulate a flash-crowd. And then, in Section 4.2.5 we show the effects of an instability of the content placement and request distribution algorithms.

4.2.1. Single Apache web server

To evaluate the performance of the RaDaR system, we first need a reference server and determine the maximum load it can sustain. The obvious choice is to use a single Apache web server as the reference server. To determine the maximum load we start by putting a low load on the server and then gradually increase it, while measuring the latency as perceived by the client. Latency is defined as the time spent from opening the connection with the server to closing the connection after the document has been received successfully. The load is increased every hour and the 90 percentile of the latency over the past hour is calculated. We choose to measure the 90 percentile instead of the arithmetic mean or the median, to make sure that accidental high or low values do not skew the measurement results, and to get the typical latency a client would experience when the server gets overloaded.
To make sure that the Apache web server is the bottleneck, and not the benchmarking client machine, we only let the benchmarking machine send a maximum of 100 requests per second and add extra client machines when the load is increased. The request list used is based on the log file of the Minix3 web site, and Method 4 is used for benchmarking. Figure 4.4 shows the results of benchmarking a single Apache web server.

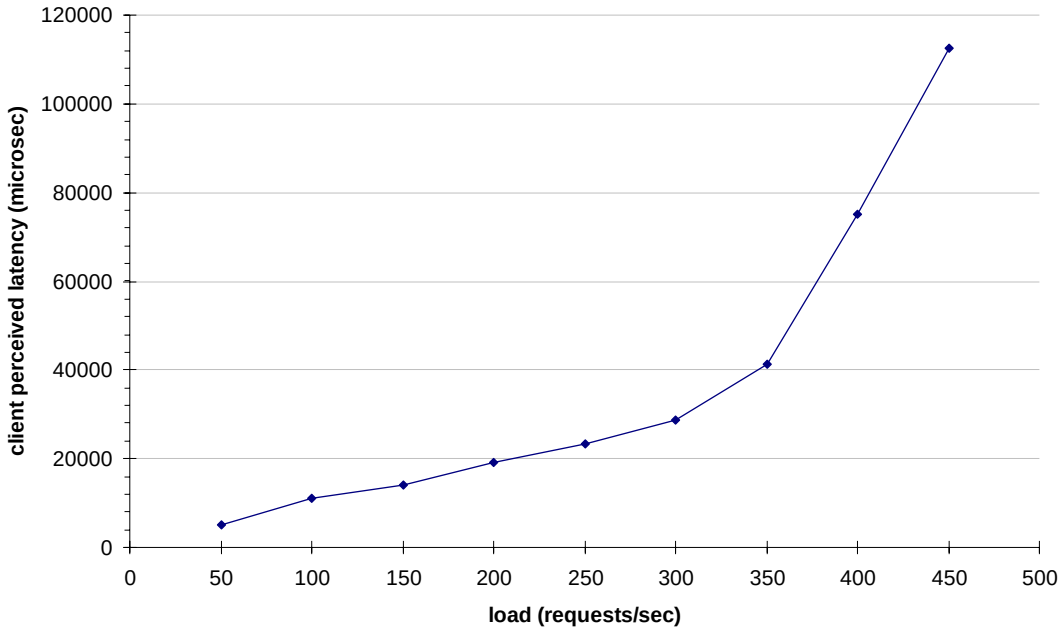

Figure 4.4: benchmarking a single Apache web server

As can be seen from the figure the server is underloaded up until 300 requests per second, and has no problem serving the clients. When the load reaches about 350 requests per second the server resources start to get saturated, and the server eventually starts to fail serving all incoming requests at around 450 requests per second. This teaches us that a reasonable load on the server lies around 300 requests

per second and that the absolute peak load is 450 requests per second. We also conclude that the peak latency lies around 100 milliseconds.

4.2.2. Benchmarking the separate components

To find out about where to place the low load watermark and the high load watermark, and to see the effect of the added redirections levels on the client perceived latency, one needs to benchmark the separate components of the RaDaR-like system. We start by benchmarking a system that contains only one replica server, using the same benchmark as in Section 4.2.1. Again we start with a low load and gradually increase it by adding extra benchmarking machines. The load is increased every hour and the 90 percentile of the client perceived latency over the previous hour is calculated. The results are shown in Figure 4.5.

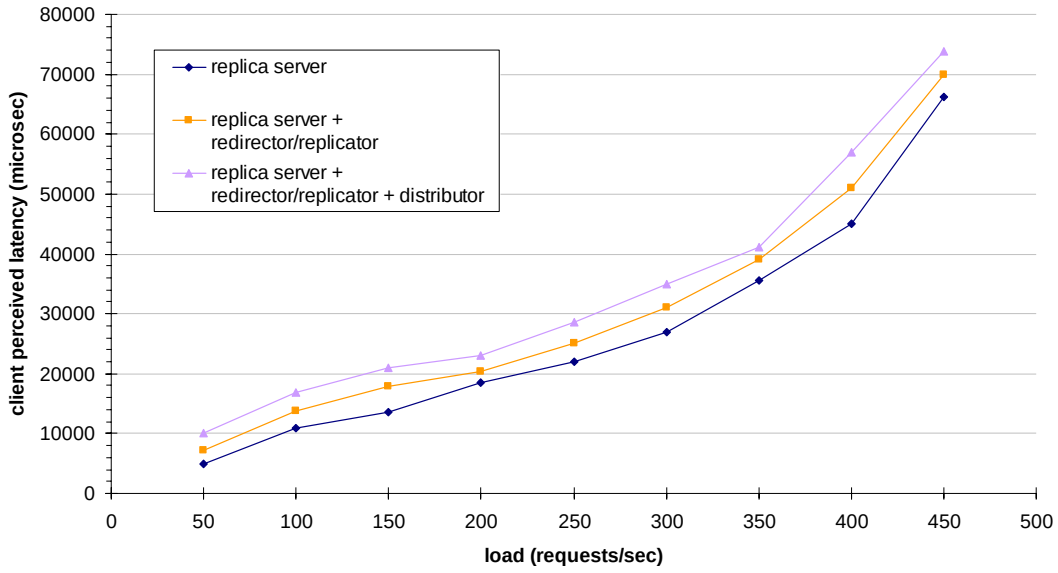

Figure 4.5: benchmarking a simple RaDaR-like system

As can be seen, the results are mostly similar to the results of the single Apache web server benchmark of Section 4.2.1. The client perceived latency scales linearly with the load up until the server resources start to get saturated around 350 requests per second. Just to be on the safe side, we decided to pick 300 requests per second as the high load watermark. A normal load for this replica server seems to be around 200 requests per second, so we take this as our low load watermark.

The next step is to benchmark a redirector/replicator. A redirector/replicator is added to the system, which will redirect all incoming client requests to the replica server. The benchmarking machines will now send HTTP requests to the redirector/replicator instead of sending them directly to the replica server. Again, the load is increased

every hour, and the 90 percentile of the client perceived latency over the previous hour is calculated. The results of this benchmark can be found in Figure 4.5.

Because of the previous benchmark, we know how much time it takes on average for a replica server to serve a single request. If we subtract this from the time it takes a redirector/replicator plus a replica server to serve a single request (which we measured in this benchmark), we know how much time the extra level of redirection (the redirector/replicator) adds to the total latency. As can be seen from Figure 4.6, the latency added by the redirector/replicator oscillates around 3 milliseconds. We can also notice a bit of a growing trend, but no clear maximum is shown. We conclude that a redirector/replicator can sustain a much higher load than a replica server. This benchmark does not succeed to put it to its maximum load.

Finally, a similar benchmark can be done for benchmarking a distributor. A distributor is added to the system, adding yet another level of redirection. This distributor redirects all incoming client requests to the redirector/replicator, which in turn redirects the requests to the replica server. Again the load is increased every hour by adding extra benchmarking machines, and the 90 percentile of the client perceived latency over the previous hour is calculated. The results of benchmarking this complete RaDaR-like system (one replica server, one redirector/replicator and one distributor) can be seen in Figure 4.5. To calculate the time the newest level of redirection (the distributor) adds to the total latency, we subtract the latency measured in the previous benchmark from the latency measured this benchmark. Figure 4.6 shows that the distributor, like the redirector/replicator, adds around 3 milliseconds.

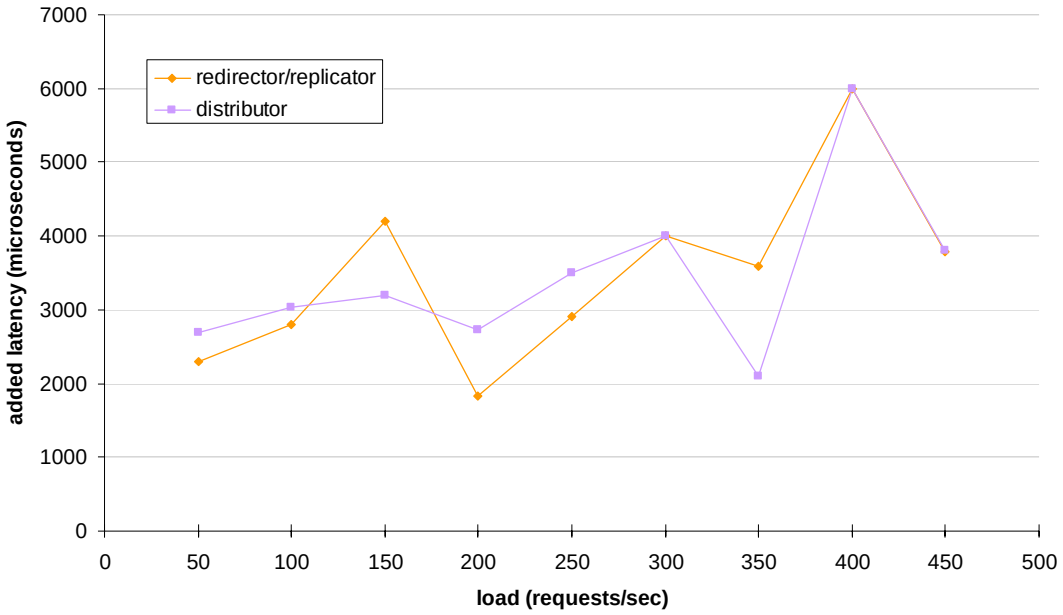

Figure 4.6: latency added by the extra levels of redirection

4.2.3. Scaling the RaDaR-like system

Because no real maximum load was found for the redirector/replicator or for the distributor in Section 4.2.2, one can conclude that the replica server becomes a bottleneck before the redirector/replicator or the distributor. This was to be expected, but leaves us with the question of how to dimension each layer of the RaDaR-like system for a given load, or for a given number of available machines.

To find out how to dimension a RaDaR-like system, we start with a simple system of one replica server, one redirector/replicator and one distributor. Again we load the system with a low load and then gradually increase it. The load is increased every hour by adding an extra benchmarking machine. We also calculate the 90 percentile of the client perceived latency over the previous hour. When the client perceived latency reaches a certain maximum, we consider the previous load put on the system as the maximum load that this system configuration can sustain. For the latency maximum we used 100 milliseconds, which was seen earlier as a latency peak in Section 4.2.1. When the system reaches its full potential, we change the system configuration by adding another replica server, and keep increasing the load. If adding replica servers does not help anymore to increase the maximum load a certain configuration can sustain, we know that either the redirector/replicators or the distributors have become the bottleneck. Now we can add a redirector/replicator or a distributor, or both of them. When a system configuration consists of multiple distributors, the benchmarking machines will be split up in groups, which each target a certain distributor, to simulate multiple DNS table entries. Figure 4.7 shows the results of this experiment.

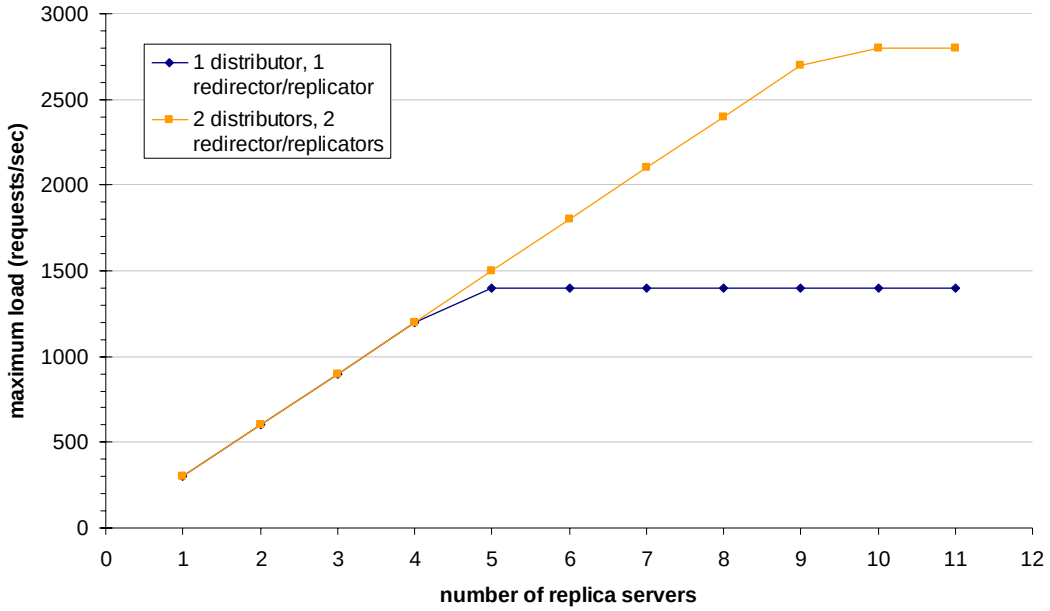

Figure 4.7: scaling the RaDaR-like system

As can be seen from the figure, the maximum load that a single redirector/replicator and a single distributor can sustain scales linearly with the number of replica servers up until four replica servers. Adding the fifth replica server adds only about 200 requests per second to the maximum sustainable load, while adding any more replica servers does not help the system at all. One can conclude from this that around 1500 requests per second, either the redirector/replicator or the distributor becomes the bottleneck of the system. However, adding just a redirector/replicator or just a distributor did not result into any noticeable gain in performance, which tells us that both the redirector/replicator and the distributor must have become bottlenecks. This means that the capacity of a redirector/replicator and the capacity of a distributor are almost equal, which lies around 1400 requests per second. More proof of this can be found when both a redirector/replicator and a distributor are added to the system, to create a configuration with two redirector/replicators and two distributors. Now the maximum load that can be put on the system scales linearly with the number of replica servers up until nine replica servers.

We conclude that one redirector/replicator is needed for every four or five replica servers. The number of distributors should be at least equal to the number of redirector/replicators, but because of the almost static nature of the distributor, they can be heavily replicated. In a very large system scenario the load reports of the replica servers to the redirector/replicators could become the only real bottleneck of this system as each replica server needs to report its load to most (if not all) redirector/replicators. However, each report only consists of a single HTTP request and needs to be issued only every few thousand client requests (depending on the report interval). We therefore have no reason to believe that the RaDaR-like system does not scale even to very high loads.

4.2.4. Performance during a flash-crowd

To see how fast the RaDaR-like system reacts to a flash-crowd, how fast it adjusts to it and how well it balances the load, we subject it to an artificial flash-crowd. We simulate this by using a step function with two intervals. During the first interval, the load put on the system is below the low load watermark. During the second interval, the load increases instantly to a level above the high load watermark. This is particularly interesting because the authors of ACDN never induced offloading in their own system by pushing replica servers above the high load watermark [14].
The experiment is conducted in a system with one distributor, one redirector/replicator and two replica servers. Initially one replica server (replica server 1) contains all documents, while the second replica server (replica server 2) is completely empty. The high load watermark and the low load watermark for the replica servers are 300 and 200, respectively (see Section 4.2.2). Replica servers send one load report to the redirector/replicators every 10 seconds. To keep things simple,

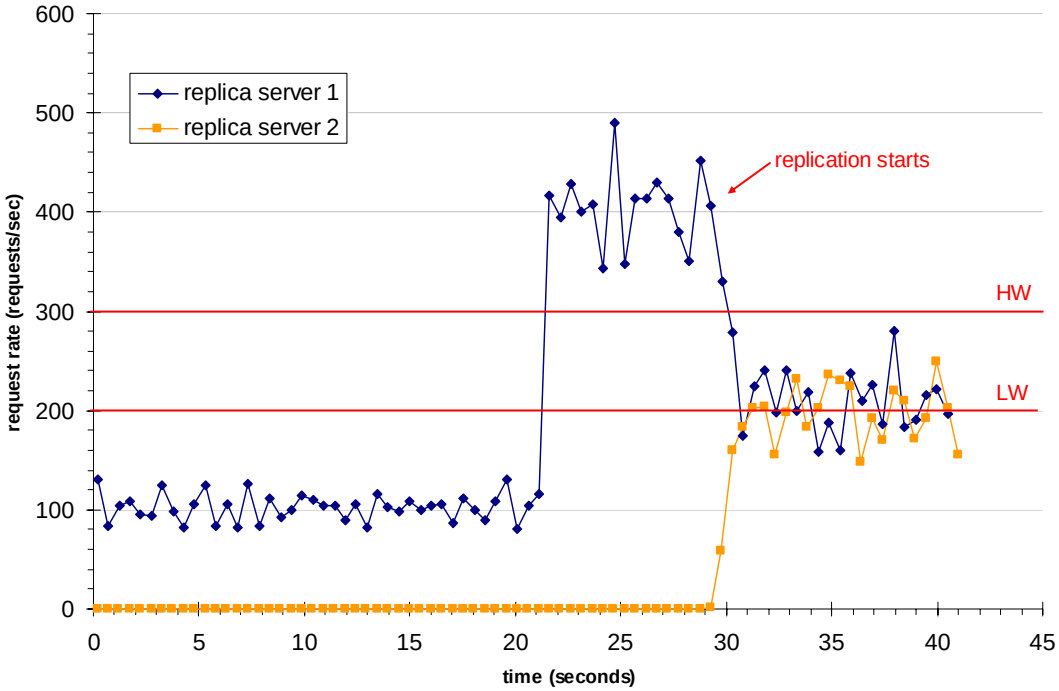

Figure 4.8: simulating a flash-crowd using a step function

we coupled the execution of the Content Placement Algorithm with the load reporting. Because we use no regions, and therefore cannot make use of the notion that every region is responsible for a certain number of requests, we use the crude estimate that creating a replica for a document will relieve the source replica server from 50% of the requests for the particular document. During the whole experiment we monitor how the load is balanced between the two replica servers. Figure 4.8 shows the transition from the first interval to the second interval. The figure plots the load on each replica server, measured in the number of requests per second, in sequential 0.5-second intervals. The flash-crowd begins around the 21st second.

As can be seen from the figure, the load on the first replica server is well below the high load watermark during the first interval of the step function. As a result of this no replication takes place, leaving the second replica server completely empty. Consequently the second replica server does not receive any traffic during the first interval of the step function. During the first eight seconds of the second interval, when the load is well above the high load watermark, the entire load is still on the first replica server, until the Content Placement Algorithm gets executed. The system now replicates the most popular documents (19 documents in total) to the second replica server, and the load on the two replica servers balances out in less than two seconds.

What we can gather from this is that the RaDaR-like system adjusts itself in a timely and efficient fashion to the flash-crowd once it gets detected; only the most popular documents are replicated and request redirection policies are updated in less than two

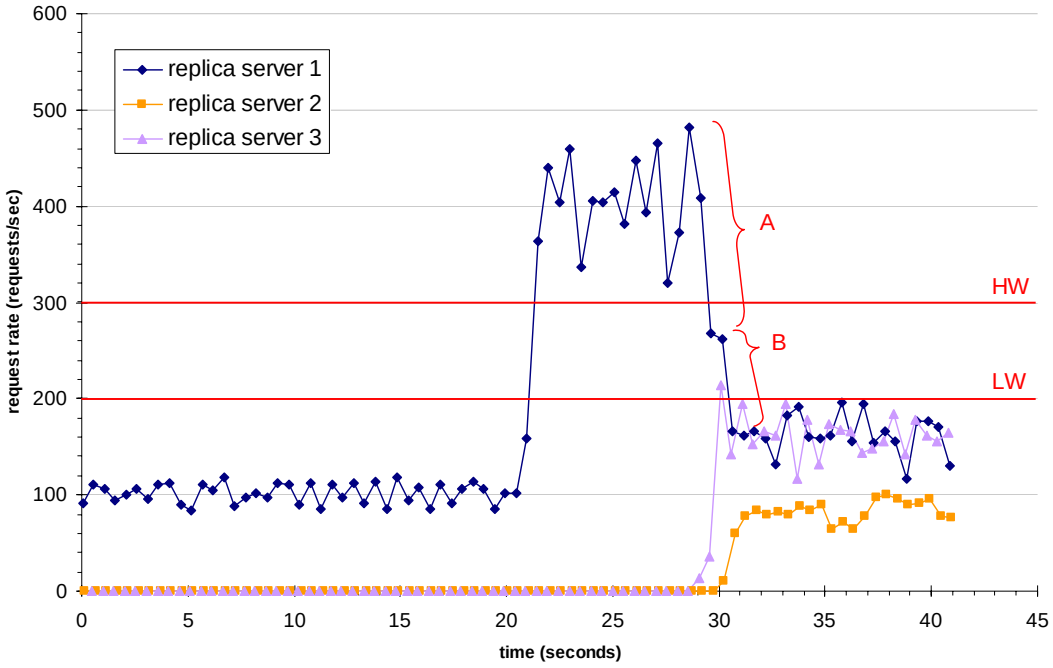

Figure 4.9: simulating a flash-crowd in a system with 3 replica servers

seconds. However, the time it takes the system to detect the flash-crowd heavily depends on how often the Content Replacement Algorithm is executed.

To take a closer look at how exactly the load gets balanced over the replica servers, we repeat the experiment but now use three replica servers instead of just two. Again, initially one replica server contains all the documents, while the other replica servers are empty. The results of this experiment can be found in Figure 4.9.

As can be seen from the figure the first replica server serves all requests during the first interval and during the first eight seconds of the second interval. When the system first detects the flash-crowd, it starts replicating documents solely to replica server 3. The figure clearly shows how the increase in traffic to replica server 3 results in a decrease in traffic of equal amount to replica server 1 (see item A in the figure). After the load of replica server 3 has been reported, replica server 2 is more attractive as a target for offloading, and therefore the rest of the documents is replicated to replica server 2. Again the decrease in traffic to replica server 1 and the increase in traffic to replica server 2 are clearly visible in the figure (see item B in the figure). The replication process stops when the load on replica server 1 falls below the low load watermark, which can also be seen in the figure.

4.2.5. Instability of the content placement and request distribution algorithms

When one closely examines the Content Placement Algorithm, it is immediately clear

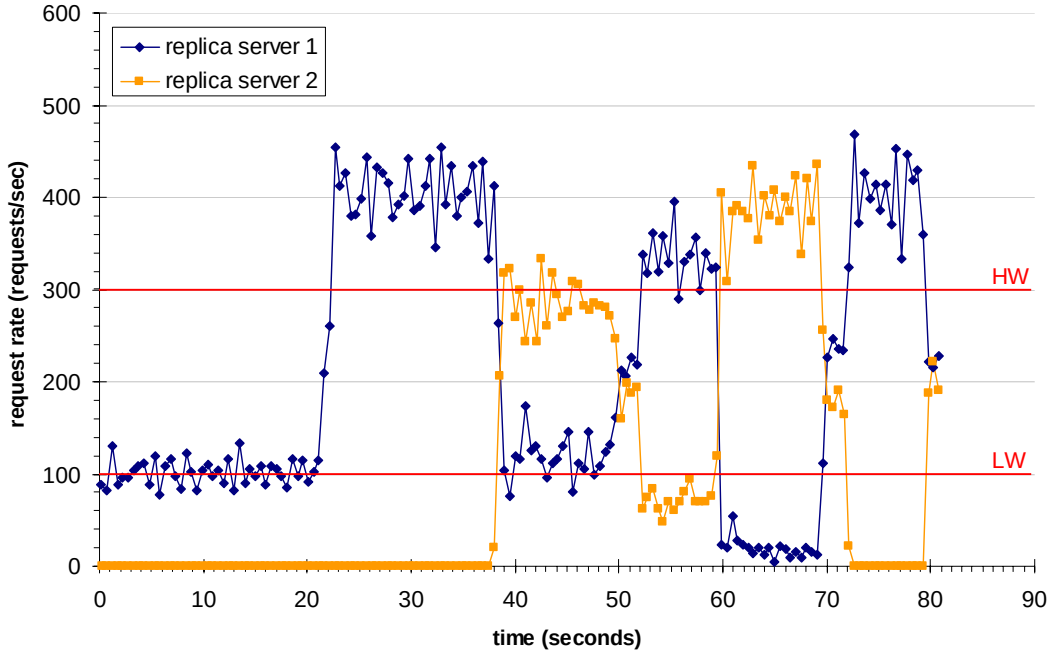

Figure 4.10: Offloading too much traffic, creating a herd effect

that the high load watermark determines the exact moment of offloading. It is however less obvious that the total amount of offloading depends on the low load watermark. This brings up the question what will happen if we set the low watermark low enough to offload an amount of traffic that would bring the fresh replica server into an overloaded state. To simulate this situation, we use a RaDaR-like system of one distributor, one redirector/replicator and two replica servers. Initially one replica server contains all the documents while the other replica server is empty. We leave the high load watermark at 300 requests per second, but lower the low load watermark to 100 requests per second. When the total load put on the system reaches 400 requests per second in the second interval of the step function, this results into offloading 300 requests per second to the second replica server, which should immediately put it into an overloaded state. Again, the report interval is set to 10 seconds, and the execution of the Content Placement Algorithm is coupled with the load reporting. The results of this experiment can be found in Figure 4.10.

As can be seen from the figure, around the 38th second, replica server 1 offloads 300 requests per second to replica server 2; this almost puts it into an overloaded state. After this load has been reported, the Request Distribution Algorithm tries to balance the load between the replica servers by sending most of the requests to replica server 1. This however leads to the load of replica server 1 exceeding the high load watermark again, which makes the situation even worse. Because servers with a load exceeding the high watermark get no requests redirected to them, almost the entire load is now put on replica server 2. The few requests still received by replica server 1, are due to the fact that replica server 2 is not hosting all of the documents yet. After

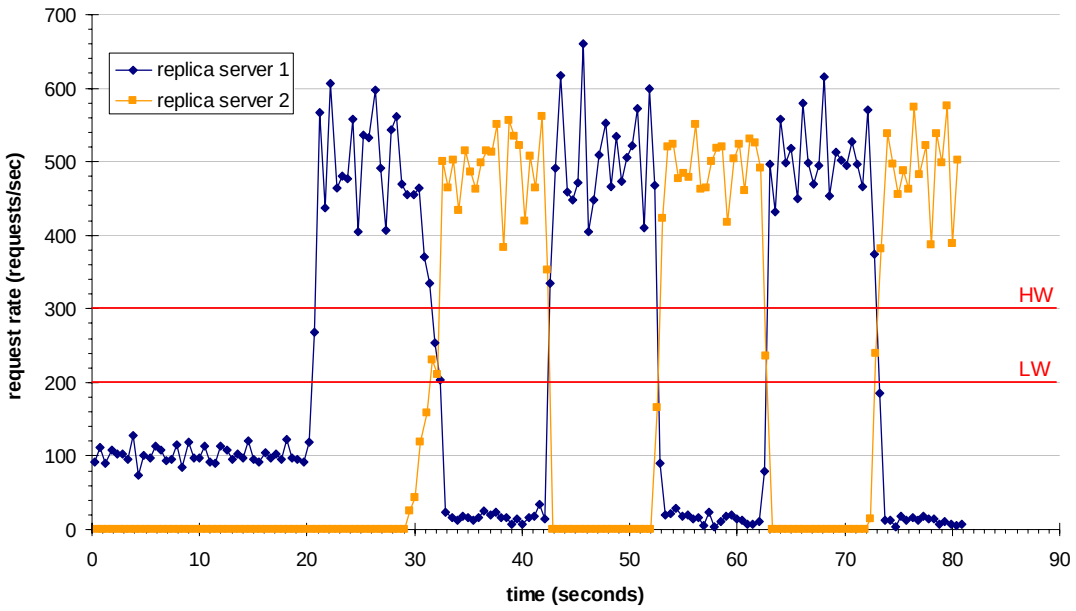

Figure 4.11: Putting a load of 500 requests per second on the RaDaR-like system

the next load report, the Request Distribution Algorithm redirects all requests to replica server 1 again, since the load on replica server 2 now exceeds the high load watermark. With one replica server in an overloaded state and one replica server in an underloaded state, the load will now keep bouncing between the two replica servers.

On a side note, to avoid this “herd-effect” the authors of ACDN propose the limitation that a replica server is only allowed to accept a request for replication if the predicted load after replication stays below the low load watermark. This however does not solve the problem shown above, but merely reverses it. In the situation above, where 2 replica servers should be capable of handling a load of 400 requests per second together, replica server 1 would only be allowed to offload a load of 100 requests per second (the low load watermark) to replica server 2. This brings the system into the exact same state with one overloaded replica server and one underloaded replica server.

The assumption that this problem can be avoided by selecting the right low load watermark is incorrect however, since the amount of offloading is determined by the difference between the total load on the overloaded replica server and the low load watermark. The system we used during the first flash-crowd experiment (with a high load watermark and a low load watermark of 300 and 200, respectively) can be brought into a state with a bouncing load as well by putting a total load of 500 requests per second on it, as can be seen in Figure 4.11.

To make the RaDaR-like system more responsive to a flash-crowd, one can make the replica servers report more often to the redirector/replicators. Trying to find a system

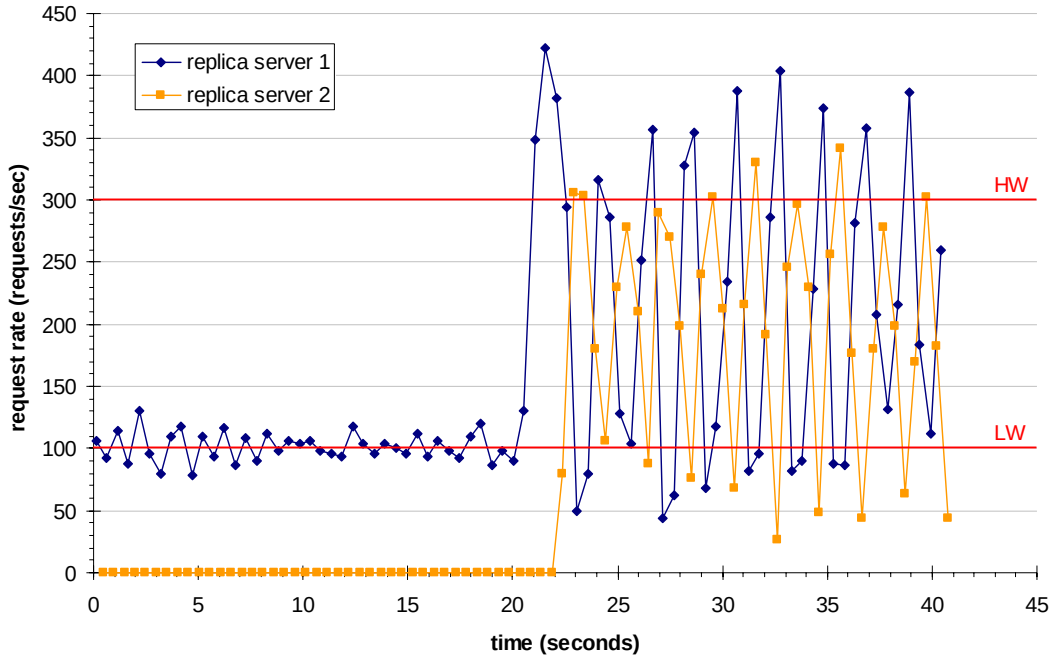

Figure 4.12: Reducing the report interval to 1 second

configuration where the system is able to recover from a state with a bouncing load by itself, we decided to run the same experiment with a low load watermark of 100, a high load watermark of 300, but with a reduced report interval of 1 second. Figure 4.12 shows the results of this experiment. As can be seen from the figure, sending more reports to the redirector/replicators does not help to avoid, or recover from, the herding effect. From this, and from the two experiments above, we can conclude that the herding effect is inherent to the part of the ACDN algorithm that handles offloading.

4.3. Trace-based benchmarking

To see how the RaDaR-like system performs in a real-life flash-crowd, and to validate the results from the synthetic benchmarking, we replay a trace from an actual flash-crowd which happened on the 3rd of February 2004. The reason of the flash-crowd was Google introducing a fractal-looking logo honoring Gaston Julia. After clicking on the logo, Google performed a search for images matching the term “Julia fractal”. The two most interesting resulting images on the top row of the list were on a server of the University of Western Australia, which abruptly experienced an enormous amount of requests for these images. Because the traffic targeting Google is much higher than any single server can sustain, the University server stood no chance, and failed.

Because, from our standpoint, the first sixteen hours of the day are not interesting, we only use the requests from the last eight hours of the day. During these eight hours the

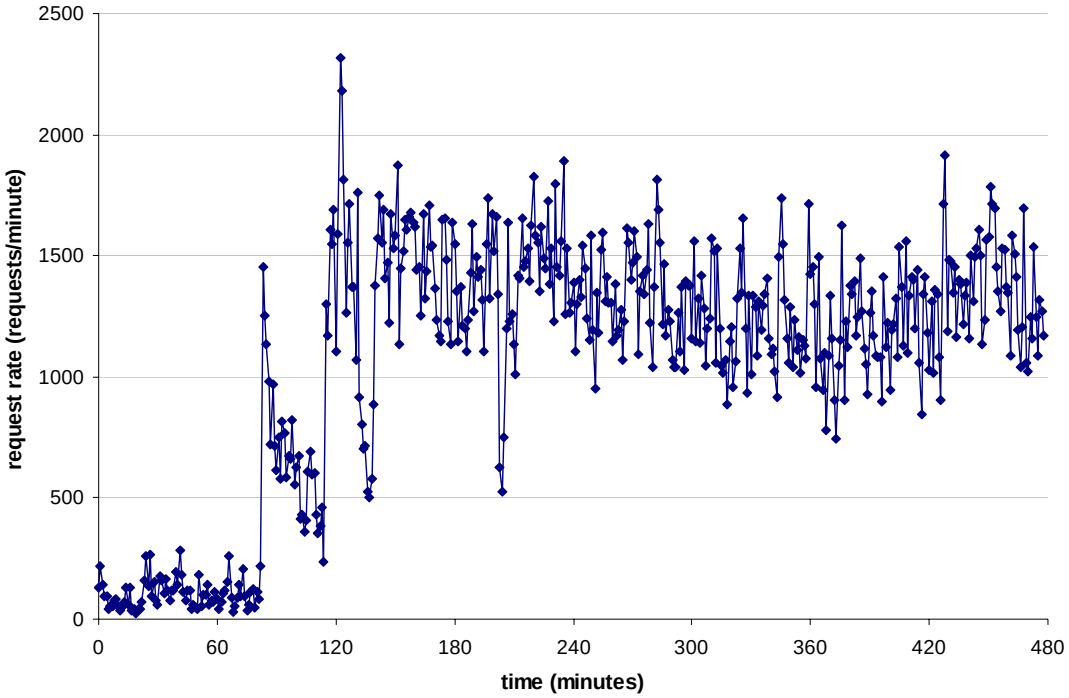

Figure 4.13: the fractals flash-crowd trace

server received a total of 511,292 requests for 18,507 unique documents. Figure 4.13 shows the request rate during this 8-hour period.

To be able to replay traces, our benchmarking tool first needs a few small adjustments. Every request in the request list is now associated with a timestamp, which defines when exactly the request has to be send to the RaDaR-like system. To make sure that the client machines that run the benchmark are not the bottleneck, we use eight synchronized client machines. The request list is divided over these eight client machines using a modulo function. Every client machine spawns a total of 200 workers that keep getting request-timestamp-pairs from the top of the request list. Instead of issuing exactly one request per second, the worker now waits until it is time to send the request, based on its timestamp; it sends the request and then gets the next pair from the request list. When we compared the load produced by benchmarking an Apache web server using the adjusted benchmarking tool with the original trace file, the results were almost an exact match.

Because replaying the trace in real-time would not put our replica servers in an overloaded state, we decided to speed up the trace by dividing the timestamps by four. The experiment is conducted in a system with one distributor, one redirector/replicator and two replica servers. The results can be found in Figure 4.14. As can be seen from the figure, the flash-crowd occurs around the 22nd minute. The RaDaR-like system reacts by replicating the 24 most popular documents to replica server 2. This results in a fairly well balanced state, with the two servers receiving an almost even amount of requests. The system stays in this state until the 48th minute.

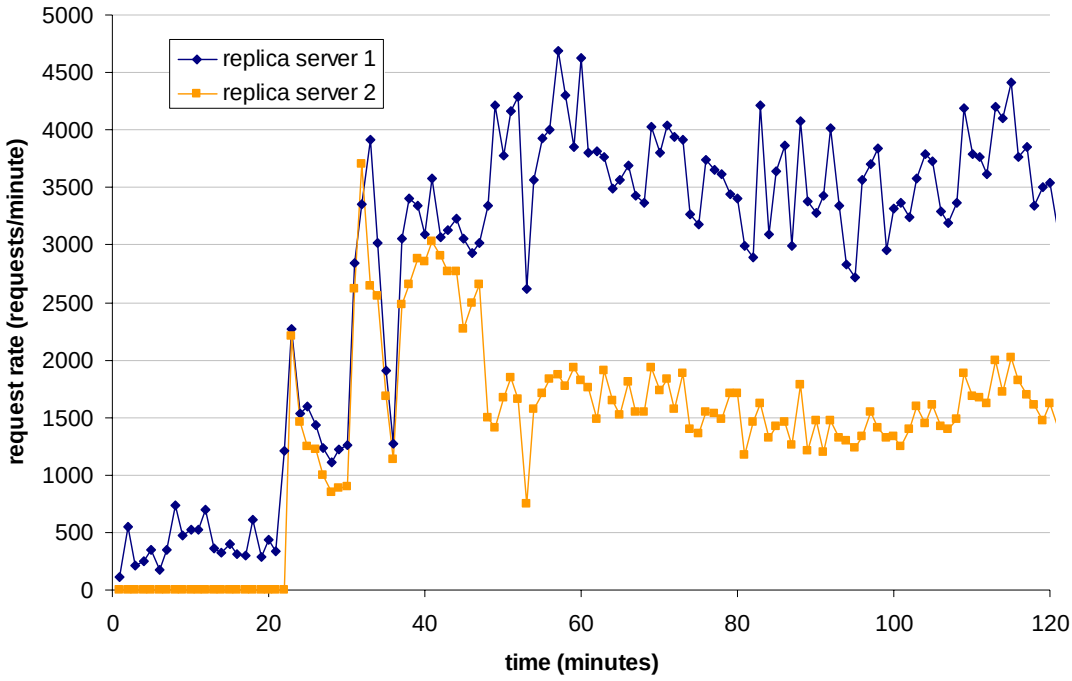

Figure 4.14: replaying the trace in the RaDaR-like system (speeded up four times)

What happens next is that the web server administrators noticed the enormous load on the server, and removed the images of the Julia fractals. They also replaced the index page with a 30K recovery page containing the Google logo, explaining what has happened to the server. From this time on, the most popular documents are the recovery page, the Google logo and the 404 error page. The RaDaR-like system immediately replicates these documents to replica server 2. However, because the users that are looking for images of Julia fractals are unsatisfied with the result, they start to look through other pages on the web server, resulting in a few thousand requests per minute for a few hundred different documents. Because no single document out of these documents is very popular by itself, none of them gets replicated. As a result, the load on replica server 1 remains at least twice as high as the load on replica server 2, which serves almost exclusively requests for the three most popular documents. It must however be noted that the load of replica server 1 remains well below the high watermark, which is the reason why the system does not adapt any further.

To take a closer look at what exactly happened during the first minutes of the flash-crowd, Figure 4.15 plots the load on the replica servers during the first two minutes in sequential 1-second-intervals. As can be seen from the figure, it takes about 9 seconds for the system to detect the flash-crowd. Then the state where the load is almost evenly balanced over the two replica servers is reached in about 20 seconds.

Concluding from Figure 4.14 and Figure 4.15, we can say that the RaDaR-like system did not have any problem with the initial flash-crowd when the requests were distributed over a small number of documents. These documents were replicated and

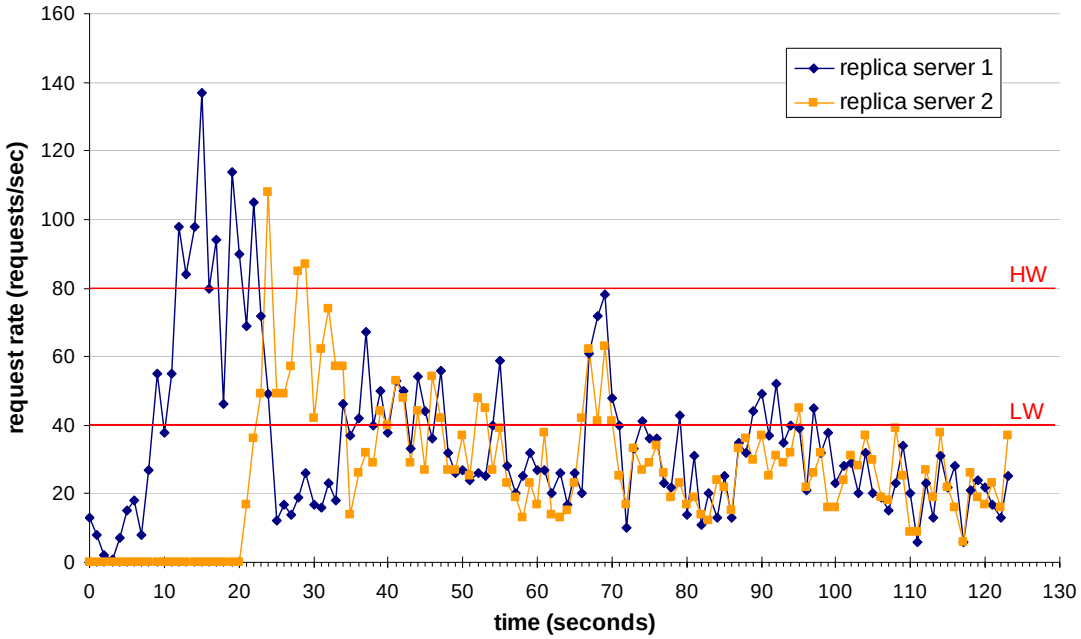

Figure 4.15: the first two minutes of the flash-crowd

the distribution of the load over the two replica servers was almost even. The system however rebalanced its load when the users started to look at a large number of different files.

5. Conclusion

Flash-crowds are becoming a growing obstacle to the further expansion of the Internet. When web servers are not prepared for the enormous growth in demand, users will experience significant delays and eventually even complete failure of the web server. Even when the flash-crowd can be predicted it is difficult to predict the exact magnitude of the extra demand and to react in time. One of the solutions to this problem is to replicate the most popular documents to different web servers, and to redirect client requests to these replicas. One promising system to handle flash-crowds is RaDaR, as it uses multiple levels of machines to redirect client requests, and can create new replicas on the fly.

In this thesis, we first proposed a benchmark that is representative for real world use of a web server. Most existing benchmarks only allow one to change the concurrency degree directly, but not the request rate. Because the request rate will effectively be adjusted to the capacity of the web server, these kind of benchmarks will not get the server into an overloaded state. On the other hand, benchmarking programs that do give the operator the option to set an even request rate merely allow to send the same request over and over again, and therefore do not produce realistic results. The benchmark we proposed combines the best of both worlds: the operator can directly set the request rate and a scenario is used to send the requests.

Secondly, we present the results of a performance analysis of a RaDaR-like system using the benchmarking tool we proposed. Because RaDaR is designed with scalability in mind rather than adaptability, we used a RaDaR-like system based on algorithms by another system, ACDN, by the same authors as RaDaR. This performance analysis goes much further than the original RaDaR measurements. First, we benchmarked the separate components of the system to measure the performance of each component. Secondly, we put an ever growing load on the system and added components when necessary. With the results of these two experiments we can answer the question of how to provision a RaDaR-like system. In our experiment one redirector/replicator and one distributor are needed for every four to five replica servers. To see how fast the RaDaR-like system reacts to a flash-crowd, how fast it adjusts to it and how well it balances the load, we then simulated a flash-crowd by using a step-function. Finally, we analyze the performance of the RaDaR-like system in a real-life flash-crowd by adjusting the benchmarking tool to replay requests from a trace of an actual flash-crowd.

From our experiments we conclude that RaDaR scales very well, even to very high loads. How fast the system detects flash-crowds heavily depends on how often the Content Replacement Algorithm is executed. During a flash-crowd our RaDaR-like system adjusts itself in a timely fashion. Once the flash-crowd gets detected, the system replicates the most popular documents and balances the load in around 2 seconds. We believe that this is fast enough to react to an actual real-life flash-crowd.

We did however discover an instability inherent to the parts of the replica placement and request distribution algorithms that handle offloading. In some cases, this instability leads to the load bouncing between the different replica servers.

Our study shows that flash-crowds are not a fatality, and that relatively simple mechanisms can allow a server to offload its exceeding demand to other rescue servers. The problems of handling flash-crowds are not solved entirely, however. In systems distributed across bandwidth-limited networks or hosting large multimedia objects, the mere creation of extra replicas can be considerably slowed down by the very flash-crowd that made it necessary in the first place. For such systems it becomes necessary to predict the upcoming flash-crowds such that replication can take place before the system is actually overloaded. Similarly, the RaDaR system only handles the replication of static documents. Replicating dynamic Web sites in front of a flash-crowd will undoubtedly require entirely new techniques.

References

[1] Adamic, L.A. “Zipf, Power-laws and Pareto – a ranking tutorial”. http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[2] Aggarwal, A., and Rabinovich, M. “RaDaR: A Scalable Architecture for a Global Web Hosting Service”. *The 8th Int. World Wide Web Conf*, May 1999.

[3] Arlitt, M.F., and Williamson, C.L. “Web Server Workload Characterization: The Search for Invariants (Extended Version)”. In *Proceedings of the 1996 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Philadelphia, PA, USA, pages 126-137, May 1996.

[4] Cunha, C.R., Bestavros, A., and Crovella, M.E. “Characteristics of WWW Client-based Traces”. *Tech. Report BU-CS-95-010*, Boston University Computer Science Dept, June 1995

[5] Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. “Globally Distributed Content Delivery”. *IEEE Internet Computing*, pages 50-58, September-October 2002.

[6] The Distributed ASCI Supercomputer 2 (DAS-2). http://www.cs.vu.nl/das2/

[7] Felber, P., Kaldewey, T., and Weiss, S. “Proactive Hot Spot Avoidance for Web Server Dependability”. In *Proceedings of the 23rd International Symposium on Reliable Distributed Systems,* Florianpolis, Brazil, pages 309-318, October 2004.

[8] Flood. http://httpd.apache.org/test/flood/

[9] Freedman, M.J., Freudenthal, E., and Mazieres, D. “Democratizing content publication with Coral”. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.

[10] HammerHead. http://hammerhead.sourceforge.net/

[11] Jung, J., Krishnamurthy, B., and Rabinovich, M. “Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites”. In *Proceedings of the 11th International Conference on World Wide Web 2002,* Honolulu, Hawaii, USA, pages 293-304, May 2002.

[12] Niven, L. “Flash Crowd”. *The Flight of the Horse,* pages 99-164, 1973.

[13] Pitkow, J.E. “Summary of WWW characterizations”. *Journal of Computer Networks and ISDN Systems,* volume 30(1-7), pages 551-558, April 1998

[14] Rabinovich, M., Xiao, Z., and Aggarwal, A. “Computing on the Edge: A Platform for Replicating Internet Applications”. *The 8th Int. Workshop on Web Content Caching and Distribution*, March 2004.

[15] Sivasubramanian, S., Szymaniak, M., Pierre, G., and van Steen, M. “Replication for Web Hosting Systems”. In *ACM Computing Surveys 36*, number 3 (September), pages 291-334, September 2004.

[16] Wikipedia, “Pareto distribution”. http://en.wikipedia.org/wiki/Pareto_distribution

[17] Wikipedia, “Zipf’s law”. http://en.wikipedia.org/wiki/Zipf%27s_law

[18] Zhao, W., and Schulzrinne, H. “Dotslash: A Self-configuring and Scalable Rescue System for Handling Web Hotspots effectively”. In *International Workshop on Web Caching and Content Distribution (WCW ’04)*, Beijing, China, October 2004.