



UNIVERSITÀ DEGLI STUDI DI ROMA “LA SAPIENZA”

DOTTORATO DI RICERCA IN INFORMATICA

XIX CICLO – 2008

Distributed Dynamic Replica Placement and Request  
Redirection in Content Delivery Networks

Claudio Vicari





UNIVERSITÀ DEGLI STUDI DI ROMA “LA SAPIENZA”

DOTTORATO DI RICERCA IN INFORMATICA

XIX CICLO - 2008

Claudio Vicari

Distributed Dynamic Replica Placement and Request  
Redirection in Content Delivery Networks

Thesis Committee

Prof. Chiara Petrioli (Advisor)  
Prof. Giancarlo Bongiovanni  
Prof. Francesco Lo Presti

Reviewers

Prof. Michele Colajanni  
Prof. Ravi Sundaram

AUTHOR'S ADDRESS:

Claudio Vicari

Dipartimento di Informatica

Università degli Studi di Roma "La Sapienza"

Via Salaria 113, I-00198 Roma, Italy

E-MAIL: [vicari@di.uniroma1.it](mailto:vicari@di.uniroma1.it)

WWW: <http://reti.dsi.uniroma1.it/eng/claudio-vicari>

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Commonly used solutions . . . . .	5
1.2	CDNs definitions . . . . .	10
1.3	Problems to address in CDNs . . . . .	12
1.4	Implemented CDNs . . . . .	13
1.5	Problems definitions . . . . .	15
1.6	Thesis contributions . . . . .	18
1.6.1	Thesis organization . . . . .	19
<b>2</b>	<b>State of the art</b>	<b>21</b>
2.1	Replica placement . . . . .	21
2.1.1	What to replicate . . . . .	21
2.1.2	General formulation of the Replica Placement Problem . . . . .	21
2.1.3	Static replica placement . . . . .	23
2.1.4	Dynamic replica placement . . . . .	26
2.1.5	Other replication scenarios . . . . .	30
2.2	Request redirection and load balancing . . . . .	31
<b>3</b>	<b>Reference solutions</b>	<b>35</b>
3.1	Problem formulation . . . . .	35
3.1.1	Centralized redirection mechanism . . . . .	37
3.2	Optimal solution . . . . .	39
3.3	Dynamic, instantaneous, centralized replica placement heuristic . . . . .	43
3.4	Static Replica Placement . . . . .	46

<b>4</b>	<b>Distributed, dynamic replica placement</b>	<b>51</b>
4.1	Dynamic distributed replica placement heuristic . . . . .	52
4.1.1	Handling replicas under the target utilization level . . . . .	56
4.1.2	The bootstrap case . . . . .	57
4.2	Performance evaluation . . . . .	60
4.2.1	Simulation environment . . . . .	60
4.2.2	Simulating the Request Redirection System . . . . .	61
4.2.3	Topology Generation . . . . .	63
4.2.4	Simulation results: comparing greedy static, centralized heuristic, distributed heuristic . . . . .	66
4.2.5	Distributed Heuristics Evaluation . . . . .	70
	Triangular traces . . . . .	70
	Dynamic behavior with Pareto processes . . . . .	72
4.2.6	Impact of the number of contents . . . . .	75
4.2.7	Sensitivity to a constrained replica neighborhood . . . . .	81
<b>5</b>	<b>Request redirection schemes</b>	<b>85</b>
5.1	Distributed load balancing . . . . .	85
5.2	Validating the distributed algorithm . . . . .	92
<b>6</b>	<b>Conclusions</b>	<b>99</b>

# List of Figures

1.1	Simple representation of a Content Delivery Network . . . . .	11
3.1	Small network topology generated with GT-ITM . . . . .	36
3.2	A model for the redirection scheme . . . . .	38
4.1	A model for the redirection scheme . . . . .	62
4.2	AT&T backbone topology . . . . .	64
4.3	“1299” backbone topology . . . . .	65
4.4	Access nodes coverage . . . . .	66
4.5	SIMULATION RESULTS one content and $d_{max} = 6$ . . . . .	68
4.6	SIMULATION RESULTS one content and $d_{max} = \infty$ . . . . .	69
4.7	Simple topology, “triangular” traces . . . . .	71
4.8	“1299” , varying $d_{max}$ and $U_{mid}$ . . . . .	73
4.9	“1299” , varying $U_{mid}$ and $d_{max}$ . . . . .	75
4.10	“1299” , varying C, low load, $d_{max} = 18$ . . . . .	79
5.1	Probability of load inflation . . . . .	90
5.2	Simulation trace, first set . . . . .	93
5.3	First set, loads . . . . .	94
5.4	Simulation trace, second set . . . . .	95
5.5	Second set, loads . . . . .	96
5.6	Simulation trace, third set . . . . .	97
5.7	Third set, loads . . . . .	97



# Chapter 1

## Introduction

During the years, the Internet and in particular web services have seen an increasing commercial success and an ever increasing user traffic. Frequently accessed services, such as very popular web sites, have to deal with a huge and still growing amount of requests, thus often suffering from network and server congestions and bottlenecks problems. This in turn has paved the way for the development of systems overcoming the traditional client-server paradigm. Hosting a web content at a single server results in the impossibility to provide the services with an acceptable level of client perceived quality when the content is highly popular: This can be either because of high response times, or because some user requests are lost. Server administrators could try to improve web services performance by improving server hardware, or by increasing the bandwidth available to the server. However, trials in this direction proved to be not enough to react to the increasing user demands.

### 1.1 Commonly used solutions

Many solutions have been proposed for the problems making content accessible by an increasing number of users without affect negatively the users' perceived quality of the service. Employed techniques are based on either reducing the load managed by a server or by distributing it among a plurality of hosts. The first solution is often referred to as *mirroring*. Mirroring consists in replicating a content in many locations (the *mirrors*) across the Internet. The users can then choose from a list of different URLs the mirror more suitable for them. Mirroring is therefore not a transparent approach, although it is still used especially for long downloads (e.g., ftp).

A second solution, more important for the objectives of this dissertation, is the technique called *caching*.

## Caching

According to the HTTP/1.1 standard ([33]), a cache is defined as a local store of response messages, along with the subsystem that controls local message storage, retrieval, and deletion. Caching has been used since the very beginning of the web. The first web server at the CERN Lab in Geneva ([60]) already had an associated proxy server that included a cache. Caching most widely cited goal is to reduce the user-perceived latency [51], that is, the time difference between the moment a web request is issued and the time the response is displayed by the client software. Caching also aims at reducing the load on the network, by avoiding multiple transmissions of the same response. The original server providing the content (the origin server) gains benefits from caching, as this effectively adds an intermediary that handles requests on its behalf.

Caching is commonly implemented by ISPs, by using a host situated close to the customers, called *caching proxy*. Clients can be configured to use this proxy as an intermediary either manually or automatically. Clients configuration can be avoided by making use of a so called *interception proxy*. This is a proxy that either examines client messages directly so to intercept web requests, or that receives the traffic flow from another network element in charge of performing traffic interception.

Upon the reception of a request the proxy decides whether it can satisfy it using its local storage. If this is the case, it returns the cached response. In case this is not possible, it directly contacts the origin server on behalf of the user: In this case the cache returns the response to the user only after having received it by the origin server. The cache must decide whether the message is cacheable (it could not be because of non-reusability or because of privacy issues), must check if there is local space available to store it, and in case there is no space available, it has to decide whether it is convenient to delete some of the objects already stored.

On the client side, since the requested objects are brought closer to the users, the customers perceive reduced latency. The system is also considered more reliable because the clients can obtain a copy of the content even when the remote server is not available.

From the point of view of an ISP, caching is not only able to reduce the latency

experienced by its customers, but has also another significant benefit: It reduces the load inside its backbone. Web events, such as sport events, tend to generate sudden spikes in the user traffic (*flash crowds*). When these events occur, deploying caches can be cheaper than increasing the backbone capacity. The bandwidth that is freed as a result of caching also enables the ISP to support more customers without improving the existing infrastructure. Note also that ISPs typically pay their upstream provider based on the bandwidth. As a consequence, by reducing the bandwidth they need, they can lower the overall costs.

Many issues must be considered when exploiting caching. First, as mentioned, the proxy must decide whether the retrieved content is cacheable. To this aim, it must first examine protocol specific information. The HTTP/1.1 RFC ([33]) specifies a series of directives to provide response cacheability information that should be considered by the caches.

Apart from protocol restrictions, a cache has its own set of rules to decide on the cacheability of objects. This can be affected by various properties of the response message, such as its size, or the fact that this message is dynamically generated.

On the other hand, the cache should ensure the freshness of a cached response before returning it, and this is not a trivial task. Policies are typically based on a time-to-live assigned to objects by the proxy itself, on data contained in the HTTP/1.1 header (e.g., cacheability information, expiration times), on the frequency of requests, and on the frequency of content updates as obtained by checking the origin server.

Another important issue is how to manage disk space in the cache. At some point the cache can become full, and some policy has to be implemented for making room for new objects. According to [70], in practical implementations caches use two thresholds,  $H$  and  $L$ ,  $H > L$ , to guide the replacement process. As soon as the total size  $S$  of locally cached objects exceeds the *high watermark*  $H$ , the cache starts deleting objects, stopping the process when  $S < L$ . Cache replacement policies can take into account many parameters such as the cost of fetching the object, the cost of storing it, the past number of accesses to it, the probability of future accesses, and the time the object was last modified (a resource that has not been modified for a long time is less likely to be modified in the future). Many policies have been employed and investigated. In [88] the author shows that there is not a clear winner among the various approaches. As a consequence, cache developers should carefully consider many factors before choosing a policy: These factors include the amount of resources

available at the proxy (CPU, disk space, bandwidth), and the set of clients that will access the proxy, behaving differently depending on whether in a small organization, or in an ISP-level proxy. For instance, if the external network bandwidth is limited or expensive, a policy should be chosen that maximizes the number of saved bytes (size based policies are not suitable for this). As a second example, in ISP-level proxies, user traffic results in considerable short-term temporal locality, so that policies such as LRU are preferable ([61]).

According to what described so far, ISP-side caching results in performance advantages. However, caching has also several drawbacks.

As explained in [51], it may occur that the origin servers do not want to have the objects delivered by proxies. The content provider may be interested in having a strict control on cached resources, i.e., it may tag several responses as not cacheable from its point of view. The reason is that caching may prevent the provider from tailoring contents to the specific user, and from obtaining accurate hit counts. These two things are very important for advertising. Caching could even provide users with out-of-date content according to policies that are out of the control of the content provider. Although the HTTP/1.1 protocol gives control to the server over what should be considered cacheable, not all caches respect this directives. The goal of reducing user latency and saving bandwidth on the ISP side is sometimes in contrast with the content provider goals.

All these reasons motivate many servers to use *cache busting* techniques that prevent responses from being cached. These techniques include using and exploiting the existing HTTP/1.1 cache control features (e.g., setting the EXPIRES header to a value in the past), altering a page without altering the user visible content (through minor modifications to the HTML source), and changing objects URLs in a pseudo-random way.

In [48] the authors analyze the URL aliasing phenomenon. An object is said to be aliased when the same content can be retrieved by accessing two different URLs. By analysing client traces collected for different client populations, the authors discovered that a relevant portion of retrieved web objects ( $\sim 5\%$ ) is aliased, accounting for as much as  $\sim 36\%$  of transferred bytes. The authors also investigate the different reasons for aliasing. Aliasing can be caused by web authoring tools when inserting clipart images in web pages, but also by flawed metadata provided by the origin server, such as when it replies with impossible dates for expiration. The important

point is that this phenomenon effectively reduces the number of hits in ISP-controlled proxy caches.

In [89] it is shown that improper HTTP setting is another significant reason for inefficient use of caching. As much as 30% of objects that are considered uncacheable could be turned cacheable by properly setting HTTP headers. It further shows that, for objects having an associated HTTP lifetime, more than 78% of them were found to be unchanged at the lifetime expiration. This means that caches could avoid checking objects in the vast majority of the cases.

Proposals to discourage cache busting ([38, 66]), require a certain degree of communication between the proxies and the origin server in order to report statistics or coordinate in delivering advertisements. In [48] a specific method to limit the effects of URL aliasing is proposed, but this involves modifying the HTTP protocol in order to transmit a checksum of the retrieved object. These approaches have not obtained much success. As of today cache busting is still the easiest choice for origin server administrators.

Finally, note that there are many cases in which the content provider itself wants to offer a better service to the users, whatever ISP the users use to access the objects. Caching cannot be used to guarantee the levels of performance, availability, and reliability that the content provider desires.

**Beyond caching** In the 1997's paper [6], Baentsch, Baum, Molter, Rothkugel, and Sturm, analysed the logs of the University of Kaiserslautern's main caching proxy, collected for 6 months. Analyzing these data, the authors pointed out several relevant problems of current caching mechanisms, including the additional latency clients have to experience each time their requests are intercepted by a proxy experiencing a cache miss.

They propose combining caching and replication as a way to overcome these limitations. Replicating in this paper means dedicating some portion of the caching servers to a complete mirroring of a manually selected list of URLs which are important to the local users. Objects are pushed to the mirroring servers at every update. The authors simulated such a scenario, assuming the mirroring of the documents most frequently accessed in their web trace, and varying the amount of space dedicated to mirroring; the remainder of the disk space is dedicated to caching. They found a 4% improvement in the object hit ratio when the cache size is 850Mb and the replication

quota is 40%, but an even higher number of saved bytes.

**Server farms** Server farms (see [16] for a survey) are another widely used class of solutions to decrease servers load. A server farm is a collection of servers, placed in the same physical location, along with a mechanism to distribute the requests among them, that is usually enforced by a single device (the *switching device* or *switch*). Users see the server set as a single server so that they don't need to be concerned about the names and the location of the mirrors, the switching device is placed on the path from the users to the servers. This device intercepts every request, decides what server should serve it (according to some policy), and sends it to the selected server. Many popular web sites employ server farms. Because of their nature, they are managed by the content provider, and are meant to be used by all customers accessing the provided contents.

Server farms are effective in reducing origin servers' load, and can manage also uncacheable contents (e.g. dynamic, read only contents, that change only depending on values stored in cookies). As every server in the farm can perform the same computation based on the same data, the client will obtain the same response regardless of the switch decision. The content can thus be replicated, even if the response itself could not be cached easily.

Although widely used, server farms are not a ultimate solution to the problem of accessing web contents and services. They fail to reduce the length of the paths traversed by both the requests and the replies, and the switching device itself could possibly become a bottleneck.

## 1.2 CDNs definitions

Content Delivery Networks (CDNs, [49, 8]) have been introduced to overcome the limitations of the aforementioned techniques, and offer the content providers with a complete solution for their problems. They are now commonly deployed, the CDN provided by Akamai ([2]) is perhaps the most famous, however many different CDN providers are operational today (see [28] for a complete list).

A CDN is a combination of various subsystems (in figure 1.1 you can see a very schematic representation): the most important ones from our point of view are the replica infrastructure and the request distribution system. The replica infrastructure

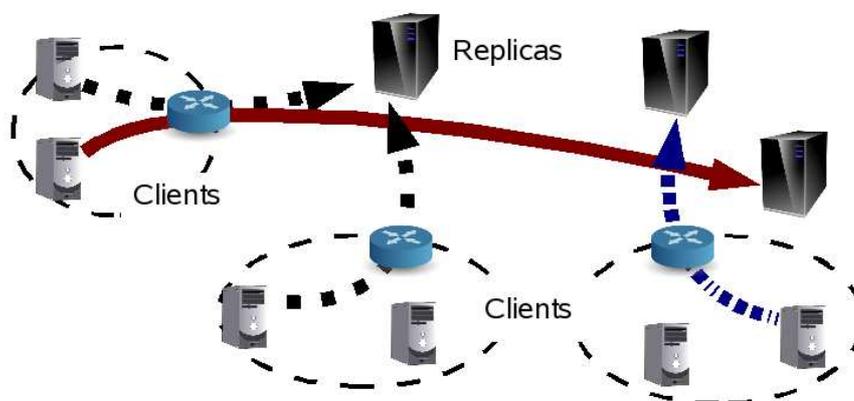


Figure 1.1: Simple representation of a Content Delivery Network

consists of a number of edge servers or replicas, each one hosting copies of some of the provided contents (in the picture, the black boxes at the top represent the replicas). Contents are replicated over the CDN, in order to move the contents closer to the users.

The request distribution system, also called request routing system (RRS), is in charge of intercepting user requests, and transparently redirecting user requests to the replica which is able to serve the request with the best possible user perceived quality. It must therefore be aware of the contents hosted at the different replicas and of network conditions, in order to redirect requests appropriately. In the picture, users (the gray boxes) access the network via the hosts represented with a cylindrical shape, that are the nodes of the RRS running the request redirection system, deciding where to route each request based both on location and type of the request.

CDNs systems thus succeed in improving the experience of users when accessing web contents, as content is moved closer to the user, while meeting the Content Provider need for content distribution control. It is also easier to maintain statistics. The provider has full control over the replicas. Another advantage is that many non cacheable objects can be replicated, so replicas can effectively serve more objects than caches do (see [77] for a thorough analysis of the possible approaches to replicate dynamic contents for web applications). Finally, to ensure the freshness of the contents, content distribution sites may avoid using HTTP, and choose protocols which allow more efficient communication with the origin server. These mechanisms

are transparent to the rest of the network.

### 1.3 Problems to address in CDNs

Designing a CDN requires addressing many problems ([8]). As the CDN must be efficient for a very large number of users, the edge servers must be deployed in a very large number in a significantly wide area.

A replica placement mechanism is needed, in order to decide where to place replicas, and how to store proactively and adaptively the appropriate contents, before the request arrival (this is in sharp contrast with traditional caching). The mechanism should meet the goal of minimizing the costs for the CDN provider, while maximizing the user satisfaction.

Another important component is the content update mechanism, that is in charge of checking whether the content has changed at the origin server, and guarantees the freshness of the replicas. The existence of such a mechanism also motivates the need for minimizing the number of replicas, in order to limit the number and complexity of updates.

The request distribution system is in charge of intercepting user requests, selecting the replica most appropriate for serving the request, and directing the user request to it. This service should locate a replica that is as close as possible to the user, while avoiding replica performance degradation - e.g., by balancing the load among replicas.

The interception and delivery of the requests can be done by means of a variety of mechanisms, the most commonly used ones ([52, 84, 31, 79]) being DNS redirection and URL rewriting, that are sometimes used in combination.

In DNS redirection, the CDN provider manages the DNS for some domains (ADNS, Authoritative DNS), and when the user requests for a domain name managed by the CDN provider, the system translates the name to the IP address of the chosen replica. Remember that when making a DNS resolution the client usually contacts its primary DNS server, that in turn contacts other DNS servers traversing the DNS hierarchy, and that DNS queries are usually cached by the DNS servers traversed by resolutions. The main drawbacks of this technique depend on the hierarchic structure of the DNS, and limit the capability of the DNS to perform different decisions for each different request:

- whenever a client asks for a DNS resolution, this goes to its primary DNS server, that in turn eventually queries another DNS server until reaching the CDN DNS server for the specific domain. Thus, this last server does not know the IP of the client, but only the location of the last DNS that queried it, limiting the capability of the RRS in exploiting the client location information for redirecting its requests to the closest replica
- DNS queries are typically cached by DNS servers, the ADNS can set a time-to-live on the responses it transmits. Anyway, many DNS implementations do not honor the TTL value so that the RRS has a coarser control on resolutions.
- users that make use of the same primary DNS server will be redirected to the same set of replicas during the whole TTL interval.

The other frequently used technique is URL rewriting, also called content modification. With a web page being made by an HTML file, the web server can modify both the references to the embedded objects and the anchors, in order to let the client fetch these resources from a specific surrogate. This technique can be applied statically, or (more interestingly) on-demand. The main drawbacks for on-demand URL rewriting are (1) that it imposes a performance overhead every time the page is generated, and (2) that the first request must always be served by the origin server.

Other interesting, although less used techniques include: application level header modification (e.g., using HTTP redirection), anycasting (see [8]). Of particular interest is IP level anycast, in which a single IP address is advertised from different hosts providing the same service. This allows a CDN to direct requests for the same content through different possible routes. Anycast services in CDNs are usually implemented as BGP anycasting ([9], [14]), in which routes are advertised through the Border Gateway Protocol.

## 1.4 Implemented CDNs

In the following we will first review the solutions so far adopted by CDN providers and we will then summarize which are the objectives of this thesis.

Although the solutions employed by CDN providers are usually proprietary and thus not of public knowledge, some papers provide insights on the algorithms and solutions they actually employ.

In the 2002 paper “*Globally Distributed Content Delivery*” ([31]), the authors disclose details about the actual implementation of the systems used by Akamai. According to them, Akamai uses a network of more than 12000 servers, spread over 1000 networks (as of a more recent whitepaper [3] from the company’s website, the network now comprises 25000 servers).

They also say that Akamai allocates more replicas in the locations where the load is higher, not giving further details on this topic. Note that other CDNs use a much smaller amount of servers: for instance, MirrorImage in its whitepapers ([64, 63]) claims that owning only a few servers ( $\simeq 20$ , [64]) is an advantage as configuration and content changes can be deployed faster.

The procedures followed by the RRS for selecting servers are not disclosed in [31], but they describe the goals of their redirection system. The claim is that it aims at redirecting every client request to a server that is close (as a function of the client location, the network conditions, and the dynamic loads of the servers), available and that is likely to already retain a copy of the requested content.

The request redirection is employed by means of the DNS system. A control application receives periodical reports from each server, aggregates the data, and sends them to the DNS dispatcher system. If the load of a server exceeds a certain threshold, the DNS system itself assigns some of the contents allocated in the server to another server, eventually lowering its load. When exceeding a second, higher, threshold, the server is considered not available to clients. To avoid this situation, the monitoring system can inform the request redirection system, and the server itself can decide to share a fraction of its load with another server.

Akamai systems have the capability to serve many types of contents, such as static contents, dynamic contents, and streaming data.

In some work ([42, 79, 52]), extensive probes have been done on the most popular CDN networks (Akamai seems to be the most studied), giving an experimental validation of these notions. These papers also sum up relevant information disclosed during the years and confirm these notions about CDNS.

The first important thing to note is that most CDNs seem to make use of the DNS redirection, mixed with URL rewriting. Some companies also adopts BGP anycasting, that opens new interesting possibilities, but is not as widely deployed: the most relevant CDN providers using such a service are CacheFly [14] and BitGravity [9].

CDNs offer either full- or partial- site content delivery services. Full-site content delivery means that the customer's content is completely served by the CDN, partial-site content delivery means that only a certain subset of the web objects are cloned in the CDN. Usually partial-site content delivery relies on the content provider's web pages having URLs rewritten in order to make use of the URL-based redirection. In [52] it is put in evidence how, in 2001, most of the objects in CDNs were images (96%-98%), although they accounted for about 40-60% of the total served bytes.

Johnson et al. in [42] put in evidence as the RRS of Akamai and Digital Island ([30]) is not always able to redirect to the current best server, anyway it meets the goal of avoiding redirections to "bad" servers.

The recent work [79] goes more in depth on the Akamai CDN. By taking accurate measurements from different hosts spread in the world, and decoupling network effects from server related effects, they were able to provide many interesting data. First, they confirm that clients in different locations are served by server sets of different size, containing different elements: during a single day, the RRS returned 20 different servers to some clients, while some other client resulted in as much as 200 different servers. Another important point is that Akamai exhibits different performance for different customers ("content providers"), some customer results in being hosted in more than 300 different server, some other one in as few as 43 out of the many thousands the CDN provider owns. The time delay before the RRS changes the selected server for a client varies for different geographic areas. For some client located in the United States, 80% of the redirections are shorter than 100 seconds; very long redirection times indeed occur, but this events are very rare and their occurrence is strongly related to the time-of-the-day. For nodes located in Brazil redirection times are much longer. The last important finding is that the RRS usually routes the requests through a path that is less congested than the average (from the experiments, this is untrue only for requests coming from Brazil), thus supporting Akamai's claim of being able to make decisions based on both the network conditions and the server health.

## 1.5 Problems definitions

We already analyzed the problems that must be addressed in order to successfully deploy a CDN. In this dissertation we start from the proposal of a modelization and

a formulation for managing the replica placement inside a CDN. In the following paragraphs, we will first give some general definition that we will widely use during this dissertation, and we will then define the problems that we addressed in our work.

**Definitions** It is important to clarify what is to be treated as a replication unit, that is, what's the structure of the objects that can be replicated. Karlsson and Mahalingam in [46] define objects as “*data aggregates, such as entire web sites, directories, or single documents*”. In [69], it is stated that in CDNs “*a typical practice is to group Web content based on either correlation or access frequency and then replicate objects in units of content clusters*”.

Note also that these definitions are still valid when dealing with dynamic contents, as in replication it is possible to replicate both the application and the data ([77]). This is a very important point, as most served pages are dynamic nowadays.

Throughout this work, we will use the following definitions for “object”, content and “replica”:

- the word **object** denotes an element that can be served by the CDN. It is the minimal unit to be considered for replication.
- with the word **content** we denote an aggregate or cluster of objects.
- a **replica** is the clone of a content. A replica can be created in any of the servers of the replica infrastructure, and is able to reply to queries for its content, on behalf of the origin server.

**Static Replica Placement** The replica placement problem in its static variant can be formulated as follows:

Given a network topology, a set of CDN servers and a given request traffic pattern, decide where content has to be replicated so that some objective function is optimized, while meeting constraints on the system resources.

The solutions so far proposed typically try to either maximize the user perceived quality given an existing infrastructure, or to minimize the CDN infrastructure cost while meeting a specified user perceived performance. Examples of constraints taken into account are: limits on the servers storage, on the server sustainable load, on the maximum delay tolerable by the users etc. A thorough survey of the different objective functions and constraints considered in the literature can be found in [45].

Despite this problem may at first seem very similar to the problem of filling up caches in the proxy scenario, it has some important differences. In caching the available disk/memory space is usually filled with contents until the storage limit is reached: it is not important to minimize the number of copies, it is more important to use all the available disk space in order to maximize the probability of retaining the requested contents. In CDNs instead, it is better to limit the number of different copies of the same object in order to save bandwidth (the more replicas the more updates) and CDN costs. Having many servers also negatively influences other processes, such as statistics collection, server maintenance, server software configuration...

As a second point, in caching each request is intercepted by the first surrogate in the path from the user location to the origin server, or by the host configured as a proxy in the client, whether or not this proxy already has a copy the requested object. In CDNs instead, requests are intercepted by the system, that possibly knows some details about the requested object and the placement of replicas. If the content is available in a service node that is not the closest to the user, but that is good enough to satisfy the desired constraints, the request routing system will succeed in directing the request to it.

**Dynamic Replica Placement** The user request traffic is dynamic and changes over time. Adopting static solutions in a realistic setting where users traffic changes over time either results in poor performance or into high maintenance costs. The former occurs in case replica placement is computed only once (or seldomly recomputed) and the same replica configuration is used for long times independently of current user requests. The latter reflects the case in which static algorithms are executed frequently to try to follow users dynamics, demanding for frequent replicas add/removals.

Dynamic replica placement schemes explicitly consider the current replica placement and the reconfiguration costs when deciding which replicas to add or remove, to reflect the current and expected future users needs, while aiming at minimizing the long-run costs associated to replica adds, removals and maintenance.

**Load balancing** The RRS is able to direct requests to replicas, and has to choose appropriate replicas, based redirection on many factors: not only the proximity of

the client to the serving replica, but also the capability of the replica itself to serve requests with a low latency.

In a distributed system, if some hosts remain idle while others are very busy, system performance can be affected drastically. To prevent this, a load balancing policy can be employed, that balances the workload of the nodes in proportion to their capacity, thus supporting the effort to minimize the user perceived latency.

## 1.6 Thesis contributions

This thesis work started from the observation that the majority of the previous solutions presented in the literature specifically address static replica placement, in a single-content, centralized fashion. We therefore proposed a new model for dynamic replica placement, considering a number of important constraints and of important features. Our model considers the possibility for a CDN to separately manage multiple hosted contents. If replica allocation is managed separately for each content, the system can exploit the behavior of different users populations and place replicas in the locations where the demand is higher for that specific content. We base our results on this model and on the centralized optimal solution based on a Semi Markov Decision Process, proposed in [7]. After the observation of the behavior of this optimal scheme we proposed a centralized, dynamic heuristic.

Using a centralized heuristic is not practical, because it has to continuously gather data and communicate the decisions. We then decided to address the problem in a distributed way: we assume that each replica is able to monitor its own load, and based on this to decide when to clone or remove itself.

Our heuristic strongly relies on the RR system. The RRS provides load balancing among the available replicas redirecting requests only to close-by replicas. A replica can give the RRS a false feedback on its load (providing an inflated value) in order to give the information that it wants its load to be reduced, if this does not impair the system capability in serving requests.

This is exploited by underutilized replicas to progressively decrease their load (when a replica does not serve traffic, it is deleted).

Our simulations compared the behavior of the distributed, dynamic load balancing heuristic against the optimal dynamic heuristic, the centralized heuristic and a greedy heuristic that has been widely used as a reference in literature. We will show

that our distributed heuristic jointly addresses dynamic replica placement and request redirection being able to serve requests with good quality, to minimize the number of allocated replicas, keeping the replicas within a desired utilization level and resulting in a limited number of replicas add/removals.

We also propose a novel distributed algorithm for request redirection that can be executed by the nodes of the CDN infrastructure that are in charge of redirecting the requests (“access nodes” from now on), by using only information that can be obtained by monitoring nearby replica load conditions. We first modeled the load balanced request redirection as an optimization problem, from which we derived a distributed scheme that minimizes the target function. The request redirection scheme is then integrated into our scheme for dynamic replica placement, providing a solution which minimizes the number of replicas, load balancing traffic among the allocated replicas and keeping the load of allocated replicas within a desirable range of utilization. Such range of utilization can be configured by the CDN provider.

### 1.6.1 Thesis organization

The remainder of this thesis is organized as follows:

- In chapter 2 we review the state of the art about the static replica placement problem, starting from the graph theoretical formulation, then describing how the problem has been addressed in the replica placement literature and how the formulation has been extended during the last years. We will also review existing papers that address the dynamic scenario, and we will briefly describe some problem strictly related to the replica placement. Finally, we will analyze various solutions proposed in literature for balancing the traffic among servers. We will first review load balancing in locally distributed systems, then in geographically distributed systems, discussing the problems that arise when dealing with real redirection techniques.
- In chapter 4 we will describe and analyze approaches for optimum replica placement. We will review the model, the centralized dynamic heuristic and the distributed dynamic heuristic we have proposed, and we will show by means of simulation that the performance of our distributed solution is close to the performance of the centralized ones.

- Our distributed load balancing scheme is presented in chapter 5. We discuss how the distributed scheme can be employed to support the replica placement heuristic and we describe the request redirection scheme in details. We finally verify by means of simulations that the distributed algorithm for request redirection is able to perform load balancing while controlling the replica load within a predefined level of utilization.
- Finally, in chapter 6 we sum up the issues we addressed and the relevance of our results, along with a discussion of the possible future directions for our research.

## Chapter 2

# State of the art

### 2.1 Replica placement

#### 2.1.1 What to replicate

The majority of papers dealing with replica placement do not directly address the problem of what is going to be replicated. The replica can be seen as a mirror of a unit of aggregated data (selected based on correlated content or similar access frequency). Current schemes assume that some policy for aggregating data exists, and that the whole system knows how data is aggregated in contents. We will make the same assumption in this work. In the following sections we review the solutions that have been proposed for replica placement and in particular for dynamic replica placement which is the objective of this thesis.

#### 2.1.2 General formulation of the Replica Placement Problem

The static version of the replica placement problem can be mapped to well-known graph theoretic problems, such as the facility location problem and the K-median problems. Both problems are NP-hard ([26]) but good solutions can be obtained in practice by approximation algorithms ([17, 40, 62, 37]). In particular Charikar and Guha developed a 1.728-approximation algorithm for the facility location problem, and a 4-approximation algorithm for the K-median algorithm in the metric space (see [17]). A 1.52-approximation algorithm has been designed for the facility location in metric space [62].

The K-median and facility location problems are formalized as follows: let  $G =$

$(V, E)$  be an undirected graph, and let  $R \subseteq V$  be a set of locations at which facilities (centers) can be hosted. Let us also denote with  $A \subseteq V$  a set of clients, and be  $d_{a,r}, a \in A, r \in R$  the distance between a client  $a$  and possible facility location  $r$ . Each client  $a \in A$  has an associated demand  $q_a$ .

**Facility Location Problem** Given a set of facility locations and a set of customers, the general facility location problem concerns deciding which facilities should be used and which customers should be served from each facility so as to minimize the total cost of serving all the customers. Several variants of the problem have been proposed e.g. based on the definition of specific objective functions and on the interaction between demand and facilities. Typically building a facility at vertex  $r \in R$  has a cost of  $f_r$ . Each client  $a$  assigned to a facility  $r$  incurs a cost of  $q_a \cdot d_{a,r}$ . Constraints can be imposed on the total customer demand that can be served from a facility; variants of the problem can be defined depending on whether a customer demand can be splitted among different facilities.

**K-median problem** In the minimum K-median problem, we must select a set of centers  $F \subseteq R$  s.t.  $|F| = K$ , and then assign every vertex  $a \in A$  to the nearest center. Assigning  $a$  to  $r$  incurs a cost  $q_a \cdot d_{a,r}$ . The objective is to select a set of centers  $F$  that minimizes the sum of the assignment costs.

The difference between these two problems can be summarized as follows. The K-median problem well represents the case in which the number of facilities that can be opened is fixed. Therefore no cost is associated to the opening of new facilities. The facility location problem has no bounds on the number of facilities, but aims at minimizing the costs associated to opening the facilities and serving the clients demands with a given set of facilities.

Many variations of these problems have been proposed and many are being studied. There are capacitated versions, in which facilities have an upper bound on the number of serviceable clients, [21, 18, 50, 20], and uncapacitated versions. Related problems have been defined relaxing some constraints, e.g. allowing unserved clients ([20]), or, in case of the capacitated version, allowing to exceed the capacity of the allocated facilities ([21]).

### 2.1.3 Static replica placement

Of the solutions proposed in the literature for replica placement, most concern the static traffic case, i.e. given a traffic pattern they address how to compute the 'best' replica placement. Basically, given the topology of the network, the set of CDN servers as well as the request traffic pattern, replicas are placed so that some objective function is optimized while meeting constraints on the system resources (server storage, server sustainable load, etc...). Typical problem formulations aim either at maximizing the user perceived quality given an upper bound on the number of replicas, or at minimizing the cost of the CDN infrastructure while meeting constraints on the user perceived quality (e.g., latency).

While static replica placement can be modeled as a facility location or  $k$ -median problem (making it possible to reuse the extensive results and solutions available in the literature) it does not well capture the traffic dynamics that are expected in a realistic scenario. In such realistic scenarios the static replica placement algorithms have to be periodically rerun for the system to adapt to changes in the traffic patterns, resulting in trade-offs between the quality of replica placement and the overhead associated to frequent replica configuration changes. In this section we review some of the major solutions proposed for static replica placement. In the following section we will see how it is possible to account for traffic dynamics.

For the static case, it has been shown that simple efficient greedy solutions result in very good performance [71], [41], [74] [43]. In [71] Qiu et al. formulate the static replica placement problem as an uncapacitated minimum  $K$  median problem, in which  $K$  replicas have to be selected so that the sum of the distances between each user and their nearest replica is minimized. They propose a simple greedy heuristic. In the first iteration, the cost of adding a replica at each replica site is evaluated, making the assumption that such replica will serve all the clients in the network: this cost is the sum of the distances from every client to the first replica. The algorithm places a replica at the site that yields to the lower cost. At the  $i$ -th step, by making the assumption that clients are served by the nearest replica, the algorithm chooses to add a new replica in a node that, along with the already chosen ones, yields the lowest cost. The algorithm stops after having chosen  $K$  replicas. This simple greedy scheme is shown to have performance within 50% of the optimal strategy, as computed by means of the relaxation of an integer linear programming problem formulation. As the  $K$ -median problem implies choosing exactly  $K$  replicas, it is not trivial how to

select an adequate value for  $K$ . An evaluation of the greedy scheme to assess the impact of  $K$  on the user satisfaction (expressed in terms of distance between the user access site and the serving replica), for different traffic patterns, is presented in [57]. Results confirm that the gain of adding new replicas diminishes as the number of replicas increase. It is important to stress that this result is valid when modeling the problem in its uncapacitated form, that is, when each replica can serve an arbitrary number of clients.

Qiu et al. have also proposed “hot spot,” a solution for placing replicas at nodes that along with their neighbors generate the greatest load [71]. The potential replica sites are sorted according to the amount of traffic generated in their vicinity, defined as the number of client sites within a certain radius. The hot spot algorithm simply places replicas at the top  $M$  sites in the ranking. This algorithm is shown to have slightly higher cost, having performance within 1.6-2 times the optimal cost.

In [80] and [81], “hot zone” is presented as an evolution of the “hot spot” algorithm. The authors note that the average user-replica latency is strongly influenced by outliers, that is, few nodes that are far from most replicas and experience bad latency. This motivated their use of the median latency as the primary quality metric to identify sites for replica placement. The idea is to first identify network regions made of nodes which can communicate with each other with low latency. This requires relying on some methods to obtain an estimate of these latencies, and carefully choosing the criterion to determine the zones size. Regions are then ranked according to the content request load that they generate and replicas are placed in the regions high in the ordering. At every step after a replica is allocated, all nodes in its region are marked in order to avoid considering them in subsequent steps. This prevents from allocating replicas close to each other. Experimental results show that Hotzone performs slightly worse than the greedy method, while it performs better than Hotspot, mainly because the latter usually places most of the replicas close to each other. On the other side, Hotzone’s computational complexity is significantly lower than both greedy and Hotspot, resulting in much lower computation times.

In [41] Jamin et al. propose the use of optimization conditions more complex than the simple condition of minimizing the sum of the distances among clients and the nearest available replicas: possible alternatives are trying to minimize the maximum RTT, the 95 – *th* percentile of the RTT, the mean RTT. Furthermore, a new “fan-out based” heuristic is presented, in which replica sites are placed one by one at

the nodes with the highest number of incident edges, irrespective of the actual cost function. The rationale is that such nodes are likely to be in strategic places, closest (on average) to all other nodes, and therefore suitable for replica location, while the resulting algorithm is much simpler to implement and requires much less network information than the greedy scheme. The performance evaluation shows that the new heuristic performs slightly worse than the greedy one.

Radoslavov et al. in [74] further extend the idea of a fan-out based heuristic. They consider various client placement models, and various realistic internet topologies, in which they have been able to identify the set of client locations, the Autonomous Systems (AS), the inter-AS routers. Relying on this decomposition, they propose a variant on the fan-out heuristic, called “Max-AS/Max Router”: instead of simply choosing the maximum fan-out node for replica allocation, Max-AS/Max Router chooses the maximum fan-out AS and picks the maximum fan-out router inside it. A performance evaluation based on real-world router-level topologies shows that the “Max-AS/Max Router” based heuristic has trends close to the greedy heuristic in terms of the average client latency, especially in Internet like topologies, even though it doesn’t make any use of potentially important information such as the customers location. An important problem with fan-out based approaches, is that adding clusters of replicas close to the backbone routers which are handling very high quantities of traffic is likely to change the volume of traffic at these already highly loaded locations. Further investigation on this impact is needed.

In [82] the authors present a solution which minimizes the costs paid for replication, while satisfying the requirement that every request is served by a surrogate within a bounded graph distance (that models maximum tolerated network latency). The replication cost is decomposed into costs due to storage and costs due to updates. The authors investigate three different objective functions, that is, minimizing the storage cost, minimizing the update cost, minimizing a linear combination of both. They examine a simplified case, in which the request routing system only knows the location of the origin server and nothing about the location of replicas, so that each replica can serve a specific request only if this request is routed through it. The topology in this case can be seen as a tree, rooted at the origin server. The authors present dynamic programming based algorithms of polynomial complexity to solve this specific problem.

The previous works focus on allocation of homogeneous replicas, assuming com-

plete replication. In [43] instead, Kangashariu, Roberts and Ross perform replication on a per-object basis. The work adapts existing replica placement algorithms to this new scenario. The set of locations where to store the objects is fixed, each location has a limited amount of storage space, the decision to be taken is what contents to store at each location. The authors then propose three simple heuristics:

A distributed one in which every replica site decides to host the contents that are most popular.

A greedy-local heuristic in which this decision also exploits knowledge of the topology, choosing to host the objects that yield the minimum value of a utility function that combines content popularity and the distance from the candidate replica site to the clients.

A greedy-global variant, in which a CDN supervisor computes the utility function for all objects and replica sites and makes global decisions.

Experiments show that the greedy-global scheme has the best performance, while greedy-local results in little improvement over the distributed heuristic.

In [44] the authors, based on their previous work in [46], try to unify and compare many of the described static placement approaches. They classify the heuristics based on the set of constraints and on the needed inputs, and evaluate them by means of simulations. They simulated the various algorithms in an Internet-like topology. The workload has been derived from a real web access trace. The main contribution is the proposal of a methodology to assist system designers in choosing the best replica placement heuristic to meet their performance goals. They conclude that carefully choosing replica placement policies is effective in reducing the overall structure maintenance costs.

#### **2.1.4 Dynamic replica placement**

A major limit of all these solutions is that they neglect to consider the natural dynamics in the user requests traffic. When network or user traffic changes occur which would make a different placement less costly or more satisfying for the users, the only possible solution is to re-apply the placement algorithm from scratch.

Depending on how often the algorithms are executed, the replica placement may react slowly to the system changes, so that the new placement of the replicas is not

the best one for the current user request traffic, or may result in significant overhead. Moreover, the replica placement happens every time from scratch, i.e., without considering where replicas are currently placed: this could possibly lead to non-negligible reconfiguration costs.

A few papers (e.g., [72], [73] and [19]) have addressed the problem of dynamic replica placement. However, the proposed schemes are embedded in specific architectures for performing requests redirection and computing the best replicas. No framework is provided for identifying the optimal strategy, and for quantifying the solutions performance with respect to the optimum.

In RaDar [72], [73] a threshold based heuristic is proposed to replicate, migrate and delete replicas (with fine-grained granularity) in response to system dynamics. This work contains many interesting ideas, the proposed architecture is distributed and every replica is able to make local decisions for the deletion, clone or migration of the contents it retains. The overall proposed solution combines dynamic replica allocation with servers load-aware redirection to the best replica to achieve low average users latency while empirically balancing the load among the CDN servers. However, no limits on the servers storage and on the maximum users latency are explicitly enforced.

In [19] two schemes designed for the Tapestry architecture [90] are presented. This work takes into account latency constraints and limits the number of requests that can be served by a replica. The objective is to allocate the minimum possible number of replicas while satisfying both latency and load constraints. The key idea is that upon a content request the neighborhood of the user access point in the overlay network is searched. If there is a server hosting a replica of the requested content within a maximum distance from the user, and such server is not overloaded, the request will be redirected to this server (or to the closest server if multiple servers meet such constraints). Otherwise a new replica is added to meet the user request. Two variants called “Naive” and “Smart” are introduced depending on the neighborhood of the overlay network which is searched for replicas, and on the scheme used to select the best location for the new replica. Going into more details, the “Naive” placement checks whether the destination server  $s$  is able to serve the request according to the load constraint and to the latency constraint: if not, the server will place a new replica in a node as close as possible to the client, situated in the overlay path traversed by the request. The “Smart” placement considers the destination server along

with its neighborhood in the overlay as candidates for serving the requests: if none of these is able to meet the constraints, a new replica will be added in the overlay path, as far as possible from the client. Although the ideas presented in the paper appear promising, they are tightly coupled with the peer to peer Tapestry architecture. The approach does not explicitly account for neither the costs of reconfiguration nor for possible servers storage limits. Finally, no information is provided in [19] on the rule to remove replicas, making it hard to compare with other dynamic approaches.

Recently, a new interesting approach has been applied to dynamically placing, removing and migrating copies of a given content in a hierarchical proxy-caches network (see [35]). The idea is to attract (push back) copies of a content from (to) a given zone in the network depending on the number of requests for that content originated in the zone, overall dynamically optimizing the contents stored in the caches. The optimization aims at minimizing the total distance between the content replicas and the locations where requests for the content are originated, while constraining the number of replicas to a constant amount. A centralized authorization mechanism is queried when there is the need to add new replicas: this mechanism ensures a maximum amount of duplication. The work in [35] does not account for limits on the number of requests a replica can serve, and on the maximum latency a client can tolerate. Finally, even though the allocation algorithm has two parameters to control the dynamicity of the allocation process, the costs due to reconfiguration have not been investigated.

The authors of [1] address dynamic replication in scenarios in which the users are mobile and, thus, requests are highly dynamic. The main contribution of this thesis is the proposal of a dynamic and distributed heuristic, in which each CDN server continuously monitors the incoming traffic and makes predictions based on its observations. In the proposed model, requests are always addressed to the closest server, regardless of whether the server hosts a replica for the content or not; every content is given a cost in terms of bytes needed for its maintenance, and a cost for its replication. At the end of fixed-length reconfiguration periods, for each content each server uses a statistical forecasting method to estimate the number of requests in the next period. Based on these predictions, the server computes the traffic needed to maintain a copy (if it already has it), to clone it, and the traffic that would be saved by copying it. By comparing these values it decides whether to keep or drop the local clone. This approach is interesting but strongly relies on some critical point: the

reconfiguration period, the reliability of the prediction, the fact that the underlying redirection scheme always addresses the requests to the closest server.

**Game theoretical approaches** Some authors ([32, 53]) addressed the problem of allocating contents in replicas using game-theoretical models.

In [53] they address the design of a cooperative scheme for minimizing the user perceived latency. Their algorithm needs an a-priori ordering of the replicas. As a first step, each replica will decide which subset of the available contents to hosts (by replicating the most popular contents). In the second step, each replica  $i$  will receive from replica  $i - 1 - th$  the placement computed by replicas  $[0..i - 1]$ . It will then compute (using a greedy policy) the set of objects it should host, and forwards the computed placement  $[0..i]$  to the  $i + 1 - th$  surrogate. This requires synchronization among cooperative nodes.

The paper in [32] addresses content placement and the request distribution problems separately. Their solution is based on a game in which the players are both the publishers that want to publish contents, and the surrogates hosting them. Surrogates act independently, trying to maximize their revenue based on the prices that are fixed by the publishers; publishers on the other hand try to maximize their revenue. The request routing problem is addressed in a similar way, publishers purchase bandwidth from the surrogates and try to maximize their revenue. The placement problem as such is quite different, as the surrogates do not cooperate and the publishers are involved; actually, the authors did not compare the proposed strategy against classical replica/content placement heuristics, but against the game theoretical optimal solution.

**Other applications/issues for replication in CDNs** Replica placement has also been investigated in slightly different scenarios.

Buchholz et al. in [13, 12] present the idea of Adaptive CDNs. In Adaptive CDNs there is an heterogeneous population of devices with different capabilities, e.g. different levels of support for multimedia contents (translating into need to scale images, transcoding video files). Possible approaches (see [13]) include: having a single server providing adapted contents, on-the-fly or by retaining adapted copies, or having proxies close to the clients, doing this task on behalf of the central server. The proposed architecture, called “adaptation path“ is quite different: a proxy, upon the reception of a request, evaluates if it can adapt its stored copy of the content in

a form suitable for the requesting device: if not, the request is passed to its parent, until a node able to answer the request is found. This work has been extended in [12]. Here the problem of composing the adaptation path is addressed. The authors suggest using a rather complex ranking function, that combines the length of the path for serving each request and the quality of the representation with respect to the quality expected by the client, to statically choose in a greedy fashion what representation of the contents to store in each node.

### 2.1.5 Other replication scenarios

Replication is a common issue to solve also in other areas. In geographically distributed systems, objects can be accessed (read/write) from multiple locations, that may be distributed worldwide. This not only happens in Content Delivery Networks, but also in other distributed systems such as peer to peer distributed filesystems, and distributed databases.

However, the problems that need to be addressed in these areas are slightly different. First of all, these systems are systems in which there are not only frequent reads but also frequent writes ([87]). Thus, it is very important to place replicas in order to minimize the sum of the costs for both reads and writes; mechanisms to synchronize the data are an important part of the system. Additionally, in the case of peer to peer systems, network nodes are not differentiated in servers and clients, and the replication can affect other behaviors: in [22] it is proposed to exploit it for speeding up the search process.

A very important work in the area of distributed data replication is the paper from Wolfson, Jajodia and Huang ([87]). In this paper the authors propose a simple distributed data replication algorithm. The algorithm converges to the optimal configuration, when executed on a tree. The basic idea is to select a set of replicas  $R$  that always induces a connected tree on the network graph. Every node  $i \in R$  monitors the number of reads and writes it receives, expanding the tree when the number of reads is higher than the number of writes, contracting it when the number of writes is higher. This algorithm can also be executed in general graphs, but in this case it is not optimal.

## 2.2 Request redirection and load balancing

So far, we have seen that CDNs require addressing the problem of how to place replicas, and we already pointed out that it is very important to find a way to intercept and route requests, assigning them to an appropriate server. Methods for intercepting requests have already been reviewed in section 1.3.

Assigning tasks to a web server is a problem that has already been addressed in locally distributed systems, such as server farms. In the already cited paper [16], after the description of the many possible architecture to redirect requests, the authors analyze the most commonly employed techniques to achieve load balancing in these systems. It is important to note that the role of directing requests in this case is taken by a single entity, implemented inside the same switch that intercepts and redirects requests.

According to the authors, the first important parameter to classify the existing algorithms is whether the algorithm is content-aware (that is, when the switch examines the requests data) or not. Furthermore, algorithms can be static (when they do not consider any state information) or dynamic. Dynamic algorithms are further classified based on the information they use: client state, server state, or both. The classical Round Robin policy belongs to the static, content-blind group. In the dynamic content-blind group, the most effective policies are the ones considering server-state information: among these, the most popular are the least-loaded server (always selecting the server with the lowest load) and the dynamic version of the Weighted Round Robin (WRR), that associates each server with a dynamic weight proportional to its load. The least loaded approach sends all requests to the same server until new information is propagated. This approach is well known to lead to servers saturation in some cases ([27, 65]). The WRR policy leads to excellent trade-off between simplicity and efficiency.

Content-aware approaches exploit the capabilities of layer-7 switches to inspect the contents of clients requests. The most efficient policy in this class is the Locality-Aware Request Distribution (LARD). It redirects all requests for the same content to the same server, until it reaches a defined utilization threshold: this maximizes the probability of the object being in the server's local cache. Upon reaching the threshold, the dispatcher will assign new requests to another node with a lower load.

In [24], the authors address the problem of assigning tasks in a distributed web server. They consider a system comprising web servers of homogeneous capabilities,

and consider a redirection system based on the Authoritative DNS (A-DNS). A-DNS is only reached by a fraction of the total requests, due to the name resolution caching employed by intermediate name servers and the client itself. The system must take global scheduling decisions under great uncertainty. They first show how the classic round robin algorithm (RR) can lead to a severe load unbalance among the servers. This is because different clients have different behaviors, and there are usually access points that exhibit a higher rate of requests. If the DNS employs the traditional RR algorithm, it could direct requests from these access sites to some particular set of servers, and then direct other requests from less traffic demanding sites to a different set of servers, causing unbalancing in the per server load. Because of the TTL, requests arriving from access sites generating a high number of requests will continue to be directed to the chosen servers for a while, while the servers that had few requests assigned might be underloaded.

Starting from this observation, the paper proposes algorithms that exploit the different behavior of clients to overcome this problem, algorithms considering past and present server load, as well as algorithms keeping track of alarms raised by overloaded servers. The simulation of the different proposed policies shows that the best performing algorithms are the ones that: (1) keep into account the behavior of the clients for each particular domain, (2) try to maintain an estimate of the load of the servers, (3) use a feedback mechanism through which servers signal the DNS when their utilization goes over a certain threshold of utilization  $T_{out}$ .

The same authors further extended their work in another paper ([23]) in which they explicitly address the problem of managing heterogeneous servers, and exploit the TTL field still considering the limitations of this parameter. Their most important idea is to assign higher TTL values for name resolutions of powerful nodes, and lower values for less capable nodes. The rationale behind this is that a powerful node is expected to react more effectively to an increase in the number of requests. The authors thus propose a class of algorithms employing adaptive TTL, study their performance and analyse what happens under the assumption of the presence of name servers not cooperating with the ADNS TTL. The results show that adaptive TTL approaches behave much better than fixed-TTL ones, even in the presence of name servers ignoring low TTL values.

Most papers on replica placement do not address the problem of balancing the load among replicas: they usually assume that each request is simply routed to the

nearest replica, and don't even consider any constraint on the rate of requests a replica can serve.

The works [72, 73], already cited for their dynamic placement policies, are the first solutions which address both replica placement and request redirection. The idea is that when a replica serves more than a certain amount of requests, the system first tries to lower the replica load and only if this is not possible it decides to allocate a new replica. This mechanism is able to achieve an imperfect load balancing, guaranteeing that each replica is not serving more than a fixed rate of requests. The proposed policy is completely dynamic and distributed.

The solution presented in this thesis improves over what has been previously proposed as it jointly addresses user requests redirection and replica placement and it introduces a way to enforce a strict control on the replicas level of utilization. Not only we minimize the costs for replicas placement and maintenance, not only we try to keep as low as possible the number of replicas adds and removals while satisfying all user requests but we do it distributively, load balancing the traffic among replicas and cloning (removing) replicas whenever their level of utilization is above (below) a desirable level of utilization. This provides a powerful tool to the CDN provider: setting the bounds of the replicas' utilization interval as well as the other parameters of our heuristics the CDN provider can have a strict control on how the CDN network will operate and can achieve different trade-offs between all the relevant performance metrics.



## Chapter 3

# Reference solutions

In this chapter we will first examine how to model dynamic replica placement (3.2). By assuming that user requests obey a Markovian model, it is possible to formulate the problem of optimal dynamic replica placement as a semi-Markov decision process accounting for the traffic, the user level of satisfaction as well as the costs paid to add, maintain or remove a replica from CDN servers ([7]). Although this model allows to achieve optimality and provides insights to the dynamic replica placement problem, it is not scalable, making it unusable in practice for controlling the operation of an actual CDN network.

Based on the outcomes of the resolution of the proposed model, we derived criteria on when and where to add/remove replicas. Such criteria are the basis for the design of a centralized heuristic described in this chapter. We also describe how to map greedy solutions for static replica placement ([71]) to our problem formulation. These schemes will be used for sake of benchmarking in the following chapters.

### 3.1 Problem formulation

We model the Internet network topology as a weighted undirected graph  $G = (V, E)$ . The vertex set  $V$  is the set of network nodes, while each edge in the set  $E$  represents a physical network link and is labeled with some kind of distance metric, e.g. the number of hops between the endpoints or a more complex function that takes into account the available bandwidth, giving lower cost to the backbone links than to the low speed access links.

We identify two subsets  $V_A$  and  $V_R$  of the set of network nodes  $V$ .  $V_A$  is the

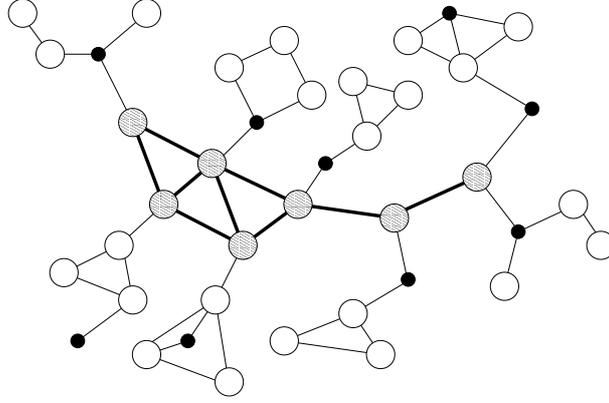


Figure 3.1: Small network topology generated with GT-ITM

set of CDN access nodes where the requests generated by the users enter the core CDN network.  $V_R$  is the set of service nodes in the core network where one or more content replica servers can be placed (we'll call these nodes either sites or nodes in the following). Figure 3.1 shows a possible 40 nodes hierarchical transit stub network topology, obtained by running the GT-ITM topology generator [36].

The white circles represent the access nodes, the gray big circles the sites that can host replicas, and the small black circles nodes only used for sake of routing. Thin and thick links reflect the low or high bandwidth of the links.

We assume that  $C$  content providers exploit the hosting service of the CDN. Customers entering the CDN through an access node in  $V_A$  can therefore issue requests for one of  $C$  sets of contents, and replicas of some of the  $C$  contents can be placed in each site in  $V_R$ . Requests entering the CDN are measured in units of aggregate requests. No more than  $V_{MAX}^A$  units of aggregate requests can be generated by an access node (to model the limited access link bandwidth). Requests for a given content are served by a suitable replica. To model user satisfaction, we assume that user requests cannot be served by a replica at a distance above a given threshold  $d_{max}$ . Users requests are redirected to the best available replica. This can be accomplished by several means, i.e., anycast, or DNS based redirection. We assume that each replica can serve up to  $K$  unit of aggregate request for that content (replica service capacity limit). No more than  $V_{MAX}^R$  replicas can be hosted at a given site (site resource limit).

We describe a given configuration of requests and replicas by means of a state vector  $\mathbf{x}$  of size  $C(|V_A| + |V_R|)$ :

$$\mathbf{x} = (\mathbf{a}, \mathbf{r}) = \left( a_1^1, a_2^1, \dots, a_{|V_A|}^1, a_1^2, a_2^2, \dots, a_{|V_A|}^2, \dots, a_{|V_A|}^C, \right. \\ \left. r_1^1, r_2^1, \dots, r_{|V_R|}^1, r_1^2, r_2^2, \dots, r_{|V_R|}^2, \dots, r_{|V_R|}^C \right)$$

in which the variable  $a_i^c$  represents the number of request units for a content  $c \in \{1, \dots, C\}$  at node  $i \in V_A$ , and  $r_j$  is the number of replicas of content  $c \in \{1, \dots, C\}$  placed at site  $j \in V_R$ .

We associate to each state a cost - paid per unit of time - which is the sum of a cost derived from the users perceived quality (users to replica distance, number of unsatisfied requests) and of the CDN infrastructure costs for hosting and maintaining replicas. We measure users perceived quality by means of a function  $A(x)$ . This is given by the sum over all users requests of the distance between the access node where the request is originated and the replica serving it. Since requests are served by the best available replica (i.e., the closest according to links metric) the redirection itself requires the solution of a minimum cost matching problem between the users requests and the available replicas, with the distance between access nodes and service sites as cost of the matching (we will explain this in detail in the following section). The solution of this problem yields the redirection scheme (which request is served by which replica) and the associated distance (cost).

A replica maintenance cost  $M(x)$  is used to model the costs of hosting replicas and keeping them up to date. We use a simple proportional cost model  $M(x) = C_{Maint} \sum_{j \in V_R, c=1, \dots, C} r_j^c$ , where  $C_{Maint}$  is a suitable constant. Two other costs  $C^+$  and  $C^-$  are paid by the CDN provider when dynamically adjusting the number of replicated servers, and are associated to the decision to add or remove a replica respectively.

### 3.1.1 Centralized redirection mechanism

It is worth spending a few lines for introducing how load balancing redirection can be modeled. We have represented the set of current user requests and the set of allocated replicas by a weighted undirected bipartite graph (see figure 3.2), that's been used as an input for a weighted minimum bipartite matching problem, solved using the efficient implementation from Andrew Goldberg and Robert Kennedy ([34]).

The matching is executed independently for each content  $c$ , it is performed over a

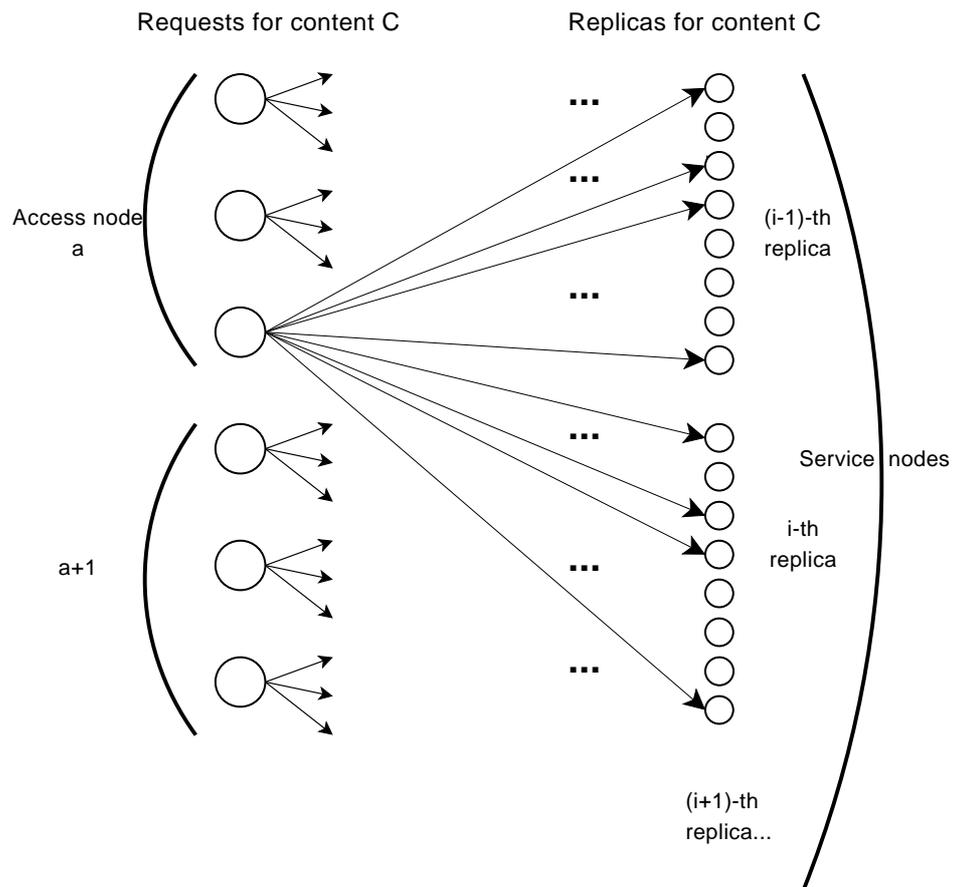


Figure 3.2: A model for the redirection scheme

bipartite graph whose nodes are partitioned into the set of user requests for  $c$  and the set of replicas of  $c$ . Since each replica can serve up to  $K$  requests,  $K$  nodes are needed to represent each replica in the second set. A finite-cost edge joins a user request and one of the  $K$  instances of a replica if and only if the distance between the user request access site and the replica hosting site is  $\leq d_{max}$ . When the distance is  $> d_{max}$ , an edge is added with infinite cost.

Load balancing is achieved by properly setting the weights of the edges in the resulting bipartite graph.

## 3.2 Optimal solution

The minimization of the long run costs in the model described above enables a decision making criterion that can be used to formulate dynamic replica placement strategies. Given a state, a cost function associated to it and the costs of replicating and deleting replicas, identify a strategy which dynamically allocates and deallocates replicas in response to users demand variations, so that the overall cost is minimized while meeting the constraints on the replica service capacity and site resources. In the following we introduce a Markov decision process to derive the optimal strategy for solving this problem as well as a centralized scalable heuristic.

The state of the Semi Markov Decision Process (SMDP) is formulated as in section 3.1 by a vector  $\mathbf{x}$  of size  $C \cdot (|V_A| + |V_R|)$ :

$$\mathbf{x} = (\mathbf{a}, \mathbf{r}) = \left( a_1^1, a_2^1, \dots, a_{|V_A|}^1, a_1^2, a_2^2, \dots, a_{|V_A|}^2, \dots, a_{|V_A|}^C, \right. \\ \left. r_1^1, r_2^1, \dots, r_{|V_R|}^1, r_1^2, r_2^2, \dots, r_{|V_R|}^2, \dots, r_{|V_R|}^C \right)$$

The states space  $\Lambda$  is then defined as:

$$\Lambda = \left\{ \mathbf{x} = (\mathbf{a}, \mathbf{r}) : \sum_{k=1}^C a_i^k \leq V_{MAX}^A; \sum_{k=1}^C r_j^k \leq V_{MAX}^R; \right. \\ \left. a_i^k, r_j^k \geq 0, i \in V_A, j \in V_R \right\}$$

Since the population of the described model is an aggregate figure of the requests traffic, the process dynamics is determined by changes in the average units of requests

at an access site for a given content. We model the aggregate request at site  $i \in V_A$  for a content  $k$  as a birth death process with birth-rate  $\lambda_i^k$  and death-rate  $\mu_i^k$ , respectively. When the system is in state  $\mathbf{x}$ , a state dependent decision can be made: a new replica of a given content can be placed at- or removed by- a site, or the system can be left as it is. The action space  $\mathcal{D}$  can be expressed by

$$\mathcal{D} = \left\{ (d_1^{1+}, d_1^{2+}, \dots, d_{V_R}^{1+}, \dots, d_1^{C+}, \dots, d_{V_R}^{C+}, d_1^{1-}, d_1^{2-}, \dots, d_{V_R}^{1-}, d_1^{C-}, \dots, d_{V_R}^{C-}), \right. \\ \left. d_i^{kx} \in \{0, 1\}, \right. \\ \left. \sum_{\forall i, k, x} d_i^{k,x} \leq 1; i = 1, \dots, |V_R|; x = +/ - \right\}$$

The indicator  $d_i^{k+} = 1$  represents the decision to add a replica of content  $k$  in site  $i \in V_R$ , while  $d_i^{k-} = 1$  stands for the decision to remove a replica of content  $k$  from site  $i$ . If all the indicators are null, the corresponding decision is to leave the replica placement as it is. For simplicity only one replica can be placed or removed at each decision time. <sup>1</sup>

The action space is actually a state-dependent subset of  $\mathcal{D}$  where a decision to add a replica is allowed only if the number of replicas hosted at the site is less than the maximum number of replicas  $V_{MAX}^R$ , and a decision to remove a replica can be made only if at least one replica is actually hosted in the considered site. If the system is in state  $\mathbf{x} = (\mathbf{a}, \mathbf{r}) \in \Lambda$  and the action  $\mathbf{d} \in D$  is chosen, then the dwell time of the state  $\mathbf{x}$  is  $\tau(\mathbf{x}, \mathbf{d})$  where

$$\tau(\mathbf{x}, \mathbf{d}) = \left[ \sum_{i=1}^{|V_A|} \sum_{k=1}^C (\lambda_i^k + a_i^k \cdot \mu_i^k) \right]^{-1} \quad (3.1)$$

The transitions that may occur in this model from an initial state  $\mathbf{x}$  to a final state  $\mathbf{y}$  can be due to an increasing aggregate request, in the form of a birth, or to a

---

<sup>1</sup>Please note that, as the model we are describing requires the system to react instantaneously to every change in the user requests traffic, adding or removing only one replica at a time is a realistic assumption. Whenever there is a single new request, there is no need to allocate more than one new replica, and whenever only one unit of requests leaves the system, there is no need to remove more than one replica.

decreasing request, in the form of a death in the underlying multidimensional birth and death process. The transition probability  $p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}}$  from the state  $\mathbf{x} = (\mathbf{x}_A, \mathbf{x}_R)$  to any state  $\mathbf{y} = (\mathbf{y}_A, \mathbf{y}_R \in \Lambda)$  under the decision  $\mathbf{d}$ , takes one of the following expressions, where  $\mathbf{e}_i^c$  is an identity vector with unary element in position  $(c \times |V_A| + i)$ .

Transitions due to an arrival of a unit of aggregate request for content  $c$  at site  $i$ :  
to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A + \mathbf{e}_i^c, \mathbf{x}_R + \mathbf{e}_j^k)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = \lambda_i^c \cdot d_j^{k+} \tau(\mathbf{x}, \mathbf{d}); \quad (3.2)$$

to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A + \mathbf{e}_i^c, \mathbf{x}_R - \mathbf{e}_j^k)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = \lambda_i^c \cdot d_j^{k-} \tau(\mathbf{x}, \mathbf{d}); \quad (3.3)$$

to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A + \mathbf{e}_i^c, \mathbf{x}_R)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = \lambda_i^c \cdot \tau(\mathbf{x}, \mathbf{d}) \cdot \left[ 1 - \sum_{j=1}^{|V_R|} \sum_{k=1}^C (d_j^{k+} + d_j^{k-}) \right]; \quad (3.4)$$

Transition due to a departure of a unit of aggregate request for content  $c$  at site  $i$ :  
to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A - \mathbf{e}_i^c, \mathbf{x}_R - \mathbf{e}_j^k)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = (\mathbf{x}_A)_i^c \cdot \mu_i^c \cdot d_j^{k+} \cdot \tau(\mathbf{x}, \mathbf{d}); \quad (3.5)$$

to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A - \mathbf{e}_i^c, \mathbf{x}_R - \mathbf{e}_j^k)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = (\mathbf{x}_A)_i^c \cdot \mu_i^c \cdot d_j^{k-} \cdot \tau(\mathbf{x}, \mathbf{d}); \quad (3.6)$$

to state  $(\mathbf{y}_A, \mathbf{y}_R) = (\mathbf{x}_A - \mathbf{e}_i^c, \mathbf{x}_R)$  with probability

$$p_{\mathbf{x}\mathbf{y}}^{\mathbf{d}} = (\mathbf{x}_A)_i^c \cdot \mu_i^c \tau(\mathbf{x}, \mathbf{d}) \cdot \left[ 1 - \sum_{j=1}^{|V_R|} \sum_{k=1}^C (d_j^{k+} + d_j^{k-}) \right]; \quad (3.7)$$

The transitions that are not considered in this list have probability 0. In order to create a decision criterion for the described model, an objective function is formulated. The state-related costs  $A(\mathbf{x})$  and  $M(\mathbf{x})$  introduced in section 3.1 are paid per unit of time as long as the system persists in the considered state  $\mathbf{x}$ . The costs  $C^+$  and  $C^-$  are instead transition-related and are only paid when a replica is actually added

or removed, i.e. when the corresponding transition occurs. Therefore a non-uniform cost function can be formulated as

$$r(\mathbf{x}, \mathbf{d}) = [A(\mathbf{x}) + M(\mathbf{x})] \cdot \tau(\mathbf{x}, \mathbf{d}) + \sum_{j=1}^{|V_R|} \sum_{k=1}^C \left( d_j^{k+} C^+ + d_j^{k-} C^- \right)$$

The uniformization technique ([5, 47, 76]) transforms the original SMDP with non identical transition times into an equivalent continuous-time Markov process in which the transition epochs are generated by a Poisson process at uniform rate. The transitions from state to state are described by a (discrete time) Markov chain that allows for fictitious transitions from a state to itself. The uniformized Markov process is probabilistically identical to the non uniform model. The theory of discrete Markov processes can then be used to analyze the discrete-time embedded Markov chain of the uniformized model. A uniform rate  $\Gamma$  can be taken as an upper bound on the total outgoing rate from each state thus obtaining a continuous time, uniform process with rate  $1/\Gamma$ . The following definition of  $\Gamma$  fits our needs.

$$\Gamma = \sum_{i=1}^{|V_A|} \sum_{k=1}^C \left[ \lambda_i^k + V_{MAX}^A \cdot \mu_i^k \right]$$

The transition probabilities of the uniformized process are formulated as in equations 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, substituting the non uniform dwell time  $\tau(\mathbf{x}, \mathbf{d})$  defined in equation 3.1 with the uniform dwell time  $1/\Gamma$ , and adding dummy transitions from each state to itself:  $\mathbf{y} = \mathbf{x}$

$$\tilde{P}_{\mathbf{xy}}^{\mathbf{d}} = \frac{1}{\Gamma} \cdot \left[ \Gamma - \sum_{i=1}^{|V_A|} \sum_{k=1}^C \left( \lambda_i^k + \mu_i^k \cdot (\mathbf{x}_A)^k \right) \right]$$

The cost function is uniformized as well, obtaining the following formulation of  $\tilde{r}(\mathbf{x}, \mathbf{d})$ :

$$\begin{aligned} \tilde{r}(\mathbf{x}, \mathbf{d}) &= \frac{r(\mathbf{x}, \mathbf{d})}{\tau(\mathbf{x}, \mathbf{d}) \cdot \Gamma} = \frac{1}{\Gamma} \cdot [A(\mathbf{x}) + M(\mathbf{x})] + \\ &\frac{1}{\tau(\mathbf{x}, \mathbf{d})} \cdot \sum_{j=1}^{|V_R|} \sum_{k=1}^C \left( d_j^{k+} C^+ + d_j^{k-} C^- \right) \end{aligned} \quad (3.8)$$

An optimal solution can be expressed through a decision variable  $\pi_{\mathbf{xd}}$  that represents the probability for the system to be in state  $\mathbf{x}$  and taking the decision  $\mathbf{d}$ . The

Linear Programming (LP) formulation associated with our SMDP minimizing the cost paid in the long-run execution is given by:

Minimize

$$\sum_{(\mathbf{x}, \mathbf{d}) \in \mathcal{S}} \tilde{r}(\mathbf{x}, \mathbf{d}) \cdot \pi_{(\mathbf{x}, \mathbf{d})} \quad (3.9)$$

constrained to:

$$\begin{aligned} \pi_{(\mathbf{x}, \mathbf{d})} &\geq 0 & (\mathbf{x}, \mathbf{d}) &\in \mathcal{S} & (3.10) \\ \sum_{(\mathbf{x}, \mathbf{d}) \in \mathcal{S}} \pi_{(\mathbf{x}, \mathbf{d})} &= 1 \\ \sum_{\mathbf{d} \in \mathcal{B}} \pi_{(\mathbf{j}, \mathbf{d})} &= \sum_{(\mathbf{x}, \mathbf{d}) \in \mathcal{S}} \tilde{p}_{\mathbf{x}\mathbf{j}}^{\mathbf{d}} \pi_{\mathbf{x}\mathbf{d}} \end{aligned}$$

where  $\mathcal{S}$  is the finite set of all feasible couples of vectors of the kind (state, decision). The problem defined in equation 3.10 can be solved by means of commonly known methods of the operations research [39]. We used the simplex method with sparse matrix support.

### 3.3 Dynamic, instantaneous, centralized replica placement heuristic

The solution to the optimization problem 3.10 is too computationally intensive but in the simplest scenarios. Therefore, in general, it is not feasible to compute the optimal policy. Here we propose an heuristic to decide the action  $\mathbf{d} \in \mathcal{D}$  to take upon transitions on the request access vector  $\mathbf{a}$ . The heuristic has been derived by closely studying how the optimal policy behaved in our experiments. In particular, we considered the case where the cost function imposes - in decreasing order - the following priorities to the resulting policy: (1) being able to serve user requests; (2) minimizing the number of replicas; (3) minimizing the distance between users and replicas. This was accomplished by setting the cost function parameters as follows:  $C_{\text{maint}} \gg \max_{e \in E} l(e)$ , with  $l(e)$  denoting the weight associated with link  $e$ ,  $C^+ = C^- = 0$ . With this choice, we expected the optimal policy to use the minimum number of replicas to serve all existing requests leaving at the same time enough spare capacity to accommodate for requests increases.

In our experiments, indeed, we observed that the optimal placement policy proactively replicates content in order to guarantee its availability in case of future requests increases. At the same time, to minimize the number of replicas, it detects and removes replicas which are not needed to serve either current requests or any possible unitary increase of them.

The SMDP model can be used only for very small scale networks. We thus proposed a centralized heuristic for the dynamic replica placement problem. The algorithm determines which replica placement/removal decision  $\mathbf{d}$  to take (if any) at each state change. The goal of the heuristics is to mimic the MDP optimal policy behavior by minimizing the number of replicas used to serve all existing requests leaving at the same time enough spare capacity to accommodate for requests increases.

---

**Algorithm 1** Centralized Replica Placement Algorithm

---

```

1:  $\mathbf{d}$  =do nothing;
2: while TRUE do
3:   wait for a change in  $\mathbf{a}$ ; take action  $\mathbf{d}$ ;
4:   if enough_replica_on_increase( $\mathbf{a}, \mathbf{r}$ ) then
5:      $\mathbf{d}$  = remove_replica()
6:   else
7:      $\mathbf{d}$  = add_replica()
8:   end if
9: end while

```

---

The centralized heuristic we propose, shown in Figure 1, works as follows. (For the sake of readability in the description below with  $\mathbf{a}$  and  $\mathbf{r}$  we mean  $a(t)$  and  $r(t)$ .) At each step, it first determines whether the current replica configuration  $\mathbf{r}$  can accommodate any possible increase in user requests  $\mathbf{a}$ . This is accomplished via the function `enough_replica_on_increase` (see Figure 2). In case that any possible increase in user requests can be accommodated by  $\mathbf{r}$  then the algorithm considers whether it is possible to remove a replica (`remove_replica()`); otherwise it tries to find a site where to add a replica (`add_replica()`). Observe that to mimic the behavior of a Markovian Decision Process, actions are decided in a given state but only taken in correspondence of the next transition.

The function `enough_replica_on_increase(( $\mathbf{a}$ ,  $\mathbf{r}$ ))` returns TRUE if any possible increase in  $\mathbf{a}$  can be served by the current replica configuration  $\mathbf{r}$  and FALSE otherwise. To this end, it uses the function `enough_replica(( $\mathbf{a}$ ,  $\mathbf{r}$ ))` which determines whether a given users requests vector  $\mathbf{a}$  can be served by the set of replicas  $\mathbf{r}$ .

---

**Algorithm 2** Centralized Replica Placement Algorithm:

boolean enough\_replica\_on\_increase( state( $\mathbf{a}, \mathbf{r}$ ) )

---

```

1: for all  $c = 1, \dots, C$  and  $i \in V_A$  do
2:   if !enough_replica( $\mathbf{a} + \mathbf{e}_i^c, \mathbf{r}$ ) then
3:     return FALSE
4:   end if
5:   return TRUE
6: end for

```

---

The function `enough_replica` itself is computed by solving a minimum matching problem between users requests and the available replicas from the solution of which we can determine whether all request in  $\mathbf{a}$  can be served by  $\mathbf{r}$  - more details about this will be given in section 3.1.1.

---

**Algorithm 3** Centralized Replica Placement Algorithm:

action add\_replica()

---

```

1: {Algorithm for Deciding where to Place a New Replica}
2: find  $I = \{(i, c) \mid i \in V_A, c = 1, \dots, C \mid$ 
    $\text{!enough\_replica}((\mathbf{a} + \mathbf{e}_i^c, \mathbf{r}))\}$ 
3: for all  $j \in V_R, c = 1, \dots, C$  do
4:   find  $J(j, c) = \{(i, c) \in I \mid \text{enough\_replica}(\mathbf{a} + \mathbf{e}_i^c, \mathbf{r} + \mathbf{e}_j^c),$ 
    $(\mathbf{a} + \mathbf{e}_i^c, \mathbf{r} + \mathbf{e}_j^c) \in \Lambda\}$ 
5:   if  $\max_{(j,c)} |J(j, c)| > 0$  then
6:      $(j^*, c^*) = \text{argmax } |J(j, c)|;$ 
7:      $\mathbf{d} = \text{place replica content } c^* \text{ in site } j^*;$ 
8:   else
9:      $\mathbf{d} = \text{do nothing};$ 
10:   return  $\mathbf{d};$ 
11:   end if
12: end for

```

---

`add_replica()` (algorithm 3) is called to determine content and location for a new replica. To this end it first identifies which requests increase would require additional replicas. This is accomplished in line 2 by determining the set  $I$  of the pairs  $(i, c)$  such that an increase of requests for content  $c$  at node  $i$  cannot be served by  $\mathbf{r}$  (line 2). It then computes the sets  $J(j, c) \subseteq I$  of users requests increment that could be served by an additional replica of content  $c$  in site  $j$  (line 3). If not all  $J(j, c)$  are empty, the content and location of the additional replica is then chosen by finding the site  $j^*$  and content  $c^*$  which maximizes  $|J(j, c)|$  (line 7). This to maximize the

probability of being able to satisfy a request increase.

---

**Algorithm 4** Centralized Replica Placement Algorithm:

---

action `remove_replica()`

---

- 1: {*Algorithm for Deciding which Replica to Remove*}
- 2: find  $U = \{(j, c) \mid j \in V_R, c = 1, \dots, C \mid$   
 $\exists i \in V_A \text{ enough\_replica}((\mathbf{a} + \mathbf{e}_i^c, \mathbf{r})) \text{ AND}$   
 $\text{!enough\_replica}((\mathbf{a} + \mathbf{e}_i^c, \mathbf{r} - \mathbf{e}_j^c)), (\mathbf{a} + \mathbf{e}_i^c, \mathbf{r} - \mathbf{e}_j^c) \in \Lambda\}$
- 3: find  $J = \{(j, c) \mid j \in V_R, c = 1, \dots, C (j, c) \notin U \mid$   
 $\text{enough\_replica}((\mathbf{a}, \mathbf{r} - \mathbf{e}_j^c)), (\mathbf{a}, \mathbf{r} - \mathbf{e}_j^c) \in \Lambda\};$
- 4: **if**  $|J| > 0$  **then**
- 5:    $(j^*, c^*) = \operatorname{argmin}_{(j, c) \in J} |S_{j^*}|, S_j = \{i, i \in V_A \mid d_{i, j} \leq d_{max}\}$
- 6:    $\mathbf{d} = \text{remove replica } c^* \text{ in site } j^*$
- 7: **else**
- 8:    $\mathbf{d} = \text{do nothing}$
- 9:   return  $\mathbf{d}$
- 10: **end if**

---

The procedure `remove_replica()` is called to determine whether to remove a replica. To this end, first it identifies the set  $U$  of replicas which should not be removed as they would be needed to serve an increase in users requests (line 2). Then, it determines, among the remaining replicas, the set  $J$  of the candidates for possible removal, i.e., all those replicas which are not used to serve current requests (line 3). Among these, it chooses to remove a replica from a node  $j$  which serves the smallest set of access nodes (line 5). Choosing the replica which serves the smallest population should minimize the likelihood to remove a replica which is going to be added soon again.

### 3.4 Static Replica Placement

In order to compare our solutions (centralized and distributed, see chapter 4) to static schemes which have been proposed in the literature and which have been proved to lead to good performance in case of static traffic, we need to map them to our problem formulation. Among the schemes introduced for static scenarios, we have selected the greedy heuristic introduced by Qiu et al. in [71] as it combines very simple rules with excellent performance and has been used as a reference by many authors ([45], [57], [80], [81], [69]). The model proposed in [71] does not account for practical constraints included in our model, such as 1) the maximum distance  $d_{max}$  between

user requests and their serving replicas, 2) the limit on the storage available at a site (number of replicas which can be allocated), and 3) the limit on the maximum number of user requests, which can be effectively served by a replica. We have therefore slightly modified the Qiu et al. greedy scheme to account for these features of our model without changing its philosophy. The resulting greedy algorithm is described in the following. In the description we first review the original scheme and then we explain how it has been modified to map to our problem formulation.

Qiu’s scheme is based on the idea of progressively adding replicas (up to a maximum number  $k$ ) trying to maximize the access service quality perceived by the final users. Such quality is inversely proportional to the weight of the path joining the access site and the serving replica. Replicas for a content  $c$  are allocated greedily, one after another, as follows. The hosting site  $v^1$  for the first replica is selected so that  $v^1 = \min_{j \in V(R)} \left\{ \sum_{i=1}^{|V_A|} a_i^c(t) \cdot d_{i,j} \right\}$ . In other words,  $v^1$  minimizes the sum of the distances between the users access sites and the replica site. The site  $v^i$  at which the  $i$ th added replica is hosted is selected so that the set  $\{v^1, v^2, \dots, v^i\}$  minimizes the distance between the users requests and their serving replicas.

To be able to compare the static greedy scheme performance to our heuristics, we had to slightly modify it to reflect our problem formulation. We have considered two implementations of such greedy approach which differ in the events triggering a new computation of the replica placement. The first greedy scheme (“instantaneous greedy” solution, or Greedy\_inst below) re-executes the replica allocation whenever a change in the users requests occurs. The second runs periodically, every  $T$  seconds. In this case, if replicas are allocated according to user requests at the time when the algorithm is run, with no clue on future traffic demands, some users requests may not be able to be satisfied.

In order to limit the occurrence of this problem, we have estimated the user requests dynamics for the upcoming time interval (of length  $T$ ) by using RLS (Recursive Least Square) prediction [10]. This allows us to estimate, based on current and past traffic, the future user requests traffic process from site  $i$  to content  $c$ .

In our model, user requests can only be satisfied by replicas at distance  $\leq d_{max}$ . It is therefore not important to minimize the sum of the distances between the users and their serving replicas (as the quality of the user access is defined in terms of  $d_{max}$ ). What matters here is the minimization of the number of replicas needed to satisfy all the user requests while at the same time meeting the load and storage constraints.

The greedy approach in our new problem formulation results in selecting each time as a new replica site the one that still has available storage and that can best increase the number of user requests satisfied.

The described greedy criterion is depicted in algorithm 5.

---

**Algorithm 5** Function **Greedy**( $\mathbf{a}(t), V_R, V_{MAX}^R, k$ )
 

---

```

1:  $\mathbf{r}(t) = \mathbf{0}$ 
2: while  $\text{min\_matching}(\mathbf{a}(t), \mathbf{r}(t)) \neq |\sum_i \sum_c a_i^c(t)|$  do
3:    $\text{max\_serv} = 0; \text{rep\_site} = 0;$ 
4:    $\text{rep\_cont} = 0; \text{rep\_distance} = \text{MAXINT};$ 
5:   for all  $j \in V_R$  do
6:     if  $\sum_c r_j^c(t) < V_{MAX}^R$  then
7:       for  $c = 1$  to  $C$  do
8:          $\text{serv} = \left| \text{min\_matching}(\mathbf{a}(t), \mathbf{r}(t) + \mathbf{e}_j^c) \right|$ 
9:         if  $(\text{serv} > \text{max\_serv})$  or  $(\text{serv} = \text{max\_serv})$  and
10:           $\sum_c \sum_i a_i^c(j, t) \cdot d(i, j) < \text{rep\_distance}$  then
11:             $\text{max\_serv} = \text{serv};$ 
12:             $\text{rep\_site} = j;$ 
13:             $\text{rep\_cont} = c;$ 
14:             $\text{rep\_distance} = \sum_c \sum_i a_i^c(j, t) \cdot d(i, j);$ 
15:          end if
16:        end for
17:      end if
18:    end for
19:   $\mathbf{r}(t) = \mathbf{r}(t) + \mathbf{e}_{\text{rep\_site}}^{\text{rep\_cont}}$ 
20: end while

```

---

The greedy procedure takes as input a snapshot of the user requests  $\mathbf{a}(t)$ , the set of possible replica sites  $V_R$ , the maximum number of replicas per site  $V_{MAX}^R$ , and the maximum load per replica  $K$ . The output produced by the procedure is the vector  $\mathbf{r}(t)$  that indicates the number of replicas to be allocated, their location, and the content of each replica.

At the start of the procedure operation the set  $\mathbf{r}(t)$  is empty (no replica has been allocated. Line 1). A new replica is added by selecting the pair (replica site, replica content),  $(j, c)$ , that satisfies the highest number of new requests (Lines from 4 to 16). (Possible ties are broken by using the sum of the distances between the users and the replica)

In particular, all possible replica sites  $j \in V_R$  which still have storage for replicas

are examined. The number of user requests which can be satisfied by the replica vector  $\mathbf{r}(t) + \mathbf{e}_j^c$  is computed and stored in the variable *serv*, where  $\mathbf{e}_j^c = \langle 0, \dots, 0, 1, 0, \dots, 0 \rangle$  is the vector with a 1 in the  $(c-1)|V_R| + j$ th position. *serv* contains the overall number of user requests that could be satisfied by adding one replica of content  $c$  at site  $j$ .

The value of *serv* is obtained by running  $C$  minimum matching procedures (one for each content, as of section 3.1.1). Here we call “cardinality of the minimum matching” the number of edges with finite cost in the solution output by the minimum matching procedure. The cardinality of the minimum matching for content  $c$  represents the maximum number of user requests for content  $c$  that can be satisfied by the current replicas allocation  $\mathbf{r}(t) + \mathbf{e}_j^c$ . The procedure call `min_matching( $\mathbf{a}(t), \mathbf{r}(t) + \mathbf{e}_j^c$ )` produces as output the sum of the cardinalities of the  $C$  minimum matchings, namely, the overall number of user requests which can be satisfied at this time. The (site)  $j$  and the (content)  $c$  that maximize the output of `min_matching( $\mathbf{a}(t), \mathbf{r}(t) + \mathbf{e}_j^c$ )` are selected as the site and the content of the next replica. The process is iterated until all current requests are satisfied.

In case of request traffic dynamically changing, the described procedure has to be run periodically as previously explained.



## Chapter 4

# Distributed, dynamic replica placement

In this chapter, a fully distributed scheme for dynamic replica placement is introduced.

The distributed solution is based on the idea that, for the CDN providers to see the infrastructure costs paid off, replicas utilization should be within a desirable interval  $[U_{mid}, U_{max}]$ . Overloaded replicas (serving  $U_{max}$  user requests or more) should clone themselves, as, otherwise, the user access performance would degrade severely, and the load should be shared among the replica and its clone. Underloaded replicas (serving less than  $U_{low}$  user requests) are a cost for the CDN provider. Our scheme tries to discourage redirection of the requests both toward overloaded and under-loaded replicas. When a replica is totally unused (which happens in case of underloaded replicas when their assigned user requests can be redirected toward alternative replicas) then the replica is discarded. We also enforce that all users requests are served by replicas at service sites within  $d_{max}$  from the access site. Decisions are taken at the CDN servers according to a distributed algorithm and based only on local knowledge of the CDN network status. More precisely, each service site has to monitor its replica utilization by monitoring the rate of requests replicas are serving. By this, it will decide whether replica load is too high (in which case it will clone it) or too low (in which case it will try to delete the replica). Furthermore, every replica needs to know some information about other nearby service sites: how many replicas each service site is hosting, and for which content. We will show in section [4.2.7](#)

that exchanging this information with all other service nodes at one hop distance is enough.

A comparative performance evaluation between the centralized heuristic, the distributed scheme sketched above and (static) solutions previously introduced in the literature allows us to quantify the advantages that can be obtained by a dynamic replica placement scheme and to assess the effectiveness of the proposed solutions in limiting costs and providing excellent users perceived quality.

## 4.1 Dynamic distributed replica placement heuristic

We briefly recall the problem formulation as given earlier in chapter 3, and complete it as needed for the following of this chapter.

We consider a CDN network hosting a set  $C$  of contents. Users access the CDN network through a set  $V_A$  of access nodes. The number of users accesses is expressed in units of aggregate requests from that access site, for each type of content. Replicas of the  $C$  contents can be stored in one or more sites among a set  $V_R$  of CDN servers sites. Each site  $j \in V_R$  can host up to  $V_{MAX}^R$  replicas, each serving requests as long as the replica load is below a threshold  $K$  (load threshold). A weight is associated to the route from a user (access node)  $i$  to a replica  $j$ . The weight  $d_{i,j}$  indicates the user perceived quality of accessing that replica. A user is said to be satisfied when the weight of the route to the best replica is below a given threshold  $d_{max}$ . Each access node  $i$  has therefore associated a set  $\rho(i)$  which include all the server sites able to satisfy users requests generated at  $i$ .

We denote by  $x_{i,c}$  the volume of user requests originated at node  $i \in V_A$  for content  $c \in C$  and by  $\alpha_{ij,c}$ ,  $\sum_{j \in \rho(i)} \alpha_{ij,c} = 1$ , the fraction of requests for content  $c$ , originated at node  $i$ , which are redirected to node  $j$ . We denote by  $r_{j,c}$  the amount of replicas (resources) allocated to content  $c$  at node  $j \in V_R$ . The dynamic replica placement goal is to identify a strategy which dynamically allocates and deallocates replicas in response to users demand variations so that the overall cost (overall number of replicas) is minimized while satisfying the users requests and meeting the constraints on the replica service capacity and site resources.

In this section we describe a distributed scheme to allocate and deallocate replicas, so that the user requests are satisfied while minimizing the CDN costs in a dynamic scenario. Being dynamic, this scheme always accounts for the current replica

placement, adding replicas or changing replica locations only when needed. Each site  $j \in V_R$  decides on whether some of the replicas it stores should be cloned or removed. This decision is based on local information such as the number and content of the replicas stored at  $j$ , the load of such replicas, and the user requests served by the replicas hosted at the site.

**Cloning of a replica** Each site  $j$  stores information about its local neighborhood in the CDN network topology. More specifically, site  $j$  knows the set  $\alpha(j)$  of the nodes in  $V_A$  which are distant at most  $d_{max}$  from it, and the set  $\rho(j)$  of the nodes in  $V_R$  that are distant at most  $d_{max}$  from any of the nodes in  $\alpha(j)$ . The first set includes all those access sites which can generate requests that  $j$  can satisfy. The set  $\rho(j)$  is the set of serving sites that can cover for  $j$ . Of these sites,  $j$  maintains information about the replicas they host, and their content. Based on this sole information, site  $j$  is able to decide when to either clone or delete a replica, and in case of cloning, where the clone should be hosted. In particular, if (and only if) one of the replicas hosted at  $j$  is overloaded (i.e., serves more than  $U_{max}$  requests),  $j$  decides to clone it. In section 4.2.7 we'll also relax this assumption about the knowledge of the status of nearby replicas and analyze its effect.

Cloning a replica implies the selection of a host for the clone. To this aim, when cloning  $c$ , site  $j$  selects the site  $j' \in \rho(j)$  that satisfies the following requirements: it still has room for hosting new replicas, i.e., adding a new replica there would not violate the constraint on the maximum number of replicas per site, and it is able to satisfy the largest amount of user requests for  $c$  currently redirected to the overloaded replica in  $j$  (ties are broken by selecting the hosting site  $j'$  closest to the user requests). Site  $j$  then contacts site  $j'$  asking it to host the clone. Upon confirming availability,  $j$  physically sends the clone to  $j'$ , and  $j'$  informs all the access sites in  $\alpha(j')$  and all the serving sites in  $\rho(j')$  of the new replica that it is hosting (see algorithm 6).

The function to clone a replica of content  $c \in C$  (algorithm 6) is called by a server site  $j$  whenever the load of one of its replicas of content  $c$  exceeds  $U_{max}$ . The function outputs the server sites(s)  $j^*$  where the cloned replica(s) should be added. The detailed operations of the function *add\_replica* are reported below.

When the function is invoked node  $j$  first computes the number  $l_{j,c}$  of requests for content  $c$  it currently serves (line 1). If the average load of the replicas for content  $c$  hosted at node  $j$  is above the threshold  $U_{max}$  (line 2) then a new replica of content

---

**Algorithm 6** Dynamic distributed RPA:

action `add_replica(j, c)`

---

**Require:** this function is called when the load  $l_j > U_{max}$

**Require:**  $j \in V_R$

```

1:  $l_{j,c} = \sum_{i \in V_A} \alpha_{ij,c} \cdot x_{i,c}$ 
2: while  $\frac{l_{j,c}}{r_{j,c}} - U_{max} > 0$  do
3:    $n^* = 0$ 
4:    $d^* = \infty$ 
5:    $j^* = \text{undefined}$ 
6:   for all  $j' \in \rho(j)$  s.t.  $r_j < V_{MAX}^R$  do
7:      $l'_{j',c} = \sum_{i \in \alpha(j')} \alpha_{ij',c} \cdot x_{i,c}$ 
8:      $total\_distance = \sum_{i \in \alpha(j')} \alpha_{ij',c} \cdot x_{i,c} \cdot d_{i,j'}$ 
9:     if  $(l'_{j',c} < n^*) \vee$ 
        $(l'_{j',c} = n^* \wedge d_{total} < d^*)$  then
10:       $d^* = d_{total}$ 
11:       $n^* = l'_{j',c}$ 
12:       $j^* = j'$ 
13:     end if
14:   end for
15:   if  $j^* = \text{undefined}$  then
16:     exit
17:   end if
18:   ask  $j^*$  to add a replica for content  $c$ 
19:   compute  $l''_{best_{vr},c} = \min(\sum_{i \in \alpha(best_{vr})} \alpha_{ij,c} \cdot x_{i,c}, 1)$ 
20:    $l_{j,c} = l_{j,c} - l''_{j^*,c}$ 
21:   remove from the set of requests those that can be offloaded
22: end while

```

---

$c$  will be added to the network with the aim to offload overloaded replicas (lines 2–22). The new replica location  $j^*$  is chosen based on  $l'_{j^*,c}$ , which is defined as the number of user requests currently served by a replica hosted at node  $j$  which could be offloaded to a new replica added at  $j^*$ . The higher the value of  $l'_{j^*,c}$  the more suited  $j^*$  is to host the new replica. Ties are broken by selecting, among those sites maximizing  $l'_{j^*,c}$  and having space for an additional replica, the site which maximizes the satisfaction of the offloaded users requests i.e., the site which minimizes the total distance  $\sum_{i \in \alpha(j^*)} \alpha_{ij,c} \cdot x_{i,c} \cdot d_{i,j^*}$  (lines 8–12).

If a suitable site  $j^*$  for hosting the clone is found node  $j$  sends a message to it asking to add a replica of content  $c$  (line 18). Node  $j$  also computes the maximum amount  $l''_{j^*,c} = \min(\sum_{i \in \alpha(j^*)} \alpha_{ij,c} \cdot x_{i,c}, 1)$  of requests it is currently serving that can be offloaded to  $j^*$  without overloading the new replica (lines 18–21). It then checks whether the requests  $j$  is serving and that cannot be offloaded to  $j^*$  are still too many, i.e., if they still overload node  $j$  replicas (line 21). If this is the case an additional replica is added to the network and the procedure is re-executed on the requests that cannot be offloaded.

It might happen that the whole system (or a portion of it) is overloaded so that no site at which a replica can be added is identified (lines 15–17). In this case the algorithm has no way to improve the situation and the procedure is exited.

**Removing replicas** If (and only if) one of site  $j$ 's replicas can be removed without affecting the capability of satisfying all the requests served by the replicas at  $j$ , then  $j$  removes that replica. The decision of removing a replica is based on a weighted average of the user requests currently redirected to the replica and past requests. This provides some smoothing in the decision process and helps avoiding the ping pong effect for which a replica is added and soon removed. Upon deciding to remove a replica, node  $j$  informs all the nodes in  $\alpha(j)$  and in  $\rho(j)$ .

We assume (see chapter 5) that the redirection scheme employed in the CDN performs load balancing among the different replicas and prevents overloaded or underutilized replicas to be selected as “best” replicas unless needed. Such a load balancing redirection scheme allows both to divert requests from underutilized replicas (thus removing such replicas whenever possible) and it also motivates the need for cloning the replica whenever the threshold  $U_{max}$  is reached. Not only the extra replica will be able to serve some of the user requests reducing load and providing a better

service to the final users, but also, the case of overloaded replicas at  $j$  implies that none of the other replicas hosted at sites distant less than  $d_{max}$  from the user requests could cover for  $j$  without reaching the threshold  $U_{max}$  themselves. In such scenario it is unavoidable to add an extra replica.

#### 4.1.1 Handling replicas under the target utilization level

The distributed algorithm we propose tries to delete underloaded replicas. The underloaded replica first tries to direct away all requests it is currently serving, by providing the access nodes with a false feedback making them believe their requests could be better served by a different replica. If all the currently served user requests can be redirected to other replicas (the replica has no requests to serve) then the replica is safely deleted.

A less critical (but common) situation is that in which the replica serves an adequate number of user requests ( $> U_{low}$ ) even if its current load is below what desirable to justify the costs for replica maintenance (i.e. it is below a threshold  $U_{mid}$ ). This is the case of a replica under the target utilization level. To cope with this case we designed a distributed probabilistic mechanism which tries to redirect the user requests and delete replicas over time so that the load of current replicas is kept in the range ( $U_{mid} < l < U_{max}$ ). This mechanism is the core of our solution since it allows us to specify the desirable utilization range for the replicas. Replicas will be dynamically allocated and deallocated in order to satisfy all user requests but also in order to ensure that each replica the CDN provider pays for is properly utilized.

To make an example of why replicas can fall below the target utilization level, consider the case in which a high volume of user traffic requests forces the CDN provider to place many replicas to be able to satisfy all user requests. If the user requests volume decreases and user requests are load balanced among the existing replicas (via the scheme shown in the next chapter) then the utilization of the different existing replicas will uniformly decrease. Until the traffic decrease is so significant that replicas starts falling below the  $U_{low}$  threshold, no replica removal would be performed according to the replica removal algorithm. However, in such a situation it might be possible to remove a significant number of replicas while still being able to satisfy all user requests.

To achieve this goal in our scheme, a replica whose load is below the target utilization  $U_{mid}$  will try to direct requests away (without affecting underloaded and over-

loaded replicas). Let  $j$  be a site at which a replica below the target utilization level is hosted. When providing feedbacks to the access sites  $i$  on how effectively requests generated at  $i$  can be served by the replica hosted at  $j$ , node  $j$  will first toss a coin and with a probability  $p_{i,j,c}$  will decide to ‘cheat’ advertising a bad service (still better than what is advertised by underloaded or overloaded replicas). The higher the load of a replica, the lower the probability it will cheat. The cheating replicas will then be partially offloaded provided that there are other replicas that can cover for them whose load is between  $U_{low}$  and  $U_{max}$ . As their load is decreased they will have even higher probabilities of directing further requests away. Eventually such replicas will redirect away all the requests they’re serving and will be removed (if possible).

Note that instead of adding this third threshold one could have thought of using a higher value for the  $U_{low}$  parameter. However, when  $U_{low} \lesssim U_{max}$ , the problem is that many replicas would be considered underloaded, and this could possibly lead both to system instability and in too many replicas trying to have their load reduced. To this end we introduce the middle threshold as a means to consider replicas for offloading, but behaving in a milder way, in order to have a more stable system.

#### 4.1.2 The bootstrap case

The last case to be considered is the following. It might happen that a user makes a request for a content  $c$  from an access site  $i$  which does not have any replica of content  $c$  within distance  $d_{max}$  (this is often the case when the replica allocation process starts). In this case the request is directed to the origin server, that clones a copy of its content to a replica site  $j$  that is distant at most  $d_{max}$  from  $i$ . Among the possible sites, the origin server selects the site  $j$  that can satisfy the requests originated by the largest number of sites of  $V_A$ . The selected site is clearly highly likely to be able to satisfy the largest number of requests in the near future. Ties are broken by selecting a node that minimizes the overall average distance from the requests. (See algorithms 7 and 9)

**Static add replica** Procedure `static_add_replica` is executed by the origin server. It needs to know the set `unsat` of all user requests, for all contents, that were redirected to the origin server (see the first `REQUIRE` statement), and the set of access nodes `unreached[c]` for every possible content  $c$  (this is in the second `REQUIRE` statements). `unsat` is expressed as a set of tuples  $(a, i, c)$  where  $i$  is the access node of

---

**Algorithm 7** Bootstrap RPA Algorithm:

action static\_add\_replica

---

**Require:**  $\text{unsat} = \left\{ (a, i, c) \in (V_A \times [V_{MAX}^A] \times C) : \right.$

$\left. \text{were assigned to the origin server} \right\}$

**Require:**  $\text{unreached}[c] = \{i \in V_A \text{ s.t. } \forall j \in V_R, r_{j,c} > 0 \Rightarrow d_{i,j} > d_{max}\}, c \in C$

```

1: while  $\text{unsat} \neq \emptyset$  do
2:    $(j, c) = \text{static\_get\_best\_replica}(\text{unsat}, \text{unreached})$ 
3:   if  $j$  is defined then
4:      $r_{j,c} = r_{j,c} + 1$ 
5:      $\text{unsat} = \text{unsat} \setminus \text{servable\_requests}(j, c, \text{unsat})$ 
6:      $\text{unreached}[c] = \text{unreached}[c] \setminus \{i \in \text{unreached}[c] : d_{i,j} \leq d_{max}\}$ 
7:   else
8:     exit
9:   end if
10: end while

```

---



---

**Algorithm 8** function servable\_requests( $j, c, R$ )

*Helper function*

```

1:  $\text{served} = \emptyset$ 
2: for all  $(i, r, c') \in \text{unsat}$  s.t.  $c' = c$  do
3:   if  $d_{i,j} \leq d_{max}$  then
4:      $\text{served} = \text{served} \cup \{(i, r, c')\}$ 
5:   end if
6:   if  $|\text{served}| \geq U_{max}$  then
7:     exit loop
8:   end if
9: end for
10: return  $\text{served}$ 

```

---

the request,  $c$  the requested content, and  $r$  an index needed to distinguish the specific unit of requests among the  $V_{MAX}^A$  possible ones that can flow with the same  $(i, c)$  values. The loop beginning at line 1 looks for a suitable replica for these requests and continues until enough replicas were found for all of them. In line 2 the function `static_get_best_replica` is called, that is the most important point of the procedure. This procedure returns a couple  $(j, c)$  that's considered "optimal", where  $j$  is a content suitable for allocating a new replica and  $c$  the content to be allocated. If such a couple exists, the algorithm deletes from `unsat` all requests that the new replica can potentially serve (the function `servable_requests` of algorithm 8 finds them, limiting to  $U_{max}$  requests per replica), and updates the set `unreached` of clients that have no nearby replicas for that content.

---

**Algorithm 9** Bootstrap RPA Algorithm:

function `static_get_best_replica(unsat, unreached)`

---

*Helper function for algorithm 7*

```

1:  $n^* = 0$ 
2:  $best\_reached = 0$ 
3:  $d^* = \infty$ 
4:  $j^* = undef$ 
5:  $best\_c = undef$ 
6: for all  $j \in V_R, c \in C$  s.t.  $r_j < V_{MAX}^R$  do
7:    $served = \{(i, r, c) \in unsat \text{ s.t. } d_{i,j} \leq d_{max}\}$ 
8:    $reached = \{i \in unreached[c] \text{ s.t. } d_{i,j} \leq d_{max}\}$ 
9:    $d_{total} = \sum_{(i,j) \in served} d_{i,j}$ 
10:  if  $(|served| > n^*) \vee$ 
       $( (|served| = n^*) \wedge (|reached| < best\_reached) ) \vee$ 
       $(|reached| < best\_reached) \wedge (d_{total} = d^*)$  then
11:     $n^* = |served|$ 
12:     $best\_reached = |reached|$ 
13:     $d^* = d_{total}$ 
14:     $j^* = j$ 
15:     $best\_c = c$ 
16:  end if
17: end for
18: return  $(j^*, best\_c)$ 

```

---

Going into further details for function `static_get_best_replica` (algorithm 9), lines 1 to 4 are used to reset the three values that the function evaluates for all nodes before choosing the one that minimizes them, and the reference to the best

server. The loop in line 6 iterates over all service sites that have room available for a new replica. For all of them it evaluates the number of requests in `unsat` that it could serve, the number of reached access nodes and the total distance from these requests (lines 7 to 9). Finally, it compares these three values with the current minimum giving precedence to `served`, breaking ties by comparing `reached` and comparing `total_distance` as a final resort (line 10). Lines 11 to 15 are needed to update the best candidate along with its associated values. The couple (node, content) is eventually returned to the caller in line 18.

## 4.2 Performance evaluation

### 4.2.1 Simulation environment

In this section we report the results of a simulation-based performance evaluation aimed at assessing the effectiveness of the distributed heuristics we have proposed for replica placement and user requests redirection. Both the two heuristics have been implemented in the OPNET simulator, widely used and accepted by the research community for network simulations. Traffic is expressed in terms of blocks of aggregated requests. In our simulations the request redirection reacts instantaneously every time the traffic generated by an access site changes (this corresponds to major load changes in the traffic). This in turn may lead to cloning or removal of replicas. The weighted average mechanism for the replica removal (page 55) has been disabled in all these simulations, to allow immediate replica deletion.

Our performance evaluation has proceeded in steps. First, we have performed experiments using the small topology of picture 3.1 to quantify the advantages of the proposed distributed dynamic replica placement heuristic with respect to the static schemes which have been proposed in the literature, and with respect to the centralized heuristic proposed in section 3.3. We have then proceeded to investigate thoroughly the performance of the distributed heuristic for dynamic replica placement. First we have investigated the impact of the parameter  $U_{mid}$  on the distributed dynamic replica placement heuristic. Then we have performed extensive assessment of the heuristic when varying the network topology, traffic, number of contents etc...

All these experiments aimed at checking the effectiveness of the placement algorithm combined with the load balancing RRS, keeping the replicas at the desired level of utilization. They validated the ability of the proposed scheme to exploit a proper

tuning of the target utilization parameter  $U_{mid}$  to provide a fine grained control of the trade-off between number of allocated replicas, degree of utilization of the allocated replicas, distance of the users from the replica and frequency of replicas adds and removals. Furthermore, we showed what happens when increasing the number of contents managed in the CDN, assuming that the popularity of the objects follows a Zipf-like distribution (section 4.2.6). Finally, we relaxed the assumptions about the knowledge each replica must have of its neighborhood, and show in 4.2.7 how this affects the behavior of our solution.

To compare the different algorithms we focused on metrics representing the user perceived quality, and the CDN costs. In particular our investigation included:

1. **user perceived quality** defined as the average distance between the access site generating a request and the replica serving it
2. **the average number of replicas** defined as the average number of replicas in the system
3. **the number of replicas added** defined as the average number of clone operations during a period of 100000s
4. **the number of replicas removed** defined as the average number of replica removals during an interval of 100000s
5. **average replica utilization** defined as the average number of requests served by each replica during the simulation, expressed as a percentage of the replica capacity
6. **unserved requests** This is the percentage of requests that could not be served by a replica, and had to be served by the origin server

#### 4.2.2 Simulating the Request Redirection System

In the following we will describe a centralized request redirection scheme able to achieve load balancing while keeping the replica load within a desired range. We used such a centralized scheme to model request redirection in the simulations to assess the distributed dynamic replica placement. In the next chapter we will show how we can design a distributed request redirection scheme well matching the behavior of the centralized solution here described.

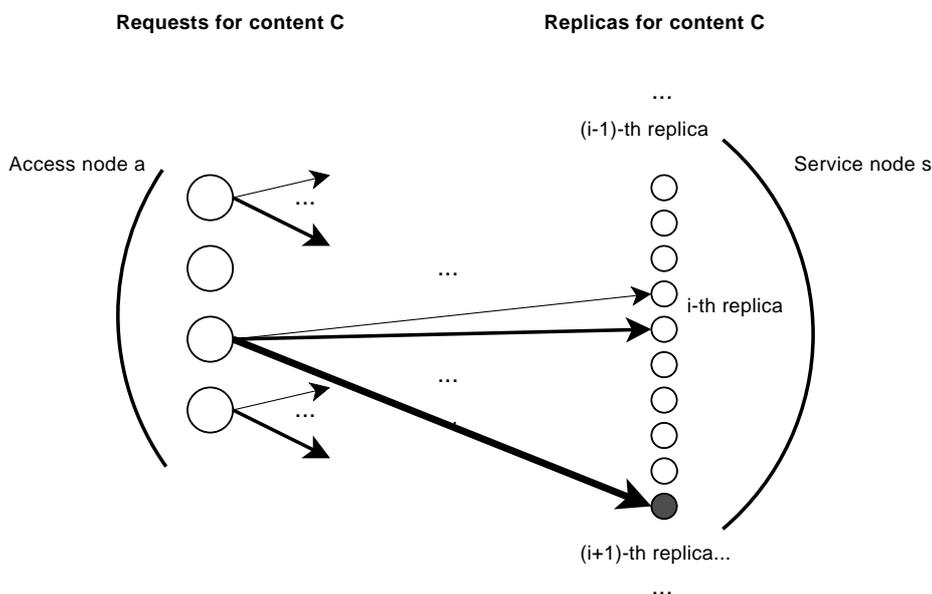


Figure 4.1: A model for the redirection scheme

The centralized request redirection is based on the minimum weighted bipartite matching, previously introduced in section 3.1.1. In picture 4.1 the nodes on the left represent the current users requests, the nodes on the right the allocated replicas. Instances of a replica corresponding to overloaded states are colored black.

We aim at achieving load balancing among the replicas avoiding to overload replicas unless needed. Also, we would like to be able to favor replica removal whenever a smaller set of replicas can satisfy users requests.

Load balancing and the restricted use of underutilized and overloaded replicas is achieved by properly setting the weights of the edges of the bipartite graph. In particular, as shown in the figure, load balancing is achieved by weighting the edges as a function of a replica's load: the higher the load, the higher the weight. Edges to the  $k$ -th instance of a replica (whose use would overload that replica) are associated to much higher weights (via the addition of a high constant). While this discourages the selection of that edge, it still makes that link available in those cases where a user request would otherwise remain unsatisfied. In the case of underutilized replicas (not shown in figure), edges leading to those replicas are associated much higher weights in the attempt of discouraging their use and to check whether requests could be satisfied without them.

When a single site hosts more than one replica for the same content, we assume that a local mechanism is employed to optimize load balancing's behavior. Let's assume given a node that hosts  $m$  replicas for a specific content  $c$ . Let us also assume that the total number of incoming requests assigned to the node by the redirection mechanism is  $n$ . The local mechanism will direct the largest possible number of requests to as few replicas as possible, leaving one replica with the least possible number of users to serve. Ranking replicas for the same content according to a fixed ordering (this could simply be given by the time the replica was allocated):

$$n_{full} = \left\lfloor \frac{m}{U_{max}} \right\rfloor$$

is the number of replicas that will serve  $U_{max}$  clients; that is, replicas whose order is in the range  $[0..n_{full}]$  will have load  $U_{max}$ .

The module of  $m$  over  $U_{max}$ :

$$l_r = m \% U_{max}$$

is the load that will be served by the  $(n_{full} + 1) - th$  replica. All replicas in the range  $[n_{full} + 2..n]$  (if any) will serve no requests.

This means that as soon as the load decreases one replica hosted at the node will get no requests and will be removed.

### 4.2.3 Topology Generation

A first set of experiments was performed on the small topology generated with the GT-ITM generator, shown in picture 3.1. We then performed experiments on the public ISP mappings provided by the RocketFuel project [75], providing a complete description of the ISP backbone topology along with link weights. Every node of the backbone network is bound to a geographic locality, where the ISP has an access point. So we considered this point as a service node, that is, a suitable location for replicas. We added to each service node a number of access nodes in its one-hop neighborhood: every service node  $s$  has a set of access nodes at one-hop distance, inversely proportional to the  $s$ 's degree in the backbone graph.

Access nodes can represent broadband users, or narrowband users. Based on the survey in [83], we assumed a probability of 43.71% of the link being broadband,



Figure 4.2: AT&T backbone topology

56.29% of the links being narrowband. The link from a broadband customer to a server site has a light weight, while a narrowband customer is linked to the backbone via a heavy weighted link.

The weight of very high data rate access links is uniformly chosen in the range 10 – 12; the weight of low data rate access links is higher, being selected randomly and uniformly in the interval 13 – 15.

We associated to every service node a *backup* service node, randomly chosen among its service nodes neighbors. We added an additional link from every access node to the backup node associated to its service node, following the same user class distinction and the same weight distribution described above.

**Characteristics of the topologies** We examined a number of topologies among the ones provided by the RocketFuel project, analyzed their characteristics and finally chose the two topologies hereby described in details, because of their different features..

The AT&T topology as from RocketFuel (see figure 4.2, from [78]) is spread almost only in the US continental territory (the heaviest link is the one connecting Los Angeles to Honolulu). The weight of a link between two localities is strongly correlated with their geographical distance, and there are only a few links connecting far away localities.

The topology of the anonymous provider identified with the number “1299” in RocketFuel (figure 4.3) is quite different: its graph spans across US, Southern, Central and Eastern Europe, and there is a higher percentage of links with a “high” weight

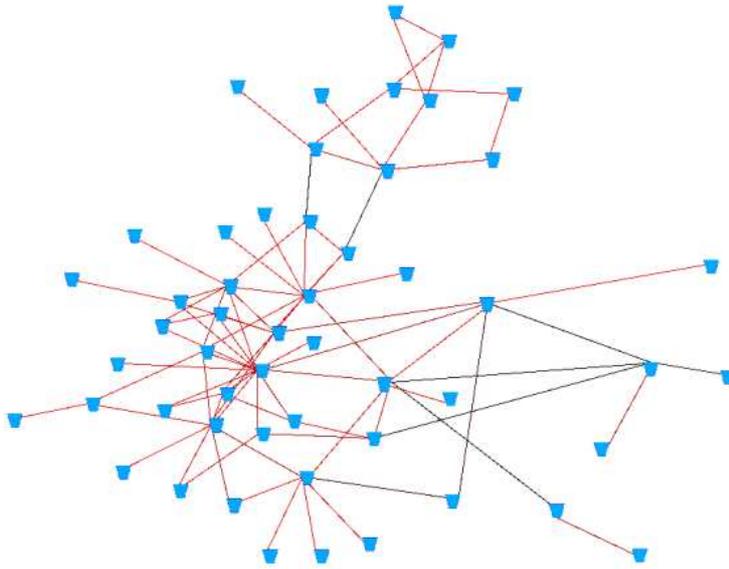


Figure 4.3: “1299” backbone topology

connecting distant localities (mainly, the links connecting different continents).

Figures 4.4a and 4.4b capture these two characteristics. In the x axis, there is the value of  $d_{max}$ , in the y axis the percentage of access nodes each service nodes has at a distance  $d \leq d_{max}$ , on average. The minimum value of x for which the curve is defined, corresponds to the value at which every access node to have at least a service node within  $d_{max}$ . In the curve for AT&T, the number of “reachable” access nodes per service node, is more than 90% when  $d_{max} = 25$ , while for  $d_{max} = 18$  every service node can serve on average more than 20% of the access nodes.

For “1299”, things are quite different. The y value increases almost linearly for values of x less than 30. Then, in the range [40 : 60], the value quickly increases and then reaches 100%.

The minimum allowed value for  $d_{max}$  corresponds to the point at which service nodes can mainly serve nearby access nodes. In this case  $d_{max}$  is not high enough to allow replicas to serve access nodes if the user to replica path has to traverse one or more of the heavy links connecting continents.

In “1299” the curve then flats and increases again for values of  $d_{max}$  at which the neighborhood of the service nodes grows over the border represented by the heavy

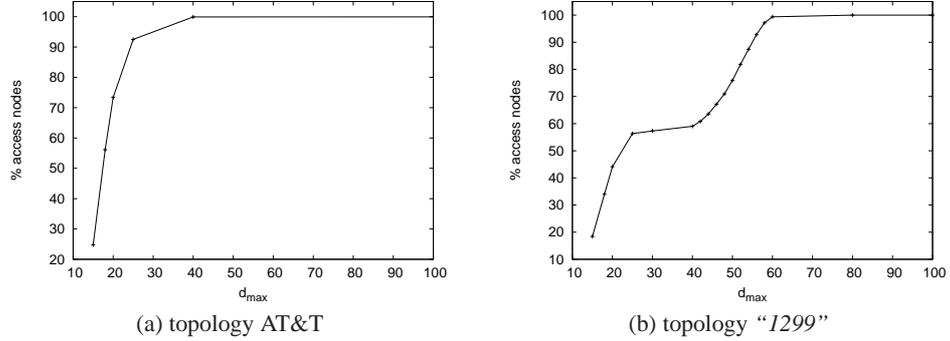


Figure 4.4: Access nodes coverage

links between the continents, gradually reaching all nodes in the network.

We studied these and many others topologies, and chose these two ones because they're very representative of the majority of the different possible network graphs.

#### 4.2.4 Simulation results: comparing greedy static, centralized heuristic, distributed heuristic

We run this first set of experiments using the topology in figure 3.1, with 24 access nodes (the white ones) and 7 service nodes (the gray ones). The thin lines represent slower links (weight 2), the thick ones faster links (weight 1). The aggregate requests at site  $i \in V_a$  are modeled as independent Markov birth-death processes. To focus on the relative algorithm behavior we considered just one content, with  $d_{max} = 6$  and  $d_{max} = \infty$ ,  $V_{MAX}^R = 10$ . For the distributed algorithm we set  $U_{low} = 20\%$ ,  $U_{max} = 90\%$  and two different values of  $U_{mid}$ ,  $U_{mid} = 20\%$  and  $U_{mid} = 90\%$ . These two values of  $U_{mid}$  represent the two possible extremes: in the first case, the algorithm behaves as a pure load balancing scheme; the second case is the other extreme with the target utilization almost as high as  $U_{max}$ . All other possible behaviors lie in between.

Results for the different metrics are illustrated in figures 4.5 and 4.6, along with the 95% confidence intervals. Replicas adds and removals are normalized to a time interval of length equal to 100000 time units. As expected, the static greedy algorithms generally result in a slightly lower number of replicas and user-replica distances than the dynamic algorithms. The difference in terms of average number of replicas and user-replica distance, with respect to the proposed dynamic replica placement is however quite limited, never topping 9% for the former, 5% for the latter. As

expected, both greedy algorithms have instead very high reconfigurations costs as their decisions are oblivious of prior states. The more often replicas are re-allocated, the better the placement, the higher the reconfiguration costs.

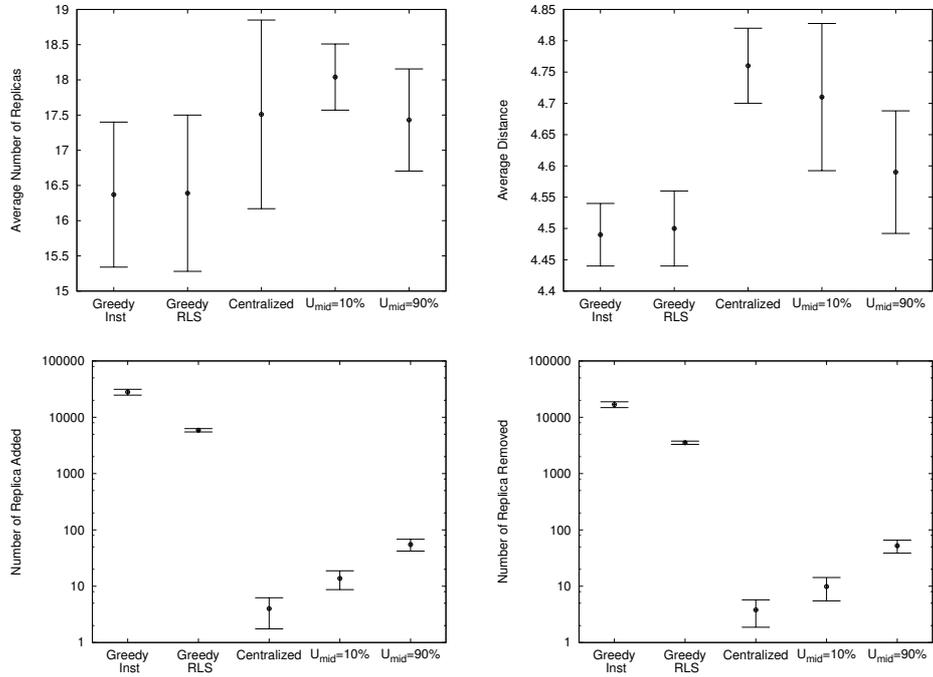
The centralized algorithm uses less replicas than the distributed one, but experiences the highest average distance. This can be explained by looking at the algorithm behavior (algorithm 1). The centralized algorithm makes placement (and removal) decisions taking into account potential future user requests increases and locating the replica where it can serve the largest population. By doing so, replicas are typically steadily placed in barycentric positions. The average distance is thus greater than allowed by the greedy algorithm which just shuffles back and forth replicas close to the users, according to the instantaneous traffic pattern. The distributed algorithm, on the other hand, uses only local information and lacks coordination among nodes: each node makes replica addition and removal decisions based on the local load.

The distributed heuristic performs comparably to the greedy ones in terms of number of replicas and distance to the best replica, but is able to reduce of up to three order of magnitude the number of replicas adds and removals.

When  $U_{mid} = 10\%$ , the distributed heuristic is characterized by a very small number of replicas additions and removals. This yields stable, low-varying replica configurations, but requires a higher number of replicas. We verified that there are many such stable configurations and the algorithm converges to one of them depending on the dynamics during the initial transient. This also explains the larger spread of the average distance between requests and serving replicas in the distributed algorithm.

The comparison of different settings of the  $U_{mid}$  threshold provides interesting results. The higher the  $U_{mid}$ , the lower both the average number of replicas AND the user-replica distance. This behavior can be explained by looking at the distributed algorithm behavior. With  $U_{mid} = 90\%$  all the used replicas tend to be fully utilized and their number closely follows the minimum required by current requests (replicas are added and removed closely following the traffic dynamics). As a consequence, the algorithm now tends to remove and add replica more frequently, and by so doing, it is more likely to place the replicas close to the users according to the instantaneous traffic pattern. This in turn reduces the user-replica average distance. The toll to pay is in terms of a higher number of replicas added/removed. This number is however order of magnitudes lower than in the greedy static scenarios.

Finally, note that there are interesting differences between the two pictures [4.5](#)

Figure 4.5: SIMULATION RESULTS one content and  $d_{max} = 6$ 

for  $d_{max} = 6$  and 4.6 for  $d_{max} = \infty$ . Increasing  $d_{max}$  has an influence on the average distance, especially for the centralized dynamic heuristic that does not try to minimize it but only to enforce the  $d_{max}$  constraint: comparing pictures in the two groups, we can see that the higher the  $d_{max}$ , the higher the average distance. At low  $d_{max}$  replicas are necessarily closer to the users issuing the request; in addition the request is to balance the load between replicas able to satisfy the requests. The difference is however very small.

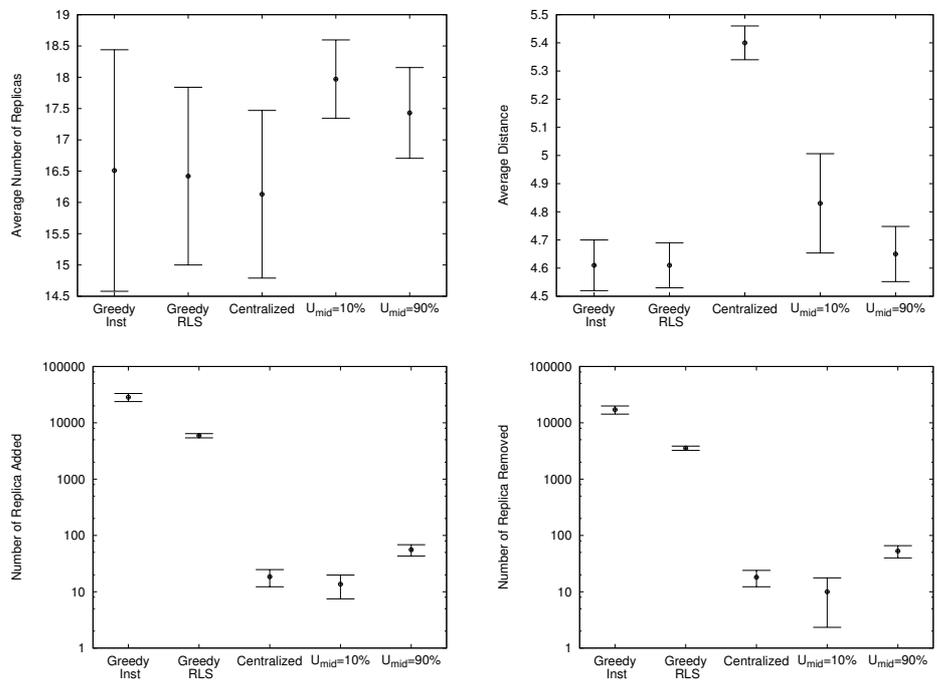


Figure 4.6: SIMULATION RESULTS one content and  $d_{max} = \infty$

### 4.2.5 Distributed Heuristics Evaluation

#### Triangular traces

In order to better understand the behavior of the distributed heuristic, we performed simulations in the simple scenario of picture 3.1, in which we gave the user traffic process a regular shape. The total rate of user requests grows linearly until a fixed amount, then is stable for a while, to start linearly decreasing. This pattern, displayed in fig. 4.7a, is repeated several times. The process is not completely deterministic, as the number of requests at any given moment is fixed, but their distribution among the access nodes is not fixed. We considered one content, with  $d_{max} = \infty$  and  $V_{MAX}^R = 1$ .  $U_{low}$  has been set to 20%, while  $U_{mid}$  varies in the range 20% – 90% of the replica capacity.

Figures 4.7a, 4.7b, 4.7c, 4.7d compare the aggregated user requests over time (“requests”), i.e.,  $\sum_{i \in V_A} x_{i,c}$  with the maximum amount of requests which can be served by the allocated replicas, i.e.,  $\sum_{j \in V_R} r_{j,c} \cdot U_{max}$  for different values of  $U_{mid}$  (“replicas capacity”). The x axis represents the time during a single simulation run. If the two curves are close to each other it means that the heuristic closely follows the traffic dynamics only allocating the minimum quantity of replicas needed to satisfy the users needs.

By examining the behavior of our heuristic when using such a simple traffic process, we wanted to put in evidence how it reacts to important load changes. In particular, it is interesting to analyze how the  $U_{mid}$  parameter influences the heuristic behavior.

In particular, figure 4.7a refers to  $U_{low} = U_{mid} = 20\%$ . In this case as the traffic increases, more and more replicas are allocated in order to fulfill user requests. After the load peak, the traffic begins to decrease, but the system is not able to delete any replica as replica loads are balanced and greater than  $U_{mid}$ .

In figure 4.7b,  $U_{mid}$  is slightly higher (50%). As the traffic decreases, some replicas experience a load below  $U_{mid}$ . Such replicas will get their load progressively decreased and they will ultimately be removed.

In figure 4.7c,  $U_{mid}$  is 75%. Our scheme in this case results in a much higher number of replica deletions, as the target utilization level that the heuristic strives to achieve is higher. It’s also worth noting that in this case the gap between the requests and replica capacity is always very small.

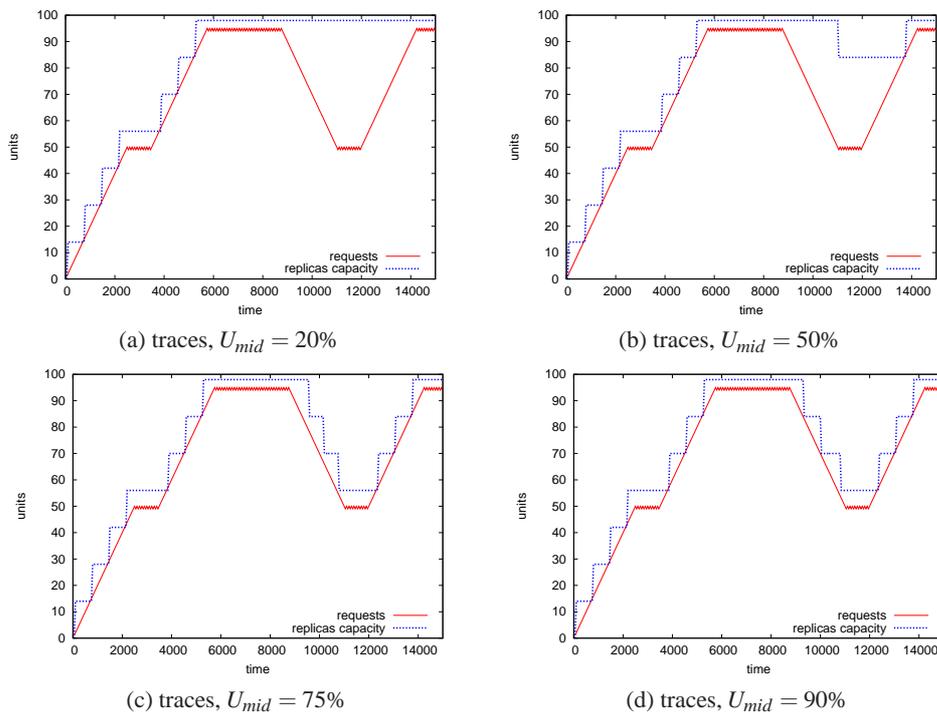


Figure 4.7: Simple topology, “triangular” traces

In the last figure (4.7d),  $U_{mid}$  is even higher (90%). The number of replicas add/removals does not increase over the  $U_{mid} = 75\%$  case. The system reacts earlier to changes in this case and the two curves (requests/replica capacity) are closer.

These observations confirm the high flexibility of our scheme. The parameter  $U_{mid}$  represents the desired replica utilization level. When the average replica load is lower than such value, the heuristic tries to reduce the number of replicas. For low values of  $U_{mid}$ , the process is never able to delete replicas. Therefore a higher  $U_{mid}$  effectively allows to reduce the number of replicas and to more quickly react to the changes in the traffic pattern.

### Dynamic behavior with Pareto processes

We now turn our attention to large scale topologies. In this set of simulations we considered a scenario based on the topology of the AT&T backbone (see Figure 4.2 and reference [75]) which comprises 184 access nodes and 115 service nodes. We have also considered the “1299” topology which comprises 153 access nodes and 31 service nodes. The requests have been modeled as the superposition of independent on-off Pareto processes.

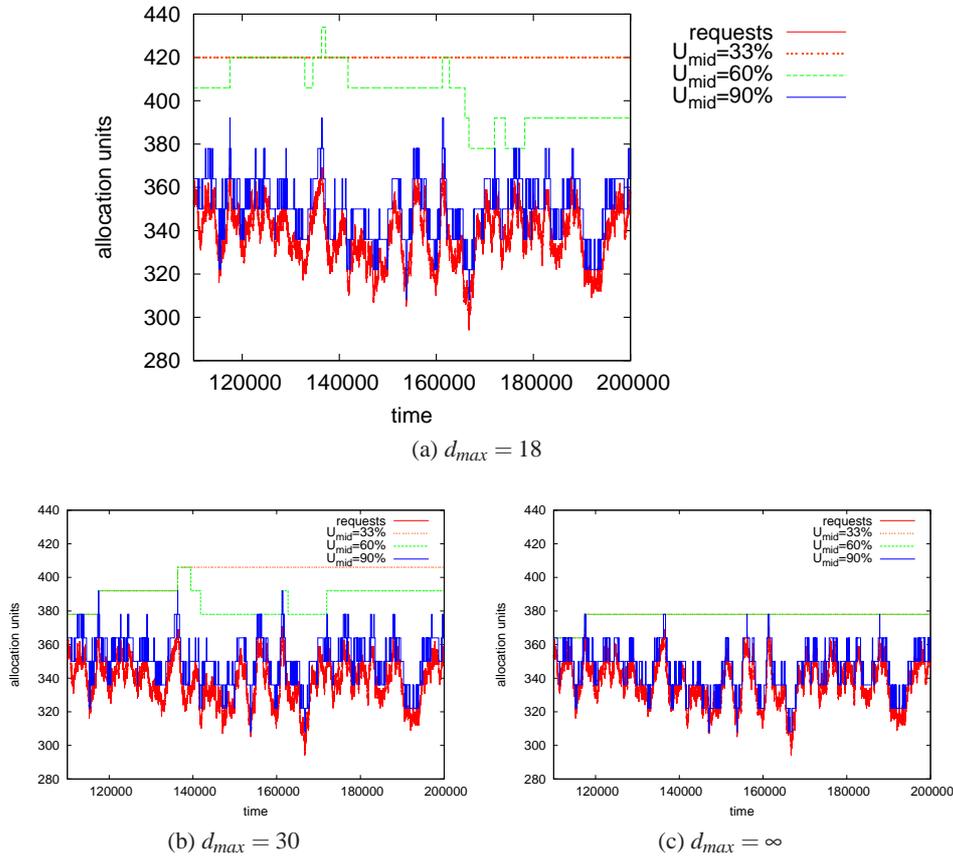
$U_{mid}$	50%	60%	90%
avg. distance	$13.01 \pm 0.163$	$12.98 \pm 0.187$	$12.78 \pm 0.093$
replica add	$0.19 \pm 0.224$	$0.54 \pm 0.234$	$3.87 \pm 0.915$
replica del	$0.15 \pm 0.162$	$0.48 \pm 0.334$	$3.86 \pm 0.956$
replicas	$54.25 \pm 8.882$	$49.10 \pm 2.349$	$45.84 \pm 1.592$

Table 4.1: AT&T,  $d_{max} = 18$

$U_{mid}$	50%	60%	90%
avg. distance	$14.19 \pm 0.353$	$14.18 \pm 0.370$	$13.26 \pm 0.150$
replica add	$0.09 \pm 0.138$	$0.09 \pm 0.138$	$3.31 \pm 0.353$
replica del	$0.00 \pm 0.000$	$0.00 \pm 0.000$	$3.31 \pm 0.496$
replicas	$47.79 \pm 2.005$	$47.79 \pm 2.005$	$45.22 \pm 1.647$

Table 4.2: AT&T,  $d_{max} = 200$

In tables 4.1 - 4.2, we summarized the relevant metrics along with the 95% confidence intervals for different values of  $U_{mid}$  and  $d_{max}$ . Values for the number of replicas adds and removals are normalized to an interval of 1000 time units. The

Figure 4.8: “1299”, varying  $d_{max}$  and  $U_{mid}$ 

behavior for the different values of  $U_{mid}$  confirms our previous observations: lower values of  $U_{mid}$  yield more stable replica configurations which in turn result into a larger number of replicas and higher user-replicas distances.

Comparing the two tables, we observe that, with smaller  $d_{max}$ , the average number of used replicas increases while the average distance decreases. This reflects the need to place more replicas to meet the stricter constraint on the maximum user-replica distance. Such replicas are necessarily placed closer to the users, resulting in lower user-request distances.

Figure 4.8 illustrates typical sample paths behavior, for topology “1299”, for

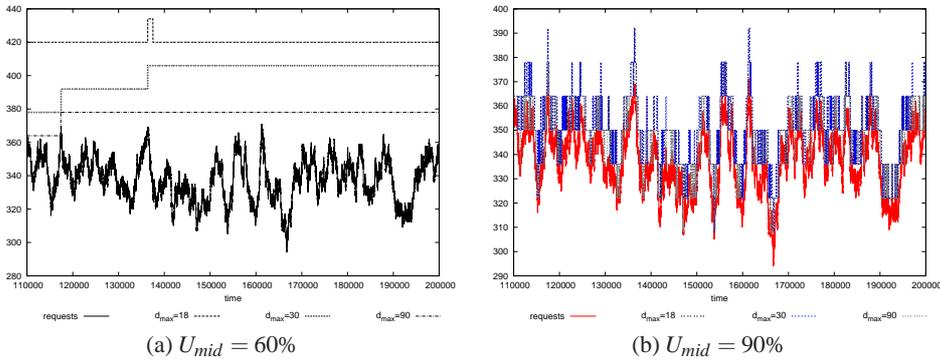
different values of  $d_{max}$  and of  $U_{mid}$ , the requests being the superposition of Pareto processes.

In the upper left corner (4.8a), for  $d_{max} = 18$ , we can see that the line for  $U_{mid} = 33\%$  is completely flat, that means that after the initial allocation no replicas are added nor removed. The curve for  $U_{mid} = 60\%$  has a different behavior: during the simulation, the overall system capacity sometimes diminishes as the average replica utilization falls below the threshold, and some other time increases: overall, the requests process is followed a bit closer. The correlation between the two curves is not very tight, as for  $d_{max} = 18$  the number of replicas is deeply influenced by the position of the requests: when many requests are concentrated in a single point, less replicas can be used. Note also that in a single short time interval, at around  $x = 140000$ s,  $U_{mid} = 60\%$  results in a slightly higher number of replicas than  $U_{mid} = 33\%$ . This is possible because of the different allocation of replicas: as higher  $U_{mid}$  values result in less replicas, these will follow the location of the requests more closely, thus it's easier to have to add a new replica in order to serve some isolated request. For  $U_{mid} = 90\%$ , the drawing shows that the capacity of the replicas follows very closely the clients demand, resulting in a much higher number of additions and removals.

In picture 4.8b the curve for  $U_{mid} = 33\%$  again shows that with this parameter the heuristic is not able to remove replicas after having allocated many. Note however that the initial allocated capacity is low, but then as the requests continue flowing it has to add more and more replicas that can not be removed. For  $U_{mid} = 60\%$ , again, although the number of replicas in the first stages is the same as for  $U_{mid} = 33\%$ , it can instead be diminished. Finally, note that the number of replicas is consistently less than for  $d_{max} = 18$  for all possible values of the middle threshold, because every service node can serve clients in a higher radius.

All these considerations are confirmed by the last picture, 4.8c. The system capacity can always be kept lower than for lower  $d_{max}$  values. The two curves for  $U_{mid} = 33\%$  and  $U_{mid} = 60\%$  are now identical, this is because there are less replicas and their average level of utilization is now always more than 33% and more than 60%, so that it's more difficult to delete them.

In picture 4.9 we see another confirmation of the already described effects of the constraint  $d_{max}$ . The data is the same as for picture 4.8, but arranged in order to compare curves for different  $d_{max}$  values. As you can see from (a), for low  $U_{mid}$  values  $d_{max}$  is very influent: the lower its value, the higher the number of replicas that

Figure 4.9: “1299”, varying  $U_{mid}$  and  $d_{max}$ 

need to be kept in the system. But as  $U_{mid}$  reaches 90% ((b)), the replica capacity and the number of requests are very near, regardless of the maximum allowed distance: this is another important proof of the effectiveness of the mechanism in real-world topologies.

#### 4.2.6 Impact of the number of contents

After having carefully studied what happens when the network hosts only one content, we decided to study the behavior when managing many contents. According to many papers ([11, 61, 68, 4]), the popularity of the contents in the Internet follows a Zipf-like distribution; other papers in the replica placement and load balancing areas make use of this assumption ([43, 15, 44]). In our simulations, when dealing with a set  $C$  of contents, we enumerated them in the range 1 to  $|C|$ , and tested this under the markovian load processes. Each content  $c$  is an independent Poisson process, all processes have the same value for the death rate  $\mu$ . The aggregated birth rate is  $\lambda$ , so the arrival rate for a single content  $c$  is:

$$\lambda^c = \beta \cdot \frac{\lambda}{c}$$

We chose the value of  $\beta$  in such a way that

$$\sum_{c \in C} \lambda^c = \lambda$$

This guarantees that fixing the value for  $\lambda$  and varying  $|C|$ , the expected number of requests is the same, although distributed over more kinds of contents.

In our experiments we have varied the number of contents  $C$  in the range 1, 5, 20. The parameter  $U_{low}$  has been set to 20% of the replica capacity, while  $U_{mid}$  is varied between 20% and 90% of the replica capacity.

**Medium load,  $d_{max} = \infty$**  The first set of results refer to a medium traffic scenario in the medium-sized “1299” topology, in which the average number of replicas needed to serve all users requests is about 30 in average, during the simulation lifetime. The value of the parameter  $d_{max}$  has been set to infinite and  $V_{MAX}^R$  has been set to 10.

In addition to the metrics already described in page 61, we also examine the ratio between the number of allocated replicas and the minimum number of replicas required to satisfy all user requests. This value has been computed as the sum, over all contents  $c$ , of the requests for content  $c$  divided by the maximum tolerable value  $U_{max}$  of requests a replica can serve, without considering the maximum distance constraint:

$$min\_replicas = \sum_{c \in C} \left\lceil \frac{\sum_{i \in V_A} x_{i,c}}{U_{max}} \right\rceil \quad (4.1)$$

From now on, we will report only the values for replica additions, because the values for replica removals are very similar for long simulations over stationary traffic processes as already verified in sections 4.2.4 and 4.2.5.

Results are depicted in Table 4.3.

We observe that the number of allocated replicas is always very close to the minimum. The gap between these two values never tops 15%, and reduces as  $U_{mid}$  increases. When  $U_{mid}$  is equal to 75% or 90% of the replica capacity, replica placement strictly follows the traffic dynamics: a replica of a given content is hosted at a service site only if it is needed to satisfy current user requests. If  $U_{mid}$  is set to a low percentage of the replica capacity, the system tends to react more slowly to traffic dynamics, and in particular tends to delete replicas only when the traffic decreases so significantly that the load of all the replicas of a given content decreases below  $U_{mid}$ . This results in an increase in the number of allocated replicas, which is particularly evident for high  $C$  values.

When the number of contents  $C$  increases, more replicas need to be allocated and the frequency of replica additions and removals increases. In the multiple contents

Number of allocated replicas/minimum required

$C$	$U_{mid} = 20\%$	$U_{mid} = 75\%$	$U_{mid} = 90\%$
1	$1.03 \pm 0.005$	$1.004 \pm 0.0022$	$1.003 \pm 0.0004$
5	$1.08 \pm 0.02$	$1.01 \pm 0.005$	$1.005 \pm 0.002$
20	$1.14 \pm 0.04$	$1.004 \pm 0.002$	$1.004 \pm 0.002$

Average replica utilization

	$U_{mid} = 20\%$	$U_{mid} = 50\%$	$U_{mid} = 90\%$
$C = 1$	$0.89 \pm 0.004$	$0.9 \pm 0.004$	$0.91 \pm 0.002$
$C = 5$	$0.78 \pm 0.013$	$0.8 \pm 0.016$	$0.84 \pm 0.007$
$C = 20$	$0.56 \pm 0.013$	$0.615 \pm 0.01$	$0.630 \pm 0.01$

Average distance to the best replica

$C$	$U_{mid} = 20\%$	$U_{mid} = 75\%$	$U_{mid} = 90\%$
1	$16.92 \pm 5.51$	$13.74 \pm 1.97$	$13.32 \pm 0.64$
5	$19.86 \pm 2.22$	$16.78 \pm 0.97$	$16.96 \pm 1.48$
20	$22.33 \pm 2.61$	$21.57 \pm 2.45$	$22.11 \pm 2.35$

Average number of replica add

$C$	$U_{mid} = 20\%$	$U_{mid} = 75\%$	$U_{mid} = 90\%$
1	$115.03 \pm 89.08$	$805.14 \pm 148.32$	$860.17 \pm 226.30$
5	$185.04 \pm 89.07$	$1220.32 \pm 465.29$	$1420.40 \pm 533.36$
20	$290.05 \pm 112.74$	$1485.32 \pm 909.57$	$1505.33 \pm 907.46$

Table 4.3: "1299", medium load,  $d_{max} = \infty$ , varying  $U_{mid}$  and  $C$

case we need to place a replica even to satisfy few requests for a low popularity content. This is confirmed also by the replica utilization results which show a significant decrease in such utilization at high  $C$ . When  $C = 20$  replicas have an average load which is 30 – 38% lower than when  $C = 1$ . More generally, the increase in the number of replicas when multiple contents are in the network reflects the fact that the traffic is heterogeneous in nature (i.e., the requests generated are for different contents) demanding for the allocation of multiple replicas, at least one for each content currently requested.

Despite our schemes closely follow the traffic dynamics both in the single content and in the multiple contents scenarios, when multiple contents are considered replicas underutilization is often unavoidable. Even if there are few requests in the network for a given content, such requests have to be satisfied, thus a replica has to be allocated to serve them. Traffic dynamics is also much faster when  $C$  is high: arrival of new user requests may result in the need to add replicas for a given content: a small decrease of user requests for a low popularity content may immediately result in the possibility to delete a replica for that content. This is clearly shown by the number of replicas additions (the number of replica removals has similar values). As the number of contents increases the frequency of changes in the replicas increases.

We finally observe that as before, the distance from the best replica tends to decrease as  $U_{mid}$  grows. At high  $U_{mid}$  we more frequently change the allocated replicas, allowing to place replica closer to the current user requests. When  $C$  increases less replicas are allocated for a given content, so that the average distance to the best replica tends to increase.

**Low load,  $d_{max} = 18$**  In the previous paragraphs we showed a simulation set in which  $d_{max} = \infty$ , and a medium load. We now move to another different scenario. In the set of pictures 4.10 we fixed  $d_{max}$  to be 18, and varied  $U_{mid}$  to be 20%, 50%, 75%, 90% of  $K$  (a distinct curve for every distinct value of  $U_{mid}$ ). The load is low ( $\simeq 5$  requests per replica).

In picture 4.10c, we can see that the number of unsatisfied requests is 0 in the  $C = 1$  case - that is, after the initial simulation warm-up phase in which we do not collect statistics, there's always at least an allocated replica in the proximity of new requests. For higher values of  $C$ , despite being very limited, the number of unsatisfied requests grows almost linearly. This is due to the fact that, when adding new contents,

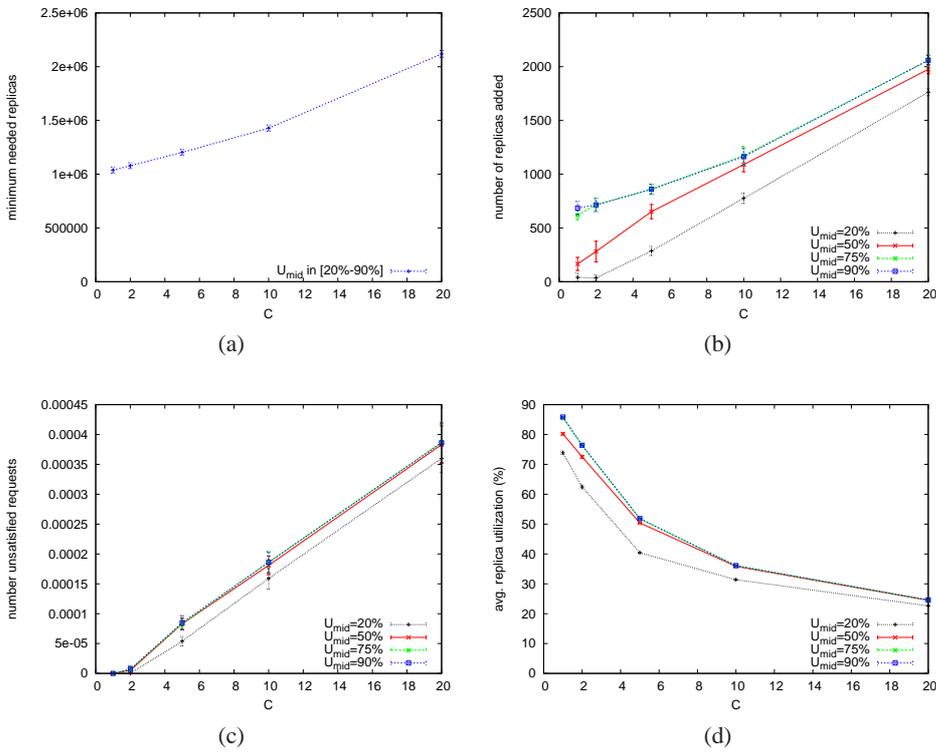


Figure 4.10: "1299", varying  $C$ , low load,  $d_{max} = 18$

the less frequent ones are less and less popular. A rarely accessed content, throughout the whole simulation, results in some interval of time with no requests in the whole system - this leads to the deletion of all replicas for that content. The next time a request will be issued an access to the origin server will be performed (unsatisfied requests) and a replica will be allocated. Furthermore, because we keep the average load constant when varying the number of contents, all contents are less popular when  $C$  is high - even the most popular content is less requested when there are 20 contents than when it's the only available one. The curve for  $U_{mid} = 20\%$  exhibits the lowest number of unserved requests (requests that have to be served by the origin server): this is because when  $U_{mid}$  is low, the system becomes very conservative and tends to keep replicas even when they serve a very small load (recall the analysis of section 4.2.5). We also measured the number of unserved requests for the most popular content (not showed), and although its traffic is lower for  $C = 20$  than for  $C = 1$ , we observed that after the initial phase all requests for it are served. The problem of unserved requests could be decreased by means of a mechanism that blocks a specific replica deletion until the exponential average of that replica load is 0 (page 55), or by adding other policies to allocate replicas before the content is requested.

From 4.10d you can see that, as the number of contents increase, the average replica utilization decreases. This is obvious as when there are more contents in the CDN, replicas are allocated for a content even if there are only a very few requests for it. This is confirmed by looking at the minimum number of replicas. This is a lower bound for the number of allocated replicas, obtained as of formula 4.1). While the load is fixed, this magnitude grows with  $C$ , so that the utilization of the replicas has to decrease. Anyway, note that the heuristic continues achieving a higher average utilization for higher values of  $U_{mid}$ .

The number of clone operations in the system is showed by diagram 4.10b. The curves show an ascending slope. For high values of  $C$  the number of replicas add/removals increases significantly. This is due partly to the higher number of unserved requests (every request served by the origin server results in a clone operation with the mechanism of algorithm 7), and partly to the already observed fact that spreading the requests over a number of contents will lead to inefficiencies.

The last thing to be considered is in fig. 4.10d. In the plot, it's evident that as  $C$  increases, each replica will serve less and less requests on average. Again, this is completely explained by the growing amount of replicas needed to serve all possible

requests.

### 4.2.7 Sensitivity to a constrained replica neighborhood

Our mechanism always guarantees every request from any given access node to be served by a replica whose distance from the access node is less than the bound  $d_{max}$ . The replicas themselves are the agents that decide whether to clone other replicas, or to remove themselves. Due to the  $d_{max}$  bound, when a replica of a given content  $c$  in a service site  $r$  decides to clone, the placement algorithm creates a new clone in a site  $r'$  that can serve as many requests as possible among the ones currently served by that replica (see alg. 6). Thus, given  $\alpha(r)$  and  $\alpha(r')$  the sets of access nodes located in the  $d_{max}$  radius by  $r$  and  $r'$  respectively, the intersection  $\alpha(r) \cap \alpha(r')$  must not be empty. This implies that every replica in  $r$  must monitor all other service sites at a distance no more than  $2 \cdot d_{max}$  from  $r$ : we called this set  $\rho(r)$ .

Due to these considerations, every service site needs to know some information about neighbor service sites: namely, topological information about each other's  $\alpha$  set, and the information about how many clones it is currently hosting for each content. If  $d_{max}$  is too large, each replica could have to collect and retain a big amount of topological information about its neighbor service nodes. Furthermore, the fact that the configuration of allocated surrogates typically changes over time implies a certain amount of message exchanges among service nodes, that is much more as  $d_{max}$  increases.

A CDN provider will possibly want to limit the impact of the exchanged messages in order to save bandwidth. So, we slightly modified our procedure in order to verify its behavior when limiting replicas knowledge. We limited each replica to know only neighbor service nodes up to  $h_{max}$  network hops away, and we ran simulations varying this parameter in order to check whether our heuristic behaves well also when this value is very low.

To this aim, we ran simulations on the "1299" topology of fig. 4.3, the traffic is the superposition of markovian processes, for a single content.  $U_{low} = 20\%$ ,  $U_{mid} = 75\%$ ,  $U_{max} = 95\%$ ,  $V_{MAX}^R = 10$ . We made simulations with two different amounts requests and markovian on-off sources, varying also the  $d_{max}$  distance threshold.

Table 4.4 refers to the first simulation set, a *low load* scenario, in which the process has been set so that the average traffic is of  $\sim 5$  requests per service node. Results are shown with the 95% confidence interval. The three sets refer to simulations with

three different values of  $d_{max}$ , namely,  $d_{max} = 18$ ,  $d_{max} = 30$ ,  $d_{max} = \infty$ : varying  $d_{max}$  is important because it is the parameter that determines the dimension of the sets  $\alpha(r)$  of neighbor nodes for each replica, that are potential candidates for hosting a new clone when a replica is overloaded. The first row of each table shows the number of hops, the columns represent the metrics for the values in the intersection.

We set  $h_{max} = 1, 2, 3, \infty$ , where  $\infty$  means that the neighborhood a replica  $r$  examines when looking for a new service site where to clone is limited, as in the original algorithm, only to the set  $\rho(r) = \{r' \in V_R \text{ s.t. } \alpha(r) \cap \alpha(r') \neq \emptyset\}$ , the effect of this being as if no  $h_{max}$  constraint was imposed.

The number of unserved requests is not reported in the tables, as it has been found to be 0 in all these simulations.

The average distance from the user to the replica only slightly changes when varying from 1 to  $\infty$ . The distance is much more influenced by the  $d_{max}$  value: for  $d_{max} = 18, 30$  it is consistently lower than for  $d_{max} = \infty$ .

Looking at the number of replicas, you can see that the  $h_{max}$  bound has no noticeable influence on it, in all the three tables.

The number of added and removed replicas (normalized over a time interval of length 100000), does not show a significant changes in its trend when varying the parameter  $h_{max}$  - it is more dependent on the value of  $d_{max}$ .

The “replica utilization” (computed as a fraction of the maximum allowed capacity of the replica  $K$ ) adds another important information to our analysis: replicas serve on average a number of requests much higher than the value  $U_{mid}$  (recall that it is 75% in these simulations), and slightly less than the maximum value  $U_{max}$ , as desired. And this behavior is absolutely not affected by the parameter  $h_{max}$ , while it is slightly influenced by  $d_{max}$ .

The second simulation set (table 4.5 ) refers to a medium load scenario, in which the average load has been set so that the average number of requests is  $\sim 15$  per service node. All other parameters are as previously specified. The three different tables show the values of the four metrics, for the three different values of the parameter  $d_{max}$ . In the first row of each table the possible values for  $h_{max}$  are shown. In this scenario there were no unserved requests.

Comparing the values in each cell of the table, row by row, you can see that the average distance, the number of replica add and the number of allocated replicas do not change much as the number of hops increases, for all examined values of  $d_{max}$ :

$d_{max} = 18$ 

<i>hops</i>	1	2	3	$\infty$
avg. distance	12.51 $\pm$ 0.06	12.50 $\pm$ 0.06	12.51 $\pm$ 0.05	12.49 $\pm$ 0.07
replica add	704.20 $\pm$ 61.37	709.57 $\pm$ 60.29	701.64 $\pm$ 80.41	711.79 $\pm$ 86.61
replicas	14.34 $\pm$ 0.23	14.34 $\pm$ 0.24	14.34 $\pm$ 0.23	14.34 $\pm$ 0.24
utilization %	0.868 $\pm$ 0.002	0.868 $\pm$ 0.002	0.868 $\pm$ 0.003	0.869 $\pm$ 0.002

 $d_{max} = 30$ 

<i>hops</i>	1	2	3	$\infty$
avg. distance	12.53 $\pm$ 0.06	12.53 $\pm$ 0.10	12.54 $\pm$ 0.08	12.52 $\pm$ 0.05
replica add	716.86 $\pm$ 37.88	719.93 $\pm$ 79.63	719.50 $\pm$ 66.93	721.22 $\pm$ 56.36
replicas	14.29 $\pm$ 0.24	14.30 $\pm$ 0.23	14.30 $\pm$ 0.25	14.29 $\pm$ 0.23
utilization %	0.871 $\pm$ 0.002	0.871 $\pm$ 0.003	0.871 $\pm$ 0.002	0.871 $\pm$ 0.002

 $d_{max} = \infty$ 

<i>hops</i>	1	2	3	$\infty$
avg. distance	13.85 $\pm$ 0.95	14.10 $\pm$ 1.06	13.87 $\pm$ 0.84	14.09 $\pm$ 1.37
replica add	356.35 $\pm$ 97.38	372.07 $\pm$ 50.67	356.00 $\pm$ 99.53	358.07 $\pm$ 66.45
replicas	14.04 $\pm$ 0.20	14.02 $\pm$ 0.23	14.04 $\pm$ 0.24	14.03 $\pm$ 0.23
utilization %	0.887 $\pm$ 0.006	0.888 $\pm$ 0.004	0.887 $\pm$ 0.005	0.888 $\pm$ 0.003

Table 4.4: 1299, low load, limited neighborhood

in particular, note that  $h_{max} = \infty$  does not provide any particular advantage in this simulations. Notice that in this table the average utilization of allocated replicas (“utilization %”) is even higher than for table 4.4: it is now more than 91% for  $d_{max} = \infty$ .

Summing up, in the above analysis we examined the behavior of our distributed, dynamic heuristic when constraining the information known at each replica site. The allocation mechanism is indeed critical for a dynamic replication scheme, as adding clones in the wrong place would eventually result in a waste of resources or in poor performance. The removal mechanism could react to limit this waste but this would lead to a higher number of changes to the system.

Simulation results show that our heuristic is hardly affected by the value of the new constraint  $h_{max}$ . This would allow a content provider to limit the burden imposed on the network by the monitoring activities needed by our distributed replica placement heuristic, without impairing the CDN performance.

$d_{max} = 18$				
<i>hops</i>	1	2	3	$\infty$
avg. distance	12.23 $\pm$ 0.11	12.17 $\pm$ 0.09	12.20 $\pm$ 0.09	12.19 $\pm$ 0.05
replica add	1192.50 $\pm$ 225.03	1210.61 $\pm$ 149.80	1216.45 $\pm$ 175.87	1170.23 $\pm$ 177.96
replicas	34.09 $\pm$ 0.49	33.99 $\pm$ 0.49	34.02 $\pm$ 0.58	34.03 $\pm$ 0.55
utilization %	0.905 $\pm$ 0.001	0.906 $\pm$ 0.001	0.906 $\pm$ 0.002	0.906 $\pm$ 0.001

$d_{max} = 30$				
<i>hops</i>	1	2	3	$\infty$
avg. distance	12.21 $\pm$ 0.13	12.18 $\pm$ 0.15	12.22 $\pm$ 0.15	12.22 $\pm$ 0.07
replica add	1186.23 $\pm$ 143.01	1246.68 $\pm$ 130.49	1183.56 $\pm$ 189.82	1191.12 $\pm$ 216.84
replicas	34.03 $\pm$ 0.57	34.02 $\pm$ 0.53	34.03 $\pm$ 0.53	34.04 $\pm$ 0.56
utilization %	0.906 $\pm$ 0.002	0.906 $\pm$ 0.001	0.906 $\pm$ 0.001	0.906 $\pm$ 0.001

$d_{max} = \infty$				
<i>hops</i>	1	2	3	$\infty$
avg. distance	12.91 $\pm$ 0.77	13.06 $\pm$ 0.86	13.10 $\pm$ 0.93	12.86 $\pm$ 0.40
replica add	695.11 $\pm$ 352.22	635.11 $\pm$ 264.99	700.00 $\pm$ 200.84	736.45 $\pm$ 222.34
replicas	33.78 $\pm$ 0.47	33.83 $\pm$ 0.45	33.78 $\pm$ 0.48	33.76 $\pm$ 0.53
utilization %	0.913 $\pm$ 0.003	0.911 $\pm$ 0.003	0.913 $\pm$ 0.002	0.913 $\pm$ 0.002

Table 4.5: 1299, medium load, limited neighborhood

## Chapter 5

# Request redirection schemes

### 5.1 Distributed load balancing

In the previous chapter we have assumed that the system is capable of redirecting the requests obtaining a best effort balancing of the load among replicas. (A perfect load balancing may be impossible due to the distance constraint.)

Here we show how redirection can be implemented in practice by a distributed algorithm that achieves both load balancing and high resource utilization. For each access node  $i \in V_A$  and content  $c \in C$ ,  $\alpha_{ij,c}$  denotes the fraction of requests  $x_{i,c}$  originated at  $i$  for content  $c$  which are redirected to node  $j \in V_R$ . For a site  $j \in V_R$ , let  $l_{j,c} = \sum_{i \in \alpha(j)} \alpha_{ij,c} x_{i,c}$  denote the aggregate demand of content  $c$  served by  $j$ . Let  $r_{j,c}$  be the number of replicas for content  $c$  hosted at  $j$  and let  $u_{j,c} = l_{j,c}/r_{j,c}$  be the utilization of content  $c$  replicas at site  $j$ .

We start by providing a formal definition of load balanced configuration. We define a *load configuration* as the set  $l_{j,c}$  with  $j \in V_R$ . A load configuration is said to be *balanced* if for each  $i \in V_A$ ,  $u_{j,c} = u_{j',c}$  ( $j, j' \in V_R$ ). When  $d_{max} = \infty$ , i.e., when each access node  $i \in V_A$  can be served by any node  $j \in V_R$ , it is always possible to achieve a load balanced configuration by properly adjusting the redirection vectors. Instead, when  $d_{max} < \infty$  load balancing among different replicas can be achieved only to the extent allowed by the distance constraint. In extreme cases when  $d_{max}$  is strongly constrained unbalanced loads are unavoidable. For instance, when each access node can be served only by one server site, each node  $j \in V_R$  would have a load depending on the number of user requests from users in its neighborhood, and there is no possibility of balancing the requests load. For these cases, we introduce the more

general notion of *feasible load balanced configuration*. With  $A_{i,c}$  we denote the set of replicas to which access node  $i$  is sending traffic, i.e.,  $A_{i,c} = \{j \in \rho(i) \mid \alpha_{ij,c} > 0\}$ . A *feasible load balanced configuration* is a load configuration  $l_{j,c}$ ,  $j \in V_R$ , such that for each  $i \in V_A$ ,  $u_{j,c} = u_{j',c}$  ( $j, j' \in A_{i,c}$ ) and  $u_{j,c} \leq u_{j',c}$  ( $j \in A_{i,c}, j' \in \rho(i) \setminus A_{i,c}$ ). In other words, a load configuration is feasible and balanced if for each access node  $i$  the nodes to which  $i$  redirects its traffic (replicas in  $A_{i,c}$ ) have the same utilization, while the nodes to which  $i$  does not send requests (replicas in  $\rho(i) \setminus A_{i,c}$ ) have higher or equal utilization than those in  $A_{i,c}$ . Observe that a *load balanced configuration* is a special case of a *feasible load balanced configuration* where all utilizations are equal.

A distributed mechanism is now needed to achieve feasible load balanced configurations. This *redirection update algorithm* (RUA) is what we present next.

Each access node  $i$  periodically updates its redirection vector  $\alpha_{i,c} = (\alpha_{ij,c})_{j \in \rho(i)}$  using solely the load information  $l_{j,c}$  of the replicas to which it is redirecting traffic and information about the number of replicas at a given site  $r_{j,c}$ . This information is made available to the access nodes by piggybacking it to the messages answering user requests. The access nodes update their redirection vector at different times, independently and asynchronously from each other. For ease of presentation here we consider periodic updates of period  $T_i$  and we denote with  $\alpha_{ij,c}(k_i + 1)$  the  $(k_i + 1)$ th update of node  $i$ 's redirection vector. In order to achieve load balancing among replicas of the same content  $c \in C$ , each access node  $i$  uses the following updating rule for each  $j \in \rho(i)$ .

$$\alpha_{ij,c}(k_i + 1) = \alpha_{ij,c}(k_i) - \delta(u_{j,c} - U_{i,c})x_{i,c}, \quad (5.1)$$

where  $\delta > 0$  and  $U_{i,c} = \frac{1}{|\rho(i)|} \sum_{j \in \rho(i)} u_{j,c}$  is the average utilization of content  $c$  replicas in sites  $j \in \rho(i)$ . The idea behind rule (5.1) is quite simple: In order to balance the requests among different replicas, each access node  $i$  periodically re-adjusts its redirection scheme by diverting traffic from over-utilized replicas (i.e., from replicas  $j$  for which  $u_{j,c} > U_{i,c}$ ) and redirecting it to those replicas which are underutilized ( $u_{j,c} < U_{i,c}$ ). Intuitively, assuming a constant volume of requests, all access nodes redirection vectors reach an equilibrium point characterized by perfect load balancing among the different replicas, i.e.,  $u_{j,c} = U_{i,c}$ ,  $j \in \rho(i)$ . In this case, from rule (5.1) we have  $\alpha_{ij,c}(k_i + 1) = \alpha_{ij,c}(k_i)$ ,  $i \in V_A$ .

Notice that rule (5.1) does not guarantee that  $\alpha_{ij,c}(k) \geq 0$ . To ensure non-negativity rule (5.1) needs to be modified as follows.

$$\alpha_{ij,c}(k_i+1) = \begin{cases} \alpha_{ij,c}(k_i) - \bar{\delta}(u_{j,c} - U'_{i,c})x_{i,c} & j \notin N_{i,c} \\ 0 & j \in N_{i,c} \end{cases} \quad (5.2)$$

where  $N_{i,c}$  is the set of replicas to which node  $i$  does not redirect requests and whose utilization exceeds the average utilization  $U_{i,c}$ , i.e.,  $N_{i,c} = \{j \in \rho(i) \mid \alpha_{ij,c}(k_i) = 0 \wedge u_{j,c} > U_{i,c}\}$ . Rule (5.1) cannot be applied to replica in  $N_{i,c}$  as  $\alpha_{ij,c}(k_i+1)$ ,  $j \in N_{i,c}$  would become negative. For such replicas we keep  $\alpha_{ij,c}(k_i+1) = 0$ . For replicas not in  $N_{i,c}$ , in rule (5.2) we apply rule (5.1) by replacing  $U_{i,c}$  by  $U'_{i,c} = \frac{1}{|\rho(i) \setminus N_{i,c}|} \sum_{j \in \rho(i) \setminus N_{i,c}} u_{j,c}$ , i.e.,  $U'_{i,c}$  denotes the average utilization of content  $c$  replicas in  $\rho(i) \setminus N_{i,c}$ . This is required to ensure that  $\alpha_{ij,c}(k_i+1)$ ,  $j \in \rho(i)$ , sum up to 1. We also need to enforce  $\alpha_{ij,c}(k_i+1) > 0$ . This is obtained by choosing  $\bar{\delta}$  as follows:

$$\bar{\delta} = \min \left\{ \delta, \min_{\{j \in V_R \mid \alpha_{ij,c}(k_i) > 0 \wedge (u_{j,c} - U_{i,c}) > 0\}} \frac{\alpha_{ij,c}(k_i)}{(u_{j,c} - U_{i,c})x_{i,c}} \right\} \quad (5.3)$$

We prove that RUA generates a sequence of redirection vector updates that converges to a load balancing equilibrium point.

**Theorem 1** *Given  $x_{i,c} \geq 0$ , and  $0 < T_i < \infty$ ,  $i \in V_A$ , and any initial redirection vector  $\alpha_i(\cdot) = (\alpha_{i,c})$ , the sequence of RUA redirection vectors  $\alpha_i(\cdot)$  converges to a feasible load balanced configuration.*

The proof is based on showing that the distributed execution of the redirection update algorithm iteratively solves the following optimization problem

$$\begin{aligned} \mathbf{LB}: \quad \min F_c(\alpha) &= \sum_{j \in V_R} l_{j,c}^2 / r_{j,c} \\ l_{j,c} &= \sum_{i \in \alpha(j)} \alpha_{ij,c} x_{i,c} \\ \sum_{j \in \rho(i)} \alpha_{ij,c} &= 1, \alpha_{ij,c} \geq 0, \quad i \in V_A \end{aligned}$$

The **LB** problem is a convex problem with linear constraints. These problems are generally solved by means of iterative gradient methods modified to account for the

presence of the constraints. The proposed redirection update algorithm, in particular, can be regarded as a gradient projection algorithm (it is not difficult to verify that it is similar to the gradient projection method of Rosen).

We first show that for any  $i \in V_A$ , if the  $\alpha_{ij,c}(k_i)$  are feasible, then the  $\alpha_{ij,c}(k_i + 1)$  are feasible as well. Indeed, (5.2) and the choice for  $\bar{\delta}$  ensures that  $\alpha_{ij,c}(k_i + 1) \geq 0$ ; moreover,

$$\begin{aligned} \sum_{j \in \rho(i)} \alpha_{ij,c}(k_i + 1) &= \sum_{j \in \rho(i)} \alpha_{ij,c}(k_i) + \delta x_{i,c} \sum_{j \in A_{i,c}} (u_{jc} - U'_{i,c}) \\ &= 1 - \delta x_{i,c} \left( \sum_{j \in A_{i,c}} u_{jc} - |A_{i,c}| U'_{i,c} \right) \\ &= 1 \end{aligned}$$

Then we show that for any  $i \in V_A$ , any step of the redirection algorithm improves the objective function, i.e.  $F_c(\alpha(k_i + 1)) \leq F_c(\alpha(k_i))$ . To this end, it suffices to show that  $\nabla F_c(\alpha(k_i))' d \leq 0$ , where  $\nabla F_c(\alpha(k_i))$  is the gradient of objective function  $F_c(\alpha)$  and  $d = (0, \dots, 0, u_{j_1,c} - U_{i,c}, \dots, u_{j_n,c} - U_{i,c}, \dots)'$  is the direction associated with a single step of the redirection algorithm,  $d$  is a direction along which the function  $F_c$  decreases.

**Theorem 2** *The following holds:*

1.  $\nabla F_c(\alpha)' d \leq 0$ ;
2.  $\nabla F_c(\alpha)' d = 0$  if and only if  $d = 0$ , i.e.,  $u_{jc} = u_{j',c}$ ,  $j, j' \in A'_{i,c}$ .

*Proof:* The proof is based on the following result:

**Lemma 1**

$$\sqrt{|A'_{i,c}| \sum_{j \in A'_{i,c}} (u_{j,c})^2} \geq \sum_{j \in A'_{i,c}} u_{j,c}. \quad (5.4)$$

Moreover, the equality holds if and only if  $u_{jc} = u_{j',c}$ ,  $j, j' \in A'_{i,c}$ .

*Proof:* It is easy to verify that  $\nabla F_c(\alpha) = (u_{j_1}, \dots, u_{j_n}, \dots, u_{j_1}, \dots, u_{j_n})'$ . Then: 1) directly follows from the Cauchy-Schwartz inequality applied to the vectors  $(u_{j,c}, \dots, u_{j,c})'$  and  $(1, \dots, 1)'$ ; for 2), observe also that the equality in the Cauchy-Schwartz inequality

requires the two vectors to be parallel. This requires all the element of the first vector to be all equal. ■

The theorem directly follows from the Lemma by observing that:

$$\nabla F_c(\alpha)'d = \sum_{j \in A'_{i,c}} u_{j,c} (u_{j,c} - U'_{i,c}) \quad (5.5)$$

$$= \sum_{j \in A'_{i,c}} u_{j,c} - \frac{1}{|A'_{i,c}|} \left( \sum_{j \in A'_{i,c}} u_{j,c} \right)^2 \quad (5.6)$$

■

Since the set of feasible points is compact and the sequence  $F_c(\alpha(\cdot))$  is non-increasing,  $\alpha(\cdot)$  and  $F_c(\alpha(\cdot))$  converge. Convergence of  $\alpha(\cdot)$  implies that all sequences  $\alpha(k_i)$ ,  $i \in V_A$  converge (convergence of a series implies the convergence of all its extracted sub-series). 2) of Theorem 2 then implies that  $\alpha(\cdot)$  converges to a feasible load balanced point.

■

The redirection scheme just presented achieves load balancing among the different servers. As previously noted, this does not necessarily imply efficient resource utilization since many servers may operate at low utilization levels. Clearly, in such a scenario, it is preferable to remove underutilized replicas without affecting the level of users satisfaction. To this end, we introduce a tuning parameter  $U_{mid}$  which represents a lower bound on the replica target utilization. Whenever the load falls below  $U_{mid}$  replicas take actions to direct requests away (which eventually results into their removal). This can be accomplished by replacing the actual utilization  $u_{j,c} = l_{j,c}/r_{j,c}$  in (5.2) with an *inflated* utilization  $u'_{j,c} \geq u_{j,c}$  as follows:

$$u'_{j,c} = \begin{cases} U_{max} - \varepsilon & u_{j,c} < U_{low} \\ U_{max} - 2\varepsilon & \text{w.p. } \frac{U_{mid} - u_{j,c}}{U_{mid} - U_{low}} \quad U_{low} \leq u_{j,c} \leq U_{mid} \\ u_{j,c} & \text{w.p. } \frac{u_{j,c} - U_{low}}{U_{mid} - U_{low}} \quad U_{low} \leq u_{j,c} \leq U_{mid} \\ u_{j,c} & u_{j,c} > U_{mid}. \end{cases} \quad (5.7)$$

The RUA algorithm is therefore applied where each replica  $c$  advertises  $u'_{j,c}$  rather

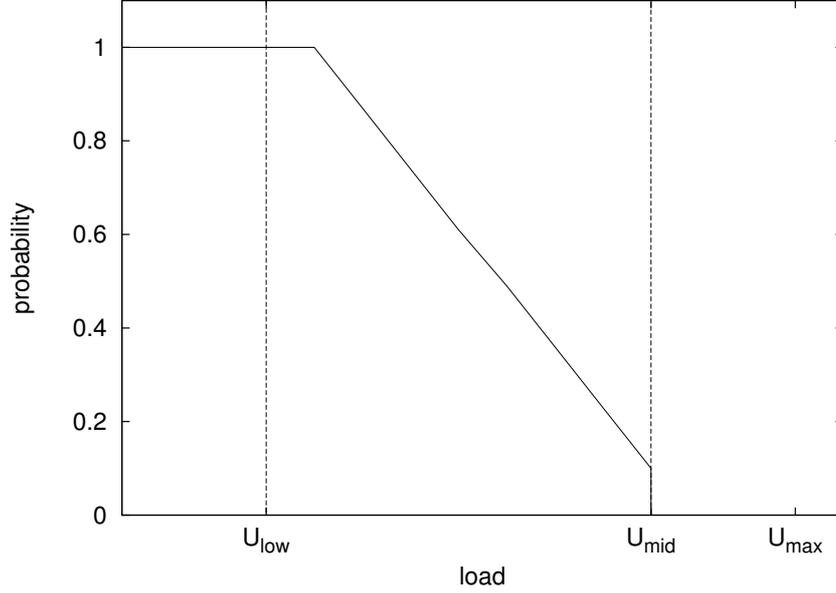


Figure 5.1: Probability of load inflation

than  $u'_{j,c}$ , i.e., the inflated utilization level rather than the actual one. Since the redirection strategy balances the load, replicas that advertise inflated utilization, i.e., those whose actual utilization is  $< U_{mid}$ , tend to be less and less utilized. In order to randomize the behavior of the different replicas so that not all those underutilized attempts to be removed, replicas with actual load in the range  $U_{low} < u_{j,c} \leq U_{mid}$  advertise the inflated value  $u'_{j,c}$  with a given probability that depends on  $u_{j,c}$ : The lower the actual load  $u_{j,c}$  the higher the probability that  $c$  attempts to be removed (see Figure 5.1).

As soon as a replica utilization level falls below  $U_{low}$  its advertised level becomes  $U_{max} - \varepsilon$  (with probability 1), that is, it becomes higher than the level of other replicas whose levels are between  $U_{low}$  and  $U_{mid}$ . This ensures that that replica will get less and less requests with respect to the others, and it will be removed before them, if possible. Note that the very definition of  $u'_{j,c}$  ensures that a replica advertising inflated utilization level, will never cause the cloning of another replica for overutilization. Since the underutilized replicas advertise a level which is  $U_{max} - a\varepsilon$ ,  $a = 1, 2$ , because of the RUA balancing, they will get a higher share of the load than replicas whose load is  $U_{max}$ . When traffic increases the RRS will proceed in its iteration, distributing more

extra traffic to underutilized replicas and replicas with load  $< U_{max}$ , converging to a situation where extra traffic is redistributed only to replicas with local load  $< U_{max}$  if possible. Cloning will therefore be performed only if needed.

## 5.2 Validating the distributed algorithm

In this section we summarize the results of a set of experiments we have performed to assess the effectiveness of the proposed distributed request redirection scheme.

In the following pictures we show the load distribution of all replicas in the system, observed over three simulation runs in which we varied the traffic. All pictures refer to the topology with 7 service nodes, 24 access nodes of figure 3.1. Simulations run for 3000 seconds each and the statistics are collected over the whole period.

In these simulations we computed the redirection decisions in an “instantaneous” fashion, in order to better analyze the distributed algorithm behavior. More precisely, as soon as the traffic changes, the simulation is frozen and all access nodes starts iterating the load balancing algorithm. The simulation is resumed after the value of the target function of the algorithm converges, this in order to check that the algorithm is able to converge in a limited number of steps, although in a real system the procedure would have to be executed periodically.

The user request process has been artificially set to be monotonically increasing and decreasing,  $d_{max} = \infty$ , we set  $V_R^{MAX} = 1$ ,  $U_{low} = 20\%$ ,  $U_{mid} = 80\%$  and  $U_{max} = 90\%$ . We want to check whether our joint load balancing and replica placement method has the desired properties of load balancing the traffic and keeping the replica load within a predefined range. To this purpose we have implemented in our simulator both the centralized load balancing and the distributed request redirection presented in the previous section.

The first set of pictures shows the outcome of the load balancing, when the traffic varies as in picture 5.2. Pictures 5.3a, 5.3b refer to two simulations employing the centralized matching mechanism and the distributed one respectively, in this scenario. We monitored all replica loads throughout the whole simulation, collected the amount of times a replica has load  $l$ , and normalized such value to the simulation timeframe. Therefore, this metric expresses the fraction of time a replica has a given load.

The x axis represents the load as a percentage of the replica maximum capacity, the y axis the probability that a replica has load  $x\%$  during the simulation. We plot three vertical lines corresponding to the values of  $U_{low} = 20\%$ ,  $U_{mid} = 80\%$  and  $U_{max} = 90\%$ . A logarithmic scale has been used to display the results.

We can see from picture 5.3a, that the majority of the times replicas load is greater than  $l > U_{mid}$ , and seldom it is  $l > U_{max}$  (in all these cases there was the need to add a replica). For the large majority of time the replicas load is between  $U_{mid}$  and  $U_{max}$

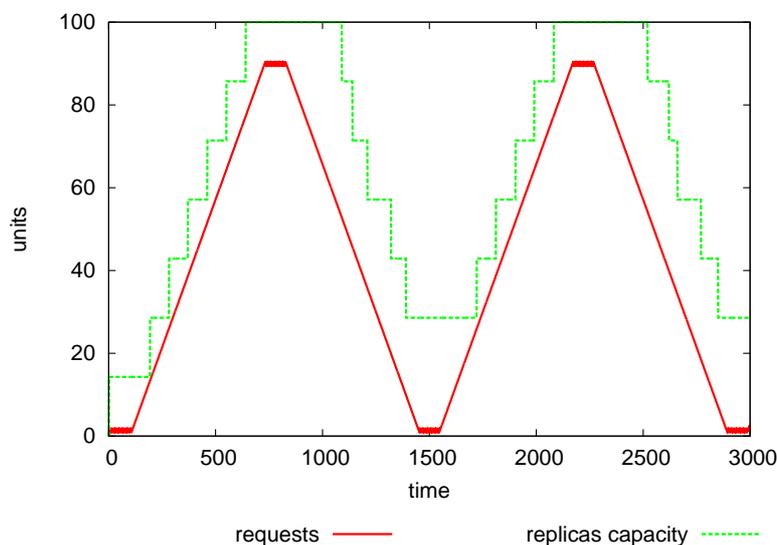


Figure 5.2: Simulation trace, first set

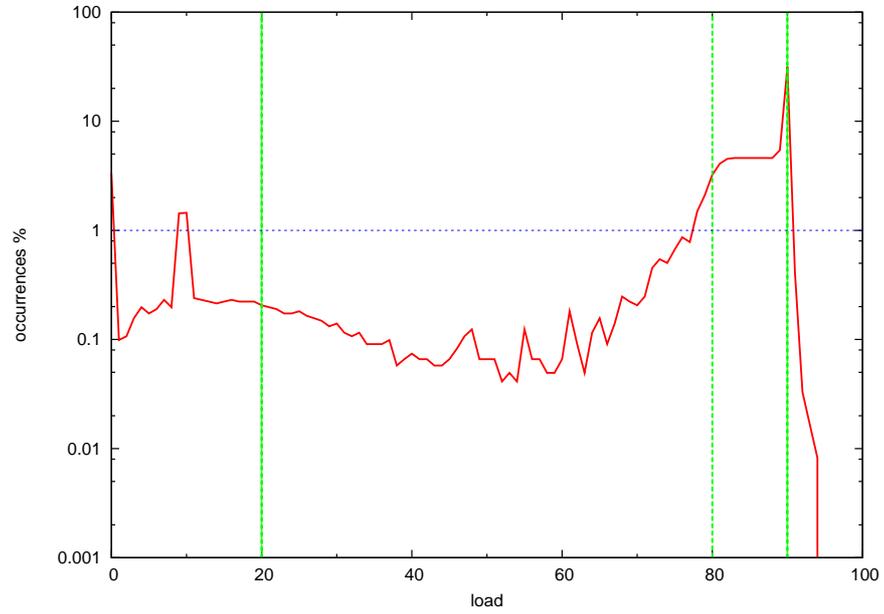
which is the specified desirable range of utilization.

Note that there is a positive probability that a replica has  $l > U_{max}$ , because this corresponds to the times when a replica is overloaded. This event is very rare and its consequence is that the replica decides to clone, thus its load is lowered in a very short time.

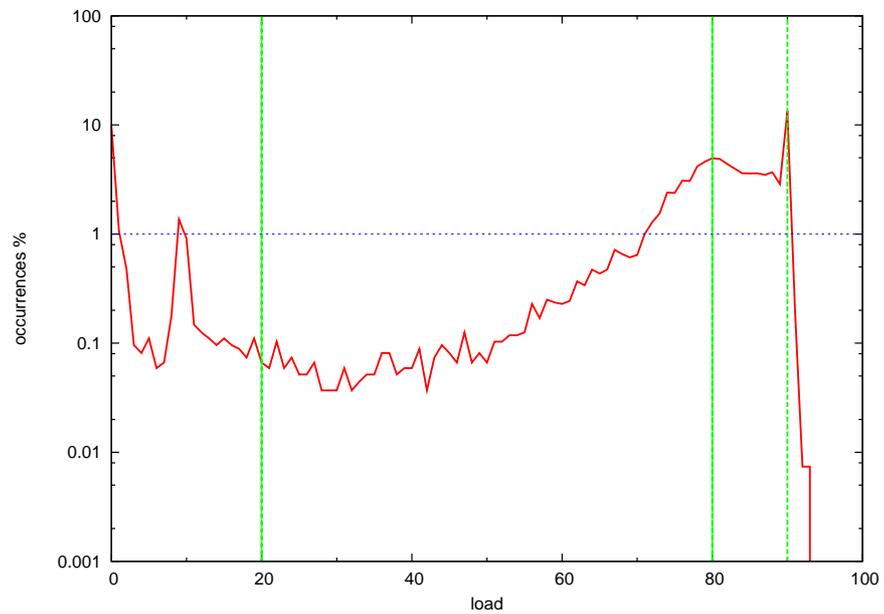
Looking at the central region of the plot, the one for which  $U_{low} < l \leq U_{mid}$ , we can see that it is more common to have loads close to  $U_{mid}$ . This is because of the probabilistic unload mechanism: in this load range, the less a replica is used, the more likely it is that it will inflate its load. This makes replica unloading faster and faster as the replica load decreases, which is confirmed by a lower fraction of time spent in the lowest values of the  $[U_{low}; U_{mid}]$  interval.

Examining the part of the graph  $0 \leq l \leq U_{low}$ , it is possible to note that there is a peak corresponding to a load  $l$  in the range that corresponds to loads  $9\% \leq l \leq 10\%$ . This is due to the particular shape of the request process (see picture 5.2), and corresponds to the portions of the simulation in which the load is very low and the allocated replicas have to serve a very small number of requests, resulting in this peak.

The last interesting thing that should be noted is that there is a relevant probability that a replica serves no requests. This is due to the underloading mechanism: every



(a) Centralized Request Redirection



(b) Distributed Request Redirection

Figure 5.3: First set, loads

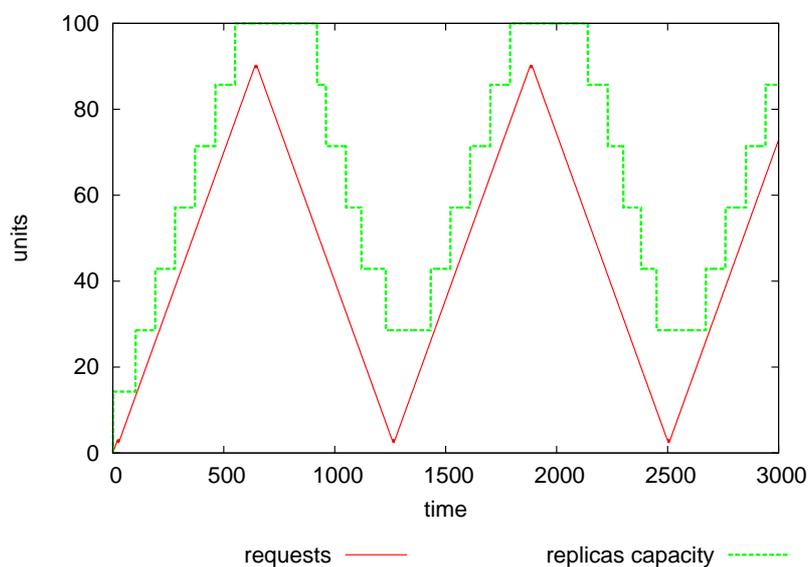


Figure 5.4: Simulation trace, second set

replica strives to have its load reduced, and if this is possible, the redirection system will assign the replica no requests to serve, so that it can later be removed from the network.

Picture 5.3b illustrates replicas load for the same traffic pattern, but using the distributed load balancing as the redirection mechanism. The behavior closely mimics what obtained using the centralized mechanism. Also in this case, replicas are almost always in the desired load range with  $l$  s.t.  $U_{mid} \leq l \leq U_{max}$ . Underutilized replicas are very rare (replicas rapidly unload themselves if possible), there is a high number of zeros and also a peak at about  $l \simeq 10\%$ , which has the same motivation discussed above.

The distributed load balancing algorithm in this simulation converges to a solution within  $1/10^5$  from the optimal value of the function, in 24.12 iterations in average, with a standard deviation of 17.45.

We have therefore shown that the distributed scheme is fast, and able to control the replica load to the desired utilization level, a significant non trivial result.

Figure 5.4 shows a second, slightly different, process. For this process we obtained the distribution of the loads as of pictures 5.5a and 5.5b. The distributed load balancing algorithm in this simulation converges to a solution within  $1/10^5$  from the

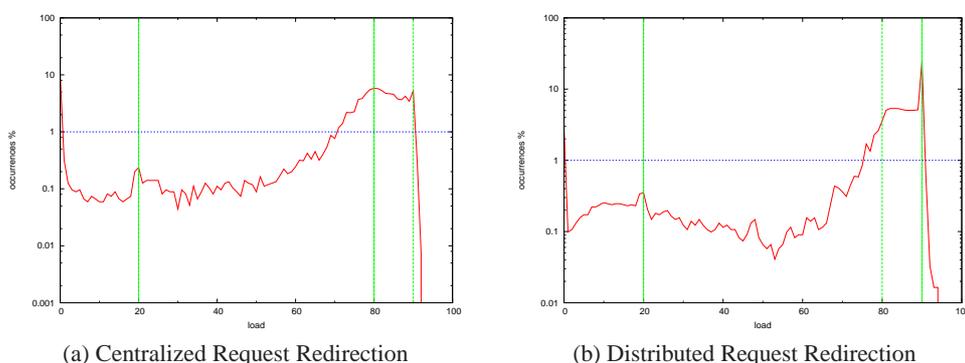


Figure 5.5: Second set, loads

optimal solution, in 25.20 iterations in average, with a standard deviation of 15.64.

For these pictures it is very important to remark the differences with the first set of pictures 5.3a, 5.3b. The peak in the number of occurrences that occurred for  $l \simeq 10\%$  is no more present, as we said that was dependent on that particular process. The distribution of the load still retains the desired properties: replicas are in the desired load region most of the time, very low probabilities of being underloaded and quite high probabilities of having no requests to serve (0 load).

Results for a third set are shown in pictures 5.6 (simulation trace), 5.7a (behavior with centralized load balancing), 5.7b (distributed load balancing). These results have been obtained from a traffic pattern identical to the second traffic pattern (5.4) in the number of request issued over the time. The difference lies in the fact that the number of requests is a function of the time, but the way requests are spread among the access nodes is not fixed: they are randomly distributed among the sources, so the two traffic patterns have randomly chosen, different sets of requests issued by each particular access site at every time. In this simulation, the number of steps needed for the distributed load balancing algorithm to converge to a solution within  $1/10^5$  from the optimal solution, is 25.50 iterations in average, with a standard deviation of 16.98.

Results for this third traffic pattern exhibit a similar behavior than for the previous ones, but are shown here in order to further confirm the trend shown in the previous pictures.

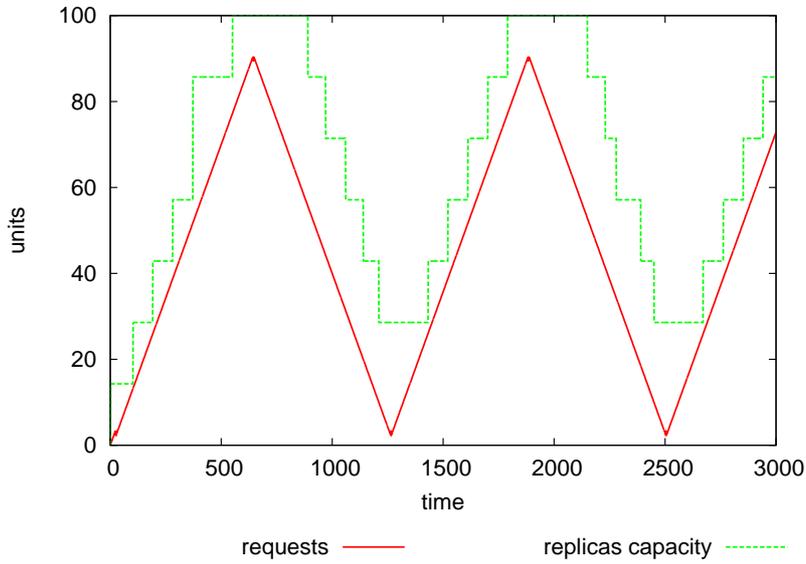


Figure 5.6: Simulation trace, third set

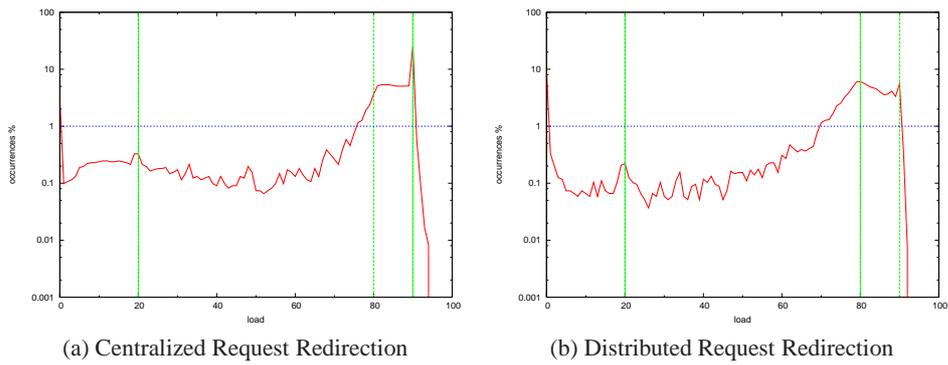


Figure 5.7: Third set, loads



## Chapter 6

# Conclusions

In this thesis we have first analyzed the main problems that must be addressed in order to deploy a Content Delivery Network. We have then reviewed the classic formulations for the problem of minimizing the number of replicas in the system (the “replica placement problem”) and for the problem of transparently redirecting requests to the replicas (request redirection scheme). We have observed that the majority of solutions so far proposed for replica placement assume static traffic. Also, the two problems (replica placement and request redirection) are usually treated in isolation.

Our contribution improves over previous results in that it considers the two problems jointly, thus providing a new overall solution that effectively trades-off among the number of replicas, the distance from the best replica, the number of replica additions and removals. We minimize the costs for replicas placement and maintenance, we try to keep as low as possible the number of replicas adds and removals while satisfying all user requests and we do it distributively, load balancing the traffic among replicas and cloning (removing) replicas whenever their level of utilization is above (below) a desirable level of utilization.

In particular, by properly setting the  $U_{mid}$  parameter we can achieve different trade-offs between the number of allocated replicas, the user request - serving replica distance and the frequency of replica additions and removals.

**Optimal formulation, dynamic heuristic** Building on this new formulation, we have provided a framework for the design of replica allocation schemes dynamically placing and removing replicas in response to changing users demand. By assuming

the users requests dynamics to obey to a Markovian model we saw how to formulate the dynamic replica placement problem as a Markovian decision process. This allowed to identify an optimal policy for dynamic replica placement that can be used as a benchmark for heuristics evaluation and provides insights on how allocation and deallocation should be proactively performed. Based on the findings obtained through the analytical model we derived a centralized heuristic which allocates and deallocates replicas to reflect the requests traffic dynamics, the costs of adding, deleting and maintaining replicas, the servers load and storage limits, and the requirements on the maximum distance of the users from the “best replica”.

**Characteristics of our replica placement distributed heuristic** We have then proposed a threshold-based distributed heuristic which dynamically adds or deletes replicas from the network depending on the load of the replicas. This heuristic relies on the behavior of the Request Redirection scheme (RRS). The RRS must be able to perform load balancing, must avoid overloading replicas unless needed, and must implement a way for replicas to provide a feedback to the access sites in order to have their load reduced (if possible without overloading other replicas). The performance of the proposed scheme has been compared with that of (static) greedy schemes which have been proven to perform well in the literature, and with the centralized dynamic heuristic.

Extensive OPNET simulations have allowed us to prove the effectiveness of our distributed heuristic. In particular, the simulations we have performed have shown that:

- The proposed solution improves over static replica placement. To achieve comparable performance in terms of number of allocated replicas, distance between the user and the serving replicas, static schemes have to be reexecuted frequently, resulting in a number of replica adds and removals that can be three orders of magnitude higher than in our scheme.
- The heuristic performs well in terms of number of replicas, replicas utilization, distance between the access site and the serving replica, frequency of configuration changes, when varying the traffic load, the network topology, the type of traffic (Poisson, long tail).
- Together with the proposed load balancing request redirection scheme, the

distributed replica placement heuristic is effective in keeping the “minimum” number of replicas in the system (cloning replicas only when needed, removing replicas whenever some replicas fall below the desired utilization level).

**Load balancing** We have proposed a distributed load balancing algorithm, which is meant to be executed by the access nodes of the CDN. According to such scheme access nodes only need to monitor nearby replicas load conditions to compute their redirections. The algorithm is executed iteratively, and it is shown to converge to the optimal load balancing in a few steps. Summing up, the algorithm:

- balances the load among replicas, thus lowering the user perceived latency due to the server performance
- is completely distributed, as every access node makes use only of local information
- is dynamic: every time the traffic or the number of replicas change, it adapts the current redirection decisions to the new state
- is asynchronous: it is not necessary for access nodes to operate at the same time
- is able to deal with servers of heterogeneous capacity
- respects the distance constraint: requests will be served only by replicas within a maximum distance

Simple modifications to the distributed load balancing algorithm allow it to cooperate with our replica placement heuristic to also enable fine grain control of the replica level of utilization. Our solution in this way intentionally breaks the load balancing in a controlled manner, as load unbalancings are limited and functional to reduce the number of allocated replicas.

The idea is that some underutilized replicas inflate the announced load (used for sake of load balancing) therefore getting lower and lower share of the users requests and being removed if possible. Here a replica is said to be underutilized if its current load is below a predefined range set by the CDN operator.

The overall solution we have proposed is fully distributed and localized and allows to minimize the costs for replica placement, maintenance, replicas adds/removals

while being able to satisfy all users requests and to keep allocated replicas load in a target range.

In particular, the redirection scheme is very effective in performing load balancing, keeping replicas within the specified interval of utilization. By setting different target levels ( $U_{mid}$ ) of utilization of the replicas we can strictly control the CDN network operations and trade-off between number of replicas, replicas utilization and frequency of changes in the replica placement.

# Bibliography

- [1] W.M. Aioffi, G.R. Mateus, J.M. Almeida, and D.S. Mendes. Mobile dynamic Content Distribution Networks. *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 87–94, 2004.
- [2] Akamai. Website. <http://www.akamai.com>.
- [3] Akamai Dynamic Site Accelerator. Online, White Paper. [http://www.akamai.com/dl/brochures/akamai\\_dsa\\_sb.pdf](http://www.akamai.com/dl/brochures/akamai_dsa_sb.pdf).
- [4] M.F. Arlitt and C.L. Williamson. Web server workload characterization: the search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.
- [5] A.T.Bharucha-Raid. *Elements of the theory of Markov processes and their applications*. McGraw-Hill, 1960.
- [6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the Web’s infrastructure: from caching to replication. *Internet Computing, IEEE*, 1(2):18–27, 1997.
- [7] N. Bartolini, F. Lo Presti, and C. Petrioli. Optimal dynamic replica placement in Content Delivery Networks. In *Proceedings of ICON 2003*, pages 125–130, Sydney, Australia, September 28–October 1 2003.
- [8] Novella Bartolini, Emiliano Casalicchio, and Salvatore Tucci. A walk through Content Delivery Networks. In Mariacarla Calzarossa and Erol Gelenbe, editors, *MASCOTS Tutorials*, volume 2965 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2003.

- [9] BitGravity. Website. <http://www.bitgravity.com/>.
- [10] G. Box, G. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, third edition, 1994.
- [11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1, 1999.
- [12] S. Buchholz and T. Buchholz. Replica placement in adaptive content distribution networks. *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1705–1710, 2004.
- [13] S. Buchholz and A. Schill. Adaptation-Aware Web Caching: Caching in the Future Pervasive Web. *13th GI/ITG Conference Kommunikation in verteilten Systemen (KiVS)*, 2003.
- [14] CacheFly. Website. <http://www.cachefly.com>.
- [15] V. Cardellini, M. Colajanni, and P.S. Yu. Redirection algorithms for load sharing in distributed web-server systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 528, Washington, DC, USA, 1999. IEEE Computer Society.
- [16] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.
- [17] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 378–388, 1999.
- [18] Moses Charikar, Samir Khuller, David M. Mount, and Giri Narasimhan. Algorithms for facility location problems with outliers. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 642–651, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

- [19] Y. Chen, R. Katz, and J. Kubiawicz. Dynamic replica placement for scalable content delivery. In *International Workshop on Peer-to-Peer Systems, IPTPS 2002*, Cambridge, MA, March 7–8 2002.
- [20] F.A. Chudak and D.P. Williamson. Improved approximation algorithms for capacitated facility location problems. *Mathematical Programming*, 102(2):207–222, 2005.
- [21] Julia Chuzhoy and Yuval Rabani. Approximating k-median with non-uniform capacities. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 952–958, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
- [22] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 177–190, New York, NY, USA, 2002. ACM Press.
- [23] M. Colajanni and PS Yu. A performance study of robust load sharing strategies for distributed heterogeneous Web server systems. *Knowledge and Data Engineering, IEEE Transactions on*, 14(2):398–414, 2002.
- [24] M. Colajanni, PS Yu, and DM Dias. Analysis of task assignment policies in scalable distributed web-server systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(6):585–600, 1998.
- [25] I. Cooper, I. Melve, and G. Tomlinson. Internet Web Replication and Caching Taxonomy. Technical report, Internet-Draft, RFC-3040, 2001.
- [26] P. Crescenzi and V. Kann. A compendium of NP optimization problems. <http://www.nada.kth.se/~viggo/problemlist>.
- [27] M. Dahlin. Interpreting stale load information. *Parallel and Distributed Systems, IEEE Transactions on*, 11(10):1033–1047, 2000.
- [28] Brian D. Davison. Content delivery and distribution services. Website. <http://www.web-caching.com/cdns.html>.
- [29] M. Day, B. Cain, G. Tomlinson, and P. Rzewski. RFC3466: A Model for Content Internetworking (CDI). *Internet RFCs*, 2003.

- [30] Level 3 Content Delivery Network (formerly Digital Island CDN). Website. <http://www.level3.com/content/services/cdn/index.html>.
- [31] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.
- [32] L. Ercetin, O. Tassiulas. Market-based resource allocation for content delivery in the internet. *IEEE Transactions on Computer*, 52(12), 2003.
- [33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol–HTTP/1.1. *Internet RFCs*, 1999.
- [34] A.V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming*, 71(2):153–177, 1995.
- [35] J. Gossa, J. M. Pierson, and L. Brunie. Fredi: Flexible replica displacer. In *Proceeding of the International Conference on Networking, International Conference on Systems and Intenational Conference on Mobile Communications and Learning Technologies (ICNICONSMCL-06)*, 2006.
- [36] GT-ITM: Georgia Tech Internetwork Topology Models. Website. <http://www.cc.gatech.edu/projects/gtitm>.
- [37] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31(1):228–248, 1999.
- [38] A. Gupta and G. Baehr. Ad insertion at proxies to improve cache hit rates. *Proceedings of the 4th International Web Caching Workshop*, 1999.
- [39] D. P. Heyman and M. J. Sobel. *Stochastic Models in Operations Research*. McGraw-Hill, 1984.
- [40] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.
- [41] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *Proceedings of IEEE INFOCOM 2001*, pages 31–40, Anchorage, AK, April 22–26 2001.

- [42] KL Johnson, JF Carr, MS Day, and MF Kaashoek. The measured performance of Content Distribution Networks. *Computer Communications*, 24(2):202–206, 2001.
- [43] J. Kangasharju, J. Roberts, and K.W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.
- [44] Magnus Karlsson and Christos Karamanolis. Choosing replica placement heuristics for wide-area systems. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.
- [45] Magnus Karlsson, Christos Karamanolis, and Mallik Mahalingam. A unified framework for evaluating replica placement algorithms. *Technical Report HPL-2002*, Hewlett Packard Laboratories, 2002.
- [46] Magnus Karlsson and Mallik Mahalingam. Do we need replica placement algorithms in content delivery networks. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, August 2002.
- [47] J. Keilson. *Markov chain models. Rarity and exponentiality*. Springer-Verlag, New York, 1979.
- [48] T. Kelly and J. Mogul. Aliasing on the World Wide Web: prevalence and performance implications. *Proceedings of the eleventh international conference on World Wide Web*, pages 281–292, 2002.
- [49] Al-Mukaddim Khan Pathan and Rajkumar Buyya. A taxonomy and survey of Content Delivery Networks. *Technical Report GRIDS-TR-2007-4*, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, February 2006.
- [50] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman. Analysis of a Local Search Heuristic for Facility Location Problems. *Journal of Algorithms*, 37(1):146–188, 2000.
- [51] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001.

- [52] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 169–182, New York, NY, USA, 2001. ACM Press.
- [53] N. Laoutaris, O. Telelis, V. Zissimopoulos, and I. Stavrakakis. Distributed selfish replication. *IEEE Transactions on Parallel and Distributed Systems*, 2005.
- [54] F.T. Leighton. HTML delivery from edge-of-network servers in a content delivery network (CDN), November 6 2007. US Patent 7,293,093.
- [55] F.T. Leighton and D.M. Lewin. Content delivery network using edge-of-network servers for providing content delivery to a set of participating content providers, April 22 2003. US Patent 6,553,413.
- [56] F.T. Leighton, R. Sundaram, R.S. Dhanidina, R. Kleinberg, M. Levine, A.M. Soviani, B. Maggs, H.S. Rahul, S. Thirumalai, J.G. Parikh, et al. Global load balancing across mirrored data centers, September 19 2006. US Patent 7,111,061.
- [57] Y. Li and M. T. Liu. Optimization of performance gain in content distribution networks with server replicas. In *Proceedings of the 2003 Symposium on Applications and the Internet, SAINT 2003*, pages 182–189, Orlando, FL, January 27–31 2003.
- [58] F. Lo Presti, C. Petrioli, and C. Vicari. Dynamic replica placement in content delivery networks. In *Proceedings of MASCOTS 2005*, September 2005.
- [59] F. Lo Presti, C. Petrioli, and C. Vicari. Distributed dynamic replica placement and request redirection in Content Delivery Networks. In *Proceedings of MASCOTS 2007*, October 2007.
- [60] Ari Luotonen and Kevin Altis. World-wide web proxies. In *Selected papers of the first conference on World-Wide Web*, pages 147–154, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.
- [61] A. Mahanti, C. Williamson, and D. Eager. Traffic analysis of a Web proxy caching hierarchy. *Network, IEEE*, 14(3):16–23, 2000.

- [62] M. Mahdian, Y. Ye, and J. Zhang. Improved approximation algorithms for metric facility location problems. *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 229–242, 2002.
- [63] Content Delivery and the Mirror Image Adaptive CAP Network. Online, White Paper. [http://www.mirrorimage.net/Site/Portals/0/MII-90014-005\\_CAP-WP.pdf](http://www.mirrorimage.net/Site/Portals/0/MII-90014-005_CAP-WP.pdf).
- [64] Powering your Web Strategy with CDN Services. Online, White Paper. <http://www.mirrorimage.net/Site/Portals/0/MII-90032-001.pdf>.
- [65] M. Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.
- [66] J. Mogul and P. Leach. RFC 2227: Simple hit-metering and usage-limiting for HTTP, Oct. 1997. *Status: Proposed standard*. URL: <ftp://ftp.math.utah.edu/pub/rfc/rfc2227.txt>, 1997.
- [67] Opnet University Program. Website. <http://www.opnet.com/services/university>.
- [68] V.N. Padmanabhan and L. Qiu. The content and access dynamics of a busy Web site: findings and implications. *ACM SIGCOMM Computer Communication Review*, 30(4):111–123, 2000.
- [69] George Pallis and Athena Vakali. Insight and perspectives for content delivery networks. *Commun. ACM*, 49(1):101–106, 2006.
- [70] Stefan Podlipnig and Laszlo Böszörményi. A survey of Web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.
- [71] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *Proceedings of IEEE INFOCOM 2001*, pages 1587–1596, Anchorage, AK, April 22–26 2001.
- [72] Michael Rabinovich and Amit Aggarwal. RaDaR: a scalable architecture for a global Web hosting service. *Elsevier Computer Networks*, 31(11–16):1545–1561, 1999.

- [73] Michael Rabinovich, Irina Rabinovich, Rajmohan Rajaraman, and Amit Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 101, Washington, DC, USA, 1999. IEEE Computer Society.
- [74] Pavlin Radoslavov, Ramesh Govindan, and Deborah Estrin. Topology-informed internet replica placement. *Proceedings of WCW'01*, June 20–22 2001.
- [75] Project Rocketfuel. Website. <http://www.cs.washington.edu/research/networking/rocketfuel>.
- [76] S. Ross. *Applied probability models with optimization applications*. Holden-Day, 1970.
- [77] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, January-February 2007.
- [78] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.
- [79] Ao-Jan Su, David R. Choffnes, Aleksandar Kuzmanovic, and Fabián E. Bustamante. Drafting behind Akamai (travelocity-based detouring). *SIGCOMM Comput. Commun. Rev.*, 36(4):435–446, 2006.
- [80] Michał Szymaniak, Guillaume Pierre, and Maarten van Steen. Latency-driven replica placement. In *Proceedings of the International Symposium on Applications and the Internet (SAINT)*, pages 399–405, Trento, Italy, February 2005.
- [81] Michał Szymaniak, Guillaume Pierre, and Maarten van Steen. Latency-driven replica placement. *IPSIJ Journal*, 47(8), August 2006. [http://www.globule.org/publi/LDRP\\_ipsj2006.html](http://www.globule.org/publi/LDRP_ipsj2006.html).
- [82] X. Tang and J. Xu. On replica placement for QoS-aware content distribution. *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, 2, 2004.

- [83] March 2005 bandwidth report online. Website. <http://www.urlwire.com/news/032105.html>.
- [84] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.
- [85] Claudio Vicari, Chiara Petrioli, and Francesco Lo Presti. Dynamic replica placement and traffic redirection in Content Delivery Networks. *SIGMETRICS Perform. Eval. Rev.*, 35(3):66–68, 2007.
- [86] P. Vixie and D. Wessels. Hyper Text Caching Protocol (HTCP/0.0). *Request for Comments RFC*, 2756, 2000.
- [87] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, 1997.
- [88] K.Y. Wong. Web cache replacement policies: a pragmatic approach. *Network, IEEE*, 20(1):28–34, 2006.
- [89] J.L. Yuan and C.H. Chi. Web Caching Performance: How Much Is Lost Unwarily. *the Proceedings of the Second International Human. Society@ Internet Conference*, pages 23–33, 2003.
- [90] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiawicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communication*, 22(1), January 2004.