Francisco Perdigão Fernandes

Licenciado em Engenharia Informática

Improving Web-Caching Systems with Transparent Client Support

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Doutor João Carlos Antunes Leitão, Assistant Professor, Faculdade de Ciencias e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente:	Doutora Margarida Paula Neves Mamede, Assistant Professor
Arguentes:	Doutor Hugo Alexandre Tavares Miranda, Assistant Professor
	Doutor João Carlos Antunes Leitão, Assistant Professor

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2017

Improving Web-Caching Systems with Transparent Client Support

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Acknowledgements

The work presented in this thesis was partially supported by the NOVA LINCS laboratory under the grant (UID/CEC/04516/2013) and by Microsoft Azure for Research Grant attributed to Prof. Dr. João Leitão. I want to sincerely thank all the support of my advisor Prof. Dr. João Leitão, for the hours he spent helping me to understand the path I had to take to achieve the goals proposed in this dissertation. His experience in the area of Distributed Systems helped me a lot to realize some points that were fundamental to the accomplishment of the work described in all these pages. I would also like to thank the PhD student Albert Linde, who was relentless in helping me during the months of both preparation and elaboration of this dissertation. It is relevant to mention that one of the components used in this work (Legion framework) was the result of his Master thesis [53].

Finally, I would like to thank all my family, my parents and brother, and close friends, specially to my girlfriend Andreia Monteiro, for always believing in me and supporting me at the most difficult times during the realization of this thesis, which supported me throughout my academic years.

Abstract

The increase in popularity of Web applications has lead to a significative increment on the load imposed on their supporting servers. To deal with this, and with the need to provide fast response to users, there has been an effort on deploying Web caching systems, which significantly reduce the end user perceived latency and the load on origin servers. Since these systems have a limit on how many content they can effectively cache and serve, large-scale providers have begun to explore the use of peer-to-peer techniques to greatly alleviate the burden on (dedicated) cache servers.

The goal of this work is to developed a solution that enriches distributed cache architectures with transparent client support in the browser. This way, the caching horizon will be transparently extended towards the clients, offering the opportunity to have clients serving content directly among them in a peer-to-peer fashion. Additionally, such system can be readily incorporated in existing applications without requiring Web applications operators to pay for specialized distributed caching services, which might not be viable for operators of small and medium scale applications.

Keywords: Web caching, peer-to-peer, transparent client support.

Resumo

O aumento da popularidade das aplicações *Web* tem levado a um aumento significativo na carga imposta nos servidores de suporte. Para lidar com isto, e com a necessidade de dar uma resposta rápida aos utilizadores, tem havido um esforço na implantação de sistemas de *cache Web*, os quais reduzem significativamente a latência percecionada pelos utilizadores finais e a carga nos servidores de origem. Visto que estes sistemas têm limites na quantidade de conteúdo que podem efetivamente armazenar e servir, fornecedores de larga escala têm começado a explorar o uso de técnicas de *peer-to-peer* para aliviar substancialmente a carga dos servidores (dedicados) de *cache*.

O objetivo deste trabalho é desenvolver uma solução que enriqueça arquiteturas de *cache* distribuídas com suporte transparente ao cliente no navegador. Deste modo, o horizonte de *cache* será estendido transparentemente para os clientes, oferecendo a oportunidade de ter clientes a servirem conteúdos diretamente entre eles de uma forma *peer-to-peer*. Adicionalmente, tal sistema pode ser prontamente incorporado em aplicações existentes sem exigir aos operadores de aplicações Web que paguem por serviços especializados em *cache* distribuída, que pode não ser viável para os operadores de aplicações de pequena e média escala.

Palavras-chave: *Web caching*, *peer-to-peer*, suporte transparente ao cliente.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Several web applications that operate on large scale use services that rely heavily on Web caching infrastructures. These systems temporarily store website's and web application's contents in order to improve their access, resulting in overall reduced bandwidth usage, server load, and user-perceived latency. The main advantage of using these systems is to provide a better user experience, without compromising server-side reliability. Nowadays, this solution is widely adopted due to the natural growth of web services that have to provide content quickly and efficiently to a large amount of users simultaneously. This requires efficient software and, specially, efficiency on network and hardware management.

Distributed cache systems can be configured properly to improve efficiency of different services. One example is a well-known simple memory caching solution: MemCached [79], which is an open-source implementation of an in-memory hash table, which provides low latency access and shared storage at a low cost. This solution is an attractive component in a large scale distributed system. There are other solutions of this type such as Redis [26] - also used as a database and message broker.

When we talk about content distribution systems two architectures have been traditionally adopted: CDNs (networks based on content delivery infrastructures) where clients download content directly from dedicated, centralized servers; and peer-to-peer CDNs, where clients download content from other clients. Recently a combination of these two systems was adopted - Peer-Assisted CDNs, which is an example of a system combining both approaches. This type of system depends on a controlled infrastructure but includes a peer-to-peer element to provide the content to the users. An example of this is Akamai NetSession, being a product from Akamai, a huge commercial CDN that exploits this architecture.

1.2 Motivation

Web caching has become a desirable solution for everyone. In addition to its advantages referred in the previous section, a relevant motivation for users that visit websites with efficient caching systems (i.e., local cache) is the fact that they significantly save mobile data, provided by their operators. This results in less financial expenses due to a reasonable reduction of network usage.

Even relying on a distributed architecture, such systems have a limit on how many content they can cache, penalizing the access to those that are eventually removed. Some Web caching services use systems such as Akamai NetSession that require users to install their software on their computers. For example, to use NetSession the required software is the NetSession Interface, which runs in the background and whenever an object needs to be downloaded this software will download it from edge servers, and in parallel obtains and controls a list of the nearest peers (i.e., other clients running NetSession) that have a copy of that object. The dependence on a software to enjoy this type of service negatively affects the user experience, first because they have to install something on their computers and second because even running in background there will always be other variables going on, such as software security, corrections, updates, etc. Additionally, small to medium scale web application operators might not have the financial resources to invest in paid services such as Akamai's CDN, while having an interest in exploiting cache layers at the edge of the network.

In order to combat this dependency, it would be beneficial and advantageous to develop new cache solutions that are totally transparent and enjoy the same properties of a Peer-Assisted CDN system (similar to NetSession), for instance, by exploiting the user's browser local storage. Such solutions would be realized and enriched through the use of WebRTC connections, so that it can be fully integrated in any browser (that supports this technology, which is true for most modern browsers), without the need to install any extension, plugin, or additional software.

1.3 Proposed Solution

To overcome network load problems on Web servers and the resulting slow access by end users, in this dissertation we propose to develop a Javascript framework that optimizes Web caching between clients, as detailed in Section 3.1. This solution can leverage on components of the Legion [54] system and WebRTC connections, which offers the possibility of using clients memory and local storage to temporarily store (cache) content, and share it in a transparent way with other users within a peer-to-peer network.

The goal is to make users' web applications access faster, making fundamental resource sharing by forming a cooperative cache between clients. This is more likely to be achieved when the origin servers are overloaded, where the cost of sending and receiving requests to them is potentially higher than asking peers who are active on the network, potentially close by.

This solution will encounter several challenges, in particular the interaction of the proposed framework with the Document Object Model (DOM), since elements of the DOM will be manipulated during loading of web pages, while at the same time sharing web content between clients and consequently storing these in the browser local cache.

1.3.1 Main Contributions

In summary, the main contributions of this dissertation are as follows:

- Design and implementation of a framework running transparently in the clients browser that perform caching between users through peer-to-peer techniques (leveraging existing components in Legion and WebRTC connections).
- A peer-to-peer search mechanism tailored for our WebRTC network to locate.
- Browser's local storage management through caching replacement strategies.
- Evaluation of the proposed solution and comparison to existing alternatives.

1.4 Document Organization

The remainder of this document is organized in the following manner:

- **Chapter** 2 describes relevant related work. Existing peer-to-peer technologies, and Web caching systems and policies are discussed. Recent technologies are also referred and briefly discussed.

- **Chapter 3** describes the proposed solution.

- **Chapter 4** presents the evaluation and results.

- **Chapter 5** discusses and conclude this dissertation. It also suggests possible features to be implemented in the proposed solution as future work.

Chapter 2

Related Work

This dissertation faces the challenges of enriching distributed web caching systems with transparent client support in the browser. However, some aspects must be considered. The following sections cover these relevant subjects:

In **Section 2.1** we briefly explain how peer-to-peer systems work, describing centralized and decentralized systems also discussing relevant solutions in this domain.

In **Section 2.2** we study web cache systems, and existing web caching replacement strategies, and recent developments. Also, we discuss existing implementations of distributed cache systems (e.g. Memcached and Redis).

In **Section 2.3** we give an overview of some recent technologies which serve as background to address challenges faced by the work presented in the thesis.

2.1 Peer-to-peer systems

Peer-to-peer systems have attracted significant interest in recent years, but they emerged around 2000. Projects like Napster [59], which is a music-sharing system, Freenet [22], which is a platform for censorship-resistant communication, and SETI@home [7], which was a volunteer-based scientific computing project, became the firsts peer-to-peer systems that took a big impact on the Internet and most of them still operate nowadays.

Peer-to-peer is a distributed application architecture with a high degree of decentralization, since most of the communication is done between the peers (i.e., client applications) directly. More specifically, system's state and tasks are spreaded over them. Peers have equal privileges, and they form a peer-to-peer network that is leveraged by those applications. These participant nodes share their resources, such as network bandwidth and disk storage, to other participants. Typically, they don't need central coordination by servers, however centralized states may exist but on a smaller scale, and is used mostly as fallback, being managed by a few dedicated nodes.

When a node is introduced to the system, little or no manual configuration is needed to maintain the system. The participating nodes do not depend of a single organization or control point, since they're operated by an independent set of individuals who join the system on their own.

In opposition to client-server systems, peer-to-peer systems have a low barrier to deployment. This means the investment needed to deploy a P2P service tends to be low, as the resources are contributed by the participating nodes. These systems grow in a organic way and tend to be resilient to faults since there are few, if any nodes, that are critical to ensure the system operation. Moreover, peer-to-peer systems tend to be resilient to attacks because there is no centralized server or operation to be attacked that could shutdown the entire system. Also, there is an abundance and diversity of resources in P2P systems, that's why it is easier to scale this type of services.

P2P systems can take different forms of serving content as a service. The most successful and popular systems address sharing and distribution of files (e.g. eDonkey [86] and BitTorrent [68]), streaming media (e.g. PPLive [80] and CoolStreaming [101]), telephony (e.g. Skype [11]), and volunteer computing (e.g. SETI@home [7]). P2P systems have also been designed for other proposes like distributed data monitoring [78], management and data mining [20], massively distributed query processing [97], serverless email [57], archival storage [70], bibliographic databases [33] and cooperative backup [31], but did not have as much development and stability as the popular ones. Also, technology developed for P2P applications has been included in other types of systems, such as Akamai Netsession [102], which uses P2P downloads to increase performance and reduce the cost of delivering streaming content.

2.1.1 Overlay networks

An overlay network is a computer network built on top of another network. Peer-to-peer systems usually resort to overlay networks, which are defined as directed graphs G=(N,E), where N is a set of participant nodes (computers) and E is a set of overlay links. A connect pair of nodes share their IP addresses so they can communicate with each other via the Internet. These communications between peers fuels a peer-to-peer system by allowing the system to have a natural organic growth when more nodes join the system.

2.1.1.1 Degree of centralization

The presence of centralized components allows to classify a peer-to-peer system, which can be centralized, decentralized, or a combination of these two, which we name hybrid system.

2.1.1.2 Partly centralized

These type of peer-to-peer systems provide a reliable and fast resource location, being relatively simple to build, characterized by centralized components that coordinate system connections and facilitate communication between peers. The centralized component can be materialized by a set of dedicated nodes or a single central server. Due to this, scalability is affected because some single points of failure might emerge. Examples of these systems include most P2P BitTorrent protocols, that have a website, also know as *tracker*, that keeps all information about peers activity and periodically provides each peer in a swarm with a new set of peers they can (attempt to) connect to. However, as soon as sharing begins, communication between peers will continue without the need of constantly communicating with the centralized tracker. Another example is Napster, a digital audio file sharing service that (in its origin server) maintained membership and a content index in their centralized component (website); and BOINC [6], which is an open-source middleware system that supports volunteer computing, consisting on a central server system and a client software that communicates with the central component in order to process work units and return the results of these computational tasks.

2.1.1.3 Decentralized

Decentralized peer-to-peer systems will typically spend more time and resources for locating resources due to the lack of a central entity that supports this operation. However, by avoiding the existence of this central unit, they avoid single points of failure. Peers have equal rights and duties and each one has only a partial view of the network. Scalability, robustness, and performance are the main reasons why decentralized peer-to-peer systems are very desirable.

Regarding the design of these systems, there are two dimensions [89]. Concerning to the structure, these systems can be classified as flat (single-tier) or hierarchical (multi-tier), and concerning to the logical network topology it can be structured or unstructured. The difference between these two designs significantly affects the operation and interaction among peers. For instance, it has a notorious impact in the way resource location queries are propagated:

- Structure
  - **Flat:** consists in a single layer of routing structures, being the load uniformly distributed among all peers. This structure composes most decentralized systems.
  - **Hierarchical:** consists in multiple layers of routing structures, providing efficient caching, bandwidth saving, and fault isolation. Examples of this category are Crescendo [38] and super-peer architectures [94]).
- Logical Network Topology

- **Structured Overlays:** typically data is placed under the control of a DHT (distributed hash table), requiring more resources to maintain the overlay but being more efficient on resolving exact match queries. Example of protocols that form and maintain a structured overlay network (i.e., DHT) include Chord [84], where queries are propagated to node's successor through a ring of connected nodes; and Pastry [75], which has slightly different routing tables at each node.
- **Unstructured Overlays:** there is no mapping between the identifiers of objects and peers, with each peer keeping information about only its own (and possible its direct neighbours) resources. Usually, this is intended as to protect the user's and publisher's anonymity. Thus, queries are propagated among all participants to get all possible results. An example of a popular unstructured peer-to-peer system is FreeNet [22], which is a platform for censorship-resistant communication, that consists on distributing encrypted information around the network and storing it on several different nodes, without using any central servers or dedicated nodes.

2.1.1.4 Hybrid

The goal of these type of peer-to-peer system architectures is to obtain the best of both centralized and decentralized architectures. To overcome the scalability issue on centralized systems, there are no servers. Peers considered more powerful are named super peers and will act as servers to provide resources to a fraction of peers. Thus, resource location can be made by a combination of decentralized and centralized search techniques (e.g. centralized by communicating with super peers), where most of the traffic and consumption of resources happens at the super-peers layer that interact with each other using decentralized approaches.

2.1.2 Query Dissemination Techniques

Many popular peer-to-peer services face a critical problem: resource location. It is not hard to locate popular resources, but less popular content that users want can become an hard challenge. Query dissemination techniques [15, 51] consist of routing algorithms that try to address this challenge on P2P systems. Centralized systems use a central component that exchange information with users to speed up the location of required resources, so there is no need to resort to these distributed query dissemination techniques. In decentralized systems, structured overlays do not require them either, as they are based on DHTs, which have efficient routing mechanisms for identifiers in each node, for example by contacting a peer whose identifier is closest to the target resource identifier. On the other hand, in unstructured overlays there is an effective need to exploit these techniques, since there is no direct correlation between the identifiers of objects and peers and their location in the network.

2.1.2.1 Flooding

Flooding technique consists on disseminating queries among participants. This method is simple to implement but scales poorly, working well on small networks with few requests.

Complete Flooding disseminates each query among all participants, this can cause overhead with an extremely large number of possible returned answers for each query. An example of a system using this technique is the Coral content distribution network [25].

Flooding with Limited Horizon is a variant of flooding, which basically consists on propagating a query but limited to a certain fraction of the overlay, with a time to live (TTL) value that is decremented on each retransmission of the query. This strategy reduces overhead introduced by the Complete Flooding, while allowing to miss relevant hits for the query.

2.1.2.2 Random Walks

The random walk technique is an alternative to flooding, that consists on forwarding (walking) the query message (walker) to a single randomly chosen overlay neighbor at each step of the algorithm, also known as Blind Random Walk. When the resource is found the walk stops. There are some extensions of this technique:

Guided Random Walks: employed to improve query efficiency, through exchange of information between overlay neighbors in order to *guide* the random walk in the overlay. A possible implementation of this technique relies on the use of Bloom Filters [13], where each peer exchanges summarized information with its neighbors about their local indexes in the form of bloom filters.

Biased Random Walks: each step of the algorithm is affected by information kept by neighbors or that was previously obtained. This information is potentially imprecise.

Parallel Random Walks: initiates with k random walks in parallel that stop when the resource is found.

Limited Random Walks: defines a threshold on the amount of total random walks that the algorithm will use, and stops even if the resource is not found.

2.1.2.3 Gossip-based

Gossip-based techniques consists on the use of interest communities to locate user resources, through a collaborative process between them. The routing process addresses the communication of the source peer with one or more neighbors, to forward a query request. This query is disseminated randomly to other peers, and this process is repeated until the query covers all (or most) of the nodes in the system.

Appropriate degree gossip search algorithm (ADGSA) is a gossip-based technique, proposed in [40]. The authors of this work prove that, with a user profiling method based on the characteristics of peers and the similarity built from an interest community, performance can be improved dramatically, including the success rate, recall rate, and search response time, as compared to other resource location techniques in P2P systems.

2.1.3 Concrete System and Framework Examples

There are several web services that use peer-to-peer systems and frameworks to scale, which improve user experience and reduce network load, without compromising system reliability.

2.1.3.1 Legion

Legion [54] is a framework that allows client web applications to replicate data from servers and propagate those among other clients through peer-to-peer interactions. Messages are propagated through the overlay network, where a communication module exposes an interface with point-to-point and multicast primitives. This allows a client to send a message to another client or to a group of clients, where these messages are secured using a symmetric cryptographic algorithm.

This framework allows the use of extensions to leverage existing web platforms, an example is that they implement an extension that interacts and exports the same API as Google Drive Realtime (GDriveRT). This allows the interaction and sharing of the same objects between legacy clients accessing GDriveRT directly and enriched clients using Legion. In the referred paper, an experimental evaluation shows that this framework provides much lower latency for update propagation among clients and a decrease of the network traffic load on servers.

In short, Legion offers shared mutable data and replication, while handling all concurrency issues, with clients interacting with each other automatically through WebRTC or by using client-server only interactions, ensuring reduced server load and scalability.

2.1.3.2 Akamai's NetSession

Akamai's NetSession [102] is a famous peer-to-peer CDN system. It operates since 2010 and has more than 25 millions users in 239 countries. The goal of NetSession is to distribute content, provided by content providers, such as software and media publishers, to Akamai users, improving their download efficiency and speed. This system is composed by an infrastructure of edge servers operated by Akamai, and users that participate through a software called **NetSession Interface**. The edge servers also support critical functions, such as content integrity verification, by generating and keeping secure content IDs and hashes of each file piece; and authorization, by providing HTTPS connection that will authenticate peers before they can receive content from other peers. The software allows Akamai to use user's idle bandwidth and computing resources to upload files to other Akamai users, by joining a peer-to-peer network. A drawback of this service is the obligation that users have to install and use additional software on their computers.

Concerning peer coordination and accounting, it is independent from Akamai edge servers, being exclusively done by a specialized group of globally-distributed servers called the NetSession control plane [102]. Each control plane server has the task of running some components. They can run **Connection Nodes (CNs)**, which are the endpoints of the TCP connections peers open when they become active; **Database Nodes (DNs)**, which keeps a database with details about peers activity and their objects availability; **STUN**, which is a component that peers periodically exchange information with about their connectivity, that is stored in the DNs, which enable Network Address Translation (NAT) traversal; and **Monitoring Nodes**, which is a component that receives peer information about operations and failures, such as application crash reports, thus helping to monitor the network in real-time and gathering useful information to solve user issues.

2.1.3.3 Peer5

Peer5 [65] is a peer-to-peer Content Distribution Network (CDN) for massively-scaled, high-quality streaming. It consists on an elastic mesh network based on WebRTC that provides resources for users to load video content from each other. Its hybrid switching algorithm decides if a viewer should load either from the P2P network or from the publisher's alternative delivery system, reducing the provider's average bandwidth usage, while improving user experience. As the P2P layer is built on top of a client-server distribution system, it is less likely for peak demand issues to ocurr, increasing reliability of the service and reducing failures in other critical situations. Regarding the system's security, Peer5 uses WebRTC data channel to transfer data between users and every single communication is secured with SCTP protocols and TLS encryption, and with the Peer5 backend, the information exchanged is secured with WebSocket that also uses TLS.

2.2 Web Cache

The Web has been constantly growing, particularly during the last decade. Web service bottlenecks and an increase of load on the Internet have led to a search for solutions to attenuate negative effects of this growth such as server overload and increased user perceived latency. Although there have been big improvements on websites itself and Web servers, this was not enough to keep up with this kind of growth.

As it is shown in Figure 2.1, in 1995 the Internet already had approximately 44 million users, in 2005 the users increased to approximately 1 billion, and last year (2016) the user base had grown to approximately 3.4 billion users, which corresponds to 46% of the world's population. This represents a growth of almost 8000% of Internet users during the last decades. Imagine that if we had not taken steps to manage this growth, it would probably be impossible to have fast access to websites nowadays. Thus, different techniques have been adopted to minimize network load and improve quality of service

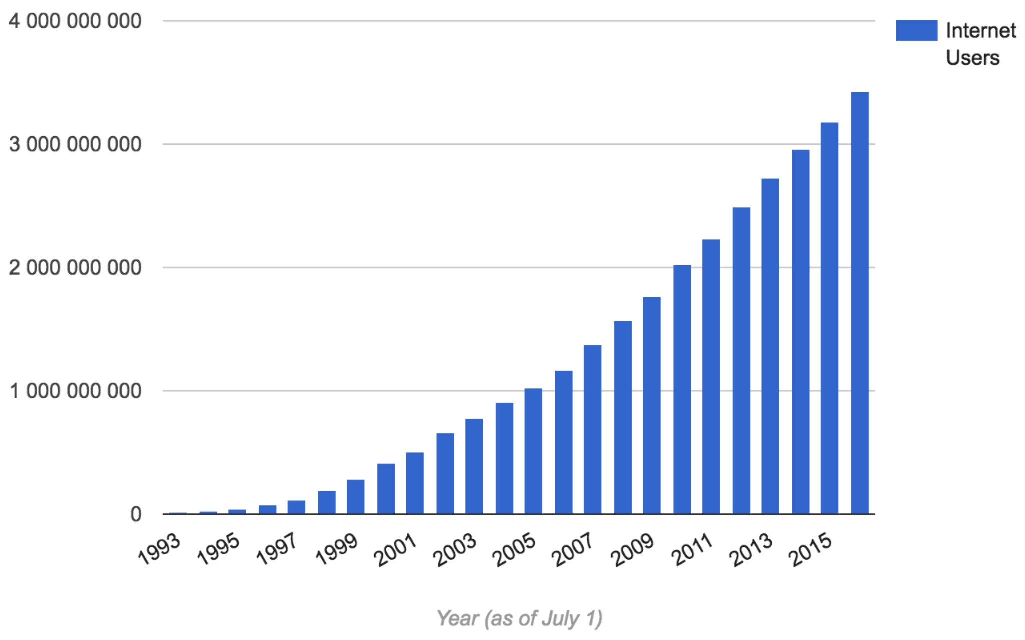


Figure 2.1: Internet Users in the World [45]

as to provide a better user experience. Web caching has proven to be an effective solution in this matter, making possible a sustained web scalability.

Web caching allows to temporarily store and serve (cache) data, like HTML pages, images, videos and other files types, at locations that are close to the clients, so they can get faster access to them. Since it's not possible to cache every web server's objects due to space availability and cache size, usually popular objects are the ones that are stored in this architectures. Web caching has a directly impact on the reduction of user-perceived retrieval latencies or delays, loads on the origin server (the number of requests effectively reaching and being processed by it), and overall network activity. This mechanism is very desirable for users, who avoid traffic congestion or delays issues when accessing websites. Other entities take great advantage of this, such as network administrators and people who create contents and manage their own websites. However, this solution is no panacea, as it faces an ambiguous problem: replacement strategy. From the point of view of cache management, the replacement strategy is fundamentally the policy that refers to the process of evicting old objects from the cache when it's full, so that it is possible to make free space to store new objects. Finding and choosing the best one, or a mixture of the existing ones, can be a hard challenge because different strategies have different design rationales and optimize different resources and aspects of systems.

Another fundamental question in Web caching is which objects should be cached or not, for example, is it worth to cache all popular objects? A good solution may be finding a trade-off between the number of references to certain objects and the time those objects are maintained in the cache. Dynamic websites have increased difficulty in measuring these situations because the content changes frequently and the number of requests to objects has huge variations.

2.2.1 Different Levels of Caching

Cache infrastructures are widely spread and located in many places across the Web. They can be referred as proxy servers, acting as intermediaries between clients and servers intercepting all Web requests at the boundaries of networks. Traffic that flows through a proxy server can be analysed, filtered, cached, modified, and secured when needed. There are two types of proxy servers: Forward and Reverse. A forward proxy provides services to a single client or a group of clients, and usually works with a firewall to provide more security to an internal network by controlling traffic from its clients that are directed to origin servers. In opposition to what a forward proxy does, a reverse proxy acts in the behalf of servers, typically as their load balancers and hiding their identities from the Internet. These proxies can act as proxy-cache servers, which in addition to the standard features of standalone proxy servers, store (cache) content to allow Web services to share those resources. Squid [81] is an example of a web caching proxy that can function as a forward proxy (default operation mode) and can also be configured to function as a reverse proxy.

However, proxy servers are not optimized for caching. They are in the way of all user traffic, which may cause some negative effects, such as network bottlenecks in the presence of heavy network loads, additionally, software or hardware failures can compromise the entire network operator. Also, this might require configuration of each user's Web browsers. Network caches [5] or Transparent caching [28] are solutions that optimize the Web caching process. They are designed to achieve transparency to clients through the WCCP (Web Cache Communication Protocol) [63], by intercepting and analyzing HTTP requests, and redirecting them to Web cache servers. Contrarily to proxy servers, this web caching solution does not require any configuration of users's Web browsers.

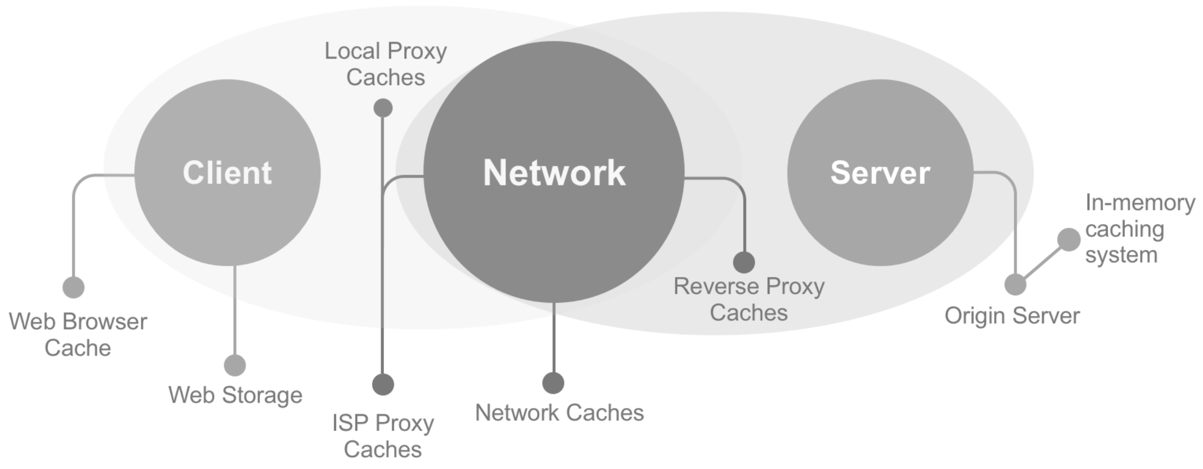


Figure 2.2: Overview of Different Levels of Caching

In terms of application-specific cache management, caches can be either implicit or explicit. Implicit cache means that all decisions are made automatically by the caching infrastructure, which usually is dependent on the network's activity and on applications in the environment that influence the traffic flow. It doesn't allow developers, with their applications, to take control of the cache. For example, Web browsers and most network cache infrastructures (e.g. proxies) manage their cache automatically, without accepting control of applications/extensions in the network. On the other hand, explicit cache means that applications can manually take control of the cache. For example, Web Storage, which is the client local storage, allows it by granting applications code privileges to manage this layer of caching. Using this application interfaces developers can optimize caching by choosing adequate replacement strategies for their application purposes.

As shown in Figure 2.2, there are different locations throughout the Web where objects can be cached. On the client side we have the Web browser's cache and Web storage. Proxy/cache servers can be located near the clients in local networks (e.g. institutional networks), inside ISPs networks, across the Internet and near origin servers (reverse proxies).

When a requested object is found on a cache server at some level, it is sent to the client and can be propagated to all previous intermediate proxy/cache servers. Thus, the next time the same object is requested, it is returned faster, as it now resides closer to the client.

This work will focus on cache infrastructures that allow developers to manage its decisions - explicit caches on the client side.

2.2.2 Caching Architectures

In the beginning of the Internet, Web caching was a solution naturally undeveloped. From local cache by a single browser to a shared cache serving all clients from a institution, caching architectures have been developed continuously. There are hierarchical, distributed, and more recently, hybrid architectures. Caching architectures allow cooperation between cache proxies. A web object present in the cache is a result of the probability of users wanting to access it. These architectures will help to ensure a structure so some form of coordination or communication might be required to exist between these different levels of caching.

2.2.2.1 Hierarchical Caching

The caching hierarchy, shown in Figure 2.3, denotes different cache layers at several levels of the network: bottom, which is the lower level of the hierarchy are the client caches, followed by institutional, regional, and national layers. If a document request is not satisfied at a certain cache level, it is redirected to the layer above it and if it is not found at any level, the national level contacts directly the origin server. When a document is found it goes down the hierarchy, potentially inserting a copy at each of the intermediate caches (this depends on the policies governing each cache layer). Several approaches based on this architecture were designed, such as the Adaptive Web caching [100] and the Access Driven cache [95].

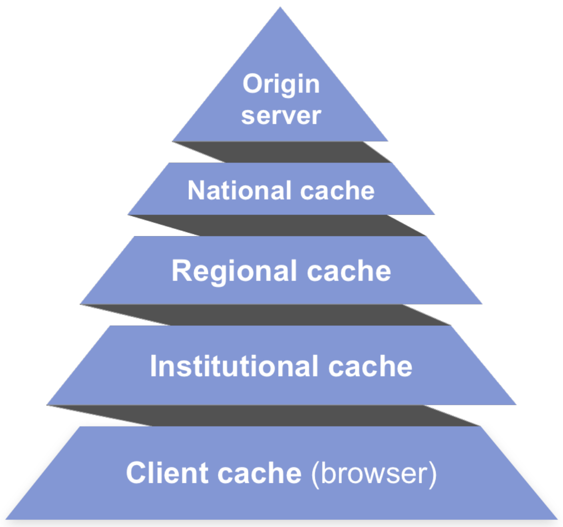


Figure 2.3: Hierarchical Caching - Pyramidal Representation

2.2.2.2 Distributed Caching

This architecture, illustrated in Figure 2.4, manages a cooperation between caches on the bottom level of the network. No intermediate copies are stored in the network, and there are only institutional caches which serve each others misses. These institutional caches store a copy of all locally requested objects. To share object copies between them, they keep metadata information of all objects of others institutional caches that are cooperating, to decide which cache will be chosen to retrieve a missing object. When a new object is fetched in any cache, its metadata is immediately updated at all cooperating institutional caches. Several instances of this architecture were designed, such as Internet Cache Protocol (ICP) [91], which sustain object finding and retrieval from neighbors as well as from parent caches, and Cache Array Routing protocol (CARP) [24].

2.2.2.3 Hybrid Caching

This architecture embodies aspects from both previously described caching architectures, where certain caches cooperate at the same or higher level of a caching hierarchy using distributed caching. Several approaches of this architecture were designed, such as in [90], where the authors propose to "employ the hierarchical distribution mechanism for more efficient and scalable distribution of metadata". In [5], the authors focus on designing "an hybrid-caching architecture for improving caching that has the lowest latency delivery

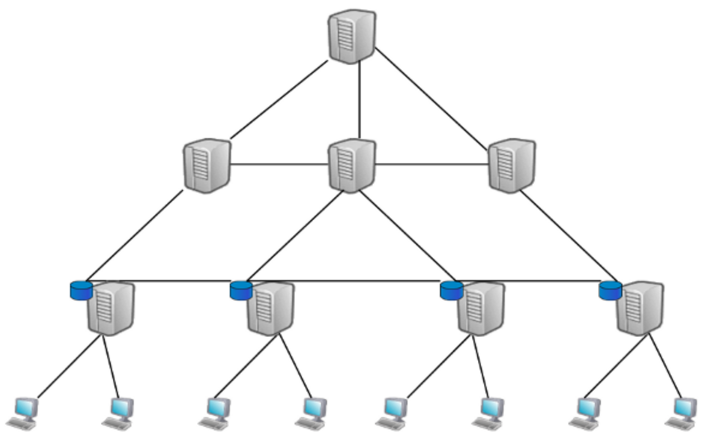


Figure 2.4: Distributed Caching Representation [17]

time for popular documents". In [17], the authors propose "a cooperative caching system that aims to achieve a broadband-like access to users with limited bandwidth, constructed from the caches of the connected clients as the base level, and a cooperative set of proxies on a higher level", as captured by Figure 2.5.

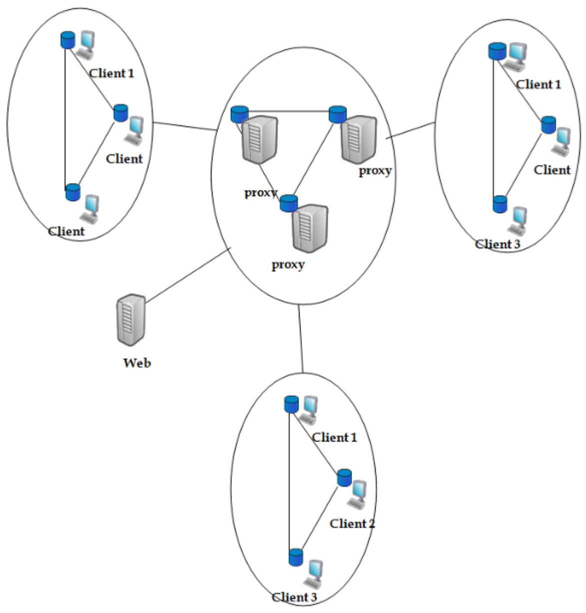


Figure 2.5: Proposed Hybrid Architecture [17]

2.2.2.4 Discussion

According to [73], hierarchical caching can be used to provide object search efficiency. This is because object metadata is distributed between caches and contains information about their location. By using intermediate caches in the network, such solutions are able to reduce the bandwidth usage and network distance to hit (i.e., find) an object.

However, they need sophisticated load balancing algorithms or powerful intermediate caches to overcome situations as high peaks of load that will cause high latencies visible to clients. This architecture can be less desirable for having redundancy of objects, by having multiple copies of the same object stored at different cache levels, as higher cache levels may turn into bottlenecks and may have long queuing delays. Distributed caching resides on traffic flowing through low network levels, which are less congested, providing better load sharing and fault tolerance. However, by only using cache at the edge of the network, administrative issues may arise. Also, large-scale deployment of this architecture may encounter some problems, such as high bandwidth usage and large network distances. A proper hybrid architecture can combine the advantages of both these alternatives, by reducing the connection and transmission time.

2.2.3 Replacement Strategies

Replacement strategies are important because they are essential to optimize caching mechanisms. Effective solutions need to adopt policies that determine the relative value of caching different objects, predicting next-request times. These decisions have to be reconciled with the type of service that is made available to customers, to provide them a better user experience. For each Web request, several different types of objects can be requested and those can influence the replacement mechanism when a caching component becomes full (i.e., has no more available space). Multiple characteristics of these objects can be considered in the design of replacement strategies, namely:

- **recency**: time since last access to the object
- **frequency**: number of past requests for the object
- **size**: size of the object
- **cost**: cost to fetch the object from the origin server (e.g. latency or number of network hops)
- **modification time**: time since last modification of the object
- **expiration time**: time when an object in the cache must be replaced (to avoid continued access to outdated versions of the object)

According to [67], a replacement process is usually guided by two watermarks: H (high) and L (low). On a cache miss (i.e., when one object is not present in the cache), the cache acquires and stores the requested object. If the size of all stored objects exceeds the cache size, some objects are removed, and if the size of all stored objects in the cache exceeds H, objects are removed until the size of all the remaining objects is bellow L.

There are different proposals of replacement strategies classifications. In this work, we follow a combined classification proposed by the authors of the referred paper, with a few adaptations: recency/frequency were moved to Combined Strategies (Section 2.2.3.11) and a size-based category was added, given in [92].

2.2.3.1 Recency Based Strategies

Recency based strategies use **recency** as the main guiding factor to select objects for replacement. Most of these strategies are an extension of the **Least Recently Used (LRU)** strategy, which discards the least recently used objects first. This strategy has two types of locality: temporal, which refers to repeated accesses to the same object in small periods of time; and spacial, which refers to access patterns where access to some objects imply access to others (considering the past pattern of access to objects). These policies take into account the time and size/cost of LRU. The rationale behind this category of replacement strategies is that recently accessed objects have more probability to be accessed in the near future.

The **LRU** strategy removes the least recently referenced object. The following strategies are some well known variants of the LRU strategy:

- **LRU-Threshold** [2]: a certain object is not cached when the size of it exceeds a given threshold, otherwise this strategy is equivalent to LRU.
- **LRU-Min** [2]: tries to minimize the number of removed/replaced documents, by setting a threshold that is the size of the requested object, and applying LRU to all objects whose size is equal to or larger than that threshold.
- **HLRU** [87]: based on the history of the number of references to a certain object. A function *hist* defines the time of the past references to a specific cached object. This policy replaces the object with the maximum hist value, and LRU is used if there are many cached objects with hist=0.
- **LRU-LSC** [41]: uses a LRU list to determine the "activity" of different objects. This value is measured by a function based on the factors: last request time, size, and cost of retrieval of the object.

The **SIZE** strategy removes the biggest object and LRU is used for objects with the same size, and the following strategies are variants of it:

- **LOG2-SIZE** [74]: results in computing the logarithm of the object size and then applying the floor function, to sort objects and differentiate them with the same value by LRU.
- **Pitkow/Reckers** [66]: objects requested on the same day are differentiated by their size and the largest files are removed first, and LRU is used for objects that were not referenced within the same period of time.

- **PSS (Pyramidal Selection Scheme)** [4]: makes a pyramidal classification based on the size of the objects (e.g. higher level has only objects with size 1, in the level bellow only objects with size 2 to 3, etc). Each level has a LRU list, and whenever there is a replacement, the object values in each level are compared.

- **Partitioned Caching** [58]: similar to PSS strategy but classifies objects into three groups: small, medium, and large. Each group has its own cache space and is independently managed using the LRU strategy.

The following strategies use time as a factor to measure the importance of the objects:

- **EXP1** [98]: LRU uses the time period between the current time and the time of the last request time for each object.

- **Value-Aging** [99]: defines a function based on the time of a new request to an object, and removes the smallest value.

2.2.3.2 Discussion

Recency based strategies are good when users desire the same Web object over the same time period, since a recently referenced object is likely to be referenced again in the near future. Most of these strategies use LRU lists, having good performance and low complexity, due to the possibility to use an implementation based on lists, where an accessed object is inserted at the head of the list and replacement takes place at the end. However, LRU does not take into account the access frequency or size of the objects. Simple LRU variants do not combine recency and size in a balanced way. In every replacement, size should be considered because typically, objects with different sizes might have different costs to be fetched from the origin and their existence in the cache takes space enough to store multiple smaller objects. The SIZE strategy is too aggressive by giving exaggerated importance to the objects size, LRU-Min also suffers from this effect in a smaller scale. A good trade-off is found in the PSS strategy, where recency and size are well balanced, as well as in a properly parametrized Partitioned Caching strategy.

2.2.3.3 Frequency Based Strategies

Frequency based strategies use **frequency** as the main decision factor when selecting objects to be removed from the cache. Most of these strategies rely on variants of the **Least Frequently Used (LFU)** strategy, that discards objects that are used least often first. Thus, this policy is based on the popularity of objects, and this results in different objects having different frequency values. This popularity reflects that most times, only a small set of objects are responsible for most of the total requests, and ensures that these objects are cached due to their regularity in being accessed.

The LFU strategy has two possible implementations: Perfect LFU and In-Cache LFU. The first one counts all requests to every object, and this counter is persistent over replacements, meaning all requests from the past are stored, which might incur in some storage overhead. The second one is different from the first just in one aspect: the counter only represents cached objects (not persistent as Perfect LFU) and therefore exhibits a lower storage overhead.

The **LFU** strategy removes the least frequently referenced object. The following strategies are some of the variants found in the literature:

- **LFU-Aging** [8]: this strategy introduces an aging effect to reduce space consumption and avoid storage overhead. If the average value of all frequency counters exceeds a threshold, these counters are divided by two.
- **LFU-DA** [8]: a problem of LFU-Aging is the dependency on the chosen parameters (threshold and maximal frequency value). This strategy recalculates these parameters whenever a request is made.
- α**-Aging** [98]: a method where a periodic aging function will decrease the value of every object to α times its original value at each virtual clock tick.
- **swLFU (Server-Weighted LFU)** [48]: uses a weighted frequency counter, and the weight for an object indicates how much the server appreciate the caching of it. LRU is used as a tie breaker on objects with the same weighted frequency value.
- **Aged-swLFU** [48]: An extension of swLFU strategy. It removes the LRU-object on every k replacement, with k=0 being the original swLFU strategy, k=1 a LRU strategy, and k>1 a mixture of recency and frequency strategies.

2.2.3.4 Discussion

Frequency based strategies are good for websites and web applications which have objects with consistent popularity and no significative changes or irregularities in access patterns. However, in comparison with LFU-based strategies, they suffer from a particular downside, which is cache pollution. This issue is characterized by having a popular object which then becomes unpopular, remaining in the cache for long periods of time without being considered for removing. This is solved with the LFU-DA strategy, through the dynamic aging technique. Although the LFU-Aging strategy includes an aging effect, it is not a good solution because it always depends on the chosen parameters which might be complex to fine tune in any arbitrary execution environment.

2.2.3.5 Size Based Strategies

Size based strategies use object **size** as the main factor when deciding which objects to evict from a cache. Objects with larger sizes are removed first to make room for multiple smaller ones. Objects are sorted by recency when they have the same size and the least recently used object with larger size is replaced by the recently requested object when the cache is full. This brings up a question: should bigger objects be removed first? If they're popular, they shouldn't. Nowadays, it's common that websites cache media files, which are typically larger than other files.

The strategies of this category were already explained on the previous categories, since they're presented in recency/frequency based policies: SIZE, LRU min, Partitioned Caching, PSS, CSS, and LRU-SP.

2.2.3.6 Discussion

These strategies work well in information-based websites that cache small files (e.g. news and articles sites) that contain mostly text and few media files. However, this strategy has poor performance for media based websites.

2.2.3.7 Function Based Strategies

Function based strategies use a particular function that computes an utility value to each object, based on their **frequency**, **size**, **cost**, and other weighting parameters. These strategies choose the object with the smallest value to be evicted from the cache first.

GD(Greedy Dual)-Size [18] is a simple operation policy: each object has a value H and at each object request this value is recalculated. The algorithm chooses the objects with the lowest H values to be replaced first, which are the objects with less cost (time or resources) or that have not been accessed for a long time. Objects with the biggest H values should stay in the cache longer. Nevertheless, there is an efficient implementation of this strategy using a priority queue, keeping an offset value for future settings of H.

The remaining strategies have GD-Size as their representative policy. For example, **Server-assisted cache replacement** [23], which proposes a caching policy that uses statistics on resource inter-request times, that can be collected either locally or at the origin server, and then forward to the proxy. They use a price function framework, that "values the utility of a unit of cache storage as a function of time". Their results show that "server knowledge of access patterns can greatly improve the effectiveness of proxy caches". Other example policies include GDSF [8], GD* [47], TSP(Taylor Series Prediction) [96], Bolot/Hoschka's strategy [14], MIX [60], HYBRID [93], LNC-R-W3 [76], LRV [72], LUV [10] and LR(Logistic Regression)-Model [35]. Many of these strategies use similar factors and weighting schemes, and choose the object with the smallest value for eviction.

2.2.3.8 Discussion

While function based strategies don't use fixed factors, considering many factors to handle different workload situations they are highly adaptive to multiple scenarios. However, it is difficult to implement a function based strategy due to its heavily parametrized nature that might require complex data structures. Additionally, complex functions might have operational overhead. Functions that consider the latency associated with fetching an object can introduce noise on replacement decisions, since they are influenced by many factors on the path between proxy servers/clients and origin servers.

2.2.3.9 Randomized Strategies

Randomized strategies use the randomness of decisions, without requiring data structures to support replacement procedures.

RAND is the simplest randomized strategy, which evicts an object from the cache randomly using a uniform distribution (all documents have the same probability of being evicted). The following policies consist of additional factors:

- **HARMONIC** [41]: where RAND uses a uniform distribution, this strategy instead evicts an object at random with a probability inversely proportional to the cost of retrieving it.

- **LRU-C, CLIMB-C, LRU-S and CLIMB-S** [83]: randomized versions of LRU policy, with assigned probabilities to evict each object depending on the cost of retrieving it.

- **RRGVF** (Randomized Replacement with General Value Functions) [69]: selects randomly N objects and evicts those that have the lowest utility value. This value is a result of a utility function and it can be selected according with the execution environment, application, or workload (e.g. based on the cost of retrieving it), since the algorithm is indifferent of the function.

2.2.3.10 Discussion

In randomized strategies, CPU and memory utilization are significantly reduced, since there is no ranking on objects and all eviction decisions are made essentially in a random fashion. These strategies avoid the use of complex data structures, further avoiding sources of overhead. Unfortunately, simple randomized policies do not have good performance, which can be solved by using utility functions to rank objects in the cache.

According to [39], the workload driven simulations have shown that to maximize the **Hit Ratio (HR)** the HARMONIC replacement policy obtains the best performance, but obtains a poor performance on **Byte Hit Ratio (BHR)**. In the BHR test, RRGVF obtains the best performance for all cache sizes. In short, HARMONIC is better for caches located near the user and RRGVF more bandwidth efficient for a proxy cache-origin server path.

2.2.3.11 Combined Strategies

Combined strategies use two or more Web object characteristics as the main factors (e.g. recency and frequency). As a result, most of them introduce additional complexity.

Recency/frequency based strategies use **recency** and **frequency**, and try to combine spatial and temporal locality. Example policies include Segmented LRU (SLRU) [9], LRU* [16], LRU-SP [21], Generational Replacement [62] and Cubic Selection Scheme (CSS) [85]). Among these strategies, only LRU* and Generational Replacement use simple LRU with frequency counts. HYPER-G [3] use **recency**, **frequency,** and **size** as factors in their replacement selection, which is an attempt to combine LRU, LFU and SIZE policies.

2.2.3.12 Discussion

These strategies, designed properly, may avoid some problems present in the individual strategies that they try to combine, effectively achieving the best of both worlds. For example, Recency/Frequency strategies avoid problems of recency based strategies and of frequency based strategies. However, they incur in additional complexity due to combination of multiple factors in their operations, which might make it extremely hard to reason about the eviction decisions taken by them.

2.2.3.13 Developing Strategies

There are several developments regarding Web caching replacement strategies. These policies are not commonly used on current caching systems. However, they represent modern ways of managing caching components that might be used in a large scale in the future. For example, **Adaptive replacement** [55] was shown to be a "self-tuning, low-overhead, scan-resistant ARC cache-replacement policy that outperforms LRU. Thus, using adaptation in a cache replacement policy can produce considerable performance improvements in modern caches". Other example is **Coordinated replacement**, the authors of [49] have observed experimentally that this strategy "significantly improve performance compared to local replacement algorithms particularly when the space of individual caches is limited compared to the universe of objects". Other examples include **Combination of cache replacement and cache coherence** [50], **Multimedia cache replacement** [32], and **Differentiated cache replacement** [67].

A more recent paper [52] has introduced a new cost-aware replacement policy called **GD-Wheel**, which is an implementation of the GD (Greedy Dual) algorithm that supports a limited range of costs. The authors of this paper describe an implementation of this strategy in the Memcached key-value store. This strategy integrates recency of access and cost of recomputation efficiently. The results of this approach demonstrates that this new replacement policy improves the performance (average latencies) of web applications.

2.2.4 Performance Metrics

Two conventional metrics can be used to compare the replacement strategies efficiency:

Hit Ratio (HR): is defined as the total number of object requests that were found in the cache (cache hits), considering the caching management policies and mechanisms (e.g. replacement strategies), divided by the total number of requests. It is appropriate to use if the objects are similar in size. This metric indicates the reduction of user-perceived latency, with an high value indicating that the cache mechanisms are effective.

Byte Hit Ratio (BHR): the ratio of the number of bytes loaded from the cache versus the total number of bytes accessed. It is appropriate to use if the objects vary in size. This metric indicates the saved bandwidth between the proxy cache and the origin servers.

Other relevant metrics that evaluate the performance of Web caching:

Bandwidth Consumption: the goal is to reduce the amount of bandwidth utilization.

Cache Server Load: CPU and disk (I/O operations) utilization, that can affect negatively end users with higher latencies.

Latency: there are several ways to measure latency: access latency for an object in a cache, end user latency, latency between two proxy servers, etc. It is hard to evaluate since this can be significantly affected by external factors to the cache (e.g. network usage). This metric can be studied through the use of a cumulative distribution function (CDF).

As pointed out, several measurements can be applied to evaluate Web caching systems. Although, it is important to realize that some of them are not very accurate due to external factors.

2.2.5 Relevant Distributed Cache Systems

Distributed cache systems [30] came to replace the traditional cache concept in a single location. A distributed cache can scale in size and might also offer transactional operations across multiple servers. Typically, this type of caching systems are used to speed up dynamic websites by relieving their database load, caching objects in the main memory of dedicated (cache) machines. This reduces the need to resort to slower accesses paths such as databases and hard disks. Nowadays, machines can have huge main memory capacity, since memory is becoming cheaper. Contrarily to conventional hard disks, main memory can potentially be boosted by high performance storage, such as flash memory. Network connection speeds are increasing in a way that bandwidth is no longer a significative problem. These observations reinforce the rationale behind distributed cache systems, since the aim is to improve caching efficiency, in order to provide faster responses to user's requests.

In these systems, objects can be replicated for achieving some combination of availability and locality-of-reference, partitioned for splitting data into sub-sets and allocating them into different machines, to consequently route the requests for the right sub-sets of machines, and invalidated to deal with object updates at the application layer.

2.2.5.1 Memcached

Memcached [34] is a free and open-source, high-performance, distributed memory object caching system, developed by Brad Fitzpatrick in 2003 for the LiveJournal website. It is an in-memory key-value store for strings and objects, resulting from database calls, API calls, or web page rendering. This system is used by high-traffic websites such as already refereed LiveJournal, a blogging and social networking system with more than 2.5 million users and running more than 70 servers; and Wikipedia, the worldwide most used free and online encyclopedia with more than 30 million users.

Its design [64] consists on a simple key-value store, in which objects are composed by a key, pre-serialized raw data, optional flags, and an expiration time. Its implementation is split between the client and server, where clients know which server they must contact in order to do read or write for a particular object. In case of connection failures, they also know how to proceed. Servers know how to store and fetch objects, and when to evict or reuse memory. There is no synchronization or other forms of communication between Memcached servers, with additional servers meaning additional available memory in the server farm. Operation complexity is very low, since they are implemented to be as fast and as lock free as possible. LRU is the default and representative replacement policy used in this system, and objects expire after a certain amount of time.

Memcached has been rewritten in C (the original implementation was in Perl) to further boost its performance. The client/server interface is simple and lightweight, with client libraries for Perl, PHP, Python, Java, C, and other programming languages. They all support object serialization using their native serialization methods.

2.2.5.2 Redis

Redis [19], is an open source, in-memory data structure store, used as non-relational database, cache, and message broker, developed by Salvatore Sanfilippo in 2009. It supports replication to scale read performance, in-memory persistence storage on disk, and client-side sharding [56] to scale write performance. Redis stores a mapping of keys to five different types of values: strings, list, sets, hashes, and sorted sets. It supports writing of its data to disk automatically in two different ways: dumping the dataset to disk every once in a while (Snapshotting), which is not very reliable due to possible system shutdowns or power fails, or by appending each command to a log (Append-only file), which is a fully-durable and reliable strategy. Also, it supports publish/subscribe, master/slave replication, and scripting (stored procedures).

Redis is written in C and works in most POSIX systems, such as Linux, *BSD and OS X. It can be used through APIs for most programming languages, such as Java, C, C++, C, PHP, among others.

2.2.5.3 Discussion

Since Redis has more features than Memcached, the best option most of the time is Redis. However, it depends on the system and environment that we want to integrate. If most content to be cached is small and static (e.g. HTML code), Memcached is potentially the best option. It is more efficient in the simplest use cases because it requires less memory resources for metadata. If most content is medium-large and/or dynamic (e.g.

video content), Redis is the best fit. It can stores several types of data natively due to data structures support, resulting in less serialization overhead. Regarding scalability, Memcached can be superior because it has a multi-threaded design. With consistent hashing it is possible to scale up without data loss. On the other hand, Redis is mostly single-threaded. It can scale by adding additional instances without loss of data, but requires more resources due to set up and operation complexity.

2.3 Recent Technologies

Recent technologies have contributed to the design of better protocols to provide efficient services in the Web. For example, HTML5 replaced Flash for reproducing video streaming, making it more efficient and lightweight to desktop and mobile Web browsers. In the following sections we discuss some recent web technologies that are relevant for this work.

2.3.1 WebRTC

WebRTC is an open-source project that makes possible for web applications to communicate directly between two browsers, without the need of installing any additional browser extension or software. This communication is possible to occur through peer-to-peer channels, without the need of using web servers as mediator to transport data. This project relies on Real-Time Communications (RTC) capabilities using appropriate APIs (e.g. Javascript API), so that developers are able to implement their own RTC web applications. It is capable of supporting high-quality communications on the web, such as audio and video chat applications, and is supported on most popular browsers, such as Chrome, Firefox and Opera, and on mobile platforms, like Android and iOS.

In the demonstration given in paper [88], they propose a peer-to-peer content distribution architecture using WebRTC Data Channels [46]. Its design is composed by a bootstrap server serving as a central instance for joining the network and the users Web browsers acting as peers. Point-to-point Data Channels allow data transfers between peers, and on top of this a protocol for joining the WebRTC network and managing user communication was implemented.

WebRTC is used to support and provide resources for the solution and prototype developed in this dissertation.

2.3.2 HTML5

HTML5 is a language for structuring and presenting content for World Wide Web (WWW), originally proposed by Opera Software [61]. It provides new features with support for the latest multimedia formats, that were only previously possible with the application of other technologies (typically external to the browser). Among all new features, some technologies were originally defined in HTML5 itself and are represented in separate specifications, such as Web Platform Working Group (WPWG) - Web Messaging, Web Workers, Web Storage, WebSocket and Server-sent events. Among them, Web Storage is a protocol that deserves special attention due to its potential usefulness in the context of the work presented in the thesis.

Web Storage is a web application software protocol used for storing data in a Web browser. Before HTML5, data had to be stored in cookies (and sent back to servers in every interaction). Web Storage supports persistent data storage, storing data locally without degrading the performance of websites, which are never transferred to the server. In most browser, it has a limit by default on storage capacity. This limit can be changed by users, but not by web applications. Using specific object abstractions, it can store data with no expiration date, and data for one session only, which is immediately lost when the Web browser is closed.

2.4 Summary

This chapter discussed previous work in the areas related to the development of this dissertation.

In the peer-to-peer context we specified degrees of centralization, and concerning to the logical network topology, described structured and unstructured overlays. These are useful to build direct browser-to-browser network to realize our caching mechanisms among clients.

In the Web caching context we described its replacement strategies, and discussed examples of distributed cache systems. Such replacement strategies will have to be employed in our cache system at the client level. Furthermore, policies in the centralized cache might need to be adapted to better leverage the cache at the edge of the system.

In the recent technologies context, some recent and commonly used technologies have been explored, which will be essential to implement our solution.

The next chapter presents our solution which consists of a framework that can be integrated into any web application, with the purpose of optimizing web caching through the exchange of resources between clients in a peer-to-peer way, directly and transparently in their browsers. As a result, our solution enhances the reduction of the vast majority of traffic imposed on the origin servers, and potentially improves end user experience.

Chapter 3

Proposed Work

In this chapter we present the proposed solution, in particular the developed framework, explaining in detail its purpose, design, and implementation. This work attempts to improve content access latency for users of small and medium scale web applications through cooperative cache between the clients themselves, and as consequence, reducing the load imposed on origin servers.

This chapter is divided as follows: Section 3.1 presents the proposal, and gives an overview of the proposed solution, showing the expected flow during its operation. Section 3.2 gives the design of the proposed framework. Section 3.3 provides details regarding the implementation of our prototype. Section 3.4 explains how the local storage is used in our implementation. Section 3.5 explains how the peer-to-peer network works in the proposed solution, how it is organized, and how resource search is executed over it.

3.1 Proposal

The proposed solution aims at developing a framework to optimize web caching between clients, running transparently in the client browser. This framework takes leverage of some components of the Legion framework, which creates a peer-to-peer network that automates creating connections and propagates updates among clients running in browsers, leveraging WebRTC to support direct browser-to-browser communication.

In a client-server model, typically each web content request and response have headers associated with it, and it's through these headers that the server can tell the client what to cache and for how long. This cache is automatically managed by the browser, usually stores these web contents in the client's local disk cache (i.e., local storage) as long as the server doesn't explicitly prevent the contents from being cached based on settings in the http headers, and, when requested multiple times in the same session are cached in memory and always fetched from there during that session. In contrast, our framework manages and takes full control of the client's local storage, using caching replacement mechanisms when necessary (i.e., cache is full), and retaining contents there to send it to other peers (when requested) in a peer-to-peer way.

This approach allows users to transparently share content directly among them, without the need of installing any kind of software, browser extensions or plugins, which favors the adoption of this solution.

Figure 3.1 shows an overview of the operations flow that can occur during execution of the proposed solution. A web request can be satisfied, following this priority list of execution, as follows:

1. by the user's own local storage, or;
2. by requesting the content to a (nearby) peer that has the content in its own local storage, or;
3. by a cache server (e.g., Memcached), or;
4. by a persistent storage server (i.e., record store of a database system).

Regarding the scheme depicted bellow, whenever an object is found in a key-value store (cache hit), it is first sent to the application/framework and only then sent to the key-value store that does not yet have it (if any). This favors a quick response to the client, by updating caches outside the critical path of the reply for the client.

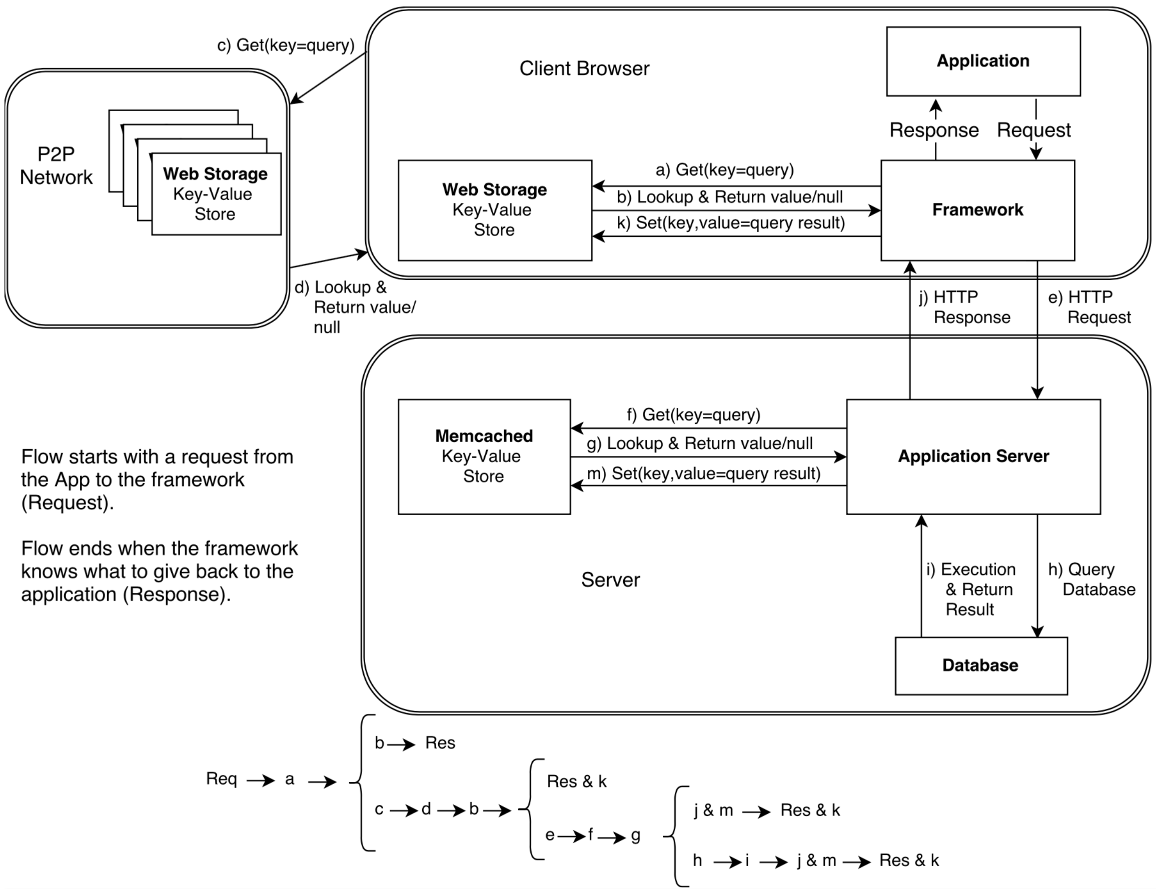

c) Get(key=query)
P2P Network
Web Storage Key-Value Store
d) Lookup & Return value/ null
Client Browser
Application
Response
Request
Web Storage Key-Value Store
a) Get(key=query)
b) Lookup & Return value/null
k) Set(key,value=query result)
Framework
j) HTTP Response
e) HTTP Request
Memcached Key-Value Store
f) Get(key=query)
g) Lookup & Return value/null
m) Set(key,value=query result)
Application Server
Flow starts with a request from the App to the framework (Request).
Flow ends when the framework knows what to give back to the application (Response).
Server
i) Execution & Return Result
h) Query Database
Database
Req → a →
b → Res
c → d → b →
Res & k
e → f → g
j & m → Res & k
h → i → j & m → Res & k

Figure 3.1: Design overview for the proposed solution

3.2 Design

The proposed framework was developed in Javascript to be easily integrated in the front-end of web applications. It can be divided into two parts: one part that is implemented at the web application level and the other part a web layer over it. In other words, the upper layer (parent layer) is a static web page for all visitors, while the lower layer (child layer) will change state while browsing. The parent layer contains the base elements of a website: header and body, optionally a footer. As shown in Figure 3.2, the header contains site metadata elements and the import of required scripts, such as jQuery and the Legion framework scripts. The body contains the iframe that points to the remote URL of the web application. The developed application is integrated into the web application, located in the footer, in order to be loaded after all elements present in the DOM are loaded. Returning to the body of the parent layer, after the iframe it contains code associated with client settings of the Legion framework, where message events enable communication with other peers and consequently communication between web layers, that is, between the Legion framework and our application (inside the iframe).

These two layers can communicate while users navigate through the web application so that P2P connections are not lost within the navigation window. This is necessary for

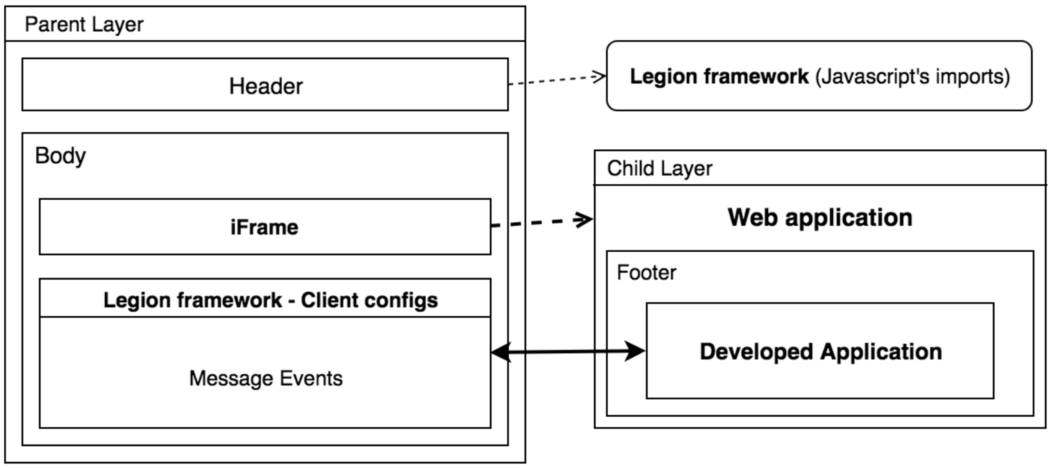


Figure 3.2: Web layers representation of the proposed framework

users to be able to share resources with each other while browsing the web application, taking advantage of the storage of these resources in their local disk cache (local storage). Exceptionally, there is only one way that the two layers can no longer communicate or lose their connection, which is the user forcing a page refresh, outside the web application's navigation space. In Figure 3.2, we show a simplified version of these web layers.

3.3 Implementation

Our implementation focuses on the concrete case of images, by caching and sharing this type of web content. Note however that we believe that this work can be generalized easily for other web content type, such as HTML code, CSS files, JSON files, etc.

As it is shown in Figure 3.3, on average images occupy over 60% of the total weight of web pages. When multiple images requests occur to origin servers, the load imposed on them is significant. Beyond this, images are typically static content on websites, which are naturally located in common parts of all of these: headers, footers, and other locations in their layout. There are other situations where images do not change very often, such as in users profile pages where we can visualize their photos/avatars, and in pages like FAQs, where instructional images do not change often. Anyway, our framework can be integrated into any web application, and works with all image formats, such as png, jpg, svg, gif, and others. However, the default limit of local storage's capacity in most browsers is reduced, not allowing to store too much of these content.

During the implementation of this framework some challenges have arisen. One of them was the way browsers render websites. In particular, it is not feasible to change the elements present in the DOM of the website before the DOM is processed by the browser, that is, to somehow get a script to trigger DOM manipulation mechanisms before the browser itself does so. The DOMContentLoaded event fires as soon as the DOM hierarchy has been fully constructed, the scripts will load when all the images and sub-frames have

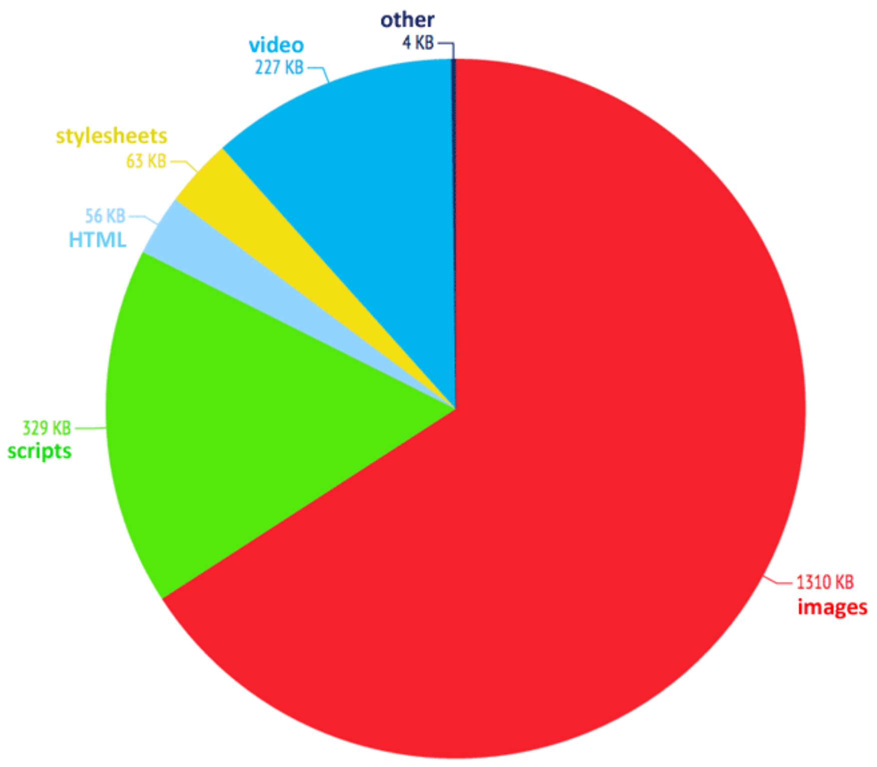


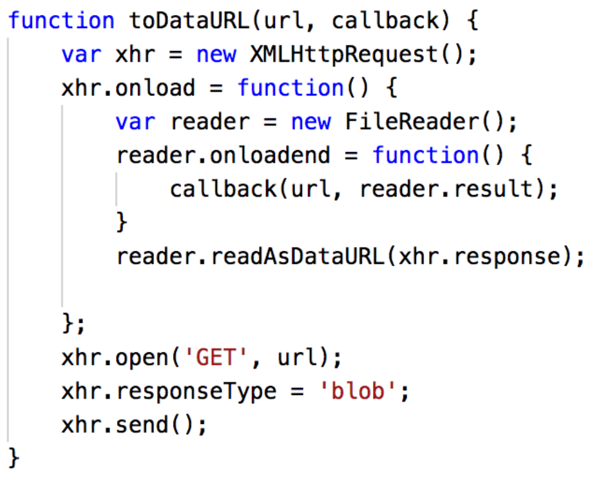
Figure 3.3: Comparison of the average weight of elements in web pages in 2015 [1]

finished loading. Taking into account the focus of this work, which are the images, this causes all images to be rendered before we can change their properties and update the DOM. The problem is that we do not want all the images to be requested directly from the source server of that website (through GET requests), but rather that we can retain them and manipulate their source before being rendered.

To work around this problem, small changes had to be made to the frontend of the web application, in particular across all image HTML tags, which the default name is **<img>**, were changed to a custom HTML tag, that we named **<notcachedimg>**. This custom tag adopted by us ensures that browsers are not able to identify this tag as an image tag, and therefore, do not request its contents to the origin server and immediately render it at the first contact with the DOM. Now we can ensure that only our framework knows that this tag is an image tag. In other words, the browser renders all the HTML tags it knows, the standard HTML tags, and the ones it doesn't know are then processed by our framework.

Given this step, the next question is whether the images to be loaded are in the user's local storage or in the local storage of other users, active in the P2P network. How to find the resources on the network is explained later. Requests that could not be satisfied in these steps will be routed to the origin server or a cache server, depending on the configuration of the web application in question.

The last step before "allowing" the browser to render the images is to convert our custom tag to the standard tag, thus making it recognized by the browser. The difference is that now the source path present in each image element in HTML will be different. This source path is a local URL (or string). This local URL can be obtained, as already mentioned, through an active (peer) user in the network, or through the origin server, carried out by the encoding of the image to **Base64**. As it is shown in Figure 3.4, the image is loaded as a binary large object (blob) via a XMLHttpRequest, which enables a Web page to update just part of a page without disrupting what the user is doing, and then use the FileReader API to convert it to a **data URL** in Base64, which is a string. The result is stored in a key-value store (i.e., local storage), where the new entry contains the image URL as key, and its data URL as value.

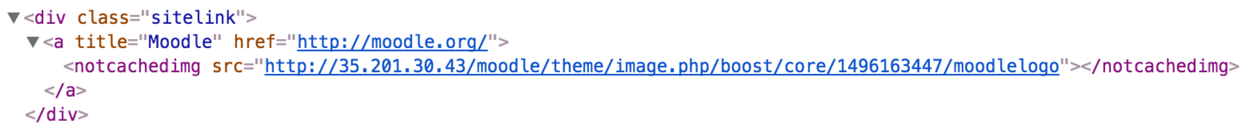


```
function toDataURL(url, callback) {
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
        var reader = new FileReader();
        reader.onloadend = function() {
            callback(url, reader.result);
        }
        reader.readAsDataURL(xhr.response);

    };
    xhr.open('GET', url);
    xhr.responseType = 'blob';
    xhr.send();
}
```

Figure 3.4: Javascript function that converts an image located remotely to data URL

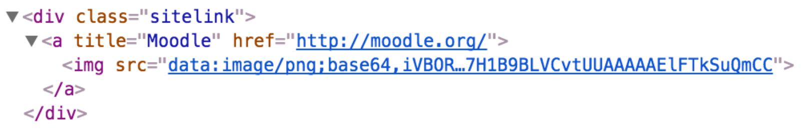
An example of this approach follows below, where we can visualize a part of the DOM in the form of HTML that has the logo image tag of the Moodle web application. Moodle is going to be used as web application in our experimental evaluation, as we will discuss later in this dissertation. It is shown in Figure 3.5, at the time when the browser has not yet loaded our framework, the source code showing the DOM presents the custom HTML tag of the image and its source path points to the origin server, and then in Figure 3.6, at the time when our framework had already been loaded, presenting the standard image HTML tag and its source path points to the Base64 string (or data URL).



```
▼<div class="sitelink">
  ▼<a title="Moodle" href="http://moodle.org/">
      <notcachedimg src="http://35.201.30.43/moodle/theme/image.php/boost/core/1496163447/moodlelogo"></notcachedimg>
    </a>
  </div>
```

Figure 3.5: Before proposed framework has been loaded

To automate the change of these HTML image tags in every web page of the application, we developed a small program that search them (i.e., <img src = "...">) and replaces



```
▼<div class="sitelink">
  ▼<a title="Moodle" href="http://moodle.org/">
     <img src="data:image/png;base64,iVBOR…7H1B9BLVCvtUUAAAAAElFTkSuQmCC">
   </a>
  </div>
```

Figure 3.6: After proposed framework has been loaded

with the custom tag (i.e., <notcachedimg src = "...">). In order to run this program, it is only necessary to indicate which web application files have these HTML tags, which typically are the ones that represent the views in a Model–view–controller (MVC), to carry out the replacements. This way we allow easy automation and preparation of integration of the proposed solution in typical web applications.

3.4 Local Storage

As previously mentioned, this solution uses the browser's local storage to store content present on the application's web pages and communicates with the Legion framework to make it available to other users in the P2P network. Each user decides whether or not to store the content in an almost random way. This is because we want to allocate different content among the peers, in order to make nodes in the network useful to each other but without incurring in high coordination overhead among the multiple clients. This randomness consists of the result of generating a random number between 0 and 1, where a custom random seed [12] is previously generated. This seed consists of the concatenation of two parameters related to the received content and also can be related to the user information. If the content is obtained within the **P2P network**, the parameters are: content key and user identification (i.e., *imageURL* + *peerID*), and the probability of being stored is 30%, which the random number must be between 0 and 0.3; if obtained directly from the **origin server**, the parameters are the *content key* and the current time (i.e., *imageURL* + *timestamp*), and the probability of being stored in local storage is 90%, so the random number must be between 0 and 0.9, which is very likely to happen.

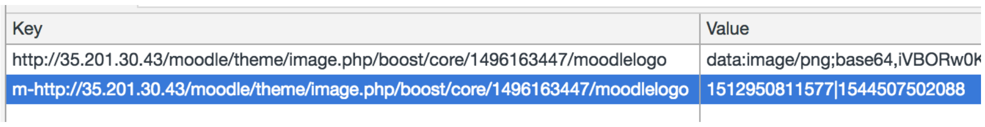
These parameters can be changed according to the context of the application that is intended for the proposed framework. We give these values to the stated probabilities because we want to prioritize content that is downloaded from the server to avoid going to the server requesting for the same content. This way, retaining as many requests as possible among users within the network, potentially leaving a few remaining requests to cross the client-server frontier.

Local storage capacity varies from browser to browser, typically the default value is 10 megabytes per origin (per domain and protocol), and it can be changed by users in a few browsers. Since 10MB is a low threshold for the needs of many small and medium-scale web applications, we had to think of a way to manage new content entries in the local storage. One way to address this is to remove content that is less useful at the moment for a specific user.

3.4.1 Adopted Cache Replacement Strategy

As detailed in Section 2.2.3, there are many policies or strategies for caching replacement. In this work, we chose to use a recency based strategy (see Section 2.2.3.1). For the proposed solution, we used **LRU strategy** which removes the least recently referenced object.

To implement this strategy we had to have some place to store this factor associated with the objects. For this, whenever we store an image into local storage, its metadata is also stored into it (see Figure 3.7), which contains its recency value, and time-to-live (TTL) value, in the respective order. The value of recency is a timestamp in milliseconds, when updated its value is replaced by a current timestamp. The value of TTL is also a timestamp, when updated its value is replaced by the result of a current timestamp plus a timestamp of how long we want the content to be cached (e.g. current timestamp + one year timestamp). This metadata is also stored into a data structure in memory, for faster readings access during a session. When a session starts, each image metadata is loaded into memory. Every time an already cached image is accessed, its metadata is updated in memory and on disk, keeping information persistent.



Key	Value
http://35.201.30.43/moodle/theme/image.php/boost/core/1496163447/moodlelogo	data:image/png;base64,iVBORw0K
m-http://35.201.30.43/moodle/theme/image.php/boost/core/1496163447/moodlelogo	1512950811577|1544507502088

Figure 3.7: Visualization of the Moodle logo's entry and its metadata entry (that is selected) in the browser's local storage

3.5 Peer-to-peer network

As already pointed in this chapter, the P2P network is managed and automated by the Legion's framework components. Whenever a user wants to join the network, it is first redirected to the server where Legion centralized component is executing. Then it is connected to other peers according to its latency to a certain end-point. This way, the network is organized into user groups that are close to each other, leveraging fast communications between them and potentially reducing end-user latency when accessing cached contents in peers through the Legion overlay network.

Legion framework uses a small API called **HTTP Pinger** [43] to measure each peer's average latency to a certain end-point, in order to get an approximate notion of relative proximity between peers. If the difference of these peers average latencies to a certain end-point is equal to or less than a certain interval (e.g. 50 ms), the Legion framework assumes that peers are close to each other and can establish a connection. This latency interval can be configured as desired by the web application owner. This way, it can be tuned depending on the number of users that are accessing the web application on a global scale. For example, considering only users located in Europe, if there are many of them concentrated in Western Europe then a low latency interval is effective as there will be more good matches between them. That is, the probability of improving their access to the web application will be higher because the latency between them is likely to be low when sharing resources. On the other hand, if users are more widely spread in the world, a low latency interval will cause them to fail to connect, and a high latency interval may cause web application access less efficient since latencies between them are likely to be higher. In turn, it might be more rewarding for them to connect directly to the server instead of fetching resources from the P2P network.

An example of this approach is shown in Figure 3.8, where 3 peers ping an HTTP end-point. Peers P1 and P2 will establish a connection, since the latency interval between them is about 36ms, so less than 50ms. The peer P3 is relatively far away from the end-point, so it will not connect to P1 or P2.

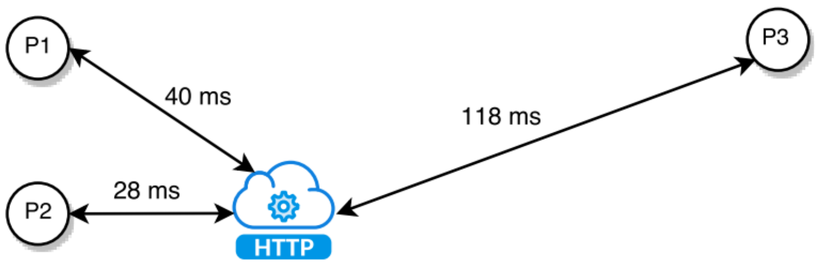


Figure 3.8: Example of peers pinging an end-point (end-point icon downloaded from [42])

Therefore, peers within the P2P network are essentially organized by their distance. Also, it is possible to configure some relevant aspects related to their close (neighbors) and far peers. These belongs to the options of the Legion overlay protocol:

- **MIN_CLOSE_NODES** - the minimum number of close peers;
- **MAX_CLOSE_NODES** - the maximum number of close peers;
- **CLOSE_NODES_TIMER** - time that takes for the network to update the peers connected to a certain peer in case the number of close peers is less than the minimum or greater than the maximum number of close peers. This avoids unnecessary fluctuations on the overlay due to the transit arrival and departure of a node.

A similiar set of parameters is used to take the management of distant peers, which are **MIN_FAR_NODES**, **MAX_FAR_NODES** and **FAR_NODES_TIMER**.

These overlay protocol parameters are relevant to the performance of the providing service. These must be suited to the dimension of the system where it is operating, taking into account its potential scalability.

3.5.1 Search

The way resources are located within the network plays an heavy role in the overall performance of a P2P service. In the proposed solution we use **Bloom Filters** [13] to know what peers have in their local cache, thus peers search is not performed in a random way. This technique allows peers to search for resources by acquiring previous feedback from their nearby peers (i.e., neighbours) and thus be able to send requests directly to them, through the communication module of the Legion framework that allows point-to-point communications.

As said, bloom filters are used to get a peripheral view of user content over the peer-to-peer network, which results in the use of a space-efficient probabilistic data structure, for instance, to know if a peer (potentially) has a certain web object, or definitely does not. This reduces complexity in searches and potentially improve end user perceived latency.

3.5.2 Bloom filters

In the proposed framework, we used a simple implementation [27] of bloom filters, developed by Jason Davies. This implementation uses the non-cryptographic *Fowler–Noll–Vo* hash function [36] for speed, being appropriate in this solution context, where we only care about the uniform distribution of hashes.

In this implementation, the basic bloom filter supports two operations: **test** and **add**. Test operation is used to check whether a given element is in the set or not. If it returns false then the element is definitely not in the set, if it returns true then the element is potentially in the set. Add operation simply adds an element to the set. The false positive rate is a function of the bloom filter's size and the number and independence of the hash functions used. The bloom filter consists of a **bit vector of length m**, and when adding an item to the bloom filter, it is fed to **k different hash functions** and set the bits at the positions denoted by the result of each hash function. Sometimes the hash functions introduce overlapping positions, so less than k positions may be set (i.e., if it had already been set by a previous element added to the bloom filter).

In Figure 3.9, it is shown an example of element testing in a bloom filter containing 10 elements, where it checks if *image3* is in the set. The result is true since it may be in one of the three buckets of the bloom filter.

In our system, users exchange information (bloom filters) to optimize and possibly reduce the number of future requests among them. For example, if a given peer needs 10 specific web objects, it is not reasonable to place requests containing these 10 objects identifiers for all his neighbors. Most of them may only have one or two, or may not have any of those objects available locally. In the worst case, it would not even be necessary to send any message or request to them. This shows that the use of bloom filters can be advantageous when applied to this framework.

Each peer's bloom filter is always updated when new objects are stored in his local storage. This update consists on removing the current bloom filter and creating another

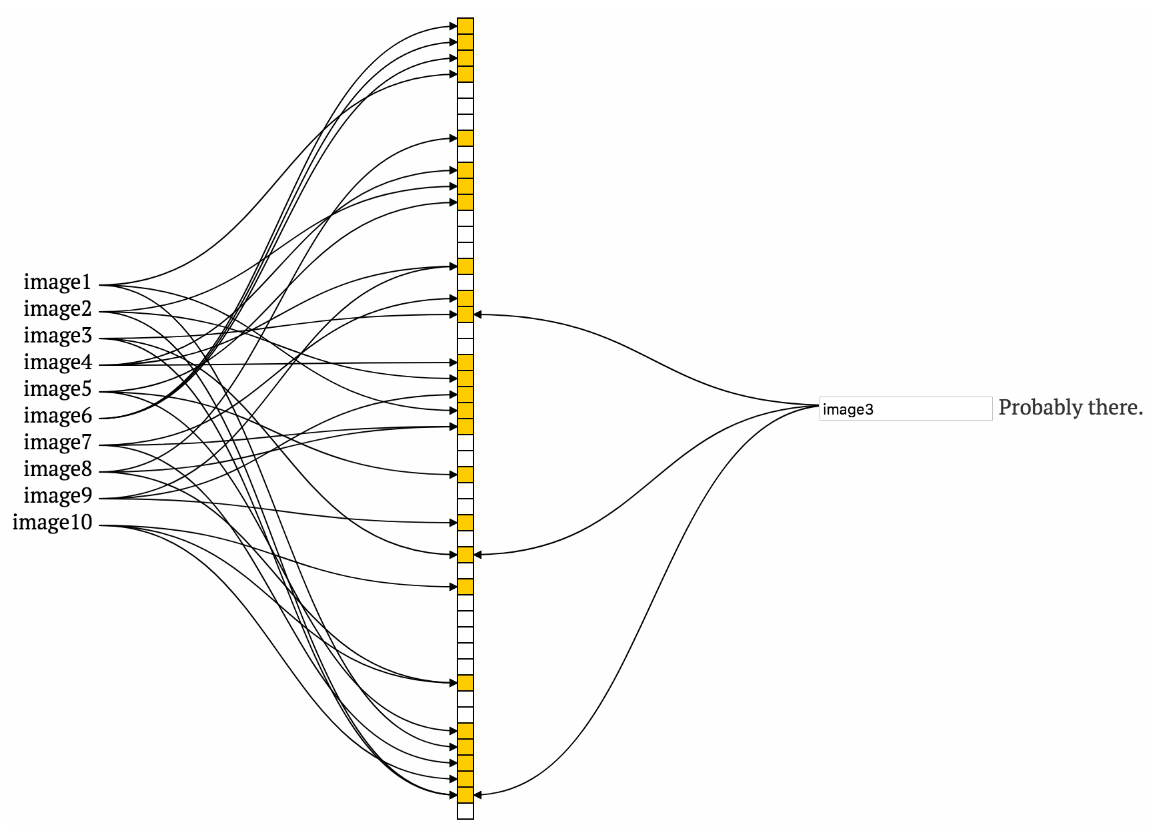


Figure 3.9: Visualization of an element testing in a bloom filter (used online simulator at [27])

.

one, adding all the current objects present in his local storage. This is necessary because during browsing certain objects can be replaced by others, and since it is not possible to remove elements from bloom filters (though this can be addressed with a "counting" filter technique) each peer's bloom filter must be re-created. As we'll see later, not all downloaded objects are stored in each peer's local storage. After updating his bloom filter, he sends it to all of his neighbors, and these neighbors will send it to their neighbors, ending this propagation within the network (i.e., bloom filters are only propagated with an horizon of two hops).

This approach allows peers to maintain updated information of their neighbors and also the neighbors of their neighbors. A peer will only send requests to peers which bloom filter validates the presence of at least one of his desired web objects. This way, bloom filters can reduce potential network overhead and clearly solves the problem of needless requests among peers who do not match with desired web objects. As it is shown in Figure 3.10, peer P1 sends his updated bloom filter to all of his neighbours (P2, P3 and P4), then P2 and P3 send it to all of their neighbours, ending there P1's bloom filter propagation.

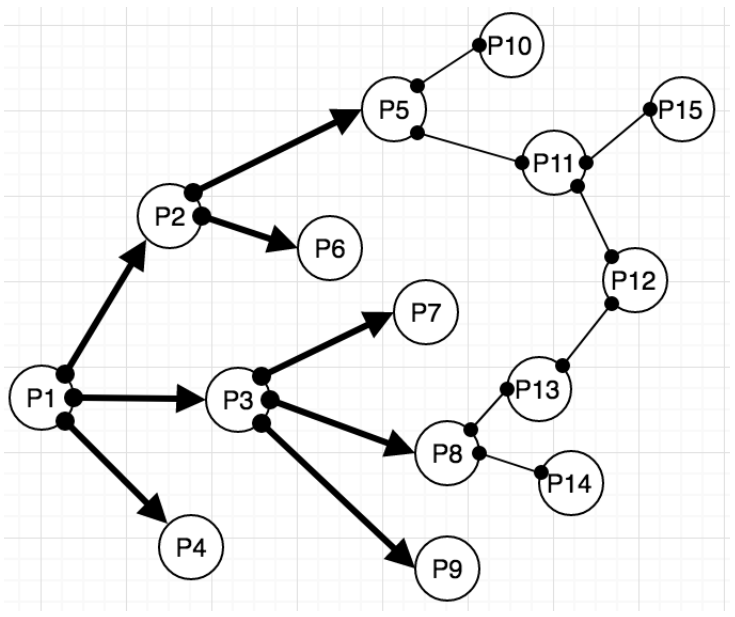


Figure 3.10: Example of a peer's bloom filter propagation

3.6 Summary

This chapter presented and explained the proposed solution, in particular the design and implementation of the proposed framework.

In the proposal context we explained the purpose and characteristics of our solution, and gave an overview of it with the expected flow during its operation.

In the proposed framework context we presented its design, explaining how our framework can be integrated into any web application, explaining how it interacts with the Legion framework in order to leverage peer-to-peer communications. Following this, we explained in detail how the proposed framework was implemented in this work, pointing out all the challenges encountered during its entire process.

In the local storage context we explained how it is used and leveraged by our framework, in particular the decisions we take to cache content which depends on where the content comes from (i.e., P2P network or origin server), and described the caching mechanism adopted to evict contents from the local storage in case it reaches its storage capacity limit.

In the peer-to-peer context we explained how peers connect to each other based on the relative distance between them, how resource searching is performed within the network, and how bloom filters are implemented and improve the process of obtaining resources from other peers.

The next chapter presents the results and evaluation of this work, the setup environment in which the tests were run, which consists of a system with three type of nodes: clients, the Legion server and the origin server (that hosts the web application). It describes the chosen web application, how clients are simulated in the experimental tests, the test conditions and used parameters. It also presents and explains the metrics used to evaluate the proposed solution.

Chapter 4

Evaluation and Results

This chapter presents the evaluation and validation of the proposed solution. The performed evaluation focuses on the end user experience and on the load imposed on origin servers. We verify our optimizations on web caching through resource sharing in a peer-to-peer fashion.

This chapter is divided as follows: Section 4.1 describes all the setup environment where the tests were run. Section 4.2 gives and explains the used parameters in the experimental tests. Section 4.3 provides and explains the used metrics to evaluate our solution and also the alternative one. Section 4.4 describes the test conditions, how were they categorized/divided. In the following sections are the results obtained in each of the test conditions.

4.1 Setup test

Experiments were performed in a system composed by three types of nodes: clients, the Legion server, and the origin server. In this context, we refer to clients as multiple automated browser instances running on virtual machines globally distributed (in our setting we chose Europe and United States), simulating users navigating on their browsers into a specific web application. This machines were provided by Microsoft Azure Cloud Services. We refer to Legion server as a virtual machine that hosts the Legion centralized component, located in an European data center, also provided by Azure. We refer to origin server as a virtual machine that hosts the web application, located in an Australian data center provided by Google Cloud Services, as we wanted clients to be as far away as possible from the origin server. To measure if, in fact, it is more efficient for clients who are nearby to share resources between them than go directly to the server that is more distant to get those resources.

These three elements are the main elements in the system but we have notion that there are other factors that can influence the results of the performed tests, such as network usage (see Section 2.2.4). In order to test our proposed solution and the currently commonly used solution for comparison purposes, we needed to set up an equal environment for both. In our experiments we rely on a simple client-server architecture as baseline to access the of benefits of using our solution.

Since multiple client instances needed to be executed at the same time on each virtual machine, it was necessary that virtual machines had enough computing power to avoid loss of performance or running out of memory, while executing multiple automated browser instances in parallel, this way we ensure that our evaluation is conducted in a fair manner to both settings. Thus, we allocated a few virtual machines for clients, with 20 CPU cores and 64GB of RAM each. For the Legion centralized component, we chose a virtual machine with 8 CPU cores and 32GB to host it. Regarding the server that hosts the web application, we chose a virtual machine with little computing power (with 1 shared vCPU and only 1.7GB of RAM), since we wanted at one point to quickly exhaust the server resources, so as to be able to compare the solutions in an overloaded state. This in some sense captures the runtime environment of a small web application.

4.1.1 Web application

It was important to choose an easy to setup web application, which allowed us to change its frontend code in a practical and clear way, to integrate the mechanisms of the developed framework within. We tried to install the Reddit web application and integrate our solution into it, but there were some problems related to its setup. This is because we have not been able to run this application in a remote domain, this being a recurring issue identified by many users over the Internet. This is due to Reddit using a proxy (HAProxy) at the front of the application, which is difficult to configure to work properly with a non-local domain. Hence, we chose the web application **Moodle** [37], which was easy to setup and has an easy to understand implementation.

Moodle is a web application widely used for academic purposes and in several situations its web pages contains multimedia contents, such as images. This way, we think this web application is suitable for evaluation purposes of the proposed solution. For testing purposes, it is relevant to try to make client behavior similar to the actual behavior of users that regularly access Moodle in their universities. Moodle also brings multiple ready-to-use modules of in-memory cache systems, such as Memcached. This way, it is easy to setup the backend with a cache server operating in front of the origin server, potentially decreasing page delivery latency when requests have to be served by the central component of the application.

To install it in the available virtual machine, we used Apache as web server and PostgreSQL as database server, since this is the required setup indicated in Moodle's installation guide. As described in Section 3.1, for the proposed solution we had to slightly modify parts of the Moodle's frontend code in order to integrate the developed framework. We needed to test the proposed solution within the adapted Moodle web application and compare to an existing solution. Thus, we chose as existing solution the one that does not use a cooperative cache between users in a P2P fashion, that is, the base Moodle web application without any code changes and without our integrated framework. Both solutions were hosted in the same virtual machine, so that comparisons between them could be reliable, as both run in the exact same setting.

4.1.2 Clients

The client side plays a key role in evaluating the proposed solution, since the proposed framework is integrated into the application's web pages and loaded transparently in the client's browser.

For the proposed solution, tests relied on two types of clients: **seeders** and **leechers**. The seeders are the clients that run first and get some content of the web pages and possibly cache them into their local storage, and remain active. Later, the leechers run concurrently and join the P2P network, communicating with these seeders, in order to obtain at least some content from them. This increases the likelihood that much of the traffic passes through the clients (leechers), potentially reducing the burden imposed on the origin server. So, leechers are the clients which we will focus on and evaluate.

For all tests, we decided that **10% of the number of leechers are seeders**. For example, for every 30 leechers that will run concurrently, 3 seeders will provide them some resources. This means that on average, 1 seeder can send resources to 10 leechers at most. We want the seeders to be peers that can satisfy many leechers, that is, could be a kind of "super peers". Obviously, there is a probability of leechers connecting to other leechers, and it is not possible to have guarantees that they are only connecting to seeders or to more seeders than leechers. Note however, that a leecher access content they can start to cache and serve that content themselves. This translates into what can happen in a real world scenario.

Each experiment boils down to obtaining metrics directly related to client (leecher) actions on the web application pages, that is, monitoring their navigation. This navigation has been defined by us, which includes visiting/opening 10 pages in total since the client starts and ends their navigation in the web application, as shown in Figure 4.1. Therefore, this navigation starts on the home page, logs into the application and goes through several pages, including "Random" pages, until the user logs out of the application. A Random page is a page chosen (randomly) from 3 possible pages: Panel, Calendar, and Private Files. We decided to use this to introduce some randomness in the middle of the clients' navigation, so that the requests are not exactly equal to each user.

When this navigation sequence starts, we set a waiting time of approximately 5 seconds in the initial page, so that clients have time to connect, in order to share resources among them. This is a limitation imposed by the Legion prototype, but in any case it is not necessary in a real setting because even with clients entering the application while some are in the middle of the navigation sequence, it will not impact others and still manage to provide resources.

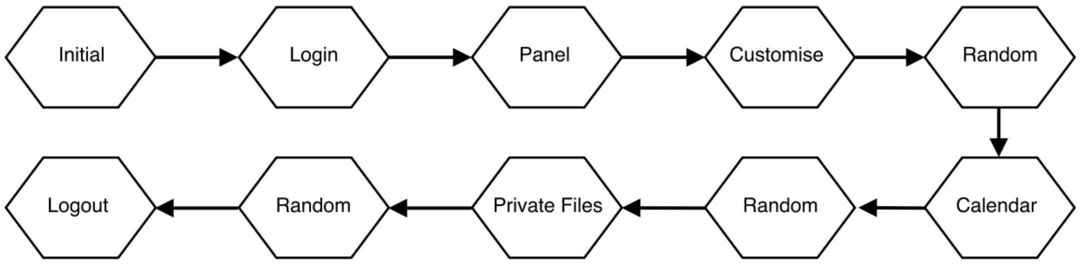


Figure 4.1: Representation of the navigation sequence executed by every client instance.

To automate a client's navigation through the web application, we developed a Java program which uses a software-testing framework that automate browsers, called **Selenium** [77]. This software allowed us to define an automated navigation sequence within web application Moodle. Since Mozilla Firefox supports Selenium and WebRTC connections, required for P2P connections to work leveraging the Legion framework, we ended up choosing Firefox as web browser for the evaluation. This browser also has a headless mode available, in order to run without its graphic interface running within the virtual machines. In addition to Selenium, a driver was used to control the UI of the web application: **Marionette**. This driver can remotely control either the UI or the internal JavaScript of a Gecko platform, such as **Mozilla Firefox**. As shown in Figure 4.2, Selenium uses the W3C Webdriver protocol to send requests to **Geckodriver**, which translates them and uses the Marionette protocol to send them to Firefox.

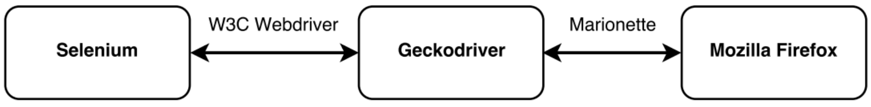
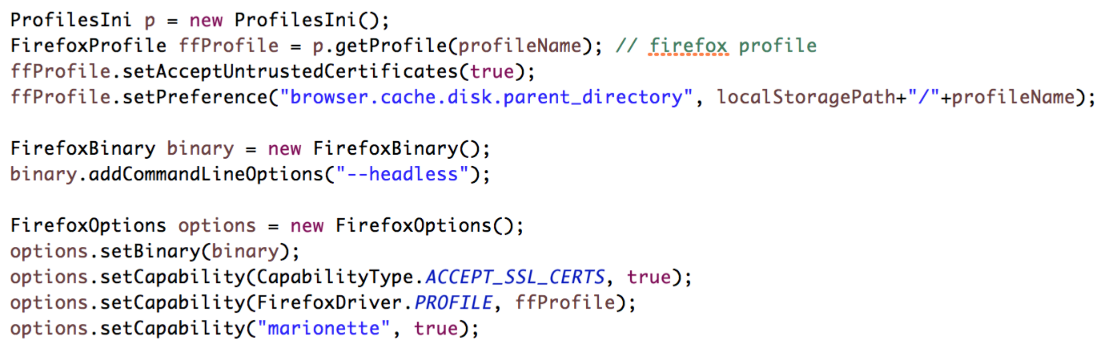


Figure 4.2: Representation of the interactions between Selenium, Geckodriver, and Firefox

In Figure 4.3, we can visualize a piece of code of the developed program. This code shows some aspects of Firefox settings, namely the association of a Firefox profile to be loaded when this browser is started, and the local storage's parent directory. This is due to each instance of Firefox being able to cache content in its own directory. In this way, each client instance loads a Firefox profile with a specific path for the local storage, not allowing resource sharing between Firefox instances running at the same time on the same machine. We want each instance to correspond to an independent client, as if it were real-life users, accessing from their personal computers in their homes or elsewhere. In the mentioned figure, we can also observe the addition of certain capabilities in order to allow the browser to run the web application features, as well as bypass unsafe certificates in test mode (this does not disable security but enables us to run our evaluation without acquiring certificates), and enable the headless mode.



```java
ProfilesIni p = new ProfilesIni();
FirefoxProfile ffProfile = p.getProfile(profileName); // firefox profile
ffProfile.setAcceptUntrustedCertificates(true);
ffProfile.setPreference("browser.cache.disk.parent_directory", localStoragePath+"/"+profileName);

FirefoxBinary binary = new FirefoxBinary();
binary.addCommandLineOptions("--headless");

FirefoxOptions options = new FirefoxOptions();
options.setBinary(binary);
options.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);
options.setCapability(FirefoxDriver.PROFILE, ffProfile);
options.setCapability("marionette", true);
```

Figure 4.3: Visualization of a piece of code from the developed Java program

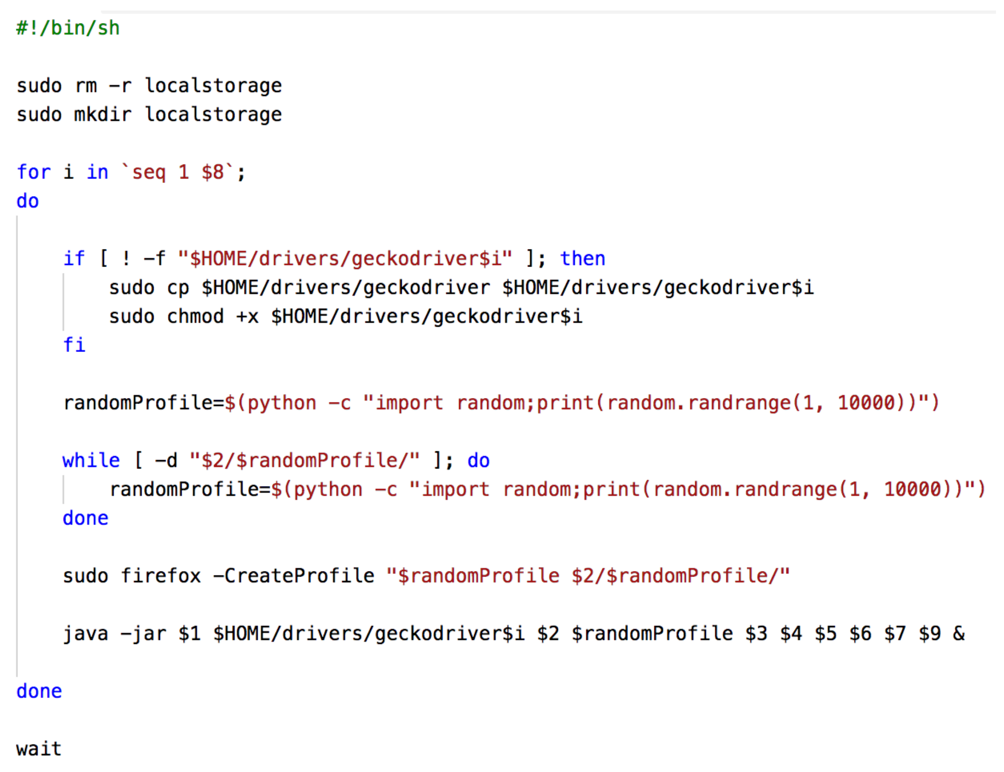
In order to paralize multiple instances of this program in a virtual machine, we developed a small script in Bash to generate multiple processes by the operating system instead of using Java's threads management for this purpose. We tested these two options and in fact we noticed that the parallelization using the Bash script is capable to manage system resources much better than using Java. As we can observe in Figure 4.4, firstly a new directory is created for storing the cached content, and for each instance a Firefox profile is created and the developed Java program (with Browser/Selenium actions) is executed, in parallel with the others. Also, as we can see in the code, it is generated a specific driver (Geckodriver) for each Selenium instance to use, this is because we encountered problems running when we shared the same driver between different instances of browsers and Selenium.

4.2 Parameters

Our solution, which integrates the Legion framework, has several parameters which are relevant for a given application:

End-point address(es): a parameter that defines the address(es) of the end-point(s) to which the peers will ping, and thus the Legion centralized component is able to obtain the relative distance between them. We used an end-point of httpbin, which the address is https://eu.httpbin.org.

Maximum latency interval for connecting peers: a parameter that defines the proximity that the connected peers will have (explained in Section 3.5). If it is a very low value the Legion framework will only connect peers that are very close, if it is too high it will connect the majority of the peers present in the network. We chose 50ms for this parameter, since we want only users from the same continent to connect among them, potentially avoiding high latencies between them in data transfers. The smaller this value, the more likely the data transferred between them to take less time, but on the other hand it can



```
#!/bin/sh

sudo rm -r localstorage
sudo mkdir localstorage

for i in `seq 1 $8`;
do

    if [ ! -f "$HOME/drivers/geckodriver$i" ]; then
        sudo cp $HOME/drivers/geckodriver $HOME/drivers/geckodriver$i
        sudo chmod +x $HOME/drivers/geckodriver$i
    fi

    randomProfile=$(python -c "import random;print(random.randrange(1, 10000))")

    while [ -d "$2/$randomProfile/" ]; do
        randomProfile=$(python -c "import random;print(random.randrange(1, 10000))")
    done

    sudo firefox -CreateProfile "$randomProfile $2/$randomProfile/"

    java -jar $1 $HOME/drivers/geckodriver$i $2 $randomProfile $3 $4 $5 $6 $7 $9 &

done

wait
```

Figure 4.4: Visualization of the developed Bash script

cause many peers to not be able to connect to anyone, which would render our framework ineffective.

Reset timer to perform peer search: a parameter that sets the wait time until it performs a restart on peer searches within the network. We chose 15 seconds for close peers and 55 seconds for distant peers. This are default values in the Legion overlay settings. A small value forces the network management to perform peer searches more often, which may cause unnecessary network traffic.

Maximum and minimum number of close and far peers for each peer: a parameter that defines the range of number of peers close and far, for each peer in the network. For close peers we chose 10 as maximum and 3 as minimum, this means that if the number of close peers is between this range the Legion network will not attempt to connect to more when the reset timer to perform peer search fires. For distant peers we chose 0 for minimum and maximum, since we don't want peers that are far from each other to connect (we don't want to wait for the latency to traverse long distances and this could have a negative impact on the user experience, which would make a centralized system probably faster in this case).

Timeout for P2P network responses after requests are made: a parameter that defines the window of time that our framework waits for the resources coming from the P2P network. It can be tuned according to number of neighbors that a peer has at the moment, including not only his closest neighbors but also the neighbors of his closest neighbors (two layers of neighbors, as explained in Section 3.5.2), and/or the number of images to be requested. We set the following values for this parameter: 0.5 seconds when the number of images to be requested is bellow five, one second when the number of images is between six and fifteen, and a maximum of two seconds when the number of images is above fifteen. What happens is that the framework after each request continually checks, every 50 ms, the current state of the incoming answers, and if all requests are satisfied or all neighbors have already answered before the timeout has ended, it will stop the wait and if possible load the respective images. If there is no neighbor or neighbor's neighbor that have the content that has been requested, no request messages are sent to them and all the requests are forwarded directly to the origin server. This is possible by checking the neighbor's bloom filters (detailed in Section 3.5.2), preventing overloading the P2P network with non-useful messages.

Bloom filter instantiation: a parameter that defines how the bloom filter of each peer is created, which requires per application tuning of the total amount of content present in its web pages in order to maximize its search and space efficiency. It can be set by two parameters: m and k, as explained in Section 3.5.2. We used an online calculator [44] to find the appropriate values for these parameters, that given the maximum number of elements that can be inserted into the bloom filter and the probability of false positives that can occur, it gives the correct parameters for m and k. This enabled us to set up a generic bloom filter based on the total number of unique images that will be displayed on web pages when our solution is tested. As we will see later in this chapter, for the 94-images tests the resulting values were m = 2703 and k = 20, and for the 144-images tests the resulting values were m = 4141 and k = 20. Regarding the probability of false positives, we chose the default one given in this online calculator: 1.0E-6. This value tells us that the probability of the bloom filter telling us that an element is present, but actually not being present, is very low. This is an acceptable value because we are almost sure that the element we are checking for is available in that peer if the test over that peer bloom filter returns true.

4.3 Metrics

We used the following metrics to evaluate the proposed solution:

- **Average page load time**: average loading time from the time the user clicks until the web page is completely displayed (including all web content loaded, such as html, images, scripts, etc);

- **Success ratio**: we consider three values: the percentage of requests satisfied by the user's own local storage, the percentage of requests satisfied by the P2P network (i.e., local storage of user's neighbours or linked peers), and, in the case the item is not present in neither the local cache and in peers' caches, the percentage of those served by the origin server.

- **Server network throughput**: the average outgoing and incoming traffic in the origin server, giving us the network load imposed on it while client browsers are navigating through the application.

The average page load time and success ratio (for the proposed solution) were measured within our Javascript application after the web page is completely loaded (thus integrated as a small script into the web application itself). The test program (developed in Java) uses the interface JavascriptExecutor of the Selenium driver to return these values directly from the executing Javascript, which corresponds to the metrics of that web page. To make sure that all of these values were final, that is, that the page had loaded completely and the executing Javascript had reached the end of its execution, we wait for two events to happen/fire within the browser:

1. **DOMContentLoaded event has fired:** meaning the initial HTML document has been completely loaded and parsed. However this event does not wait for scripts, stylesheets, and images to finish loading.

2. **Boolean variable at the end of our executing Javascript has returned true:** meaning our framework has reached the end of its execution. Since our executing Javascript is the last element to load on the DOM of each page, we are sure that everything else has already loaded completely (including images).

The "boolean variable event" was only used for evaluating the proposed solution. In order to make sure that in the existing solution all images were completely loaded, we used a Javascript library called **imagesLoaded** [29] which detects when images have been completely loaded. We did not use this library in the proposed solution because it already uses, in our Javascript framework, a function (see Figure 3.4) that sends XMLHTTPRequests with the images URLs and waits for the responses (images) to load.

The network throughput was measured by **nload** [71], which is a monitor network traffic and bandwidth usage in real time, available on Unix based systems. It can be used to measure incoming and outgoing traffic and provides additional info like total amount of transferred data, average, minimum, and maximum network usage.

For the existing solution, we do not measure the success ratio. This is because it is a solution that does not have a decentralized component, consisting of a simple client-server model. This means that all contents present on web pages are always satisfied by the origin server when it is first requested. Instead of what happens in our solution, where most of these first time requests can be satisfied by (nearby) peers that have these requested contents in their local storage. This situation also happens when the requested contents are no longer in the local cache because they have been replaced. So, although it is not the first time that these contents are being requested, it is as if they were, because they are no longer locally available.

4.4 Tests conditions

The tests were carried out in a system with two distinct states:

- **Origin server not overloaded**: it did have sufficient resources to satisfy all the incoming requests;
- **Origin server overloaded**: it did not have sufficient resources to satisfy incoming requests. This overloading of to the origin server was achieved through the execution of a simple Java program [82] that generates load on the CPU and memory of the machine, thus not allowing it to respond to many end user requests simultaneously in a timely fashion.

For each of these origin server's states, we tested two scenarios:

- **One where it is possible to put all images in local storage:** this translated into having a quantity of images to be loaded that does not exceed the quota on it (which is 10 MB in Firefox and other modern browsers). To achieve this we set a fixed number of unique images to be loaded during navigation: **94**, and the sum of the weight of these images was approximately **9 MB**;
- **Another where it is not possible to put all images in local storage:** this translated into having a quantity of images to be loaded that exceed the local storage quota, in order to evaluate the adopted cache replacement strategy (detailed in Section 3.4) for the proposed solution. To achieve this we also set a fixed number of unique images to be loaded during navigation: **144**, and the sum of the weight of these images was approximately **16 MB**;

These images did not all belong naturally to the Moodle web application, so we had to manually add some of the images to the pages so that enough images were uploaded to allow the execution of the evaluation in the desirable test conditions. More than 70% of the images were added manually, because the images that were, by default, used by Moodle were mostly icons and images located in the header, footer and other areas of the layout of the application. Note that some of these images are repeated among web pages. The manually added images comprise a weight between 50 KB and 200 KB.

For the test in which it is possible to put all images in local storage, the number of images per web page varies between 10 and 20. This makes that the total weight of images per web page does not go far beyond the average on most web pages, that was previously mentioned in Section 3.1, which is 1310 KB (see Figure 3.3). For example, 10 images with 100 KB makes a total of 1000 KB weight per page, and with this, we want to demonstrate that the tests performed try to follow the referred average (which mirrors the real world).

For the test in which it is not possible to put all images in local storage, the number of images increased naturally per web page because we simply added more images to the pages accessed in our tests with regard to the images added for the previously described tests. This distribution of images was not linear between pages, since we wanted to evaluate in a more dynamic way the operation of the content replacement mechanism in the cache layer introduced and managed by our framework.

Regarding **clients** (leechers), we made tests where we ran **16**, **32**, and **64** simultaneously. Each virtual machine ran a maximum of 16 clients at a time. We ran some tests with **clients located (only) in Europe**, and some tests with **clients located in Europe and United States**, equally divided by the two locations (e.g. 64 clients EU/US means 32 clients in EU and 32 clients in US).

The **average page load time** is the average of the page load time obtained from the clients' navigation. To get this, we first had to get the average page load time of each client (adding the latencies obtained from the pages and dividing it by the total number of pages browsed). After obtaining these latencies from all clients, we could reach the average page load time, present in the results tables (presented further ahead).

The following results are an average of the execution of the tests three times under the above conditions.

4.5 Origin server not overloaded

In these experiments the load on the origin server allows it to respond to multiple requests without major delays, since it is not overloaded and thus has enough processing power and memory available. The following results were obtained under these conditions.

4.5.1 Images fit in local storage

			Server Network Throughput	
Clients		**Average Page Load Time (s)**	**Outgoing (MB/s)**	**Incoming (KB/s)**
16 EU		2.152	9.5	780
32 EU		3.580	19.17	1041
64 EU		3.910	28.35	1120
16	8 EU	2.150	13.43	809
	8 US	1.862		
32	16 EU	3.477	18.02	971
	16 US	3.311		
64	32 EU	3.802	25.70	1004
	32 US	3.503		

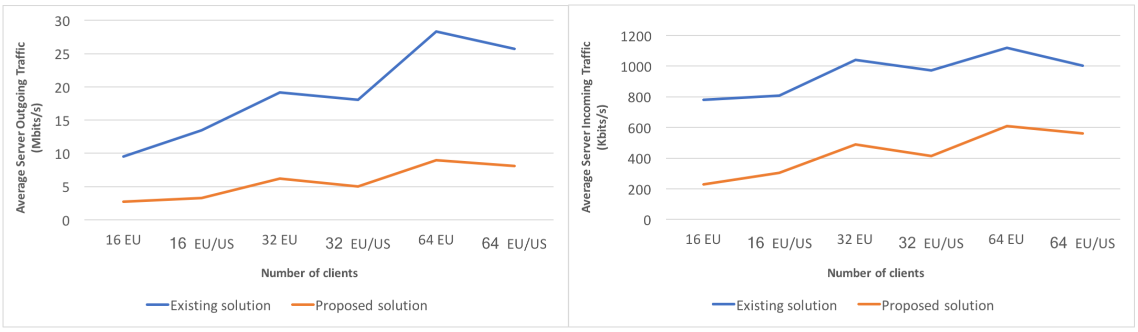
Table 4.1: Existing solution

Clients (leechers)		Average Page Load Time (s)	Success Ratio			Server Network Throughput	
			Local Cache (%)	P2P Network (%)	Origin Server (%)	Outgoing (MB/s)	Incoming (KB/s)
16 EU		2.726	49	31	20	2.7	230
32 EU		3.602	55	32	13	6.2	489
64 EU		4.309	52	34	14	8.98	610
16	8 EU	2.366	45	39	16	3.3	304
	8 US	2.186	41	40	9		
32	16 EU	3.587	60	30	10	5.02	414
	16 US	3.804	48	39	13		
64	32 EU	4.205	55	33	12	8.12	560
	32 US	3.920	58	29	13		

Table 4.2: Proposed solution

The proposed solution in terms of user experience shows to be worse than the existing solution, with higher end user perceived latencies. This is probably due to the additional execution time of our Javascript framework, which consists of several operations and flows, which P2P connections by themselves already have some additional overhead. In this case, it is better for clients to request all images directly to a central server, since it has sufficient resources to respond to multiple requests. We note however that extra latency increased in about 20% to 30%, which is not a huge overhead.

On the other hand, we can verify that in terms of load imposed on the server, the proposed solution is clearly better for the web application. The difference in the average outgoing and incoming traffic between solutions is significant. As we can see in Figure 4.5a, the proposed solution imposes about less 70% of load on the origin server than the existing solution. A similar difference occurs in the average incoming traffic (see Figure 4.5b) to the origin server, where the proposed solution imposes about less 50% to 60 % of load on the origin server, against the existing solution.



(a) Average Outgoing traffic (b) Average Incoming traffic

Figure 4.5: Server network throughput comparison

Relative to the graphs above, in the average outgoing traffic graph (Figure 4.5a) the Y axis has the units in megabits per second. This is because the traffic that effectively leaves the server is high, since it is sending data (images) to the clients. In the average incoming traffic graph (Figure 4.5b), the Y axis has the units in kilobits per second. This is because the traffic that actually enters the server, contrary to what leaves, is low, since it is the requests that the server is receiving (HTTP requests are usually small).

4.5.2 Images do not fit in local storage

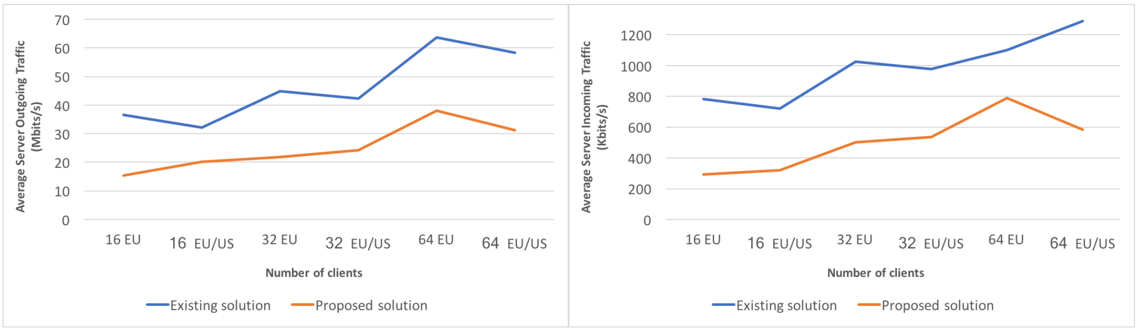
			Server Network Throughput	
Clients		**Average Page Load Time (s)**	**Outgoing (MB/s)**	**Incoming (KB/s)**
16 EU		3.202	36.56	781
32 EU		4.221	44.9	1024
64 EU		4.660	63.7	1102
16	8 EU	3.090	32.1	720
	8 US	2.064		
32	16 EU	4.378	42.36	977
	16 US	3.391		
64	32 EU	5.732	58.31	1290
	32 US	4.790		

Table 4.3: Existing solution

			Success Ratio			Server Network Throughput	
Clients (leechers)		**Average Page Load Time (s)**	**Local Storage (%)**	**P2P Network (%)**	**Origin Server (%)**	**Outgoing (MB/s)**	**Incoming (KB/s)**
16 EU		4.454	35	22	43	15.4	291
32 EU		5.112	39	21	40	21.87	502
64 EU		6.625	31	25	44	38.01	789
16	8 EU	3.544	29	20	51	20.09	320
	8 US	3.138	26	18	56		
32	16 EU	5.389	31	24	45	24.22	537
	16 US	4.130	35	12	53		
64	32 EU	6.454	36	26	38	31.15	583
	32 US	5.098	32	25	43		

Table 4.4: Proposed solution

Regarding the results in the tables above, the proposed solution demonstrates that it has a worse performance in terms of end user perceived latency. We can observe that the existing solution has lower latencies, about one second to two seconds less of average page load time compared to the proposed solution. The hit ratio of local storage, because of the evictions made by the cache mechanism adopted in this solution, is actually much smaller than under conditions where the cache does not became full. In addition, it is possible to verify that the number of requests to the server has increased because peers have not been able to satisfy so many requests among them (because they likely also suffered substitution actions in their local caches).



(a) Average Outgoing traffic (b) Average Incoming traffic

Figure 4.6: Server network throughput comparison

As verified in the test results presented previously, the traffic imposed on the server is also favorable in the proposed solution. Regarding this, the difference in the average outgoing and incoming traffic between solutions is also significant. As we can see in Figure 4.8a, the proposed solution imposes about less 40% to 50% of load on the origin server, compared to the existing solution. A similar difference occurs in the average incoming traffic (see Figure 4.8b) to the origin server, where the proposed solution imposes about less 45% to 55% of load on the origin server, against the existing solution.

4.6 Origin server overloaded

In these experiments the origin server is overloaded which does not allow it to respond to multiple requests in a timely way. The following results were obtained under these conditions.

4.6.1 Images fit in local storage

Clients		Average Page Load Time (s)	Server Network Throughput	
			Outgoing (MB/s)	Incoming (KB/s)
16 EU		4.759	5.3	260
32 EU		4.850	9.5	373
64 EU		5.910	19.75	920
16	8 EU	4.150	7.2	432
	8 US	3.862		
32	16 EU	4.477	8.65	510
	16 US	5.311		
64	32 EU	5.802	16.23	603
	32 US	5.503		

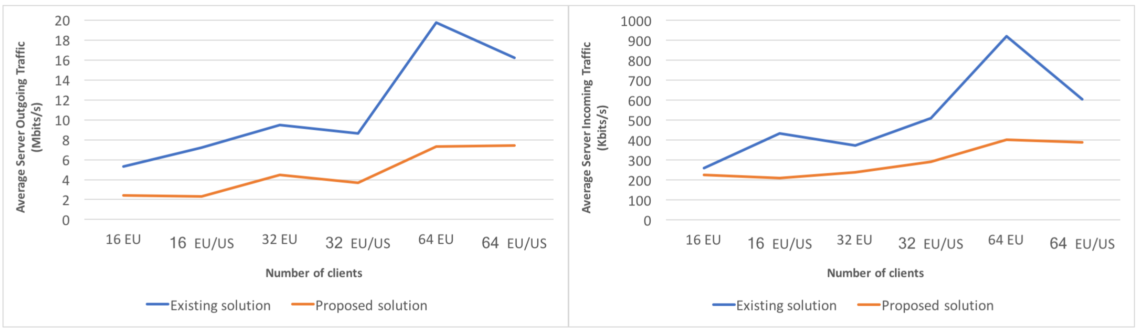
Table 4.5: Existing solution

Clients (leechers)		Average Page Load Time (s)	Success Ratio			Server Network Throughput	
			Local Storage (%)	P2P Network (%)	Origin Server (%)	Outgoing (MB/s)	Incoming (KB/s)
16 EU		3.322	46	39	15	2.4	225
32 EU		4.261	51	28	21	4.5	238
64 EU		5.122	49	31	20	7.3	401
16	8 EU	3.339	58	32	10	2.3	210
	8 US	3.218	51	28	9		
32	16 EU	3.912	52	39	9	3.7	290
	16 US	3.892	62	31	7		
64	32 EU	4.861	45	38	17	7.4	389
	32 US	4.965	48	36	16		

Table 4.6: Proposed Solution

The proposed solution in terms of user experience shows to be better than the existing solution, with lower end user perceived latencies. We can verify that our solution has average page load times with about 0.5 to 1.5 seconds less than the average page load times obtained in the existing solution. This can be justified by the fact that the centralized component is resource-exhausted and can not handle and respond to multiple requests simultaneously, incurring of more delays and additional overhead. On the other hand, our solution demonstrates that it can effectively overcome this issue through the leverage of the decentralized component - the P2P network. Network peers respond in this way more efficiently among them because they are not resource-exhausted as the central component. In this case, it is better for clients to request images among them, since the origin server is occupied handling and responding to waiting requests.

Additionally, we can verify that our solution imposes (again) much less load on the origin server. As we can see in Figure 4.7a, the proposed solution imposes about less 50% to 80% of load on the origin server than the existing solution. A similar difference occurs in the average incoming traffic (see Figure 4.7b) to the origin server, where the proposed solution imposes about less 20% to 50 % of load on the origin server, against the existing solution.



(a) Average Outgoing traffic

(b) Average Incoming traffic

Figure 4.7: Server network throughput comparison

4.6.2 Images do not fit in local storage

			Server Network Throughput	
Clients		**Average Page Load Time (s)**	**Outgoing (MB/s)**	**Incoming (KB/s)**
16 EU		5.012	21.04	525
32 EU		5.645	34.15	668
64 EU		6.483	48.01	800
16	8 EU	5.311	16.4	402
	8 US	4.729		
32	16 EU	5.424	30.69	513
	16 US	4.650		
64	32 EU	6.331	42.27	743
	32 US	6.720		

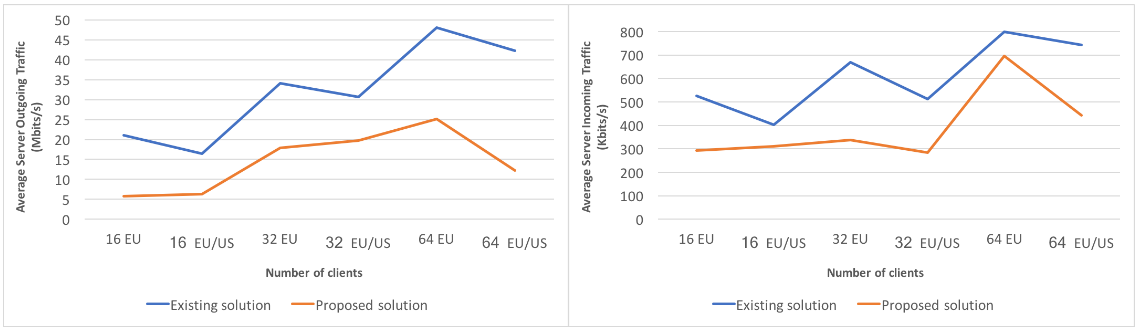
Table 4.7: Existing solution

			Satisfied Requests Ratio			Server Network Throughput	
Clients (leechers)		**Average Page Load Time (s)**	**Local Storage (%)**	**P2P Network (%)**	**Origin Server (%)**	**Outgoing (MB/s)**	**Incoming (KB/s)**
16 EU		5.451	31	20	49	5.76	292
32 EU		4.910	34	25	41	17.94	337
64 EU		7.640	26	18	56	25.12	695
16	8 EU	5.602	38	20	42	6.3	310
	8 US	4.144	39	25	36		
32	16 EU	4.833	35	19	46	19.8	284
	16 US	4.870	37	21	42		
64	32 EU	6.039	41	27	32	12.2	443
	32 US	6.805	37	24	39		

Table 4.8: Proposed solution

The results demonstrate that both solutions performed similiar in terms of end user perceived latency, since for a certain number of clients the existing solution gives lower values and for another particular number of clients gives values higher than the proposed solution. As observed in the same test but with the origin not overloaded (see Table 4.4), the cache hit ratio is lower than under conditions where the cache do not became full.

Regarding the server network throughput, our solution proves, once again, that imposes much less load on the origin server with a significant difference of traffic generated between the two solutions.



(a) Average Outgoing traffic

(b) Average Incoming traffic

Figure 4.8: Server network throughput comparison

4.7 Discussion

The results generally demonstrate that our solution performs better in terms of end user perceived latency for cases where the centralized component (origin server) is depleted of resources. In contrast, in systems where the origin server has the capability to respond to multiple requests simultaneously, our solution has proven to perform worse at end user perceived latency, although with a small overhead.

Our solution clearly demonstrated that in all tests it "took" much of the burden (usually) imposed on the origin server of a web application and passed it to the clients (decentralized component), which resulted in a cooperative cache between clients. This is a key aspect for operators of small and medium sized web applications.

The parameters used can be adjusted to the context of the applications, so our solution can perform better at some levels if they are properly adjusted and thus optimized. In addition, it is important to note that the cache replacement strategy can be changed to a different one (for example by considering additionally objects frequency and size factors) and potentially be better suited to a given service and its network users (based on their caching behaviour).

4.8 Summary

This chapter presented the results and evaluation of this work, in particular described all the setup environment where the tests were run.

In the setup test context we described and explained the components of the system where the tests were run, in particular the clients, the Legion server and the origin server that hosts the Moodle web application.

In the parameters context we described and discussed the several parameters that are relevant for a given web application, related to our framework and the Legion framework that allows us to use a decentralized component - the P2P network.

In the metrics context we presented and described all the used metrics in the testing and evaluation of the proposed and existing (alternative) solution.

In the test conditions context we explained that the tests would focus on a system where the origin server would have two states (overloaded and not overloaded), and within this category we would have two situations (test with and without the adopted cache replacement based on the number of images to be allocated in the clients local storage). We also mentioned the number of client that would be executed simultaneously during the tests and their geographic location.

In the results context we presented tables and graphs with the obtained metrics from the experiments, and discussed it.

The next chapter presents the conclusion and future work of this dissertation, where we point out what we can draw from positive and negative from this work, and discuss what can be done to improve it in the future.

Chapter 5
Conclusion

Existing Web caching solutions require financial power to support the costs of services such as Akamai, thus limiting the adoption by small and medium scale web applications. Also, some solutions, like Akamai Netsession, require installation of additional software, or plugins/extensions in the clients browsers.

In this dissertation, we propose a solution that is easy to adopt and integrate into any web application, which allows optimizing web caching through a cooperative cache between clients without having to spend money to support it.

In this work we implemented the proposed framework for a specific type of content: images, since it is (usually) a static content and occupy more than half of the total weight of web pages. Besides, our framework can be implemented and generalized to other types of web content, such as HTML code, CSS files, scripts, JSON files, etc.

We can conclude that our solution is beneficial to small and medium-scale web applications that can not support the costs of existing web caching systems, and because of the potential scalability of their services, their origin servers are unable to respond to multiple requests efficiently, due to the increasing number of users.

On the other hand, our solution has some negative aspects, which emphasize in particular the overhead that our framework (developed in Javascript) incurs during its execution. This is mainly due to the inter-layer communication at the application layer level, which is part of the design of our framework, access and updates to the disk (local storage) that are due to the maintenance of the content metadata and the sending of that content to other peers, and the additional overhead imposed on P2P communications, managed by the overlay of the Legion framework.

We think that our work can be improved in some aspects, namely the parameters that can be tested with other values in order to further optimize the access to given web applications, as well as the mechanisms of cache replacement, which can be used other caching techniques based on the caching behavior of clients that are part of the network.

5.1 Future Work

Design and implementation of a light coordination mechanism between the caching layer at the users and at the distributed caching system. For example, enrich Memcached (available as integrated module in some web applications, such as Moodle) using Legion based techniques. The distributed cache system can be modified to adapt to new caching mechanisms, and some synchronization methods must be integrated in the client and server side in order to establish and fulfill an appropriate and improved caching flow between them.

These new mechanisms can be managed by points of control located in the boundaries of their networks, called the Edge Control Points (ECP), a component which is naturally distributed, and the Centralized Control Points (CCP), which periodically exchange information about their cached content. This way, it is possible to avoid that the server unnecessarily caches content that is already present in the client's local caches, and is being shared among them directly. The distributed cache system chosen to integrate this solution might need to be (slightly) modified in order to adapt to this idea of cache synchronization between the center and the edge of the network.

An extra feature that can be integrated in this distributed solution is the use of a centralized mechanism (running in the server) to assist clients in establishing new connections with other clients that have local cached content that can be useful. The proposed solution follows the criteria of relative distance between peers to link them. This would be helpful by informing a peers that there are other peers that they do not know of, which potentially have content of their interest (based on previous requests). Therefore, they will be able to connect which can probably satisfy more requests in the near future. This approach would minimize load on cache server(s), and even reduce end user latency (in case users are close to each other).

Bibliography

[1] URL: https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. *Caching Proxies: Limitations and Potentials*. Tech. rep. Blacksburg, VA, USA, 1995.

[3] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. “Removal Policies in Network Caches for World-Wide Web Documents”. In: *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’96. Palo Alto, California, USA: ACM, 1996, pp. 293–305. ISBN: 0-89791-790-1. DOI: 10.1145/248156.248182. URL: http://doi.acm.org/10.1145/248156.248182.

[4] C. Aggarwal, J. L. Wolf, and P. S. Yu. “Caching on the World Wide Web”. In: *IEEE Trans. on Knowl. and Data Eng.* 11.1 (Jan. 1999), pp. 94–107. ISSN: 1041-4347. DOI: 10.1109/69.755618. URL: http://dx.doi.org/10.1109/69.755618.

[5] B. M. Ahmed, T. Helaly, and S. Rahman. *Prioritizing Documents and Applying Hybrid Caching Strategy for Network Latency Reduction*.

[6] D. P. Anderson. “BOINC: A System for Public-Resource Computing and Storage”. In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. GRID ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. ISBN: 0-7695-2256-4. DOI: 10.1109/GRID.2004.14. URL: http://dx.doi.org/10.1109/GRID.2004.14.

[7] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. “SETI@home: an experiment in public-resource computing”. In: *Communications of the ACM* 45.11 (2002), pp. 56–61.

[8] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. “Evaluating Content Management Techniques for Web Proxy Caches”. In: *SIGMETRICS Perform. Eval. Rev.* 27.4 (Mar. 2000), pp. 3–11. ISSN: 0163-5999. DOI: 10.1145/346000.346003. URL: http://doi.acm.org/10.1145/346000.346003.

[9] M. Arlitt, R. Friedrich, and T. Jin. “Performance Evaluation of Web Proxy Cache Replacement Policies”. In: *Perform. Eval.* 39.1-4 (Feb. 2000), pp. 149–164. ISSN: 0166-5316. DOI: 10.1016/S0166-5316(99)00062-0. URL: http://dx.doi.org/10.1016/S0166-5316(99)00062-0.

[10] H. Bahn, K. Koh, S. L. Min, and S. H. Noh. "Efficient Replacement of Nonuniform Objects in Web Caches". In: *Computer* 35.6 (June 2002), pp. 65–73. ISSN: 0018-9162. DOI: 10.1109/MC.2002.1009170. URL: http://dx.doi.org/10.1109/MC.2002.1009170.

[11] S. A. Baset and H. G. Schulzrinne. "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol". In: *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings* (Apr. 2006), pp. 1–11. ISSN: 0743-166X. DOI: 10.1109/infocom.2006.312. URL: http://dx.doi.org/10.1109/infocom.2006.312.

[12] D. Bau. *Opera Software*. URL: https://github.com/davidbau/seedrandom.

[13] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[14] J.-C. Bolot and P. Hoschka. "Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design". In: *Comput. Netw. ISDN Syst.* 28.7-11 (May 1996), pp. 1397–1405. ISSN: 0169-7552. DOI: 10.1016/0169-7552(96)00073-6. URL: https://doi.org/10.1016/0169-7552(96)00073-6.

[15] J. Buford, H. Yu, and E. K. Lua. *P2P networking and applications*. Morgan Kaufmann, 2009.

[16] A. M. C. Chang and G. Holmes. *The LRU*WWW proxy cache document replacement algorithm*. 1999.

[17] *Caching Architectures*. http://www.intechopen.com/books/computational-intelligence-and-modern-heuristics/intelligent-exploitation-of-cooperative-client-proxy-caches-in-a-web-caching-hybrid-architecture. Accessed: 2017-01-11.

[18] P. Cao and S. Irani. "Cost-aware WWW Proxy Caching Algorithms". In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*. USITS'97. Monterey, California: USENIX Association, 1997, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1267279.1267297.

[19] J. L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290858, 9781617290855.

[20] H. Chen, S. S. Fuller, C. Friedman, and W. Hersh. "Knowledge management, data mining, and text mining in medical informatics". In: *Medical Informatics*. Springer, 2005, pp. 3–33.

[21] K. Cheng and Y. Kambayashi. "LRU-SP: a size-adjusted and popularity-aware LRU replacement algorithm for web caching". In: *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*. IEEE. 2000, pp. 48–53.

[22] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. “Freenet: A distributed anonymous information storage and retrieval system”. In: *Designing Privacy Enhancing Technologies*. Springer. 2001, pp. 46–66.

[23] E. Cohen, B. Krishnamurthy, and J. Rexford. “Evaluating Server-Assisted Cache Replacement in the Web”. In: *Proceedings of the 6th Annual European Symposium on Algorithms*. ESA ’98. London, UK, UK: Springer-Verlag, 1998, pp. 307–319. ISBN: 3-540-64848-8. URL: `http://dl.acm.org/citation.cfm?id=647908.740142`.

[24] J Cohen, N Phadnis, V Valloppillil, and K. Ross. “Cache array routing protocol v1. 1”. In: *Sep* 29 (1997), pp. 1–8.

[25] *Coral CDN*. URL: `http://www.coralcdn.org/`.

[26] V. Das. *Learning Redis*. Packt Publishing, 2015. ISBN: 1783980125, 9781783980123.

[27] J. Davies. *Bloom Filter*. URL: `https://www.jasondavies.com/bloomfilter/`.

[28] D. R. c. Deepak Sachan. *Performance Improvement of Web Caching Page Replacement Algorithms*. 2014.

[29] desandro. URL: `https://github.com/desandro/imagesloaded`.

[30] *Distributed cache systems*. URL: `https://www.quora.com/What-is-distributed-caching`.

[31] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. “Cooperative backup system”. In: *The USENIX Conference on File and Storage Technologies*. 2002.

[32] J. Famaey, F. Iterbeke, T. Wauters, and F. De Turck. “Towards a predictive cache replacement strategy for multimedia content”. In: *Journal of Network and Computer Applications* 36.1 (2013), pp. 219–227.

[33] C. H. Fenichel. “The Process of Searching Online Bibliographic Databases: A Review of Research.” In: *Library Research* 2.2 (1980), pp. 107–27.

[34] B. Fitzpatrick. “Distributed caching with memcached”. In: *Linux journal* 2004.124 (2004), p. 5.

[35] A. P. Foong, Y. H. Hu, and D. M. Heisey. “Essence of an Effective Web Caching Algorithm.” In: *International Conference on Internet Computing*. 2000, pp. 269–276.

[36] G Fowler. “Fowler/Noll/Vo (FNV) hash”. In: *ONLINE http://isthe. com/chongo/tech/comp/fnv* (1991).

[37] M.-A. Free. *Open Source Course Management System for Online Learning*. 2008.

[38] P. Ganesan, K. Gummadi, and H. Garcia-Molina. “Canon in G major: designing DHTs with hierarchical structure”. In: *Distributed computing systems, 2004. proceedings. 24th international conference on*. IEEE. 2004, pp. 263–272.

[39] F. González-Cañete, J Sanz-Bustamante, E Casilari, and A Triviño-Cabrera. "Evaluation of randomized replacement policies for web caches". In: *Proceedings of the IEEE INFOCOM*. 2007.

[40] M. He, Y. Zhang, and X. Meng. "Gossip-Based Resource Location Strategy in Interest Community for P2P Networks". In: *Chinese Journal of Electronics* 24.2 (2015), pp. 272–280.

[41] S. Hosseini-Khayat. "Investigation of Generalized Caching". UMI Order No. GAX98-07761. PhD thesis. St. Louis, MO, USA, 1998.

[42] *HTTP End-Point Icon*. URL: http://www.nanoscale.io/wp-content/uploads/2016/09/icon-http.svg.

[43] *HTTP Pinger*. URL: https://github.com/JensRantil/http-pinger.

[44] T. Hurst. *Bloom Filter Calculator*. URL: https://hur.st/bloomfilter.

[45] *Internet Users*. URL: http://www.internetlivestats.com/internet-users/.

[46] R Jesup, S Loreto, and M Tuexen. "Rtcweb data channels". In: *IETF ID: draft-ietf-rtcweb-data-channel-05 (work in progress)* (2013).

[47] S. Jin and A. Bestavros. *Temporal Locality in Web Request Streams: Sources, Characteristics, and Caching Implications*. Tech. rep. Boston, MA, USA, 1999.

[48] J. S. KELLY T. and J. K. MACKIE-MASON. "Variable QoS from shared Web caches: User centered design and value-sensitive replacement." In: 1999.

[49] M. R. Korupolu and M. Dahlin. "Coordinated placement and replacement for large-scale distributed caches". In: *IEEE Transactions on Knowledge and Data Engineering* 14.6 (2002), pp. 1317–1329.

[50] B. Krishnamurthy and C. E. Wills. "Proxy cache coherency and replacement-towards a more complete picture". In: *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE. 1999, pp. 332–339.

[51] J. Leitao. "Topology Management for Unstructured Overlay Networks". In: *Technical University of Lisbon* (2012).

[52] C. Li and A. L. Cox. "GD-Wheel: a cost-aware replacement policy for key-value stores". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 5.

[53] A. van der Linde. "Enriching Web Applications with Browser-to-Browser Communication". PhD thesis. Universidade Nova de Lisboa, 2015.

[54] A. van der Linde, P. Fouto, J. a. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052673. URL: https://doi.org/10.1145/3038912.3052673.

[55] N. Megiddo and D. S. Modha. "Outperforming LRU with an adaptive replacement cache algorithm". In: *Computer* 37.4 (2004), pp. 58–65.

[56] A. Merchant, M. Kallahalla, and R. Swaminathan. *Sharding method and apparatus using directed graphs*. US Patent 7,043,621. 2006.

[57] A. Mislove. "POST: a secure, resilient, cooperative messaging system". In: *Notes* 20 (2003), p. 22.

[58] C. D. Murta, V. Almeida, and W. Meira Jr. "Analyzing performance of partitioned caches for the WWW". In: *Proceedings of the 3rd International WWW Caching Workshop*. 1998.

[59] *Napster*. URL: http://www.napster.com.

[60] N. Niclausse, Z. Liu, P. Nain, et al. "A new efficient caching policy for the World Wide Web". In: *Proceedings of the Workshop on Internet Server Performance*. 1998, pp. 119–128.

[61] *Opera Software*. URL: https://www.opera.com.

[62] N. Osawa, T. Yuba, and K. Hakozaki. "Generational Replacement Schemes for a WWW Caching Proxy Server". In: *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*. HPCN Europe '97. London, UK, UK: Springer-Verlag, 1997, pp. 940–949. ISBN: 3-540-62898-3. URL: http://dl.acm.org/citation.cfm?id=645561.659042.

[63] J. W. O'Toole Jr and D. M. Bornstein. *Method and apparatus for transparent distributed network-attached storage with web cache communication protocol/anycast and file handle redundancy*. US Patent 7,254,636. 2007.

[64] *Overview of Memcached*. URL: https://github.com/memcached/memcached/wiki/Overview#how-does-it-work.

[65] *Peer5*. URL: https://www.peer5.com.

[66] J. Pitkow and M. Recker. "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns". In: *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*. 1994. URL: http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/pitkow/caching.html.

[67] S. Podlipnig and L. Böszörmenyi. “A Survey of Web Cache Replacement Strategies”. In: *ACM Comput. Surv.* 35.4 (Dec. 2003), pp. 374–398. ISSN: 0360-0300. DOI: 10.1145/954339.954341. URL: http://doi.acm.org/10.1145/954339.954341.

[68] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. “The bittorrent p2p file-sharing system: Measurements and analysis”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2005, pp. 205–216.

[69] K. Psounis and B. Prabhakar. “A randomized web-cache replacement scheme”. In: *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. IEEE. 2001, pp. 1407–1415.

[70] S. Quinlan and S. Dorward. “Venti: A New Approach to Archival Storage.” In: *FAST*. Vol. 2. 2002, pp. 89–101.

[71] R. Riegel. *nload*. URL: http://www.roland-riegel.de/nload/.

[72] L. Rizzo and L. Vicisano. “Replacement Policies for a Proxy Cache”. In: *IEEE/ACM Trans. Netw.* 8.2 (Apr. 2000), pp. 158–170. ISSN: 1063-6692. DOI: 10.1109/90.842139. URL: http://dx.doi.org/10.1109/90.842139.

[73] P. Rodriguez, C. Spanner, and E. W. Biersack. “Analysis of web caching architectures: Hierarchical and distributed caching”. In: *IEEE/ACM Transactions on Networking (TON)* 9.4 (2001), pp. 404–418.

[74] S. Romano and H. ElAarag. “A Quantitative Study of Recency and Frequency Based Web Cache Replacement Strategies”. In: *Proceedings of the 11th Communications and Networking Simulation Symposium*. CNS ’08. Ottawa, Canada: ACM, 2008, pp. 70–78. ISBN: 1-56555-318-7. DOI: 10.1145/1400713.1400725. URL: http://doi.acm.org/10.1145/1400713.1400725.

[75] A. Rowstron and P. Druschel. “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems”. In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[76] P. Scheuermann, J. Shim, and R. Vingralek. “A Case for Delay-conscious Caching of Web Documents”. In: *Comput. Netw. ISDN Syst.* 29.8-13 (Sept. 1997), pp. 997–1005. ISSN: 0169-7552. DOI: 10.1016/S0169-7552(97)00032-9. URL: http://dx.doi.org/10.1016/S0169-7552(97)00032-9.

[77] *Selenium*. URL: http://www.seleniumhq.org/.

[78] I. Sharfman, A. Schuster, and D. Keren. “A geometric approach to monitoring threshold functions over distributed data streams”. In: *ACM Transactions on Database Systems (TODS)* 32.4 (2007), p. 23.

[79] A. Soliman. *Getting Started with Memcached*. Packt Publishing, 2013. ISBN: 1782163220, 9781782163220.

[80] S. Spoto, R. Gaeta, M. Grangetto, and M. Sereno. "Analysis of PPLive through active and passive measurements". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–7.

[81] *Squid-Cache*. URL: http://www.squid-cache.org/.

[82] SriramKeerthi. URL: https://gist.github.com/SriramKeerthi/0f1513a62b3b09fecaeb.

[83] D. Starobinski and D. Tse. "Probabilistic Methods for Web Caching". In: *Perform. Eval.* 46.2-3 (Oct. 2001), pp. 125–137. ISSN: 0166-5316. DOI: 10.1016/S0166-5316(01)00045-1. URL: http://dx.doi.org/10.1016/S0166-5316(01)00045-1.

[84] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[85] I. Tatarinov. "An efficient LFU-like policy for Web caches". In: *Tech. Rep. NDSU-CSORTR-98-01* (1998).

[86] K. Tutschku. "A measurement-based traffic profile of the eDonkey filesharing service". In: *International Workshop on Passive and Active Network Measurement*. Springer. 2004, pp. 12–21.

[87] A. Vakali. "LRU-based Algorithms for Web Cache Replacement". In: *Proceedings of the First International Conference on Electronic Commerce and Web Technologies*. EC-WEB '00. London, UK, UK: Springer-Verlag, 2000, pp. 409–418. ISBN: 3-540-67981-2. URL: http://dl.acm.org/citation.cfm?id=646160.680189.

[88] C. Vogt, M. J. Werner, and T. C. Schmidt. "Leveraging WebRTC for P2P content distribution in web browsers". In: *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE. 2013, pp. 1–2.

[89] Q. H. Vu, M. Lupu, and B. C. Ooi. *Peer-to-peer computing: Principles and applications*. Springer Science & Business Media, 2009.

[90] J. Wang. "A Survey of Web Caching Schemes for the Internet". In: *SIGCOMM Comput. Commun. Rev.* 29.5 (Oct. 1999), pp. 36–46. ISSN: 0146-4833. DOI: 10.1145/505696.505701. URL: http://doi.acm.org/10.1145/505696.505701.

[91] D. Wessels and k. claffy. *IETF RFC 2186: Internet Cache Protocol (ICP), version 2*. 1997.

[92] K.-Y. Wong. "Web Cache Replacement Policies: A Pragmatic Approach". In: *Netwrk. Mag. of Global Internetwkg.* 20.1 (Jan. 2006), pp. 28–34. ISSN: 0890-8044. DOI: 10.1109/MNET.2006.1580916. URL: http://dx.doi.org/10.1109/MNET.2006.1580916.

[93] R. P. Wooster and M. Abrams. “Proxy Caching That Estimates Page Load Delays”. In: *Comput. Netw. ISDN Syst.* 29.8-13 (Sept. 1997), pp. 977–986. ISSN: 0169-7552. DOI: 10.1016/S0169-7552(97)00041-X. URL: http://dx.doi.org/10.1016/S0169-7552(97)00041-X.

[94] B. Yang and H. Garcia-Molina. “Comparing hybrid peer-to-peer systems”. In: *Proceedings of the 27th Intl. Conf. on Very Large Data Bases*. 2001.

[95] J Yang, W Wang, R Muntz, and J Wang. “Access driven Web caching”. In: *UCLA Technical Report# 990007* (1999).

[96] Q. Yang and H. Zhang. “Taylor Series Prediction: A Cache Replacement Policy Based on Second-Order Trend Analysis”. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 5 - Volume 5*. HICSS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 5023–. ISBN: 0-7695-0981-9. URL: http://dl.acm.org/citation.cfm?id=820757.821888.

[97] C. T. Yu and C. Chang. “Distributed query processing”. In: *ACM computing surveys (CSUR)* 16.4 (1984), pp. 399–433.

[98] I. R. R. D. Zhang J. and M. Ott. “Web caching framework: Analytical models and beyound”. In: *Internet Applications, 1999. IEEE Workshop on*. 1999.

[99] J. Zhang, R. Izmailov, D. Reininger, and M. Ott. “Web caching framework: Analytical models and beyond”. In: *Internet Applications, 1999. IEEE Workshop on*. IEEE. 1999, pp. 132–141.

[100] L. Zhang, S. Floyd, and V. Jacobsen. “Adaptive Web Caching”. In: *In Proceedings of the NLANR Web Cache Workshop*. 1997, pp. 9–7.

[101] X. Zhang, J. Liu, B. Li, and Y.-S. Yum. “CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming”. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. IEEE. 2005, pp. 2102–2111.

[102] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. “Peer-assisted content distribution in akamai netsession”. In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 31–42.