Francisco Perdigão Fernandes

Licenciado em Engenharia Informática

Improving Web-Caching Systems with Transparent Client Support

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: João Leitão, Assistant Professor,
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

January, 2017

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 to 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 sob a 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, 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 - 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 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 or additional software.

1.3 Main Expected Contributions

As detailed in Section 3.1, this work aims to develop a distributed solution that enriches distributed caching systems. This solution can leverage on components of the Legion 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 distributed cache system might additionally need to be slightly 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.

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

- Design and implementation of caching between users through peer-to-peer techniques (leveraging existing components in Legion and WebRTC connections).
- Design and implementation of a light coordination mechanism between the caching layer at the users and at the distributed caching system (e.g. Memcached), promoting optimization of caching policies between them (performed by edge and centralized control points).
- Evaluation of the proposed solution and comparison to existing systems.

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 and related work plan. It also presents a plan for conducting the remainder work for achieving the goals proposed here.

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** studies 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 may serve as background to address challenges faced by the work to be conducted 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 **??**, which is a music-sharing system, Freenet **??**, which is a platform for censorship-resistant communication, and SETI@home **??**, 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 directly. More specifically, system's state and tasks are spreaded over them. Peers have equal privileges, and they form a peer-to-peer network of nodes. 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, through 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 by themselves.

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 operator. 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 distributing files (e.g. eDonkey [71] and BitTorrent [56]), streaming media (e.g. PPLive [66] and CoolStreaming [86]), telephony (e.g. Skype [10]), and volunteer computing (e.g. SETI@home [6]). P2P systems have also been designed for other proposes like distributed data monitoring [65], management and data mining [18], massively distributed query processing [82], serverless email [46], archival storage [58], bibliographic databases [27] and cooperative backup [25], 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 [87], which uses P2P downloads to increase performance and reduce 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 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) maintains membership and a content index in their centralized component (website); and BOINC [5], 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 spread 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 [75]. 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 [31] and super-peer architectures [80]).
- 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 queries. Example protocols include Chord [69], where

queries are propagated to node's successor through a ring of connected nodes; and Pastry [63], 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. Popular unstructured peer-to-peer systems are FreeNet [20], 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; and Gnutella [59], a filesharing service whose design is not so focused.

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 super peers), where most of the traffic and consumption of resources happens at the super-peers layer.

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 is an hard challenge. Query dissemination techniques [13, 41] 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 [22].

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 to 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 [11], 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 and stops until the resource is found.

Limited Random Walks: defines a threshold on the amount of total random walks that the algorithm will take, 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 based on a priory probability and then these neighbors forward this request to their neighbors for the peer. 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 [32]. The authors of this work prove that, with a user profiling method based on the characteristics of peers and the similarity built from interest community, performance has been 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 [43] is a framework that allows client web applications to replicate data from servers and propagate those among other clients through peer-to-peer interactions. It 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 (GdDriveRT). This allows the interaction and sharing of the same objects between legacy clients accessing GRDriveRT directly and enriched clients using Legion. In 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 others 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 **??** 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 specific a group of globally-distributed servers called the NetSession control plane [87]. 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 Transition (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 [53] 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 it already had approximately 44 million users, in 2005 had approximately 1 billion users, 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 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 type, 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

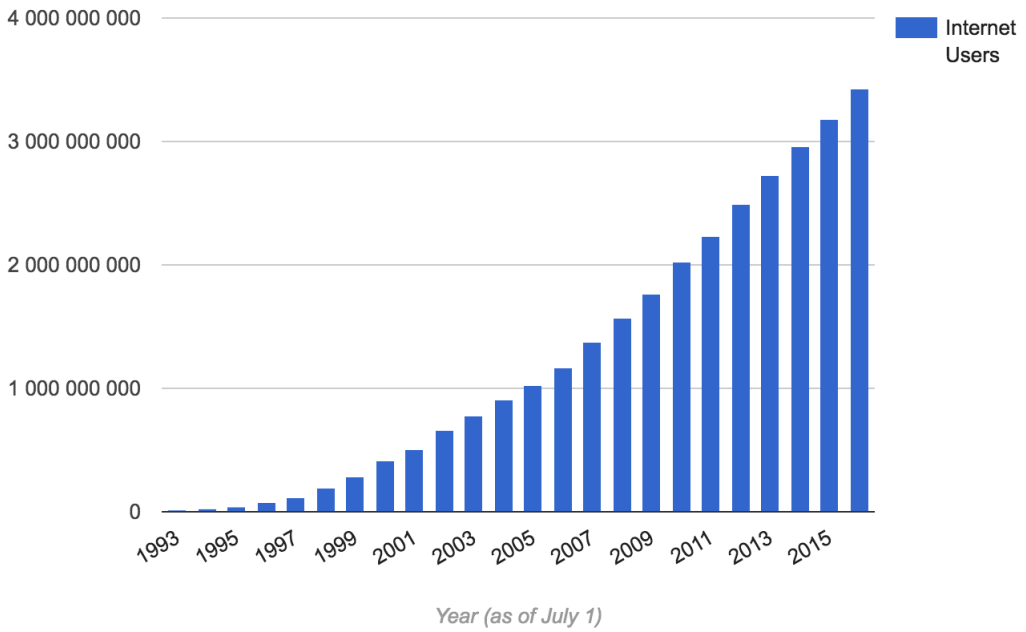

Figure 2.1: Internet Users in the World [34]

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 more 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 [67] 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 problems, 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 [4] or Transparent caching [23] are solutions that optimize the Web caching process. They are designed to achieve transparency to clients through the WCCP (Web Cache Communication Protocol) [51], 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.

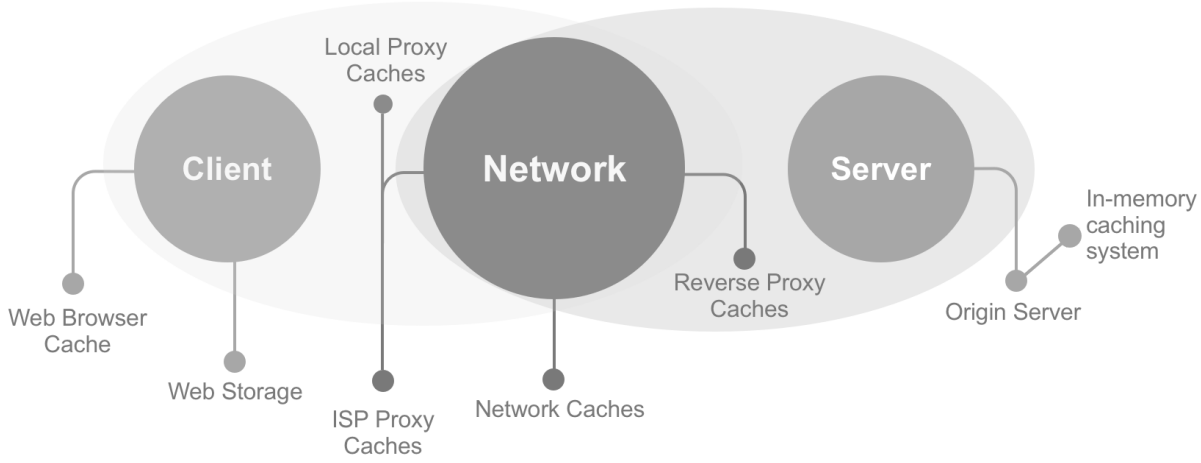

Figure 2.2: Overview of Different Levels of Caching

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 particularly 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 to 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. The proposed protocol will be deployed over the Web storage cache infrastructure. A distributed cache architecture, such as Redis (Section 2.2.5.2) or Memcached (Section 2.2.5.1), will be modified and adapted to meet the rationale behind the proposed solution to improve cache performance and management.

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 communication is also required to exist between these different levels of caching.

2.2.2.1 Hierarchical Caching

The caching hierarchy, shown in Figure 2.3, cache places 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. Several approaches of this architecture were designed, such as the Adaptive Web caching [85] and the Access Driven cache [35].

2.2.2.2 Distributed Caching

This architecture, shown 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

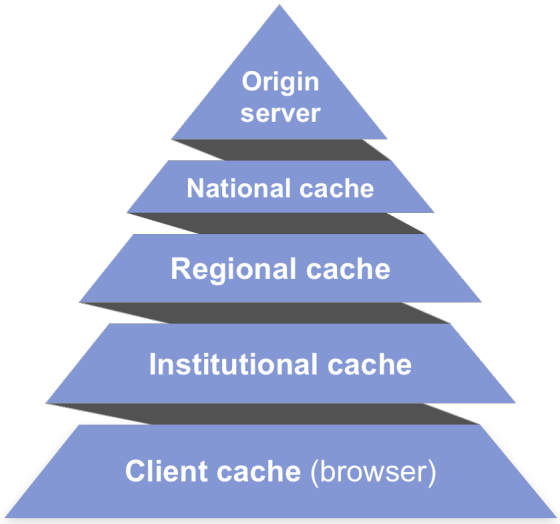

Figure 2.3: Hierarchical Caching - Pyramidal Representation

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) [77], which sustain object finding and retrieval from neighbors as well as from parent caches, and Cache Array Routing protocol (CARP) [73].

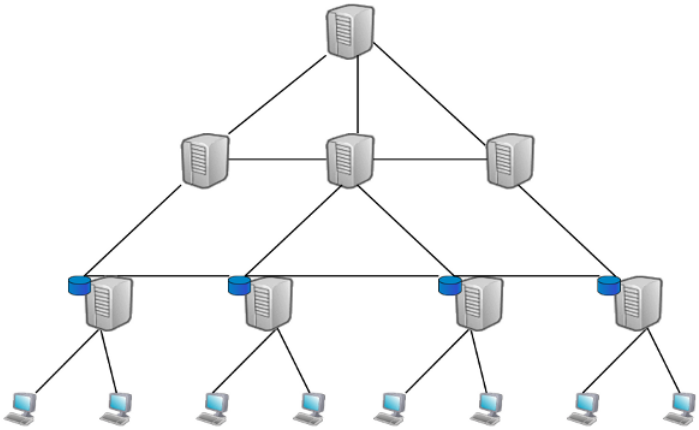

Figure 2.4: Distributed Caching Representation [15]

2.2.2.3 Hybrid Caching

This architecture embodies aspects from both previously described caching architectures, where determined 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 [76], where the authors propose to "employ the hierarchical distribution mechanism for more efficient and scalable distribution of metadata". In [4], the authors focus on designing "an hybrid-caching architecture for improving caching that has the lowest latency delivery time for popular documents". In [15], 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.

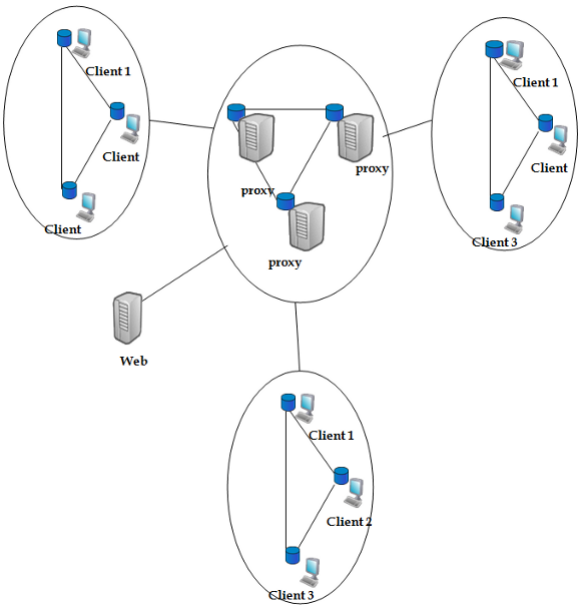

Figure 2.5: Proposed Hybrid Architecture [15]

2.2.2.4 Discussion

According to [61], 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 continues access to outdated versions of the object)

According to [55], a replacement process is 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 [78].

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** [1]: a certain object is not cached when the size of it exceeds a given threshold, otherwise this strategy is equivalent to LRU.
- **LRU-Min** [1]: 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** [72]: 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** [33]: 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 based on it:

- **LOG2-SIZE** [62]: 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** [54]: 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)** [3]: 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** [47]: 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** [83]: LRU uses the time period between the current time and the time of the last request time for each object.

- **Value-Aging** [84]: 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 use **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 of being referenced.

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 variants found in the literature:

- **LFU-Aging** [7]: 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** [7]: 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** [83]: 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)** [38]: 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** [38]: 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 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. 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 [16] 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** [21], 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 [7], GD* [37], TSP(Taylor Series Prediction) [81], Bolot/Hoschka's strategy [12], MIX [48], HYBRID [79], LNC-R-W3 [64], LRV [60], LUV [9] and LR(Logistic Regression)-Model [30]. Many of these strategies use similar factors and weighting schemes, and choose the object with the smallest value.

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 function 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** [33]: where RAND uses a uniform distribution, this 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** [68]: 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) [57]: 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 essential at random. 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 [29], 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) [8], LRU* [14], LRU-SP [19], Generational Replacement [50] and Cubic Selection Scheme (CSS) [70]). Among these strategies, only LRU* and Generational Replacement use simple LRU with frequency counts. HYPER-G [2] 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 their individual strategies. 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.

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** [44] was shown to be a "self-tuning, low-overhead, scan-resistant ARC cache-replacement policy 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 [39] 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** [40], **Multimedia cache replacement** [26], and **Differentiated cache replacement** [55].

A more recent paper [42] 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 [24] came to replace the traditional cache concept in a single location. A distributed cache can scale in size and also offers 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 is potentially 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 [28] 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 [52] 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 [17], 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 [45] 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 [74], they propose a peer-to-peer content distribution architecture using WebRTC Data Channels [36]. 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 may be used to support and provide resources for the solution and demonstration to be 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 [49]. 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 this work.

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, larger amounts of data stored locally without degrading the performance of websites, which are never transferred to

the server. 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 across 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.

In the next chapter will be explained the proposed solution, and described the work plan for the elaboration of this dissertation.

Chapter 3

Proposed Work

This chapter presents and explains the proposed solution. The work plan is discussed in the end of this chapter.

3.1 Proposed Solution

The idea of this work is to design a distributed solution to enrich an existing in-memory caching system (Memcached or Redis) with transparent client support in the browser. This solution will be designed by leveraging existing components of propagation in the Legion framework, which creates a peer-to-peer network that automates the connection and propagates updates among clients running in browsers, while leveraging WebRTC to support direct browser-to-browser communication. This approach will allow users to transparently share content directly among them, without the need of installing any kind of software or browser extensions, which favors the adoption of the proposed solution.

The use of Legion based techniques and Memcached/Redis aims at exploiting the coordination between the two layers of caching that can be directly managed by the (web) application layer. An overview of the proposed solution is presented in Figure 3.1, which gives a better idea and understating concerning the operations and flows that can occur during its execution. Briefly, a Web request can be satisfied, sorted by priority of execution, as follows:

1. by the user's own local storage, or;
2. by some peer's local storage, or;
3. by a cache server (i.e. key-value store of Memcached), or;
4. by a persistent storage server (i.e. record store of a database).

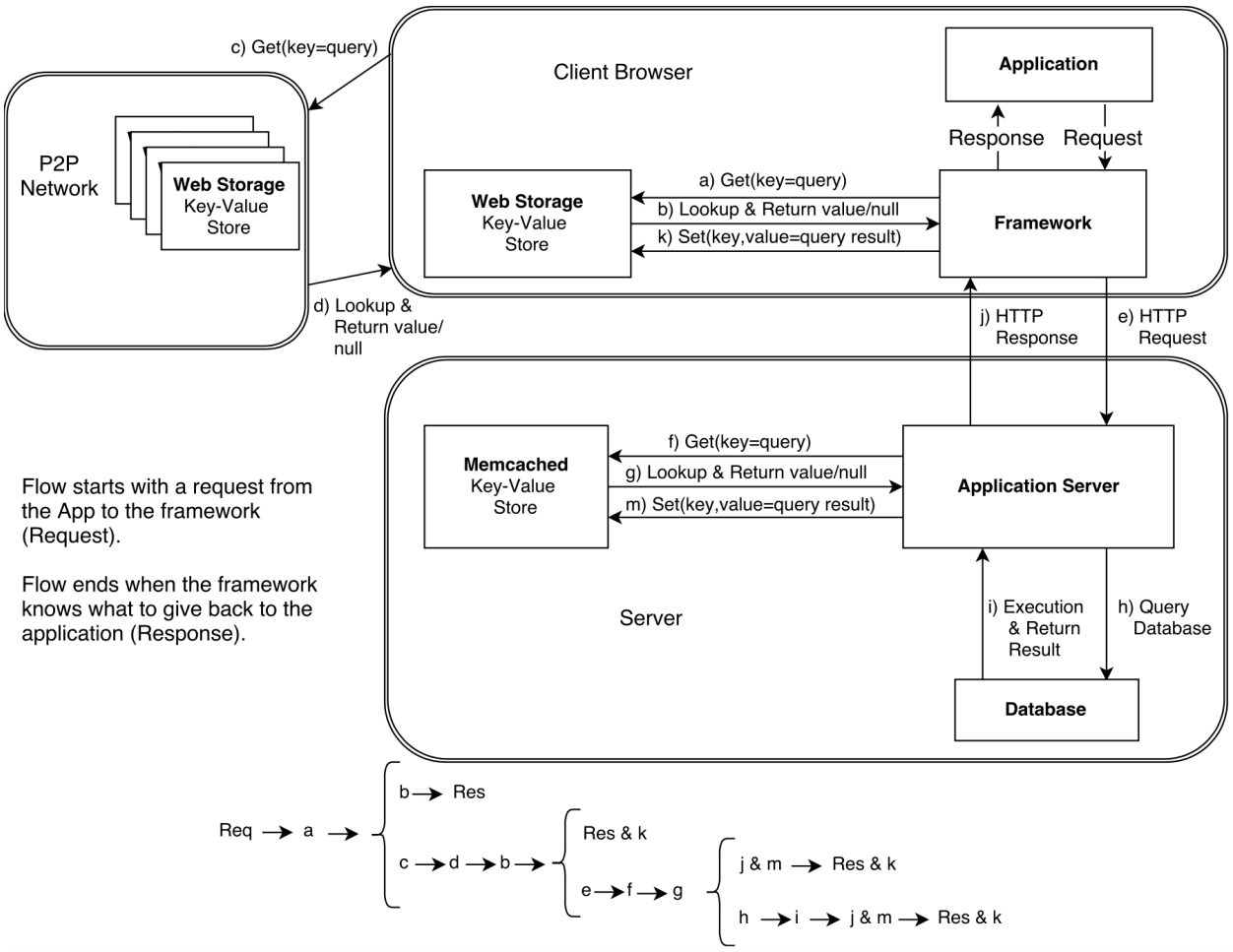

Figure 3.1: Design overview for the proposed solution

Regarding the scheme depicted above, 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. This favors a quick response to the client, by updating caches outside the critical path of the reply to the client.

In order to enrich the distributed cache system, this caching mechanisms of clients and server(s) must be adjusted. This 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.

In the peer-to-peer network users must exchange information to optimize caching decisions. Bloom filters can be used to get a peripheral view of users content over the peer-to-peer network, which would result in the use of a space-efficient probabilistic data

structure, for instance, to know if a peer (probably) has a certain object, or definitely does not. This can reduce complexity in searches and potentially improve end user perceived latency. Another aspect of the peer-to-peer network is the criteria used to choose the user's direct neighbors, which can be set in several ways. For example, if a user chooses his direct neighbors based on the amount of popular content that they have and/or by the amount of content that they have different from him. This can be achieved through appropriated heuristics based on hit rates and other performance metrics obtained within the peer-to-peer network.

An extra feature to integrate in this distributed solution is related with the use of a centralized mechanism (running in the server) to assist clients in establishing connections with other clients that have local cached content that can be useful. This would be helpful by informing users that there are one or more peer-to-peer networks that they do not know of, which potentially have content of their interest (based on previous requests). Therefore, they will be able to contact 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).

3.2 Work Plan

In this section the work plan is described for the elaboration of the dissertation. The work plan is to begin with the design of the solution in detail, then its implementation and testing, and finally, writing of the dissertation. To get an idea of the estimated time required to perform this work, in Figure 3.2 is presented the proposed work schedule.

Design: the main task is to define the solution, and the interactions between systems.

Implementation: this task can be split into two phases, without overlapping. The phases are as follows:

- **Phase one** - implementation of caching between users, including appropriate caching policies and refinements of existing ones. This is accomplished with the exploitation of the Legion framework's peer-to-peer networks.
- **Phase two** - implementation of caching between the users and the in-memory caching system (Memcached). This is performed through the control centers (ECP and CCP), which will coordinate and synchronize caching policies.

Testing and Evaluation: this task is essential to assert the effectiveness of the implemented solution, since rigorous and exhaustive tests and evaluation will be conducted, particularly comparing the existing solution with current centralized solutions.

Research Paper:: creation of an article that is directly related with this dissertation. It will be submitted to a national research conference, and (probably) to an international one, during the accomplishment of this dissertation.

Writing: the final task, which consists on writing the dissertation.

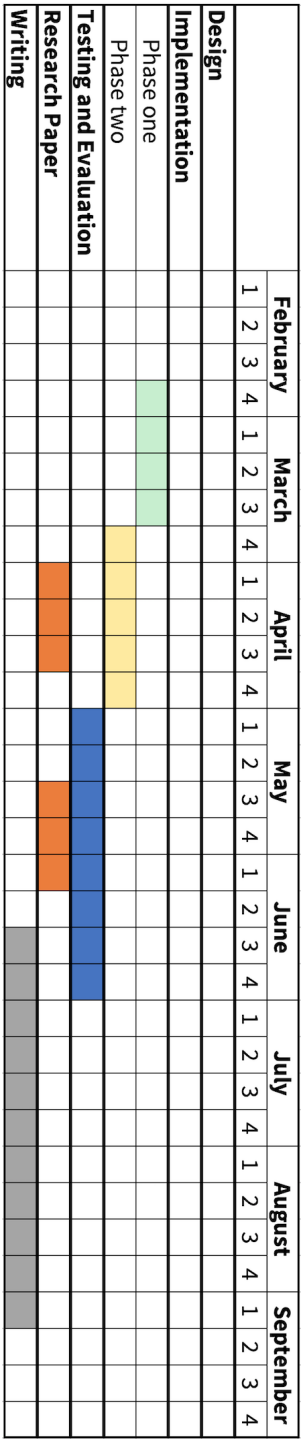

Figure 3.2: Proposed work schedule

Bibliography

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] *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.

[16] 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.

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

[18] 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.

[19] K. CHENG and Y. KAMBAYASHI. *A size-adjusted and popularity-aware LRU replacement algorithm for Web caching*. 2000.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

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

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

[29] E. C. A. T.-C. F.J. González-Cañete J. Sanz-Bustamante*. *Evaluation of Randomized Replacement Policies for Web Caches*. 2007.

[30] H. Y. FOONG A. P. and D. M. HEISEY. *Essence of an effective Web caching algorithm*. 2000.

[31] 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.

[32] 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.

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

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

[35] R. M. J. Yang W. Wang and J. Wang. *Access driven Web caching*.

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

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

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

[39] 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.

[40] 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.

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

[42] 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.

[43] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions". In: (2016).

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

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

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

[47] A. V. Murta C.D. and W. Meira. "Analyzing performance of partitioned caches for WWW". In: 1998.

[48] L. Z. NICLAUSSE N. and P. NAIN. *A new efficient caching policy for the World Wide Web*. 1998.

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

[50] 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.

[51] 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.

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

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

[54] 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.

[55] 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.

[56] 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.

[57] K. Psounis and P. B. *A randomized web-cache replacement scheme*. 2001.

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

[59] M. Ripeanu. "Peer-to-peer architecture case study: Gnutella network". In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. IEEE. 2001, pp. 99–100.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] I. Tatarinov. *An Efficient LFU-Like Policy for Web Caches*. Tech. rep. 1998.

[71] 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.

[72] 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.

[73] V. Valloppillil and K. W. Ross. *Cache Array Routing Protocol v1.1*. 1998. URL: http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-03.txt.

[74] 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.

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

[76] 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.

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

[78] 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.

[79] 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.

[80] 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.

[81] 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.

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

[83] I. R. R. D. Zhang J. and M. Ott. "Web caching framework: Analytical models and beyound". In: 1999.

[84] 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.

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

[86] 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.

[87] 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.