Filipe Alexandre Pereira Luís

Licenciado em Engenharia Informática

Distributed, decentralized, and scalable Coordination Primitives

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

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

Júri

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2018

Distributed, decentralized, and scalable Coordination Primitives

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

To my family and friends.

Acknowledgements

I would like to thank my advisor Prof. Dr. João Leitão and to my colleagues Albert Linde and Dr. Valter Balegas, which gave me all the support that i needed during this phase of my life. They gave me the opportunity to develop this work and closely contact with their experience and knowledge in this area. Despite all the adversities they do not let me down, encouraging me and making all this work possible.

My thanks go also to all my friends and to all of my colleagues in the computer science department for their support. Finally, i would like to thank all my family, especially to my mother, father, and my brother which closely contacted with me during the elaboration of this work. Their support, motivation and patience were crucial to my performance in this work.

Abstract

Nowadays, the Internet presents itself as a dynamic environment that changes according to the behaviour of its users. These users follow a centralized communication model with most of the services currently available on the Internet, since many of them only allow interaction with the servers. This distributed architecture paradigm brings some associated drawbacks, since in the eventuality of a failure (overload) in its centralized component (server) the whole service becomes unavailable.

This work addresses this problem by proposing to solve specifically the challenges associated with failures caused by exhaustion of resources. These happen, for example when the number of requests issued by clients is greater than the number of requests that the server (or the centralized architecture as a whole) can handle. To avoid these situations and make web applications more robust (greater availability and fault tolerance), this work proposes the construction of a system that implements coordination algorithms and mechanisms over a peer-to-peer model that allows direct communication between clients. It allows to achieve a more efficient and scalable management of the resources made available by the existing web applications.

Keywords: Coordination, resource management, peer-to-peer, web applications, availability, fault tolerance.

Resumo

Hoje em dia, a Internet apresenta-se como um ambiente dinâmico que se altera consoante o comportamento dos seus utilizadores. Estes seguem um modelo de comunicação centralizado com a maior parte dos serviços actualmente disponibilizados na Internet, uma vez que muitos deles permitem apenas a interação com os servidores. Este paradigma arquitectural trás algumas desvantagens associadas, uma vez que na eventualidade de existir uma falha (a exaustão de recursos) no seu componente centralizado (servidor) todo o serviço fica inacessivel.

Este trabalho vai de encontro a essa problemática propondo-se a resolver os desafios associados a falhas provocadas por esgotamento de recursos. Estas acontecem, por exemplo quando o número de pedidos dos clientes é maior que o número de pedidos que o servidor (ou a infraestrutura centralizada) consegue processar. Para evitar estas situações e tornar as aplicações web mais robustas (maior disponibilidade e tolerancia a falhas), este trabalho propõe a construção de um sistema que implementa algoritmos de coordenação em cima de um modelo peer-to-peer que permite a comunicação entre clientes. Permitindo assim obter uma gestão mais eficiente e escalável dos recursos disponibilizados pelas aplicações web actualmente existentes.

Palavras-chave: Coordenação, gestão de recursos, peer-to-peer, aplicações web, disponibilidade, tolerancia a falhas.

Contents

List of Figures **xv**

List of Tables **xvii**

Listings **xix**

1 Introduction **1**

1.1 Context 1

1.2 Motivation 2

1.3 Contributions 2

1.4 Document Organization 3

2 Related Work **5**

2.1 Distributed Systems Architectures 6

2.1.1 Centralized Architectures 6

2.1.2 Partially Centralized 7

2.1.3 Partially Decentralized 9

2.1.4 Decentralized 10

2.1.5 Relevant examples 13

2.1.6 Discussion 15

2.2 Communication Primitives 15

2.2.1 Message-oriented Communication 16

2.2.2 Stream Oriented Communication 18

2.2.3 Multicast Communication 19

2.2.4 Relevant examples 21

2.2.5 Discussion 22

2.3 Coordination Primitives 22

2.3.1 Mutual Exclusion 23

2.3.2 Elections 24

2.3.3 Consensus 24

2.3.4 Relevant examples 25

2.3.5 Discussion 26

2.4 Used Technologies 26

2.4.1 WebRTC 26
2.4.2 Legion Framework 28
2.5 Summary 29

3 System Architecture 31
3.1 Design Overview 32
3.2 Network layer 33
3.2.1 Overlay Network Logic - Architecture 34
3.2.2 Communication Module 40
3.3 Application layer 42
3.3.1 Request Capture protocol 44
3.3.2 Coordination protocol 45
3.3.3 Fault Tolerance Protocol/Resource owners management 47
3.4 Summary 49

4 Evaluation and Results 51
4.1 Network Layer Evaluation and Results 53
4.1.1 Message maximum hops Test 53
4.1.2 Join Stabilization Time Test 54
4.2 Application Layer Evaluation and Results 55
4.2.1 SMoodle vs TMoodle 56
4.2.2 SMoodle vs TQMoodle 60

5 Conclusion 63
5.1 Future Work 64

Bibliography 65

List of Figures

2.1 Client-Server Architecture . . . 7
2.2 Cloud Computing Architecture . . . 8
2.3 Geo-Distributed Architecture . . . 9
2.4 Peer-to-Peer Architecture . . . 11
2.5 Publish/Subscribe topic-based Architecture [44] . . . 18
2.6 Streaming communication Client/Server Architecture [58] . . . 19
2.7 Legion Architecture [36]. . . . 28

3.1 Client-server and Proposed model . . . 31
3.2 Proposed model design. . . . 33
3.3 Stable overlay with 4 peers. . . . 36
3.4 Overlay after peer 15 joins the overlay. . . . 37
3.5 Stabilization process after peer 15 joined the network. . . . 37
3.6 Overlay after peer 15 effectivelly joins the network. . . . 38
3.7 Overlay after peer 20 leaves the network. . . . 39
3.8 Overlay after network stabilization. . . . 39
3.9 Network with neighbours connections. . . . 41
3.10 Changes made in Moodle root directory . . . 43
3.11 Request Capture protocol flow . . . 45
3.12 Coordination mechanism example illustration . . . 48
3.13 Evolution of network state at join process . . . 48
3.14 Evolution of network state at leave process . . . 49

4.1 Experimental scheme . . . 52
4.2 P2P Networks formed during the experiments . . . 52
4.3 CPU usage of tests SMoodle and TMoodle . . . 57
4.4 Number of requests of tests SMoodle and TMoodle . . . 57
4.5 Page load latency of tests SMoodle and TMoodle . . . 58
4.6 CPU usage of tests SMoodle and TQMoodle . . . 61
4.7 Number of requests of tests SMoodle and TQMoodle . . . 61
4.8 Page load latency of tests SMoodle and TQMoodle . . . 62

List of Tables

3.1 Network state I 36
3.2 Network state II 36
3.3 Network state III 38
3.4 Network state after peer 20 leaves. 39
3.5 Network state after stabilization. 40

4.1 Message maximum hops 54
4.2 Join Stabilization Time results 54
4.3 Moodle results (SMoodle) 58
4.4 Throttle allowing 1 client per resource (TMoodle1) 58
4.5 Throttle allowing 3 clients per resource (TMoodle3) 58
4.6 Throttle-Queue allowing 1 client per resource (TQMoodle1) 60
4.7 Throttle-Queue allowing 3 clients per resource (TQMoodle3) 60

Listings

3.1 index.html 43
3.2 Request Capture Protocol 44
3.3 Coordination Structure 46

Chapter 1 Introduction

1.1 Context

Nowadays the Internet is not based on sharing static pages hierarchically organized from the home page of a website as it was once. Today, the Internet has become a dynamic environment which changes depending on the user behaviour. This environment is leveraged by the advances suffered on browsers which help to build more interactive and powerful web applications. In the same line of thought, the evolution of HTML also contributes to the choice of web applications in detriment of desktop applications. In both approaches, the interaction with servers is strictly necessary, since direct communication between users is not typically possible in the context of web applications. This behaviour, in web applications, can be contradicted by some recent API's such as WebRTC [56], which enables clients to act as a peer in a distributed peer-to-peer architecture effectively creating a browser-to-browser network. This has been shown in practice in the design of Legion framework [35, 36] that provides abstraction to enable direct interactions among clients running on user's browsers.

Decentralized architectures have gained prominence over more centralized architectures and many examples can be found in current and past peer-to-peer systems and systems that delegated some functionalities to a peer-to-peer architecture. They have aroused much interest and are currently used in several areas, such as telephony, file-sharing, streaming media, and also in volunteer computing. The most prominent and known systems that leverage it (or did so in the past) are Skype [8], BitTorrent [15], Napster [42], and BOINC [4].

The combination of powerful centralized components with peer-to-peer approaches can be very interesting and has not yet been much explored in the past. For instance, web applications could take advantage of peer-to-peer architectures in applications that need coordination to avoid having their (centralized) resources exhausted while saving money and maintaining their availability for clients.

1.2 Motivation

Typically the architecture of web applications is based on the client-server approach in which clients are mostly browsers and most of the times every coordination or communication required by clients must be performed by or through the server. In order to ensure the continuous operation of a web application, the server must be prepared to cover all the possible surges in user's activity. Resource allocation is responsibility of the service provider and often results from a bad estimate that does not address peak usage periods, which might lead the system to become resource exhausted and ultimately fail. The alternative of over-provisioning resources, can lead to waste of computational power and money.

Existing technologies, such as peer-to-peer systems and existing distributed coordination primitives offer the opportunity for the construction of solutions that allow users to directly coordinate their actions in order to never exceed the estimated capacity threshold of the centralized infrastructure resources, which would enable service providers to save money and resources. Ensuring that in the case that a user wants to access one of the resources of a service, it will eventually access it without exhausting server resources.

1.3 Contributions

This work proposes to design, build and evaluate a coordination system using as its foundation a novel distributed architecture to support web applications, which offers properties similar to that encountered today in a cloud architecture, however with lower monetary cost for application operators. The main contributions of the system are the following:

- Logical peer-to-peer network which enabled coordination between clients (i.e., browsers), based on a DHT design. Designing such overlays in the browser domain is a significant challenge due to the limitations of WebRTC [56] interactions.
- Coordination protocols used to control surges in clients activity, preventing failures caused by resource exhaustion in the centralized component (i.e., server).
- An experimental evaluation of the system, comparing with typical client-server architectures, based on a simple use case using Moodle [41] as web application.

In this coordination system, users communicate in a peer-to-peer fashion, coordinating the access to the centralized component of Moodle's web application [41]. In a general mold, our system captures all POST requests made by users to Moodle's web application, controlling the access rate of users in periods of major activity over the web application. With this requests coordination, our system reduce CPU load in the main server, mitigating exhaustion of centralized resources (i.e., Moodle's server).

1.4 Document Organization

The document is structured as follows:

Chapter 2 - Related Work - This Chapter begins with an overview of the existing architectures, being followed by existing communication and coordination primitives found in the literature, which are a fundamental aspect of the work proposed in this document.

Chapter 3 - System Architecture - Describes in detail the architecture of the developed system. Starts by describing the design overview of the developed system. We then divide the developed system in two layers, the network layer which explains in detail the peer-to-peer network architecture; and the application layer which explains how this work coordinates the access to the centralized component.

Chapter 4 - Evaluation and Results - This chapter explains in detail experimental evaluation, where we compare our peer-to-peer leveraging design to the use of a typical client-server architecture over Moodle's web application [41].

Chapter 5 - Conclusion - This chapter presents the main conclusions of this work, and also gives a brief preview of the work to be developed in the future.

Chapter 2
Related Work

This chapter presents and discusses relevant related work. In order to better understand the origin of lack of reliability and availability in centralized architectures such as the mentioned on Chapter 1 and the proposed solution presented on Chapter 3, which uses a peer-to-peer network formed by clients to avoid the exhaustion of resources.

The next Sections cover the following three topics:

- 2.1 **Distributed Systems Architectures:** This Section discusses the spectrum of distributed systems architectures starting on centralized architectures and finishing on completely decentralized ones. We discuss in detail client-server, cloud computing (partially centralized and partially decentralized), and peer-to-peer models, since they are the classical (and relevant) approaches to design distributed systems.

- 2.2 **Communication Primitives:** This Section deals with communication primitives that make interaction between two or more components in distributed systems possible. Communication is an essential aspect of any distributed architecture, and hence understanding the different existing alternatives is essential to build adequate distributed and scalable coordination primitives.

- 2.3 **Coordination and Agreement Primitives:** This Section covers existing coordination primitives and techniques that have been proposed and are correctly used in the design of distributed systems. We also discuss why existing approaches are unsuitable to achieve the goals of the work to be conducted in the context of this thesis.

2.1 Distributed Systems Architectures

The evolution of hardware, software, and network infrastructures allowed the development of more complex systems. Nowadays users expect an ubiquitous access to systems, which require to become increasingly fault-tolerant, available, resilient, and scalable. Due to the pervasiveness of computers on everybody's life, the need to make them more resilient and available has became even greater. Thus, the architectures of distributed systems had to evolve from more centralized designs to more decentralized ones, which is an important aspect to lower the dependency on centralized single points of failure and contention points, which is essential for improving fault-tolerance and availability.

We start by discussing client-server model that is the main representative of centralized architectures. Moving then to the cloud computing design that is able to spread server operations across more machines improving availability, scalability, and fault-tolerance.

Cloud-based architectures where all resources are in a single geographic location can have latency issues, for instance if only Asia had server infrastructures for a determined service, the clients in America would experience more latency than the clients in Asia. To overcome this latency issue we discuss geo-distributed cloud-based architectures, where servers are spread across different geographical areas, bringing them closer to clients. We finally address more descentralized architectures, where clients interact directly with each other avoiding central components. This type of architectures make systems more resilient, fault-tolerant, scalable and available. We will focus on these type of architectures in order to solve the challenges presented on Chapter 1.

2.1.1 Centralized Architectures

The more traditional and simple centralized architecture is the Client-Server model, that can be viewed as two main separate entities as the name itself indicates, Clients and Servers. From the Client perspective, the system allows it to access a resource or a set of resources; and from the other perspective, the server is responsible for processing and managing access to the provided resource or resources.

This model is characterized as centralized due to the fact that the control of the system fully resides on the server side. The server provides a service and all users that want to access that service must interact with it. A simple illustration of this model is presented in Figure 2.1.

In this model clients are unable to interact directly with each other, in fact each individual client is typically not aware of other clients, thus they can only interact indirectly, with each other through the service provided by the server. The scalability of this architecture is limited when the number of clients is greater than the number that can be handled by the server.

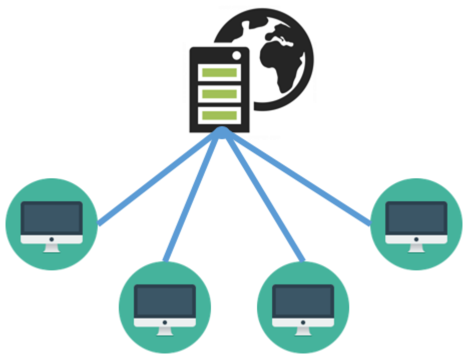


Figure 2.1: Client-Server Architecture

Clients and server communicate with each other through a server provided API. This API lists all available server methods and specify the type of reply that is emitted by the server. This API is accessed by clients using some communication primitives. In the next Section we discuss such communication primitives.

A main disadvantage of this model is that the server is a single point of failure, so when it becomes unavailable the whole system/service becomes unavailable. Next, we discuss how cloud computing paradigm addresses this problems.

2.1.2 Partially Centralized

We classify cloud computing architectures (with all servers in the same geographic location) in partially centralized systems because their organization is logically centralized, within the scope of what we call cloud. Computations are distributed over more than one machine in opposition to classical client-server model. Rather than having a single machine serving one or more clients, in cloud computing we have a centralized component (cloud), where all users are able to connect. Although client-server and cloud computing are two distinct architectures, the clients are unable to distinguish them.

This centralized component is at the same time distributed, since clouds are typically implemented on cluster's[1] to provide properties such as resource elaticity, high performance, ubiquity, availability, and fault tolerance [16]. An illustration of this model is presented in Figure 2.2.

The elasticity of resources allows developers to adapt their needs with a better cost-benefit trade off. They do not need to be concerned about resource overprovisioning, when their service does not have the expected popularity or resource underprovisioning when they did not predict a service sudden increase in popularity for their service. They are not required to know anything specific about cloud computing design to connect their computers to cloud server and use it. Thus, it is possible to develop and test their applications and services faster, without wasting costly resources with this architecture

[1] set of interconnected computers that cooperate closely to provide single and high performance computing capability.

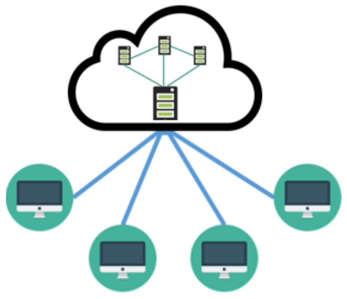


Figure 2.2: Cloud Computing Architecture

design. Developers are able to choose how much resources they need as the service grows, often called the pay-as-you-go model [6].

The virtualization of the resources made by the cloud computing model allows the elasticity of resources previously discussed. This virtualization is achieved through software that makes possible the interaction and coordination between computers, masking their physical separation [51].

This resource virtualization aspect is very important since for instance, if we have 2 machines where only 30% of each one capacity is being used, we committed an overestimated error and resources are being wasted. In cloud architecture we easily overcame this problem by virtualizing two machines over a single physical machine achieving 60% of utilization without requiring the other machine to be wasting resources.

Cloud service providers offer certain service guarantees, called service-level agreements (SLA). Typically, the SLA is a service agreement between client and service provider that includes the quality, availability, and responsibility of the service being offered by the provider [57]. For instance, the Amazon S3 plan [2] of Amazon Web Services (AWS) offers a storage service that can be up to 5 terabytes in size and is committed to a monthly uptime percentage greater than 99% of the time. Anything below this guarantee leads Amazon to pay to the service user 25% of his bill [3] as a compensation.

Cloud computing also offers ease of setup, since it is possible to easily configure a system by outsourcing computations to cloud service providers, instead of maintaining computational infrastructure and managing complex software stacks [25].

We conclude that this type of cloud architecture differs from the classical client server model, since each offers different properties. For instance, in the client-server model, if the server fails, the service provided becomes inaccessible, and clients are unable to access it, making the service useless, the same does not happen in the cloud model. Cloud architecture easily masks this type of failures and migrate the service to another machine, making the system more fault-tolerant. We consider that this property is an evidence that the cloud architecture has a distributed component, categorizing it in the partially centralized designs.

2.1.3 Partially Decentralized

According to Jakob Nielsen [47], users may notice the delay of any response from servers that takes more than 100 milliseconds. Anything below 100 milliseconds will create the illusion that the system is reacting instantaneously to clients inputs. Systems based on Cloud Computing that maintain their hardware components in a single geographic location (as partially centralized architecture) can present latency values above 100 milliseconds for users that are distant to the data center location. In order to reduce perceived latency, multiple cloud servers could be placed geographically closer to the clients, which is commonly referenced as geo-distribution. Additionally, if we want to offer access to data with lower latency regardless of their location we can replicate it among the cloud sites which is defined as geo-replication.

For instance, Google Drive service uses geo-distribution providing lower latency. They also implement geo-replication, since users that access this Google service in Europe servers observe the same content as if they access it through servers located in America. Thus, adding the geographic arrangement of servers and the replication of their data we can get low latency in multiple locations. The model of geo-distribution is presented in Figure 2.3.

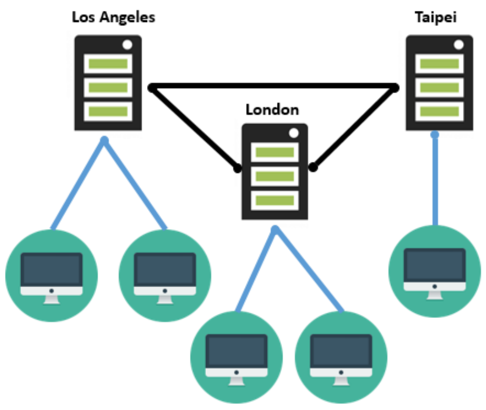


Figure 2.3: Geo-Distributed Architecture

We classify cloud computing architectures which implement geo-distribution as partially decentralized models since the system is composed by multiple servers distributed across the globe. However we do not consider it as fully decentralized because users must interact with the central component (data center servers) in a logical centralized location.

Placing servers geographically near to the clients reduce response latency, and data replication can provide enhanced performance, high availability, and increased fault tolerance, particulary to catastrophic failures that render a datacenter inoperable or inacessible [16]. Typically, the majority of distributed systems aim of having these properties, since that makes them more robust while providing better user experience.

2.1.4 Decentralized

In order to overcome the limitations of centralized architectures engineers proposed alternative architectures that evolved from the client-server model. This lead systems to start transitioning from a centralized to a decentralized model, in order to make them more fault tolerant, available, and scalable. For instance, to overcome single points of failure or availability issues, models were introduced that distribute the service through more servers. And to overcome performance issues, models choose to spread computations over more machines, simulating an entity with more computational power. This Section presents and discusses the main aspects of decentralized systems, more precisely, the architecture of Peer-to-Peer systems.

Peer-to-Peer (P2P) is considered a promising model that focus on exploitating existing resources at the edge of the Internet. These resources include computation, storage, and bandwidth, with costs handled by end-users and embracing at the same time many desirable properties, as scalability, availability, and autonomy [55]. An illustration of this architecture is presented in Figure 2.4.

The interest on P2P systems was significantly influenced by the Napster [42] music-sharing system, the Freenet [14] anonymous data store, and the SETI@home [5] volunteer-based scientific computing projects in 1999. They are mainly used for sharing and distributing files, streaming media, telephony, and volunteer computing [49].

In [49] P2P architectures are characterized by three main aspects:

- **High degree of decentralization:** Each participant acts as server and client at the same time, distributing server computation, bandwidth, and storage consumption across all nodes. The state and tasks of the system are allocated over peers and few, if any, dedicated nodes

- **Self-organization:** The system is able to adapt to new joining nodes, with little or no manual configuration needed. The same is true for nodes that depart or fail.

- **Multiple administrative domains:** The participants of the system are typically managed and owned by individuals which voluntarily join the system.

Descentralized solutions are also desirable due to their organic growth, as the resources are contributed by peers, meaning that an increase in the number of users in the system does not require a continuously infrastructure upgrade; their low barrier for deployment compared with client-server systems; the investment needed to deploy a P2P service tends to be low; resilience to faults and malicious attacks; and its diversity of systems as resources tend to be diverse among all participants.

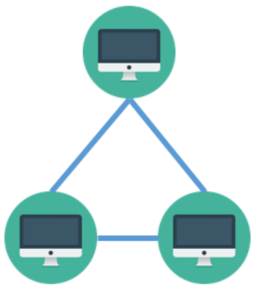


Figure 2.4: Peer-to-Peer Architecture

P2P systems can be classified in two different categories in terms of decentralization degree, partially centralized and fully decentralized. To categorize them we must take into account the presence or absence of centralized components in their designs.

Partially Centralized P2P Systems

In this type of P2P systems, there usually exists a central component similar to the client-server model (discussed in Section 2.1.1), where one or more dedicated controllers help peers to locate their desired resources, or act as task schedulers to facilitate coordination actions among clients [55].

The existence of centralized components can make this type of systems more simple to build and maintain. Since the information about resources is maintained by one or few dedicated controllers, which is more easy to manage compared with a fully decentralized model. It also has some drawbacks, as the presence of potencial bottlenecks on controller nodes and potential failure points on these components. For a very large number of peers, the dedicated controllers are not able to manage all the requests making the system slower, depending on how fast they can respond to peers. This issue of scalability is raised and this type of P2P architecture must not be recommended for very large applications. Some examples of this model are Napster [42], that relies on a central server for peer discovery and content lookup [55], and BOINC [4], which has a central server that maintains information about applications, platforms, versions, work units, results, accounts, etc. BitTorrent [15] is also an example of a partially centralized P2P system, since it relies on central servers that index contents distributed across the network.

Fully Decentralized P2P Systems

Unlike partially centralized P2P systems, this design completely avoids the use of centralized nodes for supporting special tasks. Thus, the state of the system and even information about the system membership is distributed across a logical network connecting all nodes. Each peer plays the same role, with each one having the same rights and responsibilities. This distributed approach has no native bottlenecks, having the potential to be more resilient to failures while being more scalable, since centralized components can have their resources exhausted. Additionally, coordination in these fully decentralized systems is more difficult to achieve, for instance if participants must reach any form of consensus. Algorithms such as Paxos [28, 30], that helps participants of a system to reach consensus (which are discussed later on Section 2.3) are known to have low scalability and being sensitive to continuous/frequent membership changes. An example of a fully decentralized system is Gnutella [48], where there is no central authority controlling the system organization and all the participants connect directly with each other [55].

Overlay Network

An overlay network is a logical network (e.g. operating at the application level) that enables peers to communicate with each other directly. These networks can be classified as unstructured or structured, according to the constraints imposed on the topology formed by the links among peers. In terms of overlay structures, there are flat architectures (single-tier) where peers are all at the same (logical) level and hierarchical architectures (multi-tier) where peers are organized into groups and each group has one or more intra-groups [55]. For instance, in flat designs the lookup process can be supported by any set of nodes that are part of the system, such as Chord [52] discussed further ahead. In hierarchical models, a search by the group of the target peer for that key it is made at the top-level overlay and are mostly supported by peers operating at that level. However, sometimes it is necessary to find the peer responsible for that key in the intra-group level. An example of these models is Crescendo [22], which merges Chord rings in multiple layers and routes through them hierarchically.

Unstructured Overlays

This type of overlay avoids the use of any specific arrangement structure among peers, effectively generating overlay topologies that are random.

When a node wants to find some resource in the network (e.g. a file), he must know which peers have that data. In order to obtain such information, a typical solution is to flood the overlay with a query. This search behaviour causes a tremendous message overhead in the system, since the messages are sent to all peers (most of the time unnecessarily). Also, this type of overlays is more vulnerable to malicious flooding attacks, when malicious nodes floods the network with queries. Many of them are very difficult to detect since these operate at the application level [55]. Applications that use this type of overlay include FreeNet [14], that forward requests node by node (unicast based) until a target is reached and returns a reply through the same (inverse) path, and Guntella [48], that uses flood-based techniques. This communication techniques are discussed in detail in Section 2.2.

Structured Overlays

In opposition to unstructured overlays, in structured approaches, the overlay logic imposes some constraints on node connections. These constraints shape the structure of the network graph and therefore it is common to speak about a specific structure, as its name suggests.

Typically in this type of overlay, each node has a unique identifier (chosen using a policy that makes them uniformly distributed over a key space) that identifies him in the network. This identifier is many times used to govern the arrangement of the overlay topology (e.g. a ring that respects the ordering of these identifiers). Thus, each node can know where another node is on the structure and, in opposition to the random organization strategy, target resources (that have a unique well known identifier) can be found easily using for instance, consistent hashing [49]. Examples include Chord [52] and Pastry [50] that use key based routing mechanisms to efficiently locate any desired content.

2.1.5 Relevant examples

Chord [52] - is a protocol that addresses the problem of lack of efficiency on finding the node that stores a particular data item in peer-to-peer applications. It is a fully decentralized architecture since all nodes play the same role, having all the same responsibilities over the attributed keys. It can be mapped on structured overlays since each chord node has an unique identifier based on node's IP address and maintains a link to its successor[2]. As the node with lower identifier points to the one with higher identifier, a ring topology is formed.

To achieve fault-tolerance, each node maintains a list of its first r successors nodes. When their direct successor does not responds, he contacts the next successors in order until one responds. Assuming p as the probability of a peer to fail, p^r is the probability of all peers in that list to fail simultaneously. Thus, increasing r makes the system increasingly more fault tolerant. In order to achieve efficient lookups, a finger table is stored on each node n. A finger table stores at most m entries, where m is the number of bits in the key/node identifiers. Each entry is a successor node of n, that succeeds n (on the node identifier space) by at least 2^{i-1}, where i corresponds to i^{th} entry on the finger table and $1 \leq i \leq m$. Thus, each node is aware of their nearest successors and can traverse the graph not only walking one peer at a time, but jumping through his finger table entries, increasing lookup efficiency.

[2]The next node in the ordered ring with the lowest identifier of the set of identifiers larger than the local node.

Gnutella [48] - Gnutella is a peer-to-peer decentralized protocol that builds an overlay network, typically used to find files shared by each peer. Its overlay network is characterized as unstructured since nodes are only aware of peers to which they are directly connected through TCP connections, which are established at random. Queries in this type of overlays are disseminated using a flooding technique, in which a source node propagates the query to his connected nodes, and the connected nodes propagate the query among their connections and so on. The response, when ready, is propagated back up to the source node. This type of query dissemination can lead to some lack of security, since Distributed Denial of Service (DDOS) attacks are easy to be performed. As referred before, decentralized overlays improve fault tolerance since they are not dependent of a single point or component of the system.

Skype [8] - Skype is a system mainly used for VoIP communication between users. In its early version, Skype had a peer-to-peer registration system, where each user should be registered to be able to communicate with other registered users. This registry system was a partially centralized network, since not all nodes had the same responsibilities. It had two types of nodes, ordinary hosts and super nodes. An ordinary host is a general user of the client Skype application, additionally the super nodes are peers with higher bandwidth, computation power, and memory. This design had login servers used for storing names and passwords of users. The requirement of hosts to be connected to that type of servers and to a super node also complement this partially centralized architecture. This mandatory connections generates a specific structure, thus this architecture relied on a structured overlay. Fault tolerance in this peer-to-peer architecture could be achieved increasing of the number of super-nodes.

Diamond [59] - Diamond is a recent cloud storage service that provides reactive data management and reliable synchronization across several devices. When a node updates a piece of data, these changes are automatically propagated though other nodes.

Diamond architecture adopts a cloud computing design, in which clients interact with cloud servers through Diamond's library. The Cloud component consists in a Key value store database, which employs strategies as replication and partitioning to achieve fault-tolerance and scalability. The clients are connected to stateless front end servers through Diamond's library. Additionally, this front end servers are connected with the key value stores servers. The architecture presented by Diamond includes four main components, reactive data types (RDT), reactive data maps, read-write transactions, and reactive transactions. The reactive data types are application data structures that are shared and persisted through Diamond. The reactive data map allows developers to link their RDT's and application with data keys and the diamond key value store. Read-write transactions are used to update shared RDT's, with ACID guaranties. Finally, reactive transactions are used to propagate shared application variables into local variables, making them visible to users on their own devices. This architecture allows the absence of notification mechanisms and reactive code mechanisms at server-side, since application reactive code is in the client side.

2.1.6 Discussion

In this Section we presented essentially four groups of architectures which could be used by this work. Section 2.1.1, presented an architecture based on Client-Server paradigm where all clients interact with a centralized component (server) responsible to control all the system. These architecture major drawback is its server since when it fails, all system becomes unavailable for its users. Another problem arise when the number of client requests is higher than the number of requests supported by the centralized server. Due to the presence of these points of failure, we rejected this architecture type to give support to this work. Since centralized architectures were out of our solution scope we tried to evolve to decentralized ones, Section 2.1.2 and Section 2.1.3 appear as boundary between these of opposing architectures (centralized and decentralized). However these boundary architectures are also not suitable for this work, because despite being scalable and fault tolerant (**Diamond** [59]) they do not allow interactions between clients which is an important aspect of this work.

We choose decentralized architectures (P2P) due to their common properties such as scalability, fault tolerance, high degree of decentralization, self-organization, and allow interaction between clients. In this architectures we choose structured (**Chord** [52]) in detriment of non-structured overlays because we need to delegate special privileges to a couple of peers (explained in Chapter 3). Eventually these special peers will need to be located and structured overlays are known for its efficient lookup mechanism (key based routing). We discard systems like **Gnutella** [48], which does not enjoy an efficient lookup mechanism since it has an unstructured overlay; and systems like **Skype** [8], that despite having a structured overlay relies on super nodes that depend on users hardware, potentially compromising the network structure.

2.2 Communication Primitives

The communication between two end-points in the network is built on top of two fundamental transport protocols, UDP and TCP. Both are used to transmit information between two different points connected by a directed channel (at the IP level), using the sockets intefac [16]. In this Section we focus on protocols that are build on top of these primitives, since they offer a higher level of abstraction than the support offered by the interface of the transport layer.

The exchange of messages between processes is easily achieved by implementing, send and receive operations in both processes. In order to communicate, the sender process invokes the send operation on a byte chain over the communication channel, and the receiver invokes the receive operation on the channel in which the message was sent.

In this communication process, both participants follow one of two different approaches in terms of blocking policies. They could be synchronous, in which the sender process blocks until the receiver response arrives, or asynchronous, in which the sender process sends the message and proceeds without receiving any response from the receiver [54].

We divide this Section into message-oriented communication, which is based on the exchange of discrete messages between processes, in stream oriented communication, which is based on continuous message exchange, and finally, in multicast communication that is the paradigm most used in the context of group communication.

2.2.1 Message-oriented Communication

Many distributed systems are based on message oriented communication, which is based on the exchange of discrete message units, one at a time, and unrelated with the others. In this communication type, the message queue and the event based messaging are the two most representative approaches that will be discussed next.

Message Queuing communication

Typically, message queuing communication models are point-to-point asynchronous services which enables persistent communication between two end-points. Thus, all the messages involved in the communication process are stored and both participants do not need to interact with the message queue simultaneosly. The sender have the guarantee that his message will be enventually inserted in the receiver's queue. This queue will be responsible for storing the incoming messages when the receiver is not connected to the communication channel [16].

They can be implemented in each application, one in the sender and one in the receiver, or can be shared by both applications. Thus, the sender and the receiver are decoupled in time, which would not be possible if the communication was transient/volatile. In opposition, in the transient communication approach, the participants need to be on-line simultaneously to exchange messages, since no message store mechanisms are offered to manage non delivered messages. Sockets are example of transient communication, since when one of the end-points is down, the messages that are still transversing the transmission channel are lost [16].

A simple interface of a message queuing system only include this four primitives:

- Put - Appends a message to a queue.
- Get - Removes a message from the queue following a specific policy (e.g. FIFO, priority pattern, match pattern) being typically a blocking call, since it will block if

the receiver's queue is empty.

- Poll - Non-blocking version of Get, since it will return immediately if the receiver's queue is empty.

- Notify - Mechanism invoked when a new message is placed in the queue.

In other systems, message queue mechanisms can be implemented following a centralized or a decentralized architecture. The centralized approaches typically implement a centralized queue manager, where all the messages of the system are managed. Obviously, this approach have the same advantages and disadvantages of a centralized system, due to single point of failure and inherent bottlenecks in that queue manager. Additionally, the decentralized implementations distribute the queues in order to overcome these problems. An example of the latter approach is the Java Message Service (JMS) [24] where clients remove messages from the queues assigned to hold their messages.

Event-based Communication
Event-based [38] or Publish/Subscriber [7] systems are composed by publishers, subscribers and an event dissemination system. Publishers are responsible for publishing events to the event dissemination system, and subscribers are responsible for declaring their interests regarding events published in the event dissemination system. The event system can be topic-based, in which events are published to topics and subscribers receive the messages that correspond to the topics subscribed, or content based, where subscribers are responsible for declaring their interests considering properties of events (e.g. time, size, etc) and only the events that match their interest specifications are delivered to them.

This system is an implementation of one-to-many communication approach since one topic can be delivery to many subscribers as represented in Figure 2.5.

Similarly to message queues, event-based systems can also be implemented using a centralized or decentralized architecture. In the centralized models, the publishers produce events into a single event manager, and subscribers consume the events from that central manager. Contrarily, store and forward mechanisms can be implemented in the publishers and subscribers, distributing the event manager load and eliminating the single point of failure, and the inherent bottleneck.

Systems such as MEDYM [26], choose to distribute the event manager entity among several servers. Each server maintains a data structure with the subscriptions and only events that match their requirements, will be sent to this server. Choosing this mechanism in opposition to fully replicate servers, allows to minimize event traffic load on that servers, since only the interested ones will receive the event.

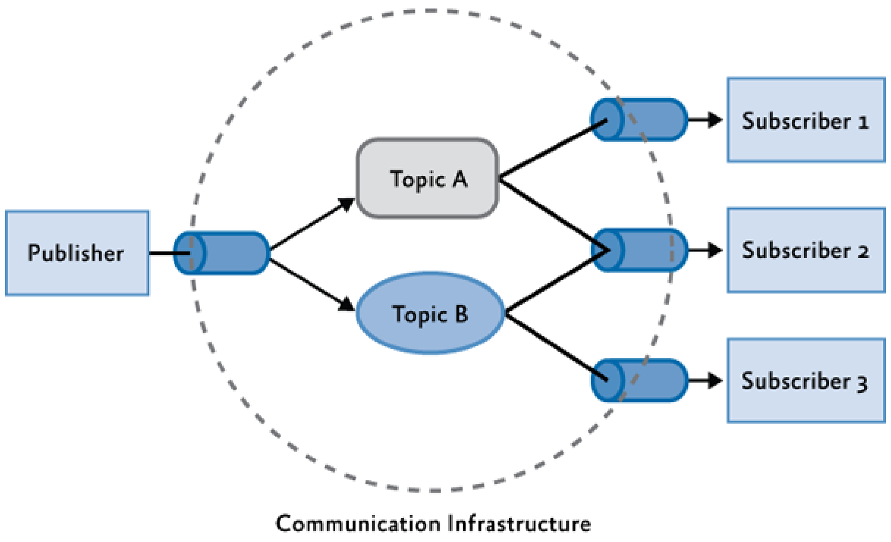


Figure 2.5: Publish/Subscribe topic-based Architecture [44]

2.2.2 Stream Oriented Communication

This communication approach is typically associated with multimedia applications, since a continuous generation and consumption of data is the normal behaviour. In these approaches, the time plays a crucial role, since all messages are related in time and must be ordered. For instance, in a live streaming audio player application, the message content must be reproduced in the same order that it was produced by its original source, otherwise the music will not sound as it should. Thus, messages delivered out of order or too late are ignored leading to errors in the reception of the data stream [16]. The TCP is a transport layer protocol that implements this type of communication, since the destination receives the message with the same order as the source send it. However, TCP cannot offer guarantees related to delivery times of messages.

These continuous transmission of data operates in three distinct modes [54]:

- **Asynchronous mode:** There are no time restrictions on point-to-point transmission. For example, when transferring a file, it does not matter whether it takes more or less time to transfer the file than to produce it.
- **Synchronous mode:** There exists an upper bound imposed on transmission time delay, usually called maximum end-to-end delay. For instance, in a live video streaming scenario, it is important that the transmission delay of messages to be constant and below the rate of production of the content, otherwise the replay will have frequent interruptions.
- **Isochronous mode:** Imposes an upper and a lower bound to the transfer time between the participants, entitled as minimum and maximum end-to-end delay.

Centralized implementations of these communication abstractions resort to servers which have the constraints and drawbacks associated with centralized solutions, such as single points of failures, bottlenecks, etc. Although, some work has been done to overcome the referred centralized problems as the case of bottleneck reduction presented in [13, 20].

The centralized approach referred in [58] is based on streaming servers supported by operating and storage systems which support continuous media storage and synchronous transmissions. The clients implement specialized algorithms for audio and video decoding. Besides that, both end-points implement QoS(Quality of service) controllers and the necessary transport layer protocols to communicate over the Internet. An illustration of this model is presented in Figure 2.6.

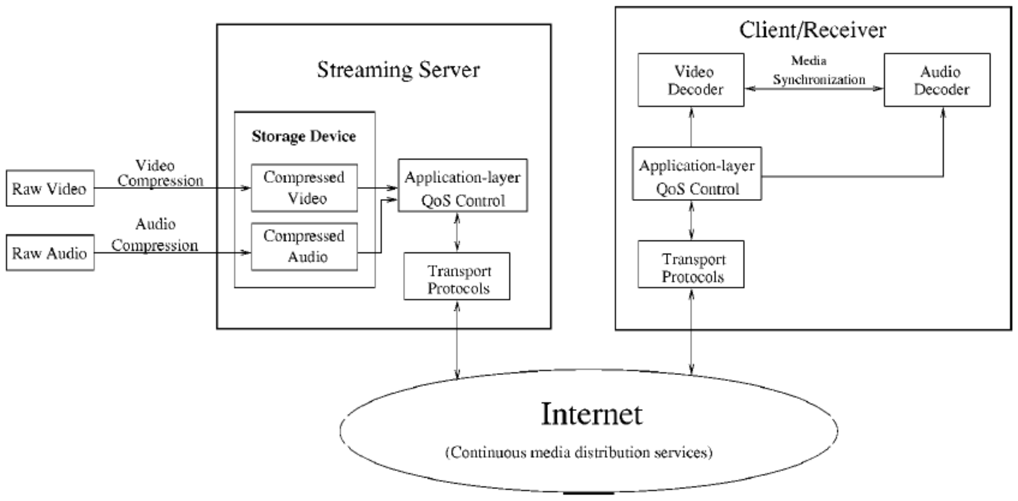


Figure 2.6: Streaming communication Client/Server Architecture [58]

Due to centralized architectures drawbacks, more decentralized architectures gained popularity. Thus, the use of peer-to-peer networks has contributed to the evolution of systems that use multimedia data diffusion thanks to their scalability and ease of deployment. Gnustream [27], is an example of a peer-to-peer media system that uses Gnutella [48] for routing media files between peers.

2.2.3 Multicast Communication

Multicast is a communication abstraction that follows the pattern of one-to-many communication, since a message is sent from one process to a group of other processes. In its simplest materialization, multicast communication does not provide guarantees of message ordering or delivery [16]. Additionally, other multicast protocols are used in systems that require stronger guarantees, as reliable multicast and ordered multicast.

Reliable multicast ensures that the messages sent will be delivered to all group members or will not be delivered to anyone, a property often called atomic delivery. Ordered multicast ensures that messages will be delivered to all group members in the same (relative) order they were sent.

Multicast is used in multiple systems from the discovery of services on the Internet, to the propagation of events. For example, it is used in publish subscribe systems, typically as a mechanism to support the notification of subscribers. To make the dissemination of information possible, this type of communication uses overlay networks, which can take the shape of a tree or a mesh [18]. An overlay organized as a mesh offers more robustness to the communication system since, when a node fails, there are more possible ways to disseminate information. This does not happen in the tree structure, forcing its reconfiguration [54]. In a tree based approach, the redundancy of messages does not exists, making this type of arrangement less fault tolerant, although having a better resource usage. Protocols such as that are presented in [32], combines both approaches. It uses a tree-based approach to achieve low message overhead and a gossip based mechanism, organized into a mesh, in order to provide better fault tolerance by transmitting redundant control information to assist on masking node failure and message omissions in a timely fashion.

Gossip Data Dissemination

This data dissemination technique reproduces the behaviour of the communication used by people in social interactions and also the dissemination of diseases in population, hence these approaches are also called epidemic-based protocols. Instead of disseminating diseases, this protocol spreads messages (i.e., information) through the network with the aim of delivering them as quickly as possible among a large collection of nodes [16].

Fundamentally, the node containing the information randomly chooses a set of other nodes in the network and passes the information that it has to them. Then, these nodes that become aware of the (new) information, randomly choose other nodes and propagate the information they have received, and so on. This process guarantees, with high probability, that all nodes will eventually become aware of the disseminated information. Systems that implement this type of protocols are more scalable, since additional nodes added to the system will not affect the communication process [54]. Although, scalability is affected by the requirement imposed by the randomly selection of nodes. In order to select the nodes that will be the target of the propagation in each gossip round, the system must know the entire network membership, and clearly this approach is not highly scalable when we are in the presence of very large number of nodes. Thus, protocols such as presented in [17, 19, 33, 43] suggests the use of partial views, that only stores a subset of the system members in order to improve scalability.

In Gossip there are typically three strategies to exchange information updates [32]:

- **Eager push approach:** Nodes send message payload to randomly selected nodes as soon as they become aware of them.
- **Pull approach:** Nodes randomly select other nodes and query them for new information. If those nodes have new information, they send it to the requester.
- **Lazy push approach:** Nodes send notification of new content to randomly selected nodes as soon as they become aware of it. The nodes which receive these notifications for content that they do not are aware, explicitly request the payload.

2.2.4 Relevant examples

Scribe [10] is an infrastructure used to support the event-notification paradigm in a large scale environments. It uses Pastry [50] to manage topics and to construct multicast trees used to disseminate the published events. These trees are formed by joining the Pastry routes from the root to each subscriber, considering the node that holds the topic as being the root. Nodes can create, subscribe, unsubscribe, and publish events. When Scribe creates a new topic, it calls Pastry route operation with a create message and topic identifier as a key (e.g. route(CREATE,topicId)), then Pastry delivers the message to the node identifier numerically closest to topicId. The same behaviour is applicable when Scribe nodes want to subscribe a topic, routing a subscribe message (e.g. route(SUBSCRIBE, topicId)). This message is routed by Pastry until it reaches the node that holds the topic (root node), creating forward nodes in its path that will be used to disseminate the events later. The unsubscribe method is analogous, since a unsubscription message with topicId is routed through the multicast tree created removing the interest in the topic.

The utilization of Pastry as routing mechanism, ensures that the distance between peers of the routes created are the closest possible, with respect to the proximity metric used in this protocol. The fact that the list of subscribers is distributed contribute as a fault tolerant mechanism and the tree/load balancing given by Pastry ensures the scalability property.

Technically **HyParView** [33] is an unstructured overlay which can be used as message dissemination protocol. Implements the gossip based communication paradigm and proposes to disseminate data efficiently with high fault tolerance, achieved through the use of two partial views of peers. A small active view is used to disseminate messages efficiently through reliable TCP connections. This view is maintained using a reactive strategy, which only changes in response to events, for instance when a peer detects a faulty peer, it removes it from the local active view and adds a new peer from the other view, the passive view. That passive view is larger and is used to replace faulty nodes of the active view. It is managed by a cyclic strategy, which is updated periodically in time. In this case, periodically a shuffle operation is performed by each peer in the system. Due to having small active views, it achieve better performance concerning message overhead than other approaches. The cyclic management of the passive view enables the passive view to contain a large sample of nodes, where the majority is correct.

2.2.5 Discussion

Following the communication primitives discussed in the presented sections of this Chapter, we decided to adopt an approach based in gossip data dissemination. The participants of the structured overlay need to communicate to particular peers in the network, and despite they do not know where these particular peers are, the network is able to locate them (using an indirect communication primitive based on gossip based approach, where the message is sent from peer to peer until it reaches its final destination).

In this Chapter we also discussed message-oriented communication primitives (Section 2.2.1) in which we presented two models of communication, message queuing communication (a persistent point-to-point message service) and event-based communication including publish-subscribe event dissemination systems. We reject both of these communication models since we do not need a persistent mechanism to exchange messages between P2P network participants and event-based communication mechanisms are generally used to contact multiple entities with that have same purpose (which is not our goal, since we want to coordinate the access to the server following a point-to-point communication strategy and not a point-to-multipoint). We also reject stream oriented communication model (Section 2.2.2) since it is mostly used to give support to multimedia applications which is out of the scope of this work.

2.3 Coordination Primitives

In distributed systems, it is very common that participants need to reach agreement on some aspect of the system operation, for instance electing a new leader to govern their actions. These activities are typically easy to implement in centralized approaches, in which a single unit is in charge of controlling and coordinate the whole system. In opposition to that, the decentralized architectures have an added difficulty, where the contribution of all participants must be taken into account.

In this Section we focus on topics related to coordination, ordering, and mutual exclusion that resolves problems raised from resource sharing. This resource sharing imply that only one process should have access to a resource of the distributed system, which is considered to be critical, in order to maintain the system consistency. Election algorithms are also referred latter, in order to overcome problems similar to resource sharing and other challenges related with centralized coordinators failure. Finally, we discuss consensus, that is widely used when a group must agree on something (e.g system's next operation) to ensure the correct progress of the system.

2.3.1 Mutual Exclusion

Mutual exclusion is used to deal with problems of consistency over shared resources. Consider a situation where we want to control the number of cars over a certain bridge using two controllers, one at each end. When a car leaves the bridge, the counter must be decreased by one, and when a car enters in the bridge, the counter must be increased by one. If there is one car leaving and another one entering the bridge at the same time, then we have a problem since both controllers are accessing the shared counter, and the counter can be decreased by two or one. As the counter is critical, since it must be accessed by only one process at a time, we define operations that manipulate its value as critical sections [16].

Adding a server that controls the access to the counter would be applicable, but it would be more efficient and fault tolerant if we enable communication between the processes. This latter approach is the most implemented in distributed mutual exclusion protocols [16], discussed next. All of them must ensure safety, liveness, and fairness properties. Safety ensures at most one process enters the critical section at a time, liveness promises that all processes that require access to the critical section will eventually access it, and fairness ensures that none of the processes that requires access to the critical section will be waiting forever.

Typically, distributed mutual exclusion protocols are divided in two main groups [46], token-based and permission-based protocols:

- **Token-based protocols [40, 45, 53] :** In these protocols, the access to the critical section is guaranteed by the existence of a unique token. The process in possession of that token, is able to access the critical section. There are two main approaches to transfer the token, or the process that has it transfers to other process, even if the receiver does not want it, causing high message overhead. Or it can be transferred after an explicit request by an interested process, hence a search token mechanism must exists. In the presence of failures, the token can be lost inducing a deadlock situation [46].

- **Permission-based protocols [1, 29, 37] :** The access to the critical section is granted by a quorum of permission votes. The process that wants to enter in the critical section must require permission from all of other participants. In conflict situations, a priority event mechanism ensures that only one will have the permission to access the critical section. In this type of distributed mutual exclusion protocols it can be difficult to gather a quorum of responses [46].

Some of those algorithms are detailed in Section 2.3.4, in order to exemplify the covered topics.

2.3.2 Elections

The permission-based protocols, in distributed mutual exclusion is closely related with the elections primitives, since both have as objective the choice of a unique process to play a specific role in the system. In fact permission-based protocols are concrete implementations of this agreement paradigm.

The election mechanism can be triggered by some process, for instance when it thinks that the coordinator process has failed. However, a single process can not trigger more than one election process at a time. During the algorithm execution the safety and liveness properties must be ensured. In this particular case, safety ensures that in concurrent elections run, only one process will be elect (e.g. the process with highest identifier). And liveness ensures that all the process eventually will vote. In order to understand better the elections approach we present the ring-based and bully algorithms in Section 2.3.4.

2.3.3 Consensus

It is often necessary for all network processes to reach agreement. For instance, in a replicated synchronous system, to choose which will be the next operation to be performed by all replicas.

Initially all the participants are undecided and propose a temporary value, that can be modified during the execution of the consensus process. Then, the communication takes place and they exchange their proposals, and after that, they achieve a final decision which may no longer be changed. To ensure the normal consensus execution, those algorithms must respect the following properties [16]:

- **Integrity:** If all correct processes proposed the same value, then this value will be chosen in the decision state by any correct process. This definition of integrity is sometimes lightened, not strictly requiring that all processes have decided the same value, thus accepting criteria of choice such as majority, minimum and maximum.
- **Agreement:** If two correct and different processes, p and q, reached the decision state then the decided value is the same for both.
- **Termination:** Eventually all processes will reach a decision state.

In utopian environments, in which no processes crash, no messages can be lost and all processes behaviour is correct, the consensus is always reachable. The same is not true in real world environments, in which the occurrence of failures is a possibility. Relatively to each process, we are in the presence of failures when it crashes or change its normal behaviour (e.g. omitting messages, transmitting wrong values, etc.) also defined as Byzantine. The famous FLP [21] result determines that consensus is impossible to achieve by a deterministic algorithm in asynchronous systems where a node fails.

In a synchronous sytem and in presence of crash failures, Lamport [31] proved that if the system has $2f+1$ correct processes working then consensus is achievable, where f is the maximum number of faulty processes. Additionally, if the failures are Byzantine they proved, using the Byzantine generals problem, that if the number of processes is greater or equal than $3f+1$ processes then the consensus is also achievable.

We are able to distinguish failures where processes crashes (fail, stop) and arbitrary(Byzantine) failures where processes do not follow their specified behaviour, as message omission and transmission of wrong values.

2.3.4 Relevant examples

The **Ring based algorithm** [11] is a non fault tolerant election algorithm, in which the processes are disposed in a ring design connected one by one through reliable communication channels in order to elect a coordinator in an asynchronous way.

Initially, all the processes are set as non-participants in an election process and anyone is able to start an election. When it happens, the process in question marks as participant and sends an election message to its clockwise neighbour with its own identifier. The receiver, if is not participating in any election, sets as participant and compares the identifier of the election message received. If its identifier is greater than the identifier in the received election message, then it is changed by its own identifier and forwards the message to its left neighbour, else it simply forwards the original received message. If it is already a participant it will not forward the message. Now, if the message was not forwarded, it means that an elections with larger identifier was already being propagated, which will effectively elect the coordinator. Then the coordinator sets as non participant and sends an elected message to its neighbour with the coordinator identifier. The same behaviour of traversing the ring until there are no participants in the election process is then repeated.

The **Bully algorithm** [23] is a election algorithm tolerant to crash failures which works in synchronous systems. In this type of systems, is possible to detect crash failures since we are able to make time assumptions and construct a reliable failure detector.

The election begins when one or more participants of the system (peers) detect that the coordinator has crashed. It is possible to detect when another peer failed if it does not responds between the time interval stipulated for message communication. Since all the peers know other peers, if a peer realizes that its identifier is currently the greatest, then itself declares as the new coordinator and sends a coordinator message to the others with lower identifier. Otherwise, a peer with lower identifier can send an election message to other peer with higher identifier, if no answer arrives, that peer will consider itself as the new coordinator and send a coordinator message to other peers with lower identifiers. Each peer that received election messages must answer with a message to the emitter, and begins another election, unless it has begun one already. The peers that received a coordinator message set their election variable equal to the new coordinator identifier in order to deal with it as new leader.

Additionally, if the time communication bound is incorrectly set to a value that is too low, the system might degenerate to high message overhead due to false positives from the failure detector component. This algorithm have problems when dealing with transmission delays, since peers are able to elect themselves as leaders concurrently, potentially disrupting safety properties. Property which ensures that in the end of an election only the peer (non-crashed) with higher identifier is elected.

Chubby [9] is a distributed lock service, in which clients are able to synchronize their activities and to agree in some information about their environment, for instance agreeing on a new leader. Typically a Chubby cell consists in a group of five replicated servers, that uses a consensus algorithm to elect a master and to propagate updates. To achieve that, the master must receive votes from the majority of the replicas. They also promise to the master that they will not start a new election process for a given time interval. Once elected, the clients interact directly only with the master. Thus, all the interaction with the system database is made by the elected master and the other replicas restrict themselves to copying updates performed by clients from the master replicas. Bigtable [12] is a distributed data storage that uses Chubby to elect masters.

2.3.5 Discussion

In this Section we presented three different coordination primitives. In Section 2.3.1, we presented the problem of accessing shared resources without any access control over it. This was the approximation used by the developed system presented in Chapter 3. We consider server as the critical section, thus we can avoid the exhaustion of its resources controlling the access rate to it, hence saving CPU load. The other two Sections, which presented elections (Section 2.3.2) and consensus (Section 2.3.3) strategies were not used in this work, but they can be seen as alternatives to future upgrades to the presented coordination system.

2.4 Used Technologies

This Section is dedicated to present two important technologies used in this work, WebRTC [56] and Legion framework [35, 36].

2.4.1 WebRTC

WebRTC [56] is a plugin-free open source project which enables real time communication (RTC) in Web and native apps. The creation of this project was motivated by the challenges imposed by web services such as Skype, which forces users to download a native application to be able to use its Voip system. WebRTC enables real time communication without heavy native app installations or extensions, it enables live communication by simply opening a web page in your browser (transparently). Actually Chrome, Firefox, Opera, Android and iOS support WebRTC technology.

Real time communication between browsers is now possible with WebRTC [56], enabling applications to connect its users via audio or video calling through the RTCPeerConnection API; exchange generic data through RTCDataChannel's API and to record/-capture audio or video through the MediaStream API in a P2P fashion. Although WebRTC was designed to support P2P applications, the real applications requires the use of some servers not only to provide the Javascript and HTML files, but also to enable applications to circumvent firewalls and NAT boxes while establishing direct connections.

The different types of server interactions which can occur in WebRTC are:

- WebRTC clients exchange personal information (i.e., identifiers) before any connection made between them.
- WebRTC clients exchange network information following the signaling protocol (explained below).
- For media connection, WebRTC clients must exchange (and agree) data and session information such as video format and resolution.
- WebRTC clients often access clients that are behind firewalls or NAT gateways, and these may have to be traversed using TURN (Traversal Using Relays around NAT) servers or STUN (Session Traversal Utilities for NAT).

Signaling is a mechanism used to coordinate communication and to send control messages among two client applications that wish to communicate via WebRTC. When two clients want to establish a connection, both connect to a known signaling server and exchange information to enable the establishment of a direct connection. The information exchanged by this mechanism is listed below.

- **Session control messages**: These messages are used to initialize or close communication channels and to report errors.
- **Network configuration**: Messages containing network data, such as host's IP address and port observable by the outside world.
- **Media capabilities**: Messages containing media metadata and settings such as supported codecs, media types, etc.

Peers can only start a connection when the exchange of information via the signaling process has been completed successfully.

2.4.2 Legion Framework

Legion is a framework [35, 36] that allows data sharing and communication among clients, and could be easily used over browsers. Legion architecture allows reducing dependency on server, since the centralized component is not responsible to spread messages with information regarding the activity of clients for all clients, letting most of this work to be distributed across peers over the network. The client side of this framework is divided into five main modules as illustrated in Figure 2.7.

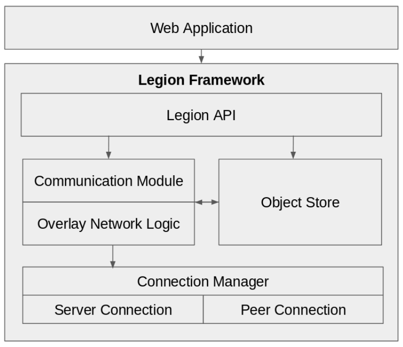


Figure 2.7: Legion Architecture [36].

- **Legion API** - This layer allows other applications to interact with the Legion framework exposing the framework interface.
- **Communication Module** - The Communication Module offers two main communication primitives: point-to-point and point-to-multipoint, used by a client to communicate with other(s) through a logic overlay maintained by the Overlay Network Logic module.
- **Object Store** - This module allows other applications to create and maintain shared objects. It uses the Communication Module to replicate and maintain objects up-to-date among web clients.
- **Overlay Network Logic** - This module establishes logical connections between clients, forming a topological structure which on the available Legion framework is unstructured. These logical connections allows clients to communicate with each other when they maintain a WebRTC connection between them.
- **Connection Manager** - This module manages client-client and client-server connections that are used by the Legion framework.

We are not focused on the **Object Store** module, since our goals are not to share updated objects among clients as this module offers. **Connection manager** is also excluded from our scope, since peer-to-peer and peer-to-server connections do not be modified by our work, being formed by WebRTC connections offered by default by Legion framework [35, 36].

Considering Legion modules previously presented, we are only focused in the **Overlay Network logic** and **Communication Module**. Since **Overlay Network logic** is responsible for the overlay topology, in this case we extended the Legion framework by adding a structured overlay, explained in Section 3.2.1; and **Communication Module** express how peers communicate with each other, allowing efficient communication between peers as explained in Section 3.2.2.

2.5 Summary

In the previous Sections we discussed existing distributed architectures. We have started in the centralized models, such as client-server which is the most used by existing web applications. We also referred a recent architecture that enables an ubiquitous access and the illusion of infinite resources, as it is the case of cloud computing architectures. Cloud servers could be placed in the same geographic location, or geographically distributed which contributes to lower the latency experienced by users. Finally we addressed decentralized designs with greater emphasis in the context of the peer-to-peer model, in which the participants organize themselves into a logical structured or unstructured overlay.

After the discussions of system architectures, we focused on communication primitives that includes multi-process communication paradigms such as multicast based models. Message queuing and streaming communication are also referred, in a context of end-to-end paradigm.

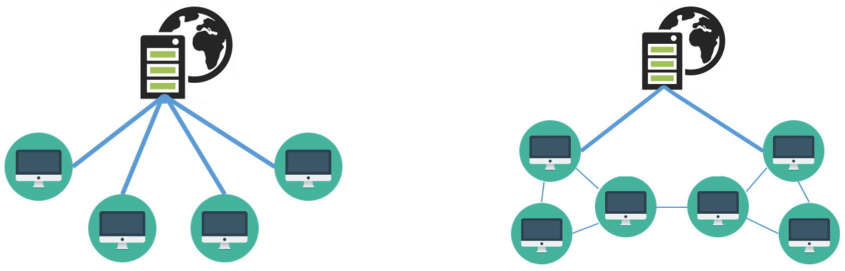
In Section 2.3, we addressed coordination primitives widely used in current systems, such as election primitives which allows a group to reach an agreement on a new coordinator (e.g. leader). Mutual exclusion is also discussed as a mechanism for managing shared resources maintaining system's consistency. Closing this Section is the consensus paradigm which is a very important aspect regarding coordination in distributed systems.

In the last Section of this Chapter we presented two important technologies used by this work, WebRTC and the Legion framework. WebRTC is not directly used in this work, but instead it is indirectly used through the Legion framework. Finally we introduced the Legion framework, which is used by this work to create a structured P2P network to give support to our coordinated system explain in the next Chapter.

Chapter 3

System Architecture

This Chapter presents the system implemented to resolve the lack of availability in centralized architectures as discussed on Chapter 1. The model proposed in this work, presented in Figure 3.1b, follows a different architecture than the classical client-server architecture as most web applications do, presented in Figure 3.1a.



a Client-server architecture b Proposed Model

Figure 3.1: Client-server and Proposed model

We propose a distributed model to achieve great availability levels with lower monetary costs required by cloud scale solutions. Combining a peer-to-peer network, formed by clients, with coordination primitives we can save CPU load of the centralized component, avoiding the exhaustion of resources with lower hardware monetary costs. In this case browsers play the role of clients and together create a network which coordinate the access to the central component avoiding the exhaustion of server resources and potentially increasing server availability.

The Legion framework was used to create P2P connections between clients of a web application, forming a connected network of browsers. This network has special participants which coordinate the access to resources of the web application. These special participants need to be easy to locate by other clients. Unfortunately, the unstructured overlays offered by Legion does not offer efficient among participants. Thus, we extended Legion, creating a DHT over it which enables two relevant functionalities to our proposes: i) deterministically assigning the responsibility of managing resource access to a given participant and ii) enabling any participant that wants to access a given resource to route requests to the client that is responsible for that resource. This DHT network was inspired by the Chord protocol [52] which was not trivial to implement in Legion framework since all connections between peers need to be mediated by a signaling server as explained in Section 2.4. This connection behaviour introduces significant restrictions in the way that peers can inter-connect among them to maintain the DHT topology (a discussion on the complexity of this can be found in [34])

This Chapter is divided in three Sections:

- 3.1 **Design Overview** - Provides a high level overview of the designed system architecture.
- 3.2 **Network Layer** - Explains, how the P2P network is organized (i.e., the DHT), how clients are disposed in the network (architecture) and how they communicate with each other (communication).
- 3.3 **Application Layer** - Explains, how special clients in the network coordinate the access to the centralized component, lowering its CPU load and increasing its availability.

Before explaining how the proposed system operates, we present three definitions inherent to the developed system, which will be useful to better understand its implementation:

- **Resource owner** - Client which is responsible for mediating the access to a resource in the server.
- **Resource solicitor** - Client which wants to access a resource on the centralized component.
- **Pre-submited form** - An HTML Form which was submitted by a client but stays in an intermediary stage controlled by our system (Request Capture Protocol). Hence not being directly (nor immediately) submitted to the server.

3.1 Design Overview

The system architecture is divided into two main layers, the application layer and the network layer. The next Sections will explain these layers in detail, but essentially the network layer (Section 3.2) is responsible for the organization of the network and for message routing mechanisms; and the application layer (Section 3.3) is in charge of the coordination between clients.

The proposed approach depicted in Figure 3.2, can be easily integrated with most web applications, since all the application logic is de-coupled from the logic of the web application. As explained further ahead, we encapsulate all web application logic into an HTML iframe, and all the logic to coordinate the access to the server (Request capture protocol, coordination protocol, and fault tolerance protocol) is outside of that iframe. We use lists and queues as data structures to support the decentralized coordination mechanisms to control the access to resources, and Legion [35, 36] is used for network management and message routing mechanisms.

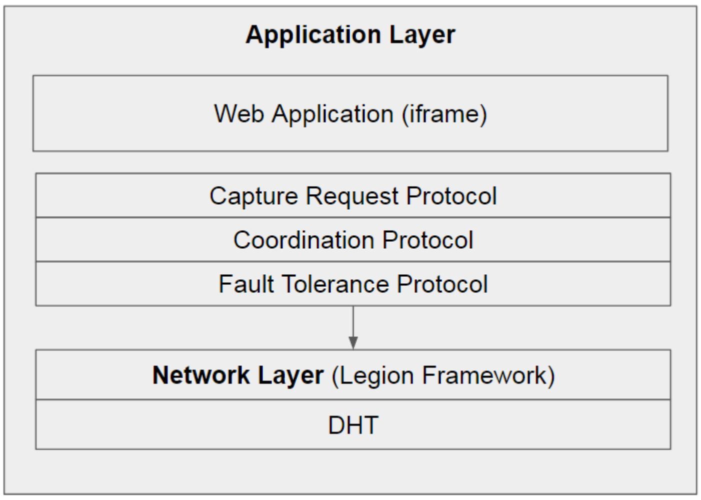


Figure 3.2: Proposed model design.

3.2 Network layer

The network layer is an important module of this work, since it shapes the network into a specific topology, enabling communication between clients. Since we want to improve server availability by distributing coordination responsabilities among clients, we choose a partly centralized P2P topology between the decentralized architectures presented in Section 2.1.4. The chosen network is not fully decentralized, since Legion [35, 36] relies on a central server responsible for user authentication, key management, durability of the application state, and for assisting clients in joining the system.

As stated before, to implement the P2P structured overlay we rely on the Legion framework, developed in the NOVA LINCS laboratory where the work presented in this dissertation was conducted and explained in detail in Section 2.4.2. The following sections explain how we shape our overlay network (Section 3.2.1) and how peers communicate with other peers using the overlay network (Section 3.2.2).

3.2.1 Overlay Network Logic - Architecture

As stated before (Section 2.4.2), the Legion framework is divided in five main modules and this Section focus on our work to enrich the **Overlay Network Logic** module. This module offers the use of three types of network overlays, an all-to-all overlay, a random graph overlay, and a geographic optimized random overlay.

As the name implies, the all-to-all overlay is formed by connecting all peers to all other peers. Thus, if *P* is the total number of peers in the network, all peers have *P-1* connections, which is not scalable for large number of clients. The random graph overlay does not have a particular topology since all connections are made in a random fashion, and the geographic optimized overlay is built taking into account the geographical location of peers, giving preference to peers which are geographically closer. These last topologies (random and geographic optimized) can not efficiently locate which peers are responsible for specific resources, since their topologies are random in nature, the network needs to be flooded to locate a particular peer, which is not scalable for large number of clients.

Hence, as none of the offered topologies is suited to tackle our problem, we need to extend the offered overlays and create a new overlay which fits better within our problem context. We need an overlay which enables fast and effective searches of resources owners over a distributed network. Inspired in an expression stated by Rodrigues et al. [49], "structured overlays are good at finding "needles"", we created a structured overlay which relies on unique identifiers to store data items and has as main advantage its key-based lookup process. Our overlay is based on the design of Chord [52], since it uses unique identifiers to form a ring topology enabling effective searches for a large number of members in the network.

In the overlay implemented by our work, peer connections are not made directly through TCP connections, but rather using connections managed by the Legion framework, which are WebRTC connections between peers. Due to the fact that peers can not be connected directly through TCP connections, each WebRTC connection must have the mediation of a central component that will allow the creation of the connection between the peers (explained in Section 2.4.1). Every peer has knowledge of its immediate successor and its immediate predecessor, in order to form the ring topology. This topology depends the behaviour of its participants, and to explain how the developed network is formed, we distinguish join, stabilize, and leave as key components of the overlay management protocol. This is in line with the design of the original Chord protocol

Join mechanism

When a peer (i.e., a web application instance running on a client web browser) joins the network, it communicates with a known Legion server to spread a Find Successor Request (FSR) message in the network, in order to find its immediate successor. If the joining peer is the first peer in the network then the join process finishes immediately, otherwise, all network nodes which received FSR message verify if the joining peer is its immediate successor (i.e., the peer with the lowest identifier in the group of peers with higher identifier than the joining peer is its immediate successor) and, if it is, then the peer starts a connection with the Legion server in order to create a direct WebRTC connection to the joining peer (which is achieved by performing the WebRTC signaling protocol). When the joining peer is connected to its immediate successor in the network the stabilization process starts, enabling the joining peer to find its immediate predecessor (i.e., the peer with the highest identifier in the group of peers with lower identifier than the joining peer is its immediate predecessor).

Stabilize mechanism

This mechanism is not only used when the join process ends but also occurs periodically, so that the direct predecessor and successor of the peers are continuously updated according to the arrival or departure of network participants in the system. In the stabilization phase, peers ask their immediate successor who they perceive as being their immediate predecessor by issuing a Successor Predecessor Request (SPR). Upon receipt of the response, the peer checks whether the peer identifier of the response is between itself and its direct successor. If so, it changes its direct successor and notifies (NOTIFY request) its new successor of its own existence. If the peer which received the notification verifies that the notifier is its predecessor, he updates its predecessor directly. This ensures that, periodically, changes in the network (due to failures or newly entering nodes) are correctly added to the overlay.

Leave mechanism

This mechanism is employed when a peer leaves the network, either by failure or simply because the user terminates its client application. To deal with this behaviour, each participant in the network stores a list of its close successors. Each peer asks periodically who are the close successors of its immediate successor, keeping the n closest successors, delimited by a network parameter called `MAX_CLOSE_SUCCESSORS`. Thus, when a peer leaves the network, its predecessor connects to the peer that will be its new immediate successor. Eventually, the stabilization process starts, refreshing all predecessors and successors according to network participants context.

In order to understand better how these mechanisms interact among each other, we present two examples, one demonstrating a peer joining the network and another exemplifying a peer departing the system.

Peer Joining Example

Figure 3.3 shows a P2P network with 4 peers and `MAX_CLOSE_SUCCESSORS` = 3. **JP** is the joining peer and **S** is the Legion server responsible to mediate the creation of connections between peers. Initially, each peer knows its two closest successors (besides its imediate successor), its imediate successor and its imediate predecessor as shown at Table 3.1.

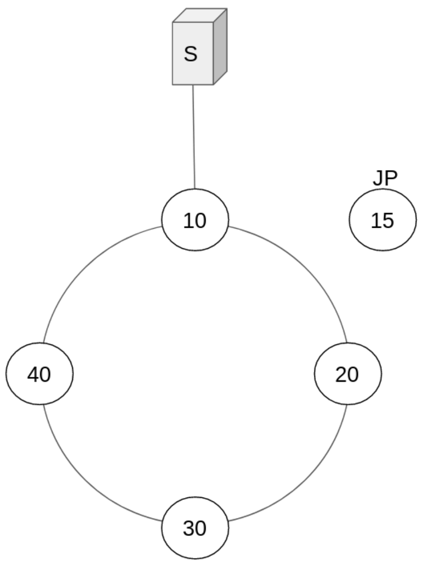


Figure 3.3: Stable overlay with 4 peers.

Table 3.1: Network state I

Peer Id	Predecessor	Successor	Close Successors
10	40	20	[20,30,40]
20	10	30	[30,40,10]
30	20	40	[40,10,20]
40	30	10	[10,20,30]

Peer 15 wants to join the system, so it connects to **S** and spreads a FSR request trough the network, in order to find its immediate successor. Peer with identifier 10 receives a FSR request, but since it is not imediate successor of **JP** it re-sends FSR request to its imediate successor (peer with unique identifier 20). Peer 20 verifies that is imediate successor of **JP**, so it connects to peer 15 and network evovlves to the configuration denoted in Table 3.2 and the shape illustrated in Figure 3.4.

Table 3.2: Network state II

Peer Id	Predecessor	Successor	Close Successors
10	40	20	[20,30,40]
15	null	20	[20,30,40]
20	15	30	[30,40,10]
30	20	40	[40,10,20]
40	30	10	[10,20,30]

JP's stabilization process starts when the join process ends, which is as soon as it is connected to node 20. Figure 3.5 illustrates the stabilization process only for peers directly affected by the join process. The joining peer (the peer 15) stabilizes, sending a SPR request to its imediate successor (peer with identifier 20). Peer 20 replies with

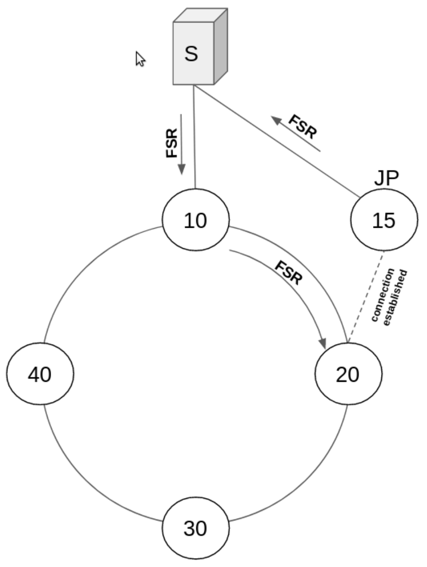
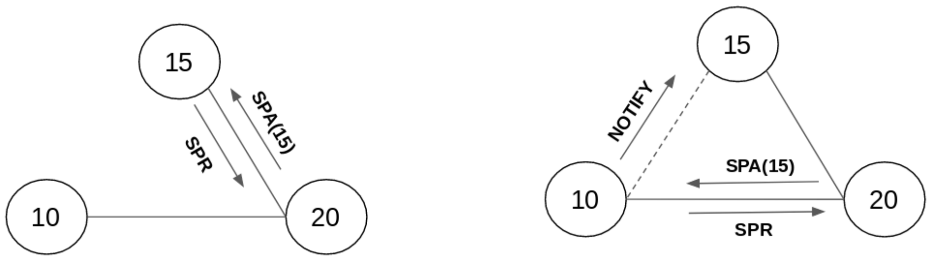


Figure 3.4: Overlay after peer 15 joins the overlay.

the identifier 15, and since 15 is not the immediate successor of itself, nothing changes (Figure 3.5a). Eventually, peer with identifier 10 will stabilize (Figure 3.5b), sending a SPR request to 20 (its known imediate successor), and 20 replies with 15. Peer 10 verifies that 15 is its actual successor and notifies 15 warning him about its existance (10 send a NOTIFY request to peer 15). When peer 15 receives peer 10 NOTIFY request, it knows that 10 is it actual predecessor, so it updates its predecessor value accordingly.



a - Peer 15 stabilizes

b - Peer 10 stabilizes

Figure 3.5: Stabilization process after peer 15 joined the network.

Eventually, the P2P network stabilizes (i.e., when all network participants finish their stabilization process the network stabilizes, assuming the ring shape) and assumes the configuration represented in Figure 3.6 and the state presented in Table3.3.

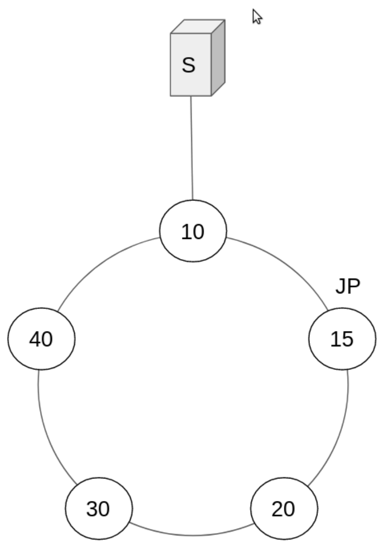


Figure 3.6: Overlay after peer 15 effectivelly joins the network.

Table 3.3: Network state III

Peer Id	Predecessor	Successor	Close Successors
10	40	15	[15,20,30]
15	10	20	[20,30,40]
20	15	30	[30,40,10]
30	20	40	[40,10,15]
40	30	10	[10,15,20]

Peer Departure Example

Starting now from the network shown by the Figure 3.6 and from the state denoted by Table 3.3, we explain how the network stabilizes when peer with identifier 20 leaves the system.

As soon as peer 20 leaves the network, peers who maintain a connection with it will notice its departure as WebRTC connections, when dropped, notify the upper layer (Table 3.4 and Figure 3.7). Each peer, after realizing the departure of a connected peer has to check if it was its successor, predecessor, or nothing (neither predecessor nor successor). If it was nothing, it just disconnects from it; if the peer that departed was its predecessor then it updates its predecessor value to null; if it was its successor, it initializes the value of the variable successor to its own id, removes the peer that has left from its list of close successors and tries to connect to the first candidate in that list.

In this case, peer 15 and peer 30 which maintained direct connections with peer 20 detect its departure. The peer with identifier 30, detects that 20 was its predecessor and sets its predecessor variable to null. Peer with identifier 15 detects that 20 was its successor, then sets itself as its successor, removes 20 from its list of close successors and tries to connect to peer 30, since it is its first successor candidate. If 30 is alive,

Table 3.4: Network state after peer 20 leaves.

Peer Id	Predecessor	Successor	Close Successors
10	40	15	[15,20,30]
15	10	15	[30,40]
30	null	40	[40,10,15]
40	30	10	[10,15,20]

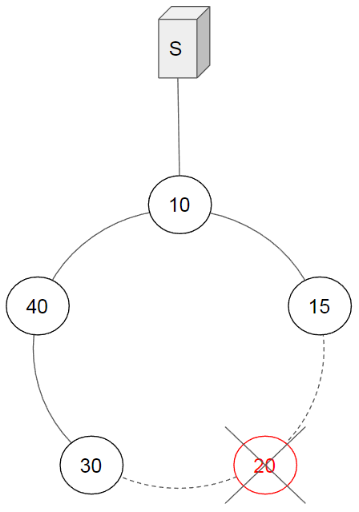


Figure 3.7: Overlay after peer 20 leaves the network.

the connection is established and 30 its marked as the new successor of peer 15. After stabilization, the peer with identifier 30 will be notified by peer 15 adopting it as its predecessor and the network assumes the topology represented in Figure 3.8 and the state denoted in Table 3.5.

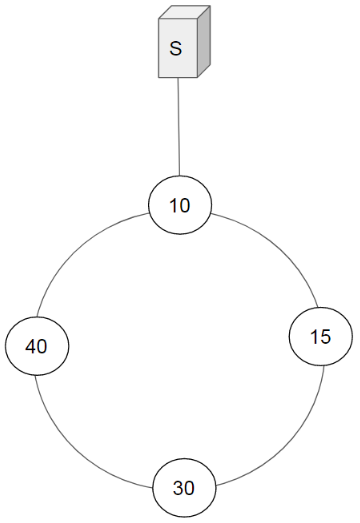


Figure 3.8: Overlay after network stabilization.

Table 3.5: Network state after stabilization.

Peer Id	Predecessor	Successor	Close Successors
10	40	15	[15,30,40]
15	10	30	[30,40,10]
30	15	40	[40,10,15]
40	30	10	[10,15,30]

3.2.2 Communication Module

In addition to the way the network is formed, the way its participants communicate is also a relevant aspect for the presented work. As Section 2.4.2 explained, the Legion framework has a **Communication module**, that is dedicated to manage how peers communicate, which had to be extended to meet the requirements of this work. This Section covers these modifications in two topics, **Message Routing and Overlay Optimization** which explains how messages are disseminated in the developed network architecture and **Message Delivery** which determines which peers in the network receives (and delivers) each message.

Message Routing and Overlay Optimization

The Legion framework offers flooding as the default message routing mechanism. Since we do not consider this technique of message dissemination scalable, we decided to extend this module of the Legion framework. However we do not totally remove this dissemination technique since it can be useful in some situations, as we will see further ahead.

Now that we have a ring like network topology that is ordered by peer identifiers (Section 3.2), we can build a more efficient way of communicating instead of flooding the network each time a message is sent between two peers within the network. In this mechanism, the message sender, instead of distributing it to all its direct neighbours (flooding), it only sends the message to the neighbour with the closest identifier to the message receiver identifier, reducing the overhead when exchanging messages in networks with many participants. For example in Figure 3.6, when peer 15 sends a message with peer 40 as destination, the message would be sent to both peers 10 and 20, where in our case we only send to peer 20 (since 20 has an identifier closer to 40 than peer 10) . In this example, where we consider a small network, this does not seem to have a huge impact, but it is easy to see that in networks with many participants (and more than one neighbour), message transmission overhead can be reduced substantially.

With the ring-organized network, as shown in Figure 3.6, messages would have to traverse almost all the ring (worst case), but we can lower the number of hops if there were overlay links that could allow messages to skip large segments of the ring (i.e., network jumps). Network jumps are connections with peers that are not successors or immediate predecessors, in addition to the ring topology. Thus, messages pass through fewer peers, being delivered in a more efficient way. The network architecture presented in Section 3.2.1 contains not only the direct connections between a peer and its successor and immediate predecessor, but also contains connections to more distant peers, enabling such jumps in the network. Although these connections are managed by the **Overlay Network Logic** module, they are explained in this Section because they have become necessary with the requirements imposed by this **Communication module**.

To manage these extra network connections, each peer periodically checks the number of direct connections it has with other peers (called neighbours). If that value is below the `MIN_CONNECTIONS` parameter, that peer chooses one of its neighbours and requests a list with its own neighbours. Upon reception of the response the peer with fewer neighbours than defined by `MIN_CONNECTIONS` will connect to a random chosen peer of the received list of neighbours. We also control the maximum number of connections of this type with the `MAX_CONNECTIONS` parameter. Similar to the management of the minimum connections, each peer also periodically checks if the number of its neighbours is greater than the allowed maximum number of connections. If this is the case, the peer chooses one of its neighbours (based on key distance) to eliminate the connection between them, giving more value to the most distant and relatively close connections.

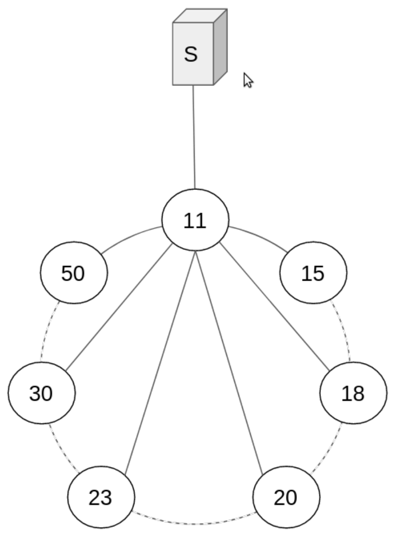


Figure 3.9: Network with neighbours connections.

For instance, considering `MAX_CONNECTIONS` = 5 in a network with more participants as the one shown in Figure 3.9, peer with identifier 11 has as its successor peer 15 and as its predecessor peer 50. Peers 15, 18, 20, 23, 30, and 50 are its neighbours. Since there are more neighbours than allowed, one peer connection must be removed. The peer will never cease a connection with its immediate predecessor and its immediate successor, so peer 11 can only eliminate a connection with 18, 20, 23, and 30. But since we consider the most distance and relatively close jumps must remain, the only connections that can be eliminated will be the connections with peer 18 or peer 30. From these two we choose a random one and remove it from peer 11's neighbours.

Communication between peers becomes more efficient with this type of architecture which allows messages to be transmitted faster, since they can skip several peers instead of traversing almost all peers in the ring (in the worst case). Because peer failures can occur during message transmission, sending the message only to the nearest neighbour can be dangerous, since if one of the peers in the path fails, the message may never be delivered. Thus, instead of choosing the neighbour with nearest identifier to the identifier of the message destination, we add some redundancy and choose the two closest neighbours overcoming issues that could arise more frequently otherwise.

The network architecture is important to allow efficient communication between peers. Since this point-to-point communication technique is based on the order of peer unique identifiers, peers that are entering in the network (i.e., which are not positioned in the right place in the ring) imposes some problems. Thus, the peers that are not well positioned in the ring, trade messages using flooding as communication protocol. The peers that are already in the correct position in the ring communicate using the point-to-point protocol described above.

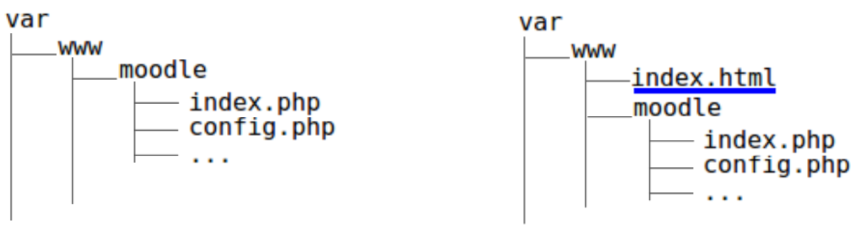
Message Delivery

In the given examples, the message receiver is always a network member, so it is obvious who is the responsible for receiving the message. But, when the message receiver is not a network member who is responsible for receive that message? To answer this question, we decided that the successor of the non existing member should be responsible for receiving the message. Resuming the Figure 3.9, if a message is routed for peer 21, since it does not exists in the network the responsible for its messages is its virtual successor, peer with identifier 23. Thus, if the peer does not exists in the network, the responsible of its messages is its immediate successor which is a network member that always is bound to be available. Notice that, contrary to the Chord protocol, in our solution, messages can be routed in any direction over the ring.

3.3 Application layer

As stated in Section 3.1, the application layer gives support to coordination primitives which contribute to increas server availability. This application layer is implemented within the front page of a chosen web application. In this particular work, Moodle [41] was chosen as a use case and test application, since it is open source and normally is managed by educational institutions. Most of these educational institutions often have low monetary resources to maintain robust, and reliable systems to support web applications with high access rate. Note however that the design choices apply to many applications, detailing the use of Moodle in our design here is to ease the reader's understanding of the system as a whole.

Our system logic can be easily integrated in this web application by inserting a script inside Moodle's front page. The index page (index.php in the web application root folder) which works as starting point for its users, can be encapsulated inside a new index (index.html created by us), containing a HTML iframe with all original web application logic. The change made in Moodle's directory are represented in Figure 3.10, with the only notable difference is the creation of index.html. The contents of this file are summarized in Listing 3.1, in which **Request Capture Protocol** captures all POST requests made by Moodle users; **Coordination Protocol** gives responsibility to resource owners for coordinate the access made by resource solicitors to a Moodle resource (e.g., Login, User Creation); and **Fault Tolerance Protocol** is responsible for maintain the consistency of coordination lists in the presence of failures (i.e., when a resource owner fails).



a Original Moodle Structure   b Changed Moodle Structure

Figure 3.10: Changes made in Moodle root directory

Outside the iframe we incorporate the logic to capture the requests made by users, and the logic to coordinate the accesses to the server, thus achieving the coordination desired in the application. We also use Legion enriched (simply by importing its Javascript file in the index.html file) with the network protocols described previously to provide the adequate peer-to-peer support required in the client side. The main structure of index.html (used to encapsulate the original web application's logic) is summarized in Listing 3.1.

Listing 3.1: index.html

```
<!DOCTYPE html>
<html>
  //Import Legion Framework
  <body role="document">
    <iframe id="moodle_iframe" src="moodle/index.php"></iframe>
    <script type="application/javascript">
      //Request Capture Protocol {}
      //Coordination Protocol {}
      //Fault Tolerance Protocol {}
    </script>
  </body>
</html>
```

3.3.1 Request Capture protocol

Most web applications follow a client-server architecture in which clients communicate with the server via HTTP requests (POST, GET, etc), and Moodle is not an exception. Moodle uses HTML forms to send data to the server via HTTP POST requests and to control the access rate to the centralized component of Moodle, we need to capture all the requests made and then coordinate them by deciding which requests should be (or not) sent to the server. Thus, we create this request capture protocol which is responsible for capturing all HTML POST forms that were pre-submitted by Moodle's web application. HTTP POST requests performed by HTML forms cease to be direct to the server, since they are caught by this intermediary layer and passed to our coordination protocol. The coordination protocol is in charge of decide which requests should be sent to the server. After the coordination protocol ends, if approved, the request it is passed again to the request capture protocol to be sent to the server for processing. Thus, we maintain an identifier of this forms to effectively send them later. This capture protocol is show in Listing 3.2.

Following this context, we are only focused on HTTP requests which induce more server side processing effort, such as POST, PUT, and DELETE requests, since usually they are used to change the state of application database (usually considered a slow operation when compared to GET requests, which are also often resolved by a cache).

Listing 3.2: Request Capture Protocol

```
<!DOCTYPE html>
<html>
  <body role="document">
    <iframe id="moodle_iframe" src="moodle/index.php"></iframe>
    <script type="application/javascript">
      //Request Capture Protocol
      $('#moodle_iframe').load(function() {
        $(this).contents().find('form').submit(function(e) {
          form = this
          if(form method == "POST"){
            create formId for form
            resource = sha1(form.action)
            If (resource_owner != me){ //Coordination Protocol (remote)}
            else{ //Coordination Protocol (local)}
          }
        });
      });
      //Coordination Protocol {}
      //Fault Tolerant Protocol {}
    </script>
  </body>
</html>
```

This request capture protocol is also responsible for creating resource unique identifiers. Essentially, clients have unique identifiers which identifies them in the network, such as resources. Client identifiers are generated by the sha1 hash function applied to a random number managed by the network layer while resource identifiers are generated by the application of the same sha1 hash function to the URL of the pre-submitted form. Both key spaces starts in 1 and ends in 99999.

Figure 3.11 presented below, summarizes the role of the request capture protocol in our system, using as example the Login action in Moodle's web application. When an user wants to login in Moodle web application, he must fill the login form fields and then submit the form through a click on the *"Login"* button. As stated before, the form data will not be submitted to server immediately, but it will be captured by our request capture protocol.

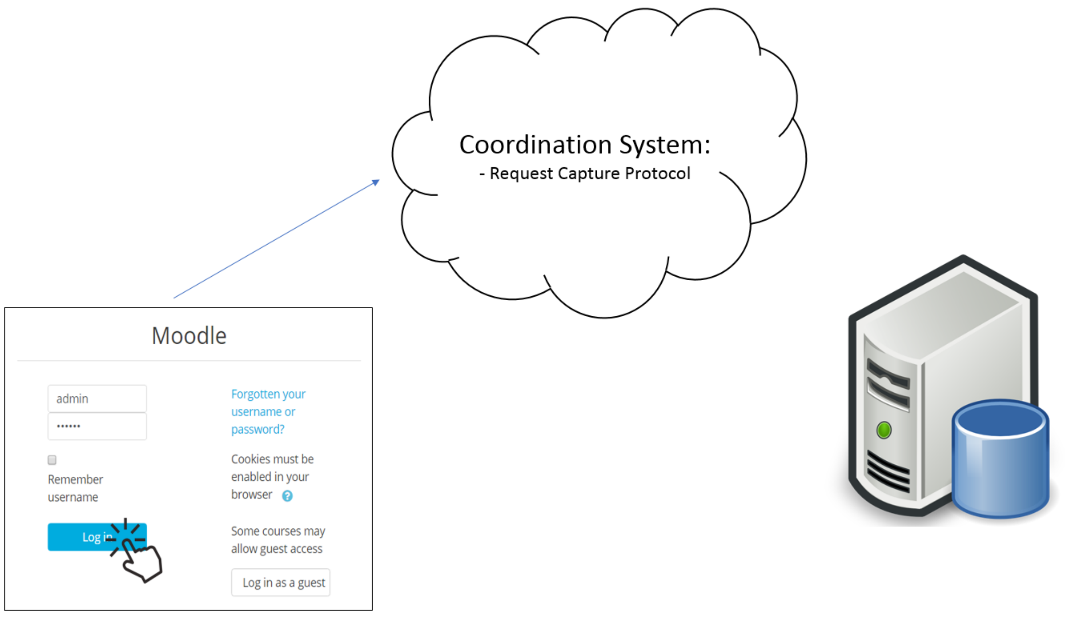

Coordination System:
- Request Capture Protocol
Moodle
admin
Forgotten your username or password?
Cookies must be enabled in your browser
Remember username
Log in
Some courses may allow guest access
Log in as a guest

Figure 3.11: Request Capture protocol flow

After determining the process that is responsible for managing the resource being accessed by the client (i.e., the resource owner) and sending a message requesting access to the resource, the request capture protocol ends and the coordination protocol begins.

3.3.2 Coordination protocol

At this stage, we have presented the request capture protocol responsible to capturing all HTTP POST requests made by the web application. Once these HTTP POST requests (pre-submitted by clients) are captured, they will be processed by the coordination layer and routed to the resource owners through the P2P network described previously. In order to achieve a decentralized coordination model, clients (i.e., peers) coordinate and communicate with each other using the following messages:

- **CanI** - Used by resource solicitor to request access to a resource.
- **YYCan** - Used by resource owner to grant access to the resource.
- **Release** - Used by resource solicitor when released the resource (i.e., after performing its operation over the centralized component).

The coordination protocol is responsible for coordinating the access to the web application's server, and assign to resource owners (i.e., assignment of resources to peers in the network). Access coordination is achieved by maintaining a list of the clients that are currently using a resource of the web application. Each peer maintains, for each resource identifier managed by it, a list with the identifier of the resource solicitor and the time-stamp of the access granted to that resource. Additionally, it maintains a queue that operates as a waiting list, for refused requests. When a request is refused, it is placed in the waiting queue, and when a client releases its access over a resource, the first peer in the queue is notified and granted access to the desired resource. The structure of data at a resource holding peer is summarized in Listing 3.3.

Listing 3.3: Coordination Structure

```
{resourceId:
  { accesses:  [{Rsolicitor: integer, timeOfAccess: integer}],
    waitingQueue: [{message: message}]
  };
};
```

Ignoring the request captured by the request capture protocol, the coordination protocol behaviour can be divided in three phases: the access request, the decision process, and the release phase.

Access Request - The coordination layer is responsible to verify if the resource solicitor and the resource owner are the same client or not. If they are different clients, the resource solicitor must send a **CanI** request (through the Legion DHT network) to the resource owner. Once received a **CanI** request, the resource owner must decide if the resource solicitor can or can not access the pretended resource. In fact this request not always exists, because if the resource solicitor and the resource owner are the same client the decision is made locally (without sending a request through the Legion DHT network).

Decision process - The decision whether or not to access the resource is based on the number of clients that are using the given resource. In the coordination layer we define a specific threshold that controls the number of simultaneous clients that can access the resource. Thus, when the resource owner receives a **CanI** request he simply needs to verify if the number of clients currently accessing the resource is greater than the threshold defined. If it is lower, the resource owner sends a **YYCan** request to the resource solicitor. Otherwise, the resource owner puts the resource solicitor in a waiting list, which is being served as clients accessing the resource release the resource.

Release - After a resource solicitor accesses the intended resource and does not need it anymore, he sends a **Release** request to resource owner, warning him that it is not occupying the resource anymore, letting other clients access the resource in the future. The resource owner has a timeout to release the resource if this **Release** request is not received (for instance due to a crash of the client which is accessing the resource).

Detailed Example

The network used in this example is formed by three peers, represented in Figure 3.12 as Client _1, Client _2, and Client _3. Client _1 is being used by the user "admin"to login in the system and Client_2 is the resource owner which manages the access to the Login resource. Since admin wants to login in Moodle's web application, he fills the login form fields and then click on the *"Login"* button. This action will trigger the request capture protocol of Client_1 which captures the request (1) and delivers it to local coordination protocol(2). The local coordination protocol of Client_1 verify's that it is not the owner of the Login resource. Thus Client_1 sends a **CanI** request through the Legion P2P network, which will find that Client_2 is the Login resource owner (3). Once Client_2 received the **CANI** request (sent by Client_1), its coordination protocol instance checks that Client_1 can access the Login resource and sends him a **YYCan** message (4). When Client_1 receives the **YYCan** request he can effectively submit the HTTP POST to the server since he has access granted (5).

3.3.3 Fault Tolerance Protocol/Resource owners management

The coordination structures need to be consistent to maintain a correct access rate to the server allowing savings of its CPU load. We note that the coordination structure is affected by when clients join or leave the network, thus we consider Join and Leave events into the coordination protocol and we explain how to deal with these two events next.

Join - When a client joins the network he could be or not, a resource owner depending on its position on the DHT ring. In this case we only focus on the case of clients joining and becoming resource owners, since they need to obtain an updated coordination list of the resource (or resources) that they become owners of. This join operation will trigger an event on other clients that will check if the new client is its predecessor. If the stated condition is true, it should means the new client needs to update its coordination structure (which initially is empty). Thus, the new client receives a coordination structure of its

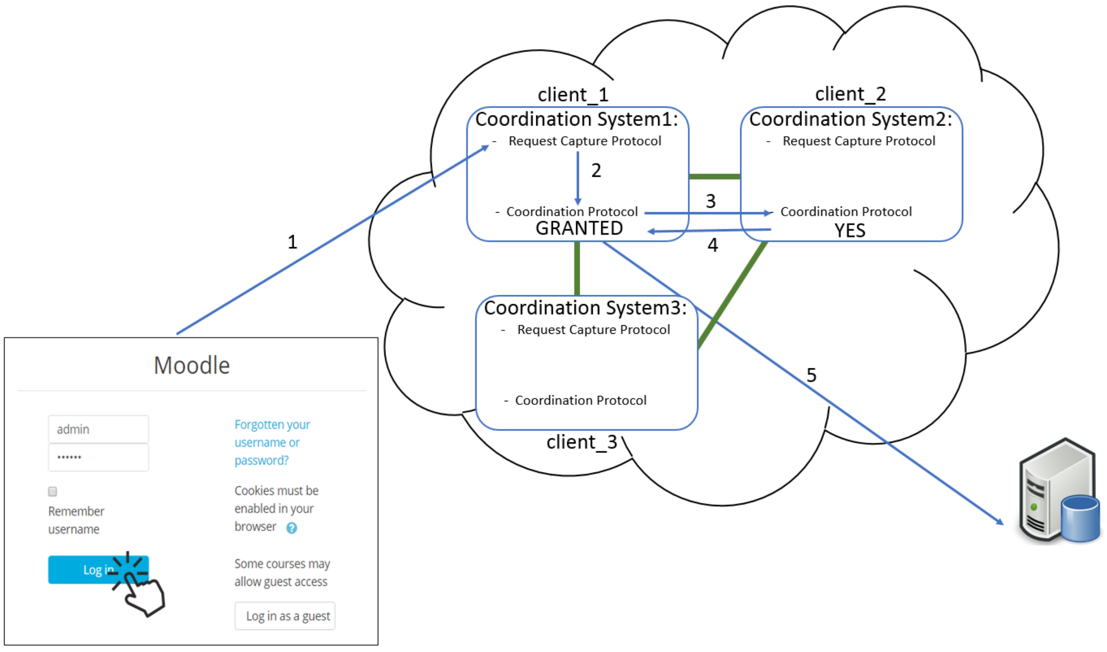


Figure 3.12: Coordination mechanism example illustration

network successor to verify which resources are now in his charge, becoming a resource owner. Imagine a part of the network containing two peers as Figure 3.13 illustrates. Peer with identifier 10 is owner of resources 9 and 10, and the peer with identifier 20 is owner of resources 12,15 and 20 (Figure 3.13a). Now, when a new peer with identifier 15, joins the network the peers 10 and 20 will check if peer 15 is their predecessor. This condition is only true for peer 20, thus the peer with identifier 20 sends its coordination structure and the new peer chooses which resources are now in his charge, which are the resources with identifier 12 and 15 (Figure 3.13b).

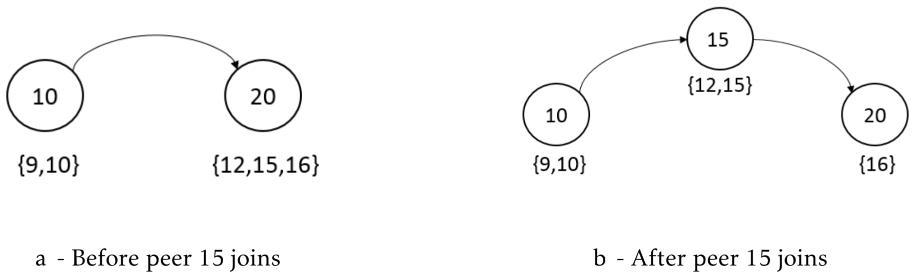


Figure 3.13: Evolution of network state at join process

Leave - This event may also cause a change on resource owners, depending on the leaving peer. If the leaving peer was a resource owner, then its resources must be held by some other peer, otherwise no change to resource management structure is required. In order to overcome possible peer leavings we use a simple fault tolerance protocol that replicates coordination structures. This protocol plays an important role in the context of our system since it is responsible for maintaining the consistence of the coordination structure among clients, when something wrong happens (e.g., peer failure). We use a proactive strategy to maintain the coordination structure consistent in the event of peer failures, since we replicate these structures over successor peers periodically. Thus, when a resource owner fails or leaves the network, its immediate successor will have an updated coordination structure, and is thus able to maintain a correct access rate to the server even as the original owner of the resource failed. It is possible to define how many successors must have the resource owner list in order to grant a greater fault tolerance, by changing an application parameter called `REPL_SUCCESSORS`. Thus, each peer does not only maintains its coordination structure, but also maintains an updated coordination structure of each close predecessors.

Keeping in mind the network example used to explain the join process (Figure 3.14a), `REPL_SUCCESSORS` = 1 and peer with identifier 15 leaves the network. Now, each peer maintains its coordination structure and the updated coordination structure of its immediately predecessor. Peer 15 has its own coordination structure which contains the resources with identifier 12 and 15, and the coordination structure of its immediately predecessor. Similarly peer 20 has its own coordination list which contains the resource with identifier 16, and the coordination structure of its immediately predecessor which contains the resources with identifiers 12 and 15 (Figure 3.14a). When peer 15 leaves the network, the resources controlled by it must be controlled by other peer which should maintain an updated coordination list of the resources concerned. This is the case of peer 20, which understands the departure of peer 15 and becomes the resource owner of the peer 15 resources, as illustrated by Figure3.14b.

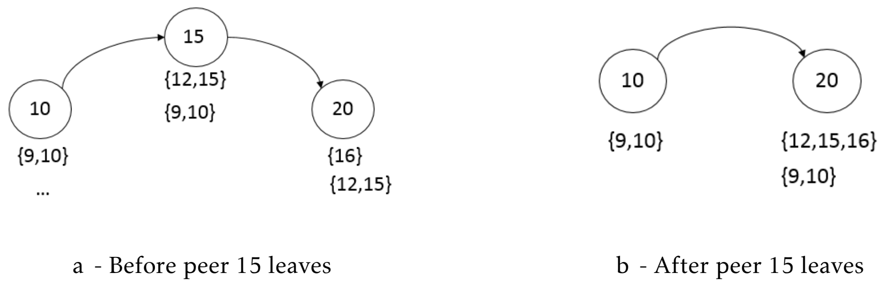


Figure 3.14: Evolution of network state at leave process

3.4 Summary

In this Chapter we have presented how the network and application layer are organized. The network presented, was created with a framework developed in our academic laboratory (Legion), it assumes a ring shape, ordered by the identifier of its participants (peers). Each peer is not only aware of its immediate neighbours (successor and predecessor), but also maintains a connection to peers further away in the network (neighbours). This connection type with peers which are more distant in the ring, allows messages to be exchanged in a more efficient way, since it makes possible jumps in the network when routing messages among peers. These jumps on the network are randomly created as the network grows and are controlled by two network parameters, known as `MIN_CONNECTIONS` and `MAX_CONNECTIONS`. For the case of peer failure, each participant also keeps a list of potential successors (close successors), connecting to one of them when its direct successor fails. We have also seen that peers enter the network through **Find Successor Request** requests sent with a flooding policy while network participants (i.e., peers that already belong to the network) directly communicating with others following a protocol based on the proximity of the identifier of the destination node of the message. Thus, messages are always propagated to the two neighbours with the nearest identifier of the message destination (instead of one, for fault tolerance). Lastly in the network layer, we saw how the messages are delivered to its responsible peer, and in the case of a message which has a destination that does not exist, the responsible peer for that message will be the peer that acts as direct successor of this non-existent peer (i.e., resource allocation is done by delivering to the closest peer id above resource id). This overlay is highly based on the design of the Chord protocol.

Related to the application layer, we have seen how the application coordinates the access to the centralized server. The web application resources are managed by special peers in the network formed by web application clients. These peers, called resource owners, verify if resource solicitors can access the pretended resource. This process starts when a resource solicitor wants access to a resource. Our system captures the request made by this resource solicitor and routes a message to the resource owner. The resource owner verifies if the resource solicitor can access the pretended resource by checking how many clients are using it in that precise moment. If that number is less than the maximum clients allowed, the resource owner will grant access to the resource solicitor, otherwise the resource solicitor it will be placed in a waiting queue.

In order to avoid lack of consistency in coordination structures (which can happen when nodes who are responsible for some resource leave the network), we also implemented a simple fault tolerance protocol, which is in charge of replicating the coordination lists to a predefined number of successors, controlled by the `REPL_SUCCESSORS` parameter.

CHAPTER 4

Evaluation and Results

To evaluate the system developed in this work, we created specific experimental setups that are explained later in this chapter. Since the present work acts at two different layers (application/coordination layer and network architecture layer), we created different tests to evaluate each layer individually. We dedicate Section 4.1, to report the experiments and results regarding execution of the network architecture layer. And Section 4.2, explains the experiments and the obtained results for the coordination layer scope, which is the main focus of this work.

All reported experiments in Section 4.1 and Section 4.2, were executed three times in a distributed environment (4 cluster clients and 2 servers) supported by Microsoft Azure platform [39]. Following we present the list of hardware resources used in these experiments to maintain each component of our tests:

- **Clients** - Standard D8s_v3 (8 vcpus, 32 GB memmory)
- **Moodle Server** - Standard D2s_v3 (2 vcpus, 8 GB memory)
- **Legion Server** - Standard D2s_v3 (2 vcpus, 8 GB memory)

The tests conducted on the work developed were not trivial and involved the use of multiple technologies. Since the application was developed in a web environment, we used the java version of the SeleniumHQ (Web driver), which is a framework that automates all interactions with a browser (in this case Firefox). This type of interaction includes navigation in the Moodle web application, collection of metrics, launching firefox instances, etc. In this way, the experiments explained further ahead were conducted following the scheme presented in Figure 4.1.

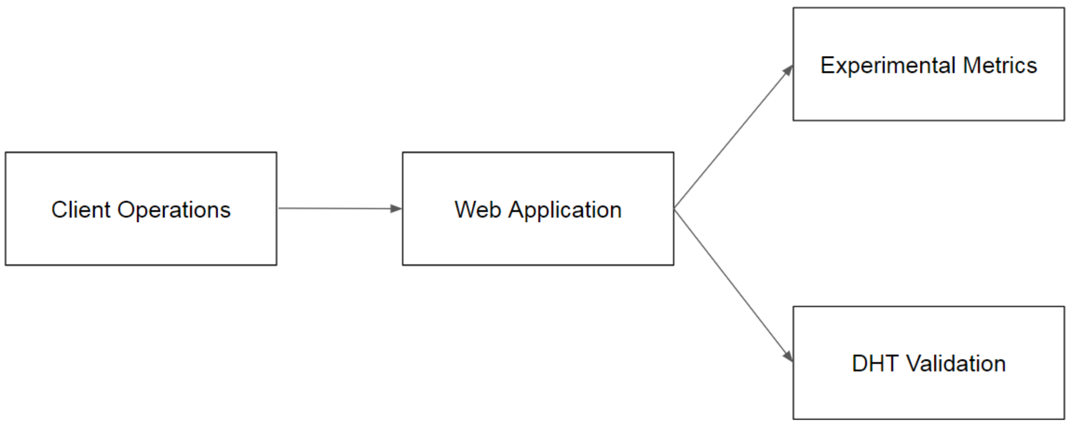
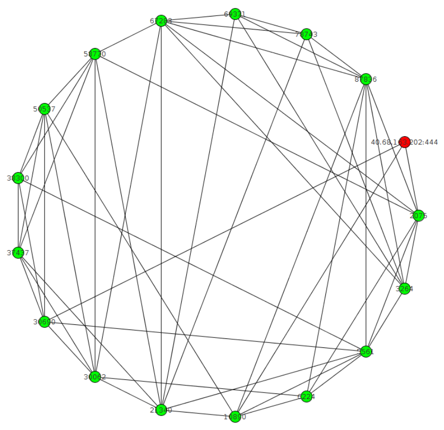


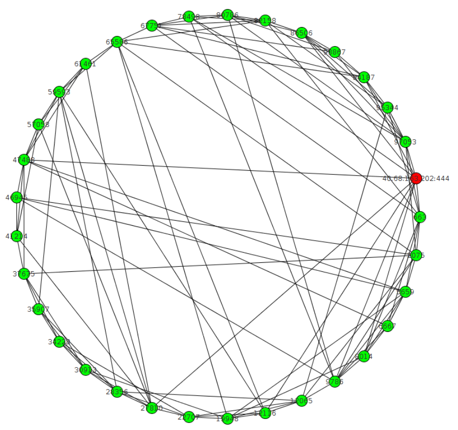
Figure 4.1: Experimental scheme

Moodle users are represented by multiple instances of firefox browsers running in each Microsoft Azure [39] client machine and perform the same operations type over the web application (which implementation is different for each test). These operations over the web application produce two different groups of outputs for each test; one regarding to experimental metrics and another used to validate the structure of the DHT formed by the clients of each test (i.e., DHT Validation).

The outputs regarding DHT validation were used to recreate and to visualize the network formed by the clients of each test. Each DHT validation file contained all peer identifiers and their respective neighbours (i.e., immediate predecessor, immediate successor and network jumps), which allowed us to recreate the network formed during the test. Figure 4.2 illustrates two networks formed during the experiments, in which peers are represented by green color and the Legion server is represented by the red color.



a - P2P network with 16 clients



b - P2P network with 32 clients

Figure 4.2: P2P Networks formed during the experiments

Each test consists in a executable jar (containing the user behaviour for each test) which was run in each client machine separated by three seconds. Considering for instance, an experiment which launch four peers and t represents the starting time of jar execution, client 1 launches one peer (result of dividing four peers to launch by four clients) at $t + 0$ seconds, client 2 launches one peer at $t + 3$ seconds, client 3 launches one peer at $t + 6$ seconds and client 4 launches one peer at $t + 9$ seconds. Each peer is launched between 1 and 4 seconds in all the tests presented in this Chapter, meaning that we do not have a fixed time to separate the launch of each peer.

4.1 Network Layer Evaluation and Results

This Section is dedicated to the tests conducted to evaluate the P2P structured overlay network, both at performance and structural levels. To evaluate network efficiency we conducted two different tests, one for counting the maximum number of hops given by a message exchanged between two peers, and other to count the network stabilization time after a new peer joins the system. In the end of each test we verified network structure by running validation program created by us, ensuring the ring shape. The setup of each test is explained below.

4.1.1 Message maximum hops Test

This test consists in counting the maximum number of required peers to spread a message changed between two peers (maximum hops). The test started launching a fixed number of peers (16, 24, or 32 peers), after that we waited 1 minute to ensure network stabilization, and then we choose 100 random pairs of peers (one peer to send and another peer to receive a message). Each message had a counter which was incremented each time the message passed through a peer that was not the message receiver.

For this test setup we do not performed tests for networks with 4 and 8 peers since we had `MIN_CONNECTIONS` set to 5 and `MAX_CONNECTIONS` set to 8, ensuring a full connected network with high probability. A full connected network is a network where all peers are connected with all other peers, resulting in maximum number of hops of one, since all message are changed directly.

For networks with 16, 24 or 32 peers the probability of a full connected network, with the used parameters is low. Since connections between peers are made randomly as the network grows, as explained in Section 3.2.2, we cannot predict the maximum number of hops for all possible networks. However, we conducted this test to get closer to values which allows us to evaluate the transmission efficiency of messages, this values are presented in Table 4.1.

Note that random network jumps were introduced to increase the efficiency in the transmission of messages, relatively to a network which would only be composed of a

Table 4.1: Message maximum hops

Clients	Max Hops
16	4
24	6
32	9

single ring topology. In this network, where each peer would only have knowledge of its predecessor and its successor, a message could be dissipated in three different ways: predecessor in predecessor, successor in successor or both. For instance, in a network with 32 peers, the maximum number of hops for the worst case of each dissemination perspective would be 29, 29 and 15 respectively. Compared with the maximum number of hops obtained for a network of 32 peers (Table 4.1), we saved about 7 hops (comparing to the best scenario of the dissemination processes presented). These results allow us to conclude that this optimization has an impact on the transmission of messages, making the dissemination process more efficient with the presented random jumps.

4.1.2 Join Stabilization Time Test

This test evaluates the average time taken by a peer when joining the system. We started by launching a fixed number of peers (4, 8, 16, 24, or 32 peers). Then we waited for 1 minute after all peers were launched to ensure network stabilization. After that minute we launched the new peer. In the end of the test we collected the time when the last peer connected to the Legion server (the moment when peer enters the network) and the time of the peer when it connects to its predecessor (meaning that it is corrected positioned in the network ring).

A periodic task that allows network stabilization is part of our overlay management protocol, similarly to the Chord protocol presented in [52], and the period between the execution of this network stabilization can be parameterized. In this test, the stabilization task was executed every three seconds and the results of the new peer entry times in the network are presented in Table 4.2, according to the setup presented previously.

Table 4.2: Join Stabilization Time results

Clients	Time (s)
4	0,72
8	1,86
16	1,74
24	1,38
32	2,49

Comparing the results obtained with the periodicity of the network stabilization, we can verify that the entry time of the new peer in the network, is always lower. This leads us to conclude that it takes only one stabilization phase to fit the peer into the ring-structured network. The times presented in Table 4.2 do not follow any trend (ascending or descending) because they are related with the instant the peer enters the system. For example, if the peer enters 2 seconds before the stabilization task runs, the network will take 2 seconds plus the time taken by the peer to connect to its direct neighbours, to stabilize. But since the network has a maximum of 32 peers, the connection time of the new peer to its direct neighbours is very low. For networks with a lot more participants, this time of connection to the direct neighbours would present an upward trend because it would be more difficult to find the successor of the joining peer.

4.2 Application Layer Evaluation and Results

This Section is dedicated to the tests conducted to evaluate the developed system performance. The goal of our system is to avoid the exhaustion of the centralized component (Moodle server), distributing its CPU load along the time. To evaluate our goal we conducted two types of tests, one in which our system was not integrated with Moodle's web application and other which we integrated our system with Moodle's web application. Inserted in the second category, we conducted two experiments; one where we established a maximum number of simultaneous client accesses to a Moodle resource, ignoring all the other requests by dropping them and requiring clients to attempt again later (i.e., throttle the maximum number of requests in a decentralized way); and other where was also established a maximum number of simultaneous client accesses, but the rejected ones were placed in a waiting queue which ensures that resource solicitors will be eventually served according to a FIFO policy (Throttle with queue). Each test collects the GET and POST requests made to Moodle's web application, the CPU load of Moodle's server, the average latency added by legion, and average of page load latency.

Contrary to the tests performed to evaluate the overlay network topology, over which was not imposed a fixed duration of time, the tests that evaluate the coordination scheme at the application level were limited to 5 minutes. During these 5 minutes, each client/peer creates a user in Moodle's web application every 5 seconds. Clients also wait for server response (at maximum of 3 seconds) before continue to the next operation. Note that this clients behaviour was the same for all the following tests, varying only the implementation of the web application.

The Moodle's web application was modified in each test, resulting in three different tests: i) Moodle without modifications; ii) Moodle with our coordination system in throttle mode; and iii) Moodle with our coordination system in throttle with queue mode. These tests are henceforth referred as SMoodle, TMoodle and TQMoodle respectively. Note that, each one of the latter two tests allow one client simultaneously per resource or allow three clients simultaneously per resource, thus henceforth we refer them as TMoodle1, TMoodle3, TQMoodle1 and TQMoodle3 respectively.

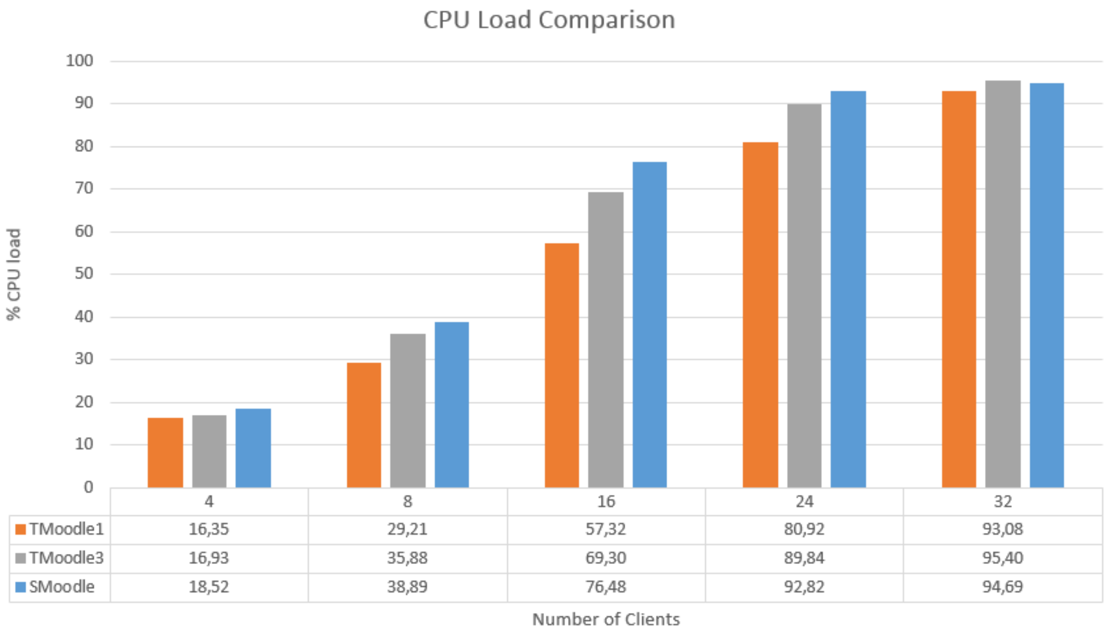
4.2.1 SMoodle vs TMoodle

Here we compare SMoodle with TMoodle, two different tests conducted to evaluate the coordination scheme at the application level. These tests simulate resource concurrency, since all clients attempt to access the same resource, which is the user creation resource. We limited this concurrency, by allowing only one or three simultaneous clients accessing the resource at any given time, depending on the test. Imagining the scenario of allowing only one client per resource, each client tries to create a user in the Moodle's web application, if the resource is available, the resource owner gives access to the resource solicitor and all other requests to this resource are rejected while the resource solicitor do not free the resource (or the 2 seconds timeout expires). If the request of a resource solicitor is rejected the resource solicitor is warned and can perform another operation in the Moodle's web application.

The results obtained by these tests (SMoodle, TMoodle1, and TMoodle3) are shown in Figure 4.3 which represents the percentage of CPU load of the Moodle's server per number of clients; Figure 4.4 represents the number of requests (GET and POST) executed in the Moodle's server per number of clients (the requests rejected are not represented here); and Figure 4.5 represents the page load latency per number of clients, measuring the time which clients wait for a page after a request has been submitted. All this figures join three different tests representing by three different color bars. The orange bars represent the TMoodle1 test (which allows one client per resource), the gray bars represent the TMoodle3 test (which allows three clients per resource) and blue bars represent SMoodle (Moodle without any changes to its original version (Moodle 3.2)).

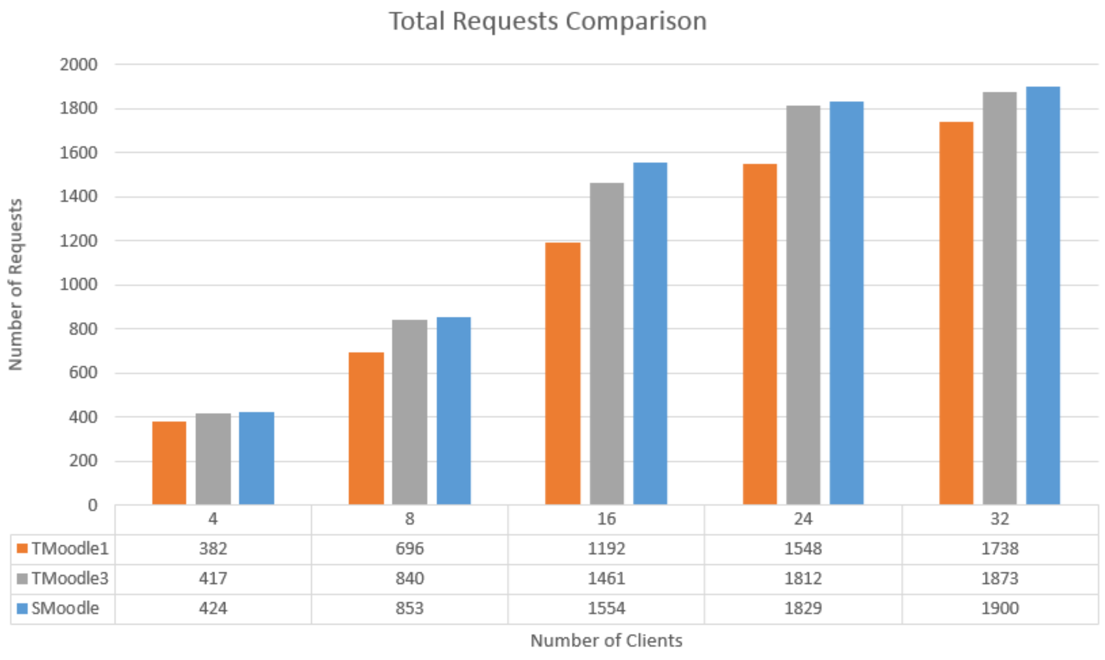
Tables 4.4, 4.5, and 4.3 summarize the data illustrated in Figures 4.3, 4.4 , and 4.5 in table form, adding the number of rejected requests, and making the distinction between POST and GET requests, and latency due to the communication used for coordinating accesses through the Legion network.

Note that the data presented in the tests refers to requests made on the server, they do not represent the total requests made by the clients. Thus, the total number of requests made by clients will be equal to TotalRequests plus Rejected ones. Since the data presented reflects only the work done by the server, for example for 4 clients in the Table 4.4, we can assert that of 411 requests (382 + 29) about 29 were rejected, and should not be interpreted as, at of 382 requests 29 were rejected.



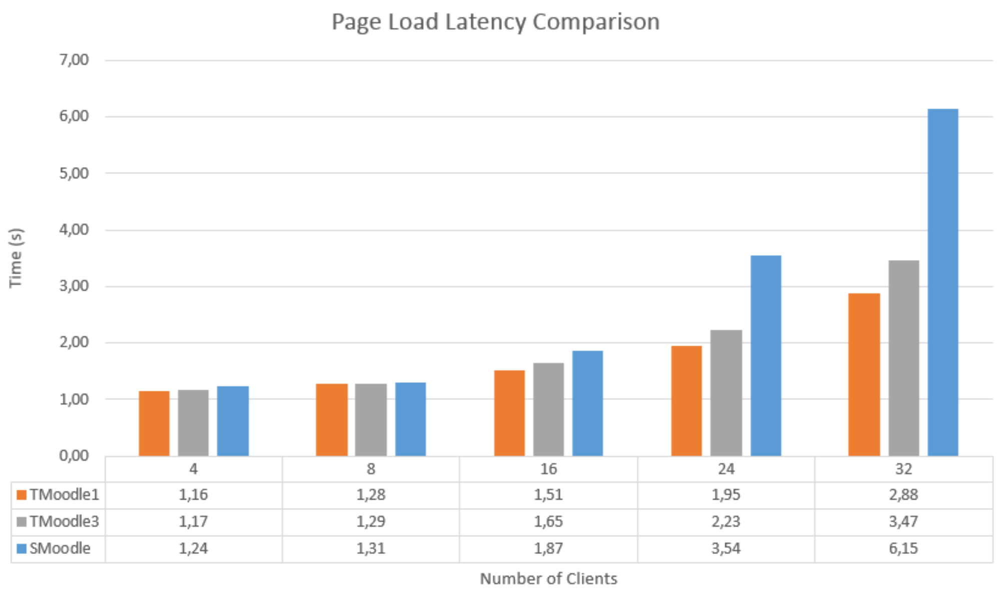
	4	8	16	24	32
🟧 TMoodle1	16,35	29,21	57,32	80,92	93,08
⬜ TMoodle3	16,93	35,88	69,30	89,84	95,40
🟦 SMoodle	18,52	38,89	76,48	92,82	94,69

Figure 4.3: CPU usage of tests SMoodle and TMoodle



	4	8	16	24	32
🟧 TMoodle1	382	696	1192	1548	1738
⬜ TMoodle3	417	840	1461	1812	1873
🟦 SMoodle	424	853	1554	1829	1900

Figure 4.4: Number of requests of tests SMoodle and TMoodle



	4	8	16	24	32
🟧 TMoodle1	1,16	1,28	1,51	1,95	2,88
▨ TMoodle3	1,17	1,29	1,65	2,23	3,47
🟦 SMoodle	1,24	1,31	1,87	3,54	6,15

Figure 4.5: Page load latency of tests SMoodle and TMoodle

Table 4.3: Moodle results (SMoodle)

Clients	Total Requests	GET	POST	%GET	Page Load Latency (s)	% Server Load
4	424	317	107	74,76	1,24	18,52
8	853	638	215	74,79	1,31	38,89
16	1554	1162	392	74,77	1,87	76,48
24	1829	1365	463	74,63	3,54	92,82
32	1900	1414	486	74,42	6,15	94,69

Table 4.4: Throttle allowing 1 client per resource (TMoodle1)

Clients	Total Requests	GET	POST	Rejected	%GET	Page Load Latency (s)	Legion Latency (s)	% Server Load
4	382	305	77	29	79,84	1,16	0,01	16,35
8	696	580	116	102	83,33	1,28	0,02	29,21
16	1192	1047	145	279	87,84	1,51	0,07	57,32
24	1548	1408	140	439	90,96	1,95	0,07	80,92
32	1738	1610	128	555	92,64	2,88	0,09	93,08

Table 4.5: Throttle allowing 3 clients per resource (TMoodle3)

Clients	Total Requests	GET	POST	Rejected	%GET	Page Load Latency (s)	Legion Latency (s)	% Server Load
4	417	317	100	0	76,02	1,17	0,02	16,93
8	840	639	201	3	76,07	1,29	0,03	35,88
16	1461	1151	310	80	78,78	1,65	0,05	69,30
24	1812	1483	329	206	81,84	2,23	0,07	89,84
32	1873	1594	279	323	85,10	3,47	0,06	95,40

In Figure 4.3, we can observe that the objective of reducing the load in the centralized server was achieved, since in both tests of TMoodle (TMoodle1 and TMoodle3) the percentage of CPU utilization is always below the percentage of CPU utilization presented by the original version of Moodle (i.e., SMoodle). The CPU load percentage is lower in Moodle's Throttle versions because we are rejecting requests above a given threshold (1 client or 3 clients), making fewer requests to the centralized component, as can be seen in Figure 4.4. Consequently, page load latency is also smaller in the TMoodle tests, since the server in these tests is less overloaded than the original version of Moodle.

In the Tables 4.4, 4.5 and 4.3 we can also relate the percentage of GET and POST requests made in all tests. Since Moodle's original version accepts all requests that are made, the ratio of GET requests is more or less constant assuming a value of about 75%. The same does not happen in the Throttle versions (i.e., TMoodle1 and TMoodle3), since the P2P network/resource owners block some requests made to the server, saving CPU load on the centralized component.

In the TMoodle1 (Table 4.4), the contention generated to access the resource is greater, since only one client at a time can access it. In this way, TMoodle tests present a higher number of GET operations (i.e., higher percentage of GETs) than the number of GET operations presented in the SMoodle test, due to the rejection of POST operations by resource owners in this high contention environment. This percentage increases with the increasing of network participants, this tendency reaches the value of about 93% in a 32 clients network. Note that, in the TMoodle3 test the percentage of GET operations also presents a increasing tendency, but at a lower rate than TMoodle1 GET operation percentage (table 4.5). This is due to the fact that the contention for accessing the resource is lower than the contention present in TMoodle1, since there can be 3 clients simultaneously accessing the resource (instead of only one as TMoodle1) decreasing the number of rejected POST operations.

It is possible to extract more interesting conclusions from the SMoodle and the TMoodle results. From Tables 4.4, 4.5, and 4.3, and from Figure 4.5, observind data regarding the Legion network latency and page load latency, we can observe that the time that the P2P network takes to process the requests (Legion Latency) is residual, never reaching values higher than 0,1 seconds and in conditions of greater contention (24 and 32 clients) the page load latency of Moodle's original version (i.e., latency perceived by clients when executing the operations) is always higher than the one presented in both Throttled tests (i.e., TMoodle1 and TMoodle3). This page load latency difference can be explained with the percentage of GET operations since these operations require less server effort, hence reducing its load. This implies that the higher the percentage of GET operations, the effort imposed on the server (CPU Load percentage) and consequently the page load latency will become lower, as shown in the last two rows of Tables 4.4, 4.5, and 4.3. For example, in the case of 24 clients in the TMoodle1 test, the percentage of GET operations is around 91%, the page load latency is about 1,95 seconds and CPU usage is about 81%. For 24 clients in TMoodle3 test, the values for the same fields are about 82% of GET's, 2,23 seconds of page load latency, and about 90% of CPU usage. For the original version of

Moodle the values are 75%, 6,15 seconds and about of 95%. These values show the effect of the percentage of GET operations on the effort imposed on the server and consequently on page load latency.

4.2.2 SMoodle vs TQMoodle

Similar to the previous comparison, we compare the results of SMoodle test(i.e., Moodle's original version) with the results of the Moodle's web application integrated with our coordinate system, in this case TQMoodle. What changes from the tests previously presented is the usage of the waiting queue, instead of simply reject requests. After each POST request, clients wait for a maximum of 8 seconds to perform the next operation, until this threshold is reached the clients are interested in the resource, after 8 seconds have passed, clients ignore any response referring to the performed request. With this test we wanted to improve the racio of accepted POST operations, adding a queue to all requests made by clients that want to access a resource. Eventually the request for a given resource will be answered positively and that client can use the resource or, if not interested in the resource anymore, manifest its disinterest releasing it. In some way, the TQMoodle test gives to the user the power to decide if he want to continue with other operation or if he is willing to wait for the desired resource, since the access will eventually be granted.

Similar to the previous tests, this tests also collected server CPU load (Figure 4.6), the total number of requests executed on the server (Figure 4.7), and page load latency (Figure 4.8). Tables 4.3, 4.6, and 4.7 resume the values presented in the figures.

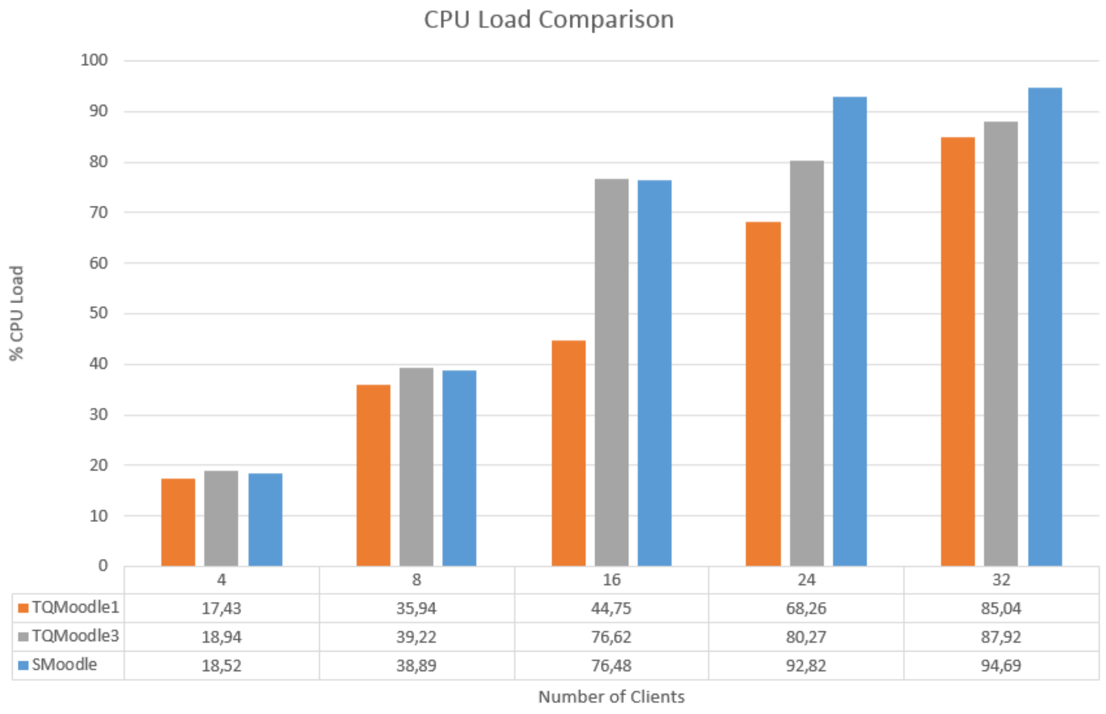
Table 4.6: Throttle-Queue allowing 1 client per resource (TQMoodle1)

Clients	Total Requests	GET	POST	Rejected	%GET	Page Load Latency (s)	Legion Latency (s)	% Server Load
4	416	316	100	45	75,96	1,26	0,29	17,43
8	801	608	193	151	75,91	1,44	0,85	35,94
16	858	777	81	336	90,56	1,79	9,99	44,75
24	1216	1143	73	464	94,00	1,89	10,17	68,26
32	1503	1428	75	599	95,01	2,67	10,88	85,04

Table 4.7: Throttle-Queue allowing 3 clients per resource (TQMoodle3)

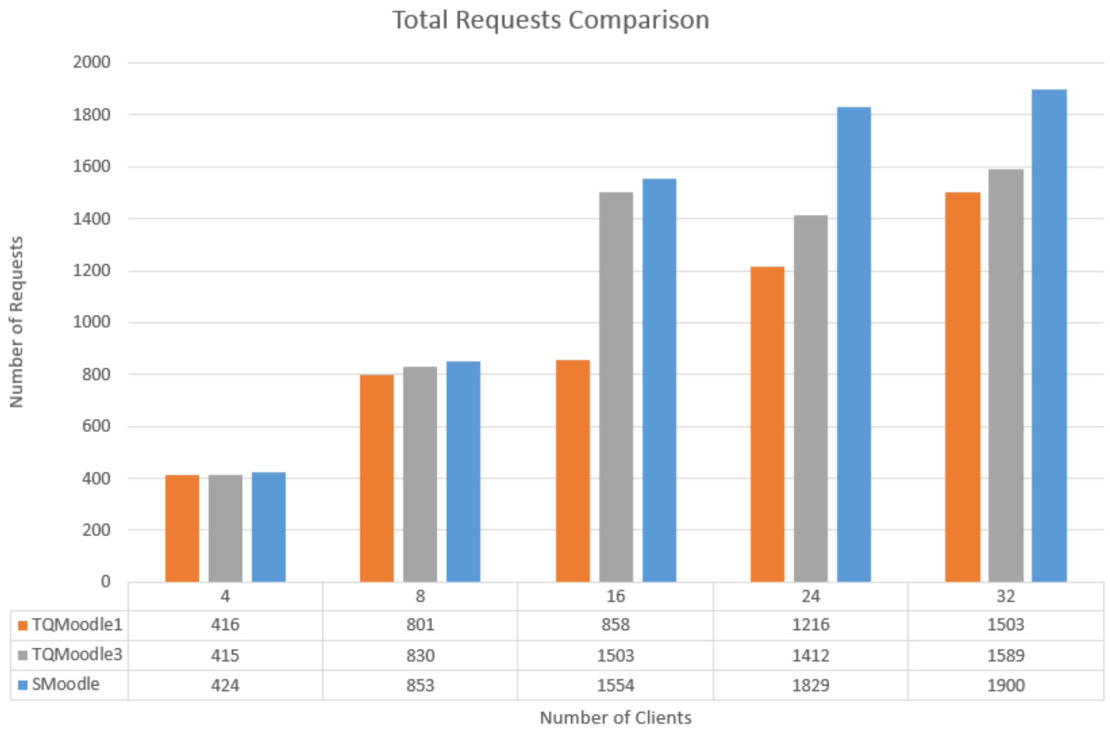
Clients	Total Requests	GET	POST	Rejected	%GET	Page Load Latency (s)	Legion Latency (s)	% Server Load
4	415	315	100	1	75,90	1,41	0,03	18,94
8	830	631	199	6	76,02	1,44	0,05	39,22
16	1503	1142	361	126	75,98	1,95	0,28	76,62
24	1412	1157	255	448	81,94	2,35	8,05	80,27
32	1589	1398	191	567	87,98	3,03	9,43	87,92

In these tests, observing Figure 4.6 and Figure 4.8, the SMoodle test presents higher CPU usage and page load latency than the TQMoodle1 and TMoodle3 test. Similarly to the previous tests, these decreases are due to the fact that some of the POST operations are being blocked by the P2P network, with fewer operations being performed on the



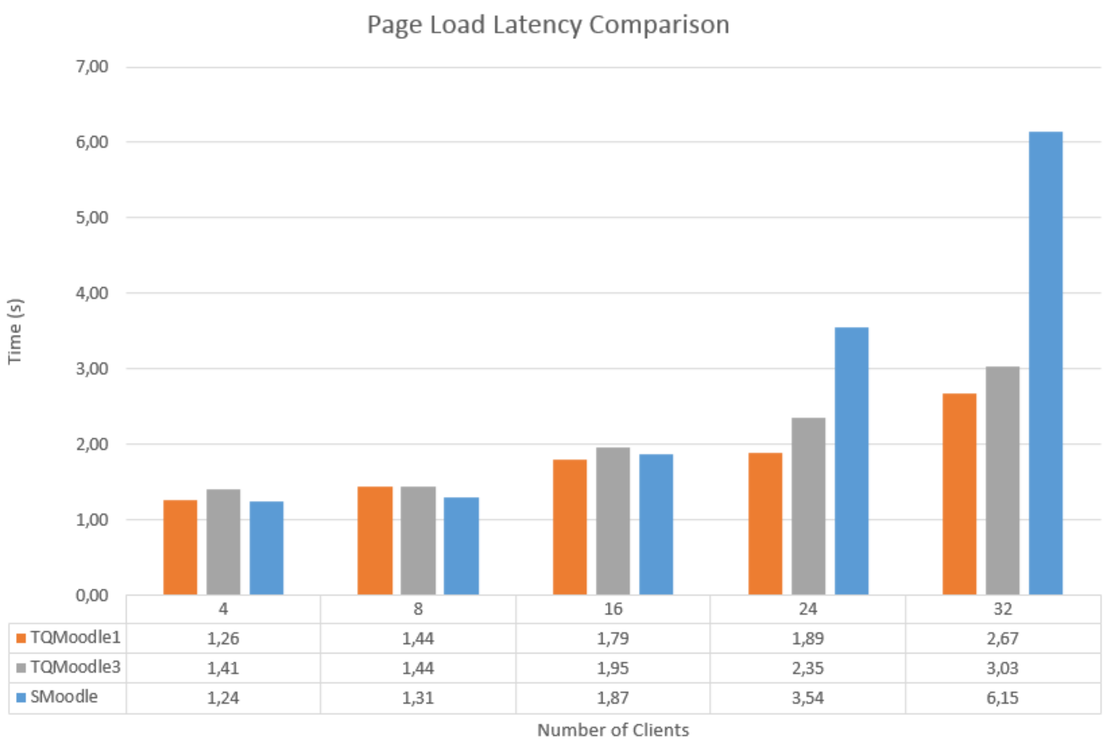
	4	8	16	24	32
🟧 TQMoodle1	17,43	35,94	44,75	68,26	85,04
⬜ TQMoodle3	18,94	39,22	76,62	80,27	87,92
🟦 SMoodle	18,52	38,89	76,48	92,82	94,69

Figure 4.6: CPU usage of tests SMoodle and TQMoodle



	4	8	16	24	32
🟧 TQMoodle1	416	801	858	1216	1503
⬜ TQMoodle3	415	830	1503	1412	1589
🟦 SMoodle	424	853	1554	1829	1900

Figure 4.7: Number of requests of tests SMoodle and TQMoodle



	4	8	16	24	32
🟧 TQMoodle1	1,26	1,44	1,79	1,89	2,67
■ TQMoodle3	1,41	1,44	1,95	2,35	3,03
🟦 SMoodle	1,24	1,31	1,87	3,54	6,15

Figure 4.8: Page load latency of tests SMoodle and TQMoodle

server in the throttle queue tests compared to the operations performed in the original version of Moodle (Figure 4.7). Remembering the previous results shown in 4.2.1, the results present here are roughly similar when considering periods of higher concurrency (16, 24, and 32 clients); since TQMoodle results also present a lower CPU usage, lower number of operations, and consequently lower page load latency than the SMoodle.
Note that, in TQMoodle test, the resource owner does not rejects the access requests instead, it places the requests of clients in a queue that will eventually be servedby having the resource owner respond affirmatively to the client who asked for the resource. Passing the control to the resource solicitor, whether it still desires or not access the asked resource, induces an increasing of the response time of the P2P network, reaching very high values when the queue reaches large sizes. For instance, in Table 4.6 in periods of high concurrency (16, 24, and 32 clients) the P2P network takes an average of over 9,99 seconds to respond to the resource solicitors. This time exceeds the 8 seconds that the client in this test is willing to wait for accessing the resource, causing the number of requests rejected by the resource solicitor to be very high, which consequently increases the percentage of GET operations that the clients do. The same happens in TQMoodle3 clients version 4.7, in which periods of higher competition located in networks of 24 and 32 clients, accounting for a mean response latency above the waiting time of the clients.

Chapter 5

Conclusion

Accessed anywhere and on any device with an Internet connection, web applications are nowadays seen as a better alternative to desktop applications. Most of them follow a client-server architecture, making its users follow this model of centralized communication. The availability of these services is inevitably associated with their centralized components, if user activity overloads these components, then the web application may become unavailable to all users.

This work proposed a new low-cost distributed system that can handle peaks of activity over web applications, maintaining server CPU load and allowing clients to continue interacting with the web application. As an alternative to the conventional client-server architecture, we proposed a system which coordinates access to the server in order to minimize the load in the centralized component, using coordination primitives and a Peer-to-Peer network. The proposed system was designed and implemented by taking advantage of the advancements of technologies on the browser, which allow browser-to-browser communication transparently (e.g., Javascript WebRTC API). The Legion framework used to create the Peer-to-Peer network makes use of WebRTC to enable direct connections between clients of the web application in a completely transparent way. We have designed a DHT for Legion based on the design of Chord, where peers are organized in a ring. The ring-shaped network allowed server resources to be distributed by network participants, making them resource owners. These special participants were in charge of deciding who can (or can not) access the centralized component based on a threshold set by the programmer.

Our system was evaluated in two different aspects, the architecture part and the application part. With respect to the architecture we evaluated the maximum number of hops given by a message on the network and we verified the join stabilization time of a peer. Connections between peers are restricted only by their position in the ring since peers must be connected to their successor and their predecessor. The rest of the connections, called jumps in the network were made randomly, following a minimum and maximum limit of connections. This randomness in the creation of this type of connections (network jumps) allows the messages to be delivered in less steps.

At the application level we tested the system in two different ways. One test referred as throttle (i.e., TMoodle), rejected the POST requests above a defined threshold (1 client or 3 concurrent clients accessing one resource) and another test, the throttle queue (TQ-Moodle), in which rejected requests were placed in a waiting queue and served as the server becomes available. In both tests it was possible to verify that our system reduces the CPU load on the server and the page load latency, at cost of a smaller number of requests that our system serves. It was also verified that in the TMoodle test, the latency added by the network P2P was insignificant, this is explained by the absolute rejection of the requests that exceed the defined threshold. The same has not been observed in the TQMoodle test, where latency imposed by P2P network is very high. This happened because clients requesting access to the resource will be served only when the server can serve them. Thus a client requesting access to the resource may not have immediately access to the desired resource, being necessary to wait for a affirmative response, increasing the latency of responses given by the P2P network.

5.1 Future Work

During the development of this work, we identified possible future improvements to it. The network presented in Section 3.2.1 presents some limitations and a continuous development is essential to keep its robustness according to the needs imposed by its users. For instance, in this work we kept peers joining the system in a controlled rate which is not always true in a deployed web application environment. In these type of web applications, our network need to be robust enough to deal with peaks of peers joining (or leaving) the system, maintaining the DHT consistent.

The current strategy to coordinate the access to the web application server resources, revealed that the server usage could be saved in detriment of the number of requests served. It would be interesting to test new coordination strategies in order to improve the percentage of rejected requests. The results also demonstrated that the coordination system implemented gives more relevance to GET requests in detriment of POST requests, lowering its percentage in contention environments. In future work we want to explore the combination of this work with strategies which aim to save CPU usage in contexts of GET operations(i.e., cache).

The security of the information changed between users was totally ignored by this work, since it was not its principal focus. In future work, we aim to explore security mechanisms to improve the integrity and privacy of the data exchanged by users in the context of this work.

Bibliography

[1] D. Agrawal and A. El Abbadi. "An efficient and fault-tolerant solution for distributed mutual exclusion". In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 1–20.

[2] *Amazon S3*. URL: https://aws.amazon.com/s3/details/.

[3] *Amazon S3 SLA*. URL: https://aws.amazon.com/s3/sla/.

[4] 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.

[5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. "SETI@Home: An Experiment in Public-resource Computing". In: *Commun. ACM* 45.11 (Nov. 2002), pp. 56–61. ISSN: 0001-0782. DOI: 10.1145/581571.581573. URL: http://doi.acm.org/10.1145/581571.581573.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[7] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. "Content-based publish-subscribe over structured overlay networks". In: *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*. IEEE. 2005, pp. 437–446.

[8] S. A. Baset and H. Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol". In: *arXiv preprint cs/0412017* (2004).

[9] M. Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 335–350.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.

[11] E. Chang and R. Roberts. “An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processes”. In: *Commun. ACM* 22.5 (May 1979), pp. 281–283. ISSN: 0001-0782. DOI: 10.1145/359104.359108. URL: http://doi.acm.org/10.1145/359104.359108.

[12] F Chang, J Dean, S Ghemawat, W. Hsieh, D. Wallach, M Burrows, T Chandra, A Fikes, and R Gruber. “Bigtable: A distributed structured data storage system”. In: *7th OSDI*. Vol. 26. 2006, pp. 305–314.

[13] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. “A Hierarchical Internet Object Cache.” In: *USENIX Annual Technical Conference*. 1996, pp. 153–164.

[14] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. “Freenet: A Distributed Anonymous Information Storage and Retrieval System”. In: *Designing Privacy Enhancing Technologies on International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Ed. by H. Federrath. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: http://link.springer.com/chapter/10.1007/3-540-44702-4_4.

[15] B. Cohen. *The BitTorrent protocol specification, version 11031*. 2008.

[16] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011.

[17] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. “Crew: A gossip-based flash-dissemination system”. In: *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE. 2006, pp. 45–45.

[18] A. El-Sayed, V. Roca, and L. Mathy. “A survey of proposals for an alternative group communication service”. In: *IEEE network* 17.1 (2003), pp. 46–51.

[19] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. “Lightweight probabilistic broadcast”. In: *ACM Transactions on Computer Systems (TOCS)* 21.4 (2003), pp. 341–374.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. “Summary cache: a scalable wide-area web cache sharing protocol”. In: *IEEE/ACM Transactions on Networking (TON)* 8.3 (2000), pp. 281–293.

[21] M. J. Fisher, N. Lynch, and M. S. Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM* 32.2 (1985), pp. 374–382.

[22] 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.

[23] H. Garcia-Molina. “Elections in a distributed computing system”. In: *IEEE Trans. Computers* 31.1 (1982), pp. 48–59.

[24] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. “Java message service”. In: *Sun Microsystems Inc., Santa Clara, CA* (2002), p. 9.

[25] B. Hayes. “Cloud Computing”. In: *Commun. ACM* 51.7 (July 2008), pp. 9–11. ISSN: 0001-0782. DOI: 10.1145/1364782.1364786. URL: http://doi.acm.org/10.1145/1364782.1364786.

[26] K. M.L.P.a.G.A. e. Jean Bacon David Eyers. *Middleware 2005: ACM/IFIP/USENIX 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005. Proceedings*. 1st ed. Lecture Notes in Computer Science 3790 : Programming and Software Engineering. Springer-Verlag Berlin Heidelberg, 2005. ISBN: 3540303235,9783540303237. URL: http://gen.lib.rus.ec/book/index.php?md5=13FBCD492D5C94E41F57FD68FC5B67EB.

[27] X. Jiang, Y. Dong, D. Xu, and B. Bhargava. “GnuStream: a P2P media streaming system prototype”. In: *Multimedia and Expo, 2003. ICME’03. Proceedings. 2003 International Conference on*. Vol. 2. IEEE. 2003, pp. II–325.

[28] L. Lamport. “The part-time parliament”. In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[29] L. Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[30] L. Lamport et al. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[31] L. Lamport, R. Shostak, and M. Pease. “The Byzantine generals problem”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[32] J. Leitao, J. Pereira, and L. Rodrigues. “Epidemic broadcast trees”. In: *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE. 2007, pp. 301–310.

[33] J. Leitao, J. Pereira, and L. Rodrigues. “HyParView: A membership protocol for reliable gossip-based broadcast”. In: *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*. IEEE. 2007, pp. 419–429.

[34] J. C. A. Leitão. “Topology Management for Unstructured Overlay Networks”. PhD thesis. Universidade Técnica de Lisboa, Instituto Superior Técnico, 2012. URL: http://asc.di.fct.unl.pt/~jleitao/pdf/LeitaoPhDThesis.pdf.

[35] A. van der Linde. “Enriching Web Applications with Browser-to-Browser Communication”. MA thesis. Faculdade de Ciências e Tecnologias, Universidade Nova de Lisboa, Nov. 2015.

[36] 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: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 283–292.

[37] M. Maekawa. "An algorithm for mutual exclusion in decentralized systems". In: *ACM Transactions on Computer Systems (TOCS)* 3.2 (1985), pp. 145–159.

[38] G. Mühl, F. Ludger, and P. Pietzuch. *Distributed event-based systems*. Springer Science & Business Media, 2006.

[39] *Microsoft Azure*. URL: https://azure.microsoft.com.

[40] N. Mohamed and T. Michel. "How to detect a failure and regenerate the token in the log (n) distributed algorithm for mutual exclusion". In: *International Workshop on Distributed Algorithms*. Springer. 1987, pp. 155–166.

[41] *Moodle 3.2*. URL: https://docs.moodle.org/32/en/Main_page.

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

[43] J. Pereira, L. Rodrigues, A. Pinto, and R. Oliveira. "Low latency probabilistic broadcast in wide area networks". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 299–308.

[44] *Publish/Subscribe topic-based architecture image*. URL: https://i-msdn.sec.s-msft.com/dynimg/IC141963.gif.

[45] K. Raymond. "A tree-based algorithm for distributed mutual exclusion". In: *ACM Transactions on Computer Systems (TOCS)* 7.1 (1989), pp. 61–77.

[46] M. Raynal. "A simple taxonomy for distributed mutual exclusion algorithms". In: *ACM SIGOPS Operating Systems Review* 25.2 (1991), pp. 47–50.

[47] *Response Times: The 3 important limits*. URL: https://www.nngroup.com/articles/response-times-3-important-limits/.

[48] 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.

[49] R. Rodrigues and P. Druschel. "Peer-to-peer Systems". In: *Communications of the ACM* 53.10 (2010). ISSN: 0001-0782. DOI: 10.1145/1831407.1831427.. URL: http://doi.acm.org/10.1145/1831407.1831427..

[50] A. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings*. Ed. by R. Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-45518-9. DOI: 10.1007/3-540-45518-3_18. URL: http://dx.doi.org/10.1007/3-540-45518-3_18.

[51] J. Sahoo, S. Mohapatra, and R. Lath. "Virtualization: A survey on concepts, taxonomy and associated security issues". In: *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE. 2010, pp. 222–226.

[52] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: http://doi.acm.org/10.1145/964723.383071.

[53] I. Suzuki and T. Kasami. "A distributed mutual exclusion algorithm". In: *ACM Transactions on Computer Systems (TOCS)* 3.4 (1985), pp. 344–349.

[54] A. S. Tanenbaum and M. Van Steen. *Distributed systems*. Prentice-Hall, 2007.

[55] Q. H. Vu, M. Lupu, and B. C. Ooi. *Peer-to-Peer Computing: Principles and Applications*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642035132, 9783642035135.

[56] *WebRTC*. URL: https://webrtc.org.

[57] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour. *Service Level Agreements for Cloud Computing*. Springer Publishing Company, Incorporated, 2011. ISBN: 1461416132, 9781461416135.

[58] D. Wu, Y. T. Hou, W. Zhu, Y.-Q. Zhang, and J. M. Peha. "Streaming video over the Internet: approaches and directions". In: *IEEE Transactions on circuits and systems for video technology* 11.3 (2001), pp. 282–300.

[59] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy. "Diamond: automating data management and storage for wide-area, reactive applications". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association. 2016, pp. 723–738.