Filipe Alexandre Pereira Luís

Licenciado em Engenharia Informática

Distributed, decentralized, and scalable Coordination Primitives

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

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

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2017

Abstract

Nowadays, the Internet presents itself as a dynamic environment that changes according to the behavior 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 inaccessible.

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 client requests is greater than the number of requests that the server (or the centralized architecture as a whole) can handle. To avoid this behavior 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 obtain 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 com 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 este comportamento 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

1 Introduction **1**

1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Expected Contributions . . . 2
1.4 Document Organization . . . 2

2 Related Work **5**

2.1 Distributed Systems Architectures . . . 6
2.1.1 Centralized . . . 6
2.1.2 Partly Centralized . . . 7
2.1.3 Partly Decentralized . . . 9
2.1.4 Decentralized . . . 10
2.1.5 Application examples . . . 13
2.2 Communication Primitives . . . 15
2.2.1 Message-oriented Communication . . . 15
2.2.2 Streaming Oriented Communication . . . 17
2.2.3 Multicast Communication . . . 19
2.2.4 Applications examples . . . 20
2.3 Coordination and agreement Primitives . . . 21
2.3.1 Mutual Exclusion . . . 21
2.3.2 Elections . . . 22
2.3.3 Consensus . . . 23
2.3.4 Applications examples . . . 23
2.4 Summary . . . 25

3 Proposed Work **27**

3.1 Proposed Solution . . . 27
3.1.1 First Approach: Single Delegation . . . 28
3.1.2 Second Approach: Multi-Delegation . . . 29
3.2 Evaluation . . . 31
3.3 Work Plan . . . 32

Bibliography **35**

Chapter 1

Introduction

1.1 Context

Nowadays the Internet is not based on sharing static pages hierarchically organized from the home page 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 powerful, interactive 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 [9], which enables clients to act as a peer in a distributed peer-to-peer architecture effectively creating a browser-to-browser network.

Decentralized architectures have gained prominence over more centralized architectures and a good example of existing (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 [16], Napster[41], and BOINC [4].

The combination of these two aspects 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 to 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 Expected 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. The main contributions of the system are the following:

- A logical peer-to-peer networks that enables direct coordination between clients (e.g. browsers).
- Coordination and agreement protocols used to control surges in clients activity, preventing failures caused by resource exhaustion in the centralized component (e.g. server).
- A experimental study of the system benefits, when compared with typical client-server architectures.

1.4 Document Organization

The document is structured as follows:

Chapter 2 - Describes the related work. It begins with the overview of the existing architectures and it's followed by the existing communication and coordination primitives found in the literature, which is a fundamental aspect of the work proposed in this

document.

Chapter 3 - Describes the future work and its respective scheduling.

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:

- **Section 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 (partly centralized and partly decentralized), and peer-to-peer models, since they are the classical approaches to design distributed systems.

- **Section 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.

- **Section 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 our systems which requires them to be more 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, 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 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

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 server perspective, it 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 is at 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 provider or directly with the server. The scalability of this architecture is limited when the number of clients is greater than the amount that can be handled by the server.

Clients and server communicate with each other through a server provided API. This

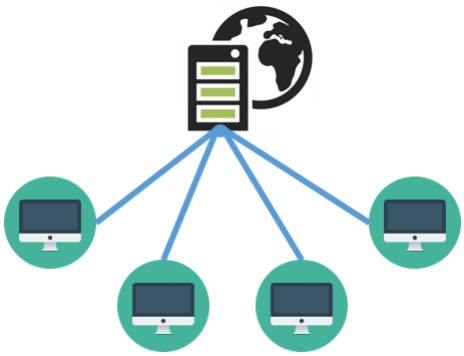


Figure 2.1: Client-Server Architecture

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 more about such communication primitives.

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

2.1.2 Partly Centralized

We classify cloud computing architecture (with servers in the same geographic location) in partly 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 connected with 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 [17]. 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 don 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. They are not required to know anything specific about cloud computing design to connect their computers to cloud server and use it. Thus is possible to develop and test their concepts faster and without wasting costly resources with this architecture design. Developers are able to

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

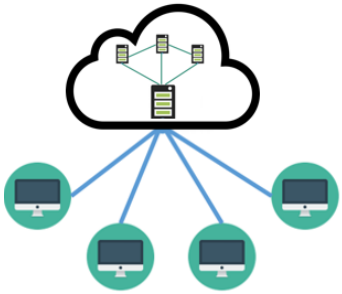


Figure 2.2: Cloud Computing Architecture

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 he have 2 machines where only 30% of each 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 provided that includes the quality, availability, and responsibility of the service concerned [56]. 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%. Anything below this guarantee leads Amazon to pay to the service user 25% of his bill [3].

Cloud computing also offers ease of setup, since it is possible to easily configure a system by outsourcing computation to cloud service providers, instead of maintaining computational infrastructure and managing complex software stocks [27].

We conclude that this type of cloud architecture differs from the classical client server model, since both enjoy 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 fault-tolerant. We consider that this property is an evidence that the cloud architecture has a distributed component, categorizing it in the partly centralized designs.

2.1.3 Partly 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. Clouds that maintain their hardware components in a single geographic location (as partly centralized architecture) can present latency values above 100 milliseconds to 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, defined 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, defined as geo-replication.

For instance, Google uses geo-distribution on GoogleDrive service providing lower latency since they have servers distributed across the planet. They also implement geo-replication, since users that access this Google service in Europe servers have the same content if they access it through America servers. 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.

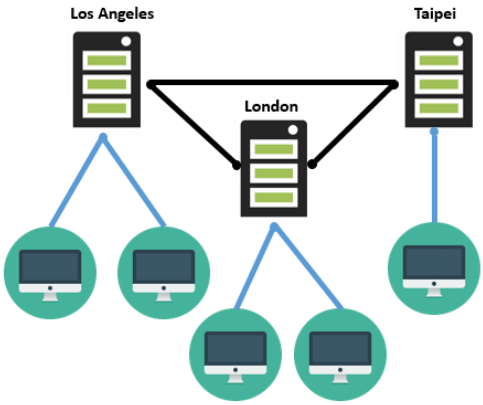


Figure 2.3: Geo-Distributed Architecture

We classify cloud computing architectures which implement geo-distribution as partly 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 [17]. 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 escape from 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 [41] music-sharing system, the Freenet [15] 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.

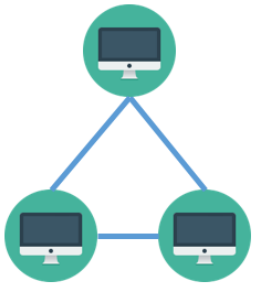


Figure 2.4: Peer-to-Peer Architecture

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

Partly Centralized P2P Systems

In this type of P2P systems, there usually exists a central component similar to the client-server model (referred 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 [41], 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 [16] is also an example of a partly centralized P2P system, since it relies on central servers that index contents distributed across the network.

Fully Decentralized P2P Systems

Unlike partly centralized P2P systems, this design completely avoid the use of a centralized node 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, which 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 as PAXOS [30, 32], that helps participants of a system to reach consensus are discussed later on section 2.3, however, such solutions 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]. 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 peer 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[24], 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 whose topology is 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 are at the application level [55]. Applications that use this type of overlay include FreeNet [15], 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 achieve the target data.

2.1.5 Application examples

Chord[52] - Chord 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 is 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 higher identifier points to the one with lowest identifier, a ring topology is formed.

To achieve fault-tolerance, each nodes 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.

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 nodes to which they are 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

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

source node. This type of query dissemination can lead to some lack of security, since Distributed Denial of Service (DDOS) attacks are easy to perform. 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, which 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 partly 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 [58] - 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, this 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 allow 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.

2.2 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 inteface[17]. 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 [17].

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 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 [17].

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) [26] where clients remove messages from the queues assigned to hold their messages.

Event-based Communication

Event-based [39] 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 it is 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

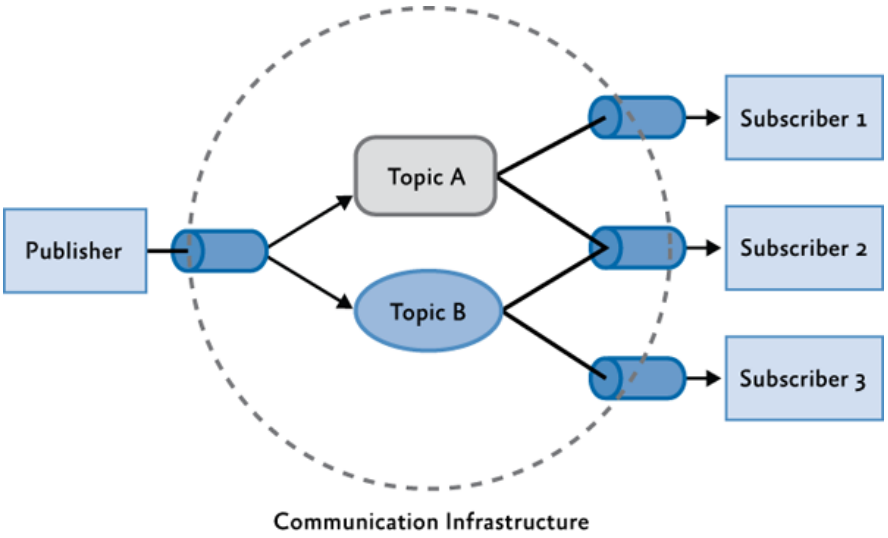


Figure 2.5: Publish/Subscribe topic-based Architecture [44]

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 [28], 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.

2.2.2 Streaming 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 [17]. 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 situation, 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[14, 21].

The centralized approach referred in [57] 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.

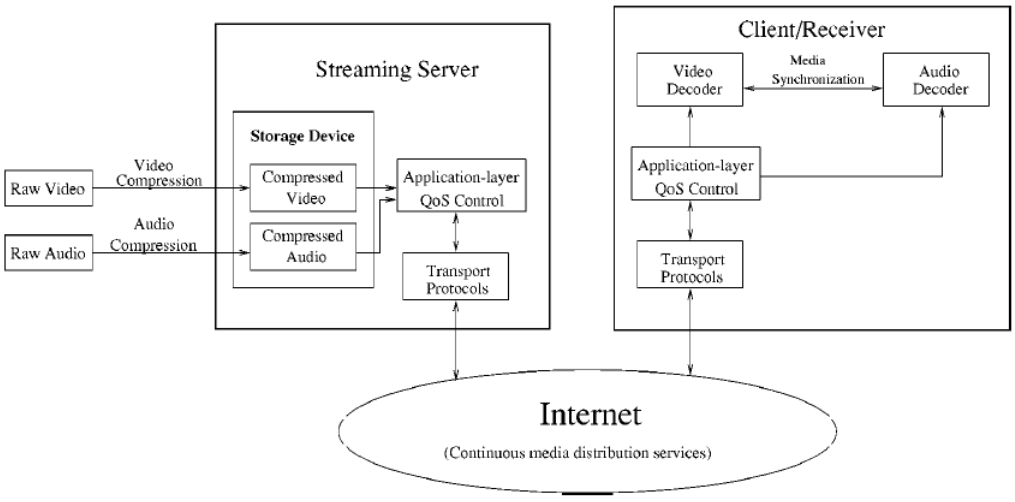


Figure 2.6: Streaming communication Client/Server Architecture [57]

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 [29], 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 [17]. 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, also 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[19]. 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 [35], 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 mask node failure and omissions.

Gossip Data Dissemination

This data dissemination technique reproduces the behaviour of the communication used by people in social networks 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 through the network with the aim of delivering them as quickly as possible among a large collection of nodes [17].

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 as [18, 20, 34, 43] suggests the use of partial views, that only stores a subset of network members in order to improve scalability.

In Gossip there are typically three strategies to exchange information updates [35]:

- **Eager push approach:** Nodes send message payload to randomly selected nodes.
- **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. Nodes receiving these notifications for unknown content, explicitly request the payload.

2.2.4 Applications examples

Scribe [11] - 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 events published. 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.

HyParView [34] - HyParView is a message dissemination protocol that implements the gossip based communication paradigm. It proposes to disseminate data efficiently with high fault tolerance, which is 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.

2.3 Coordination and agreement 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 [17].

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 [17], discussed next. All of them must ensure safety, liveness and fairness properties. Safety ensures at most one process in the critical section at a time, liveness promises that all processes that required 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, 31, 38] :** 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 sectio 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 over 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 these three properties [17]:

- **Integrity:** If all correct processes proposed the same value, then this value will be chosen in de decision state by a 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 [23] result determines that consensus is unsuitable in asynchronous systems where a node fails.

In a synchronous sytem and in presence of crash failures, Lamport [33] proved that if the system has $2F + 1$ correct processes working then consensus is achievable, where F is the 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 Applications examples

Ring based algorithm [12] - It 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.

Bully algorithm [25] - It 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 do 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, disrupting the safety property. Property which ensures that in the end of an election only the peer (non-crashed) with higher identifier is elected.

Chubby [10] - Chubby 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 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 [13] is a distributed data storage that uses Chubby to elect masters.

2.4 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 is the case of cloud computing architecture. 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 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 the last section of this Chapter 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 next chapter we discuss the work plan for the dissertation. It will address the proposed solution of the problem introduced in Chapter 1, and further discusses how we plan to tackle the challenges that are the focus of this work.

Chapter 3

Proposed Work

In order to clarify the proposed solution we dedicate this Chapter to address examples where the problem is applicable and all the effort already made towards designing an adequate solution. We further discuss the planning of future work in the context of the thesis.

It is often difficult to handle the load variations that a web application suffers. Hence, in many of these applications are resources that are wasted or exhausted, leading to money loss. Thus, in order to obtain the best possible income, the amount of computational power allocated should vary according to the expected popularity of the resources available by service. While elasticity in cloud infrastructures might solve this problem, their use might be too expensive to small and medium sized web application operators.

An example of this, could be a public BitTorrent tracker deployed in someone's house, that can not handle all the requests due to its popularity. The actual student portal used by our University, the CLIP system is also a good example where its popularity in particular periods of the semester (e.g. when students need to plan their semester schedule), could be disadvantageous. It is usual for the system to be down in periods where students are choosing their schedule and it is not propitious to malicious attacks. We could enumerate other concrete examples but such exhaustive listing is outside the scope of this document.

3.1 Proposed Solution

As already mentioned, currently, most web applications follow the traditional client-server architecture and this brings the disadvantages associated with centralized architectures. We will address in more detail the existence of bottelneck and the lack of fault tolerance in these architecture types.

Increasing the number of machines that provide the service and the associated data replication, are the current employed techniques to overcome these two challenges. But as referred before, this approach is expensive and we propose to achieve a more accessible scheme for service providers with less monetary resources.

In the next sections, we present possible approaches that could be employed to address these challenges, and that we will explore in the context of the thesis. In an initial phase we consider that all the participants are trustable and the system is not targeted by malicious attacks, such as Distributed Denial of Service (DDOS). We consider also that clients are browsers. Later we plan to address some of the security problems mentioned.

3.1.1 First Approach: Single Delegation

Due to the fact that we are dealing with centralized architectures, the major problem encountered is the centralized component (server) and the lack of coordination primitives to access its resources. In this first iteration we propose the existence of coordination primitives that allow server to be (almost) always available. Thus, clients could organize themselves into a peer-to-peer network to allow direct communication between them. Once the network is created, the server could delegate to one client the mediation of access to either a resource or a part of a resource (assigning it can be easily partitionable, for instance in the context of CLIP, imagine that a client become responsible for managing subscriptions to a single laboratory class of a course). Thus, the server only will interact with clients that already have been approved by the coordinator node.

For instance, considering that the BitTorrent tracker is a public and much popular research paper sharing service. With our first proposed architecture, the server must delegate one client node to coordinate the access to the resource maintained by the server. After that, clients that desire a popular paper could communicate with the elected coordinator in order to get permission to directly access the main server that holds the client desired resource. Figure 3.1 is an illustration of this proposed architecture, in which the red circle represents the delegated node.

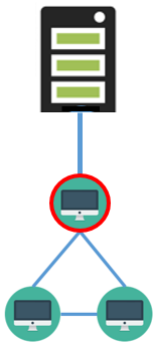


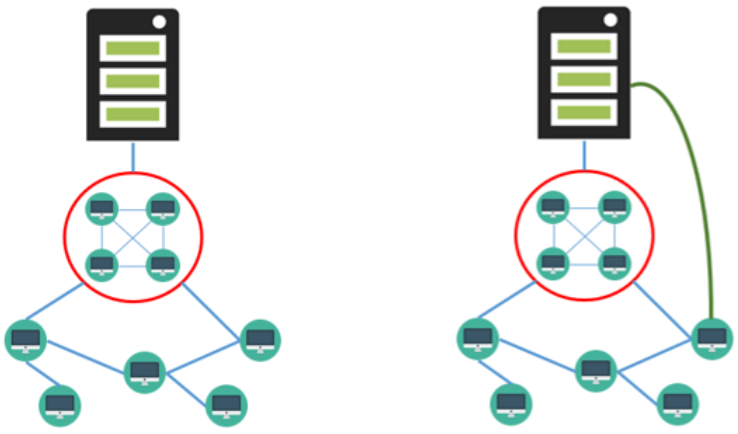
Figure 3.1: First architecture approach.

Using this architecture, the server would never be exhausted with client requests, since they need to overcome the first tier of coordination. In this way, the web application operators do not need to allocate more resources to handle the service overhead in periods of higher demand, hence saving money.

However, delegating the coordination of access to a resource maintained by the server, only to a client node is not properly removing the central component of the system. Since if the coordinator node fails the information that he had gets lost and the server will have to answer all the clients requests until a new coordinator is selected.

3.1.2 Second Approach: Multi-Delegation

The architecture presented above, maintains in some extent the existence of a centralized component (for each resource or shard of a resource), which is the case of the coordinator node. In order to overcome this issue, the main server could delegate to a set of nodes (Figure 3.2 (a)), rather than delegate responsibility to a single. This additional decentralization means that the system does not only depend on a single node to manage a resource (or shard), but on a set of nodes, making the system more robust, fault tolerant, and more balanced in terms of load. These set of nodes will act as a single virtual node, since all decisions are made using group coordination approaches referred in section 2.3, such as quorums. This architecture is illustrated below, in Figure 3.2, where the red circles represents the coordination group responsible for managing the access to a resource (or shard).



a Delegation b Server interaction

Figure 3.2: Second architecture approach.

Following the example presented in the previous section, in this proposed architecture

the server must delegate over a set of client nodes the coordination of access to a resource maintained by the server. Clients follow the behavior stated before, interacting with the coordination group in order to obtain the access to the paper desired, maintained by the main server. When granted, clients can interact directly with the server, as demonstrated in Figure 3.2 (b).

Next we explore the possible implementations for each component of the solutions described above (e.g. network, delegation of a coordinator etc).

3.1.2.1 Peer-to-Peer network

Clients that desire to achieve the resource owned by the main server need to be connected through a peer-to-peer network. In order to be part of this network, they could connect themselves to an internal or external centralized component that serves as entry point or facilitator of this join procedure, which enables clients to become part of the existing network.

The peer-to-peer network formed by the clients in both approaches, needs to locate the, or one of the node(s) managing a target resource and for that, an efficient resource discovery techniques must be used. The use of a Distributed Hash Table (DHT) is a very efficient way of locating resources in structured peer-to-peer networks, since it offers low search complexity of content, such as $O(Logn)$ achieved in Chord [52] protocol. However, due to clients activity in this type of services, the implementation of a DHT could not be practicable since nodes are always going in and out of the system (e.g. churn[1]), making DHT management very expensive, precluding the scalability of the system.

To overcome this problem, we could implement other DHT techniques such as Rollerchain [42] which relies on gossip-based mechanisms o avoid the loss of keys under heavy churn.

Additionally, the envisioned solution also requires that all nodes that materialize the coordination group be directly linked to each other. This will avoid message overhead in the coordination process and non expected behaviors, such as the situation where a malicious node changes other node's proposal.

To achieve that, we can be influenced by architectures such as Overnesia [36] (shown in Figure 3.3, in which the group with the red circle is the coordinator group), or implement techniques, such as the creation of direct connections between each group member after the server has delegated the coordination group, for example. In the latter, Legion [37] could be used as a framework to assist the construction of such overlays.

3.1.2.2 Coordination and Agreement mechanisms

In order to delegate a part of the system operation to a group of nodes, we plan to focus in coordination primitives described in section 2.3. A departure point might be the use

[1] Periods of time which the number of members in the network is unstable, high amount of nodes joining and leaving the system.

of quorums among nodes managing the access to a resource (or a shard of the resource).

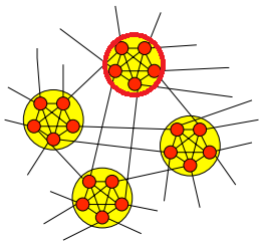


Figure 3.3: Overnesia architecture [36].

3.1.2.3 Security mechanisms

We are aware of the possible security issues, but as it was already referred it is not our main focus now. However, if we have time we will attempt to address them, thus we briefly discuss this next.

It will be necessary implement security mechanisms in order to ensure, that the delegated nodes are safe and will not adopt unusual behaviors. For instance, in order to grant the authenticity of the primary server we could implement public key algorithms. In the limit, the coordinator group could be malicious and a cyclic mechanism which elects new coordinator groups could be implemented.

Clients could try to access the main server without permission ,and to ensure that it has the permission granted by the coordination group, we could implement some token based techniques. In which the coordinator group gives an expirable token to the client with the authorization to access the main server, which clients then present this to the main server.

3.2 Evaluation

We are planning to evaluate our work in a cloud environment. Our servers will be running in a cloud platform such as Amazon AWS, and clients nodes could be implemented on the computers of the laboratories, thus achieving a good sample of high access rate to the service, or be emulated also in Amazon EC2 machines.

In order to prove that our work offers more guarantees than the actual centralized web application architectures, we will conduct comparative experiments between our approach and a typical web service architecture. Considering performance metrics we will focus on time complexity of serialized operations (actual made by servers of these centralized applications) and the time complexity of the system using our approach.

3.3 Work Plan

In this section we describe the work plan for the dissertation. Steps towards the final dissertation are divided and described below, which are roughly composed by system designing, implementation, evaluation and thesis writing.

Design - The main focus is to define the system architecture and design protocols to support the solution (leveraging some solutions found in the literature).

Implementation - This task is divided in three phases which constitute the development of our work. They are described below.

- **Phase 1**: In this first phase we will focus on the implementation of the communication layer that includes a peer-to-peer network connecting the clients in browsers.
- **Phase 2**: Once with the network is built, in this phase we will build the coordination primitives to help the main server to delegate responsibilities to clients in the network. Additionally a security module could be implemented in this phase (considering time constraints).
- **Phase 3**: Creation and integration with example applications to prove the properties of our work.

Evaluations - We plan to evaluate our work by comparing systems that use our work with systems that rely on a traditional centralized architecture.

Writing - This phase consists of writing the dissertation.

In Figure 3.4, is represented the schedule of the work planned for the following months also predicting the elaboration of scientific research papers that will possibly be submitted to national and international conferences.

Roughly the implementation will take approximately 14 weeks and is divided in three phases which varying from 4 to 7 weeks with overlapping periods. We also allocated approximately 7 weeks for the elaboration of a scientific paper to submit for inforum conference. The period of evaluation will take about 8 weeks and the last 11 weeks are allocated for the thesis writing.

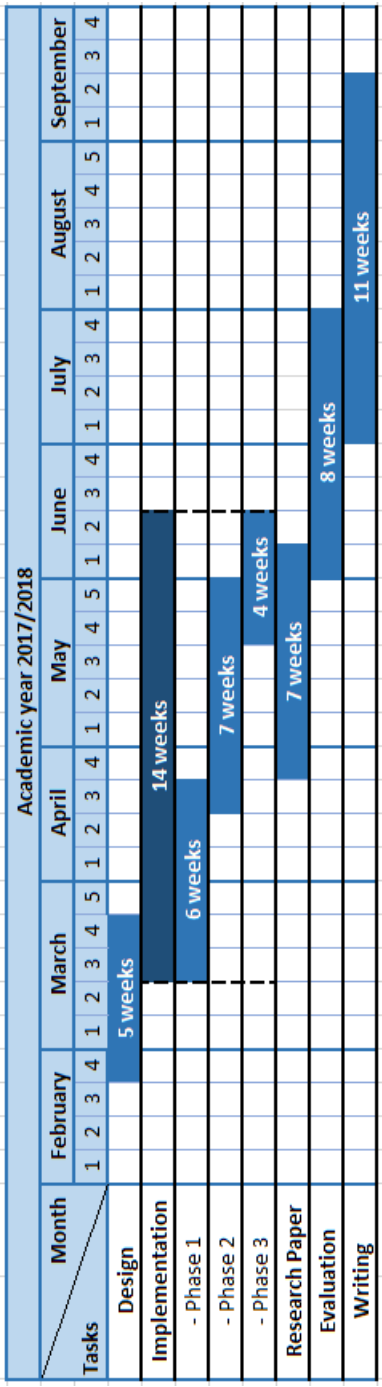


Figure 3.4: Planned work schedule.

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] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. "Webrtc 1.0: Real-time communication between browsers". In: *Working draft, W3C* 91 (2012).

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

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

[33] 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.

[35] 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.

[34] 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.

[36] J. C. A. Leitão and L. E. T. Rodrigues. "Overnesia: a resilient overlay network for virtual super-peers". In: *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE. 2014, pp. 281–290.

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

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

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

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

[42] J. Paiva, J. Leitao, and L. Rodrigues. "Rollerchain: A dht for efficient replication". In: *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*. IEEE. 2013, pp. 17–24.

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

[57] 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.

[58] 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.