TÉCNICO LISBOA

browserCloud.js - A federated community cloud served by a P2P overlay network on top of the web platform

David Dias

Thesis to obtain the Master of Science Degree

P2P Networks, Cloud Computing and Mobile Applications

Examination Committee

Chairperson: Prof. Doutor.
Supervisor: Prof. Doutor. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Doutor. João Dias Pereira

March 2015

Acknowledgements

THANK YOU EVERYBODY :D

20th of September, Lisbon

David Dias

–To all of the first followers, you undoubtly changed my life.

Abstract

Grid computing has been around since the 90's, its fundamental basis is to use idle resources in geographically distributed systems in order to maximize its efficiency, giving researchers access to computational resources to perform their jobs (e.g. studies, simulations, rendering, data processing, etc). This approach quickly grew into non grid environments, causing the appearance of projects such as SETI@Home or Folding@Home, leveraging volunteered shared resources and not only institution-wide data centers as before, giving the birth of Public Computing. Today, after having volunteering computing as a proven concept, we face the challenge of how to create a simple, effective, way for people to participate in this community efforts and even more importantly, how to reduce the friction of adoption by the developers and researchers to use this resources for their application. This thesis explores and innovates new ways to enable end user machines to communicate, using recent Web technologies such as WebRTC, creating a simple API that is familiar to those used to develop applications for the Cloud, but with resources provided by a community and not by a company or institution.

Resumo

IGUAL AO ABSTRACT MAS EM PORTUGUÊS

Palavras Chave

Computação na Nuvem, Redes entre pares, Computação voluntária, Partilha de ciclos, Computação distribuida e descentralizada, Plataforma Web, Javascript Tolerância à faltas, Mecanismo de reputação, Nuvem comunitária WebRTC

Keywords

Cloud Computing, Peer-to-peer, Voluntary Computing, Cycle Sharing, Decentralized Distributed Systems, Web Platform, Javascript, Fault Tolerance, Reputation Mechanism, Community Cloud WebRTC

Index

List of Figures

List of Tables

1 Introduction

"As Sivers highlighted, the first follower is probably more important than the leader: they validate the insanity. <3 first followers". – *Paul Campbell, founder of Tito, curator of ullconf and brio conference*

1.1 Background and Motivation

The Web has evolved considerably since its inception, specially in the last few years, with the profileration of the Web Browser, the birth of the Web Platform[1], leveraging the Browser capabilities to create a ecosystem of APIs and protocols that enabled it to be a host for first class applications. The Browser is the most ubiquitous runtime, making the Web Platform the number one target for developers to build their applications.

The Web, or the World Wide Web, as *Sir* Tim Berners Lee presented it to the world with the introduction of HTTP, was decentralized by design, every machine could act as a client and a server, however, with time, this panorama changed and although every machine ability to still run a HTTP client or server, due to network topology considerations, mainly NAT, users progressively lost the ability to connect to other user machines as it was intended, having to overcome some obstacles between interoperability of protocols and platforms. In essence, the barrier of entrance to create a P2P environment became so high, that systems moved to a pure centralized model. However, P2P was not forgotten and now more than ever, there is a chance for peers to connect between each other, perform resource discovery and sharing, due to the P2P cabilities brought to the browser when WebRTC was introduced.

Today, in the information communications technology landscape, user generated data has been growing at a large pace, with the introduction of social networks, search engines, Internet of Things, which led to innovation on home and vehicle automation. The storage, transfer,

[1] https://www.webplatform.org/

and carry out of processing and analysis of all this data brings the need for considerable new breakthroughs, enabling us to optimize systems towards a better and enhanced experience. However, how to use the information available to achieve these breakthroughs has been one of the main challenges since then.

Currently addressing these issues in part, Cloud Computing has revolutionized the computing landscape due to key advantages to developers/users over pre-existing computing paradigms, the main reasons are:

- Virtually unlimited scalability of resources, avoiding disruptive infrastructure replacements.
- Utility-inspired pay-as-you-go and self-service purchasing model, minimizing capital expenditure.
- Virtualization-enabled seamless usage and easier programming interfaces.
- Simple, portable internet service based interfaces, straightforward for non expert users, enabling adoption and use of cloud services without any prior training.

Grid computing had offered before a solution for high CPU bound computations, however it has high entry barriers, being necessary to have a large infrastructure, even if just to execute small or medium size computing jobs. Cloud computing solves this by offering a solution "pay-as-you-go", which transformed computing into an utility.

Still, even though we are able to integrate several Cloud providers into an open software stack, Cloud computing relies nowadays on centralized architectures, resorting to data centers, using mainly the Client-Server model. In this work, we pursue a shift in this paradigm, bridging the worlds of decentralized communications with efficient resource discovery capabilities, in a platform that is ubiquitous and powerful, the Web Platform.

1.2 Problem Statement

The resources required to execute a continuous, massive and significant analysis of the data available are controlled by a small subset of the companies and organizations. In order to

enable more people to use Big Data analysis, we need to reduce the cost that is inherent to process all this user information, which typically needs vast amounts of CPU cycles for processing, analysis and inference.

Unlike the conventional approach to make Cloud Computing 'green' (i.e. Green Computing) by improving datacenter's efficiency through expensive and strictly centralized control, our vision entails a shift in perspective, by enabling each user to contribute to this effort, leveraging his/her idle computing resources (sometimes up to 70% of power wasted), and thus reducing overall environmental footprint. Thus browserCloud.js resources are provided in a voluntary manner by common Internet users that want to share their idle computer cycles and storage available, while browsing the web, without having the concern to setup any application or system to do so.

Community Clouds(**?**)(**?**), are not a complete novelty in the Distributed Systems research area. However, existing models have been developed to follow the client-server model, transfering the data to the place where the computation will take place, which causes big bottlenecks in network traffic, limiting the amount of computed units done in a delimited window of time. One of browserCloud.js goals is exactly to mitigate this bottleneck by taking the computation (the algorithms that will perform operations over the data) to the machines where the data is stored.

1.2.1 Current Shortcomings

We've identified several issues with current solutions, most of which inspired us to pursue this research and the development of browserCloud.js, these are:

- Typical resource sharing networks do not offer an interface for a user to act as a consumer and contributor at the same time, specially when it comes to CPU resource sharing.
- If a user wants to consume resources from a given network, it is almost certain that user will have to develop his specific usecase for that runtime or runtimes where their tasks will be executed, interoperability is not a prime concern.
- There is a high level of entrance cost for a user to contribute to a given resource sharing network, typically passes through several steps of software installation and configuration.

1.3 Research Proposal

To accomplish this, we propose a new approach to abandon the classic centralized Cloud Computing paradigm, towards a common, dynamic. This, by means of a fully decentralized architecture, federating freely ad-hoc distributed and heterogeneous resources, with instant effective resource usage and progress. Additional goals may include: arbitration, service-level agreements, resource handover, compatibility and maximization of host's and user's criteria, and cost- and carbon-efficiency models.

This work will address extending the Web Platform with technologies such as: WebRTC, Emscripten, Javascript and IndexedDB to create a structured peer-to-peer overlay network, federating ad-hoc personal resources into a geo-distributed cloud infrastructure, representing the definition made by C.Shirky of what an peer-to-peer means:

"An application is peer-to-peer if it aggregates resources at the network's edge, and those resources can be anything. It can be content, it can be cycles, it can be storage space, it can be human presence.", C.Shirky (**?**)

We've named this system browserCloud.js. It has the possibility to grow organically with the number of users. The management of these resources is done by technologies and protocols present in the Web Platform, enabling desktop and mobile apps to use the resources available in a way that's familiar to developers.

1.4 Contributions

Our main goal with this work is to design and implement a system that is able to take advantage of volunteered computer cycles through the most ubiquitous growing platform, the browser. In order to create this system, several components will be developed:

- An efficient way to perform resource discovery, without a need for a central indexing.
- A distributed job scheduler able to receive jobs and coordinate with the nodes inside the network, without having to recur to a centralized control system.
- A job executioner able to receive different assets to perform the jobs (image/video manipulation, calculation, etc), taking advantage of the dynamic runtime available by the

predominant language in the browser, javascript.

- A server to work as the entry point for browser to download the code necessary to run browserCloud.js logic. This is the only point that is considered to be centralized in the network, due to the limitation of browsers being typically behind NAT and not having static IPs
- Structured peer-to-peer overlay network for browsers to communicate directly among themselves, without being necessary to take the data or the computation to a centralized system.

These components are fully described in section 3. After its development, a proposed evaluation is going to be executed, according to a set of assessment metrics, enabling us to compare the viability of browserCloud.js.

1.5 Structure and Roadmap

We start by presenting in Section 2, the state of the art for the technologies and areas of study relevant for he proposed work, which are: Cloud computing and Open Source Cloud Platforms (at 2.1), Volunteered resource sharing (at 2.2) and Resource sharing using the Web platform (at 2.3). In Section 3, we present thed architecture and respective software stack, moving to Implementation details in Section 4 and system evaluation present on Section 5.

1.6 Open Source Contributions

During the development of browserCloud.js, several Open Source contributions, MIT licensed, were made. Table **??** contains the list of these contributions and respective interest by the Open Source, P2P and Javascript communities based on number of downloads and project stars.

List is: https://gist.github.com/diasdavid/55f5bf71d9494caf6a08 . Will extract the numbers closer to deadline

Name	URL	Number of downloads/instalations	Number of Github stars	Number of npm stars
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e
a	b	c	d	e

Table 1.1: List of Open Source contributions

1.7 Publications, Presentations and References

We've seen a new tendency on the Javascript, Node.js, WebRTC and essencially Web Open Source communities to move to a model where contributions to the ecosystem are measured by their ability to be used by other projects, reviewed and studied from their internals and easy to use, therefore creating the need for simpler interaces, open source code and easy to install/run, inspite the tradicional reports with digests on the analysis done during the development and typically hard to reproduce results.

We've adopted this mindset since the beginning of the development of browserCloud.js, taking the project to the community and collecting feedback early and often, getting other developers excited to use the platform. In this process, we've achieved:

- Invited to the third OpoJS event, where I had the opportunity to give a 50 minutes talk on for more than 140 Web developers and P2P enthusiasts. The video of this talk was later publish, having close to 180 impressions (https://www.youtube.com/watch?v=fNQGGGE_zI)
- WebRTC Weekly Issue #60 mention, the number one WebRTC newsletter with more than 1000 subscribers.

- Number one article in EchoJS for 3 days in a row and top 5 for 7 days. (http://www.echojs.com/news/14009)
- browserCloud.js demo video - Over 200 impressions. (https://www.youtube.com/watch?v=kjwIjoENCRE)

2 Related Work

"Those who cannot remember the past are condemned to repeat it." – *George Santayana, Philosopher, essayist, poet and novelist (1863 - 1952)*

In this section, we address the background state of the art of the research topics, more relevant to our proposed work, namely: Cloud Computing, Volunteer Computing, P2P Networks and the Web Platform.

2.1 Cloud computing and Open Source Cloud Platforms

Cloud Computing is a term used to describe a large number of computers, connected through a network. The computing power from these machines is typically made available as virtual machines, without dependence to a particular real physical existence, enabling the possibility to scale up and down its resources on the fly, without affecting the end user.

Cloud Computing today is available as a set of Services, from Infrastructure(IaaS), Platform (PaaS), Software (SaaS), Network (NaaS), physical hardware (Metal as a Service) and more as described on Table **??**. However, the idea of having computing organized as a public utility just like the telephone or the electricity service is not new, it was envisioned around 1961, by Professor John McCarthy, who said in MIT's centennial celebration:

Acronym	Full Name
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
NaaS	Network as a Service
MaaS	Metal as a Service
MDBaaS	MongoDB as a Service
...	...

Table 2.1: Some of the different types of Cloud Computing services being offered

Advantage	Public Cloud	Conventional Data Center
Appearance of infinite computing resources on demand	Yes	No
Elimination of an up-front commitment by Cloud users	Yes	No
Ability to pay for use of computing resources on a short-term basis as needed	Yes	No
Economies of scale due to very large data centers	Yes	Usually not
Higher utilization by multiplexing of workloads from different organizations	Yes	Depends on company size
Simplify operation and increase utilization via resource virtualizations	Yes	No

Table 2.2: Comparing public clouds and private data centers.

"Computing may someday be organized as a public utility just as the telephone system is a public utility, Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system. Certain subscribers might offer service to other subscribers. The computer utility could become the basis of a new and important industry.", Professor John McCarthy.

Cloud computing presents several advantages comparing to the Conventional Data Center type of architecture(**?**), seen in Table **??**, similar to the vendor lock-in that lead to the adoption of open distributed systems in the 1990, moreover there are currently also security issues due to shared CPU and physical memory between different applications from different clients, which enables one of the clients to access data from the other if the application is not well confined.

2.1.1 Cloud interoperability

The lack of portability has already been identified as a major problem by growing companies, and is becoming one of the main factors when opting, or not, for a Cloud Provider, the industry realized this issue and started what is known as OpenStack[1].

OpenStack is an ubiquitous open source cloud computing platform for public and private clouds. It was founded by Rackspace Hosting and NASA. OpenStack has grown to be *de facto* standard of massively scalable open source cloud operating system. The main goal is go give the opportunity to any company to create their cloud stack and therefore, be compatible with

[1]http://www.openstack.org/ - seen on December 2013

other cloud providers since day one. All OpenStack software is licensed under the Apache 2.0 license, giving the possibility for anyone to involve the project and contribute.

Although OpenStack is free and open source, there is an underlying illusion that is the fact that you still have to use OpenStack in order to have portability, it is just a more generalized and free version of the 'lock-in syndrome'. We have currently other solutions available that give application developer an abstraction on top of different Cloud Providers, instead of changing the architecture of each Cloud, such as: IEEE Intercloud[2], pkgcloud[3] and Eucalyptus(**?**), described in the following two paragraphs.

2.1.1.1 IEEE Intercloud

pushes forward a new Cloud Computing design pattern, with the possibility to federate several clouds operated by enterprise or other providers, increasing the scalability and portability of applications. This federation is known as 'Intercloud' in which IEEE is creating technical standards (IEEE P2302) with interoperability in its core goals. Currently IEEE has already available an Testbed, the IEEE Intercloud Testbed, which provides a global lab for testing Intercloud interoperability features.

The envisioned Intercloud architecture categorizes its components into three main parts, see in Figure **??**:

- Intercloud Gateways: analogous to an Internet router that connects an Intranet to the Internet.
- Intercloud Exchanges: analogous to Internet exchanges and peering points (known as brokers in the US NIST Reference Architecture) where clouds can interoperate.
- Intercloud Roots: A set of core essential services such as: Naming Authority, Trust Authority, Messaging, Semantic Directory Services, and other "root" capabilities. This services work with an hierarchical structure and resembles the Internet backbone.

[2]http://cloudcomputing.ieee.org/intercloud - seen on December 2013
[3]https://github.com/nodejitsu/pkgcloud - seen on December 2013

2.1.1.2 pkgcloud

is an open source standard library that abstracts differences between several cloud providers, by offering a unified vocabulary for services like storage, compute, DNS, load balancers, so the application developer does not have to be concerned with creating different implementations for each cloud. Instead, just make the provision in the one that is most cost-effective. Currently, it only supports applications built using Node.js.

2.1.1.3 Eucalyptus

is a free and open source software to build Amazon Web Services Cloud like architectures for a private and/or hybrid Clouds. From the three solutions described, Eucalyptus is the one that is more deeply entangled with the concept of a normal Cloud, packing a: Client-side API, a Cloud Controller, S3 storage compliant modules, a cluster controller and a node controller, as seen in Figure **??**. Eucalyptus has all the components to build an entire cloud, however, since its compatible, specially, with Amazon Cloud, we can use Eucalyptus to migrate our services, or provision Amazon services, and work without having to deal with the application or the system itself.

This hybrid model provides a desired environment for a development, test and deploy stack, that can support Amazon Cloud with the elasticity necessary to sustain service during spikes. This way, a company that has its private cloud does not need to over provision in advance.

2.2 Volunteered resource sharing

Volunteered resource sharing networks enable the cooperation between individuals to solve higher degree computational problems, by sharing idle resources that otherwise would be wasted. These individuals may or may not have a direct interest with the problem that someone is trying to solve, however they share the resources for a common good.

The type of computations performed in this Application-level networks (ALN), are possible thanks to the definition of the problem in meta-heuristics, describing it with as laws of nature(**?**), such as: Evolutionary algorithms (EA); Simulated annealing (SA); Colony optimiza-

tion (ACO); Particle swarm optimization (PSO), Artificial Bee Colonies (ABC) and more. This process creates small individual sets of units of computation, known as 'bag of tasks', easy to distribute through several machines in and executed in parallel.

2.2.1 Hybrid and Community Clouds

A community cloud is a network of large scale, self-organized and essentially decentralized computing and storage resources. The main focus is on free economic and censorship wise, putting the user back in control of the information, giving them freedom to share content without censorship or a company interest. The term 'User Centric Cloud' appears on (**?**), where the resources are made available by individuals, but with a common API, similar to a centralized Cloud, where users that participate in the effort can also use others resources.

One major trend in Community Cloud computing is not only to share and trade computing resources, but also to build the actual physical network in which they are shared, this is known as Community Networks or "bottom-up networking". Community Networks such as guifi.net and Athens Wireless Metropolitan Network (AWMN) have together more than 22500 nodes providing localized free access to content, without the need to contract from an Internet provider.

CONFINE(**?**) is an European effort that has the goal to federate existing community networks, creating an experimental testbed for research on community owned local IP networks. From this project, resulted Community-Lab,[4] a federation between guifi.net, AWMN and Funk-Feuer (community network from Vienna and Graz, Austria), with the goal of carrying out experimentally-driven research on community-owned open networks.

2.2.2 Cycle and Storage Sharing, using Volunteer Resource Systems

When we talk about peer-to-peer applications, most people will remember volunteered storage sharing, as it most widely known for its ability to distribute content, thanks to the illegal distribution of copyrighted software and media. However if we take a look at the whole spectrum of volunteer resource systems, we will see that are two categories, one for content sharing and the second one for cycle sharing, the second is known today as Public Computing.

[4] http://community-lab.org/ - seen on December 2013

Storage and content sharing systems are the popular type from the two categories of peer-to-peer systems, specially because their ability to distribute content without legal control, which after their success, systems like Napster[5] were legally forced to shutdown. One of the key benefits of using a peer-to-peer storage sharing system is their ability to optimize the usage of each individual user limited bandwidth, enabling file partitioned transfers from multiple users, using the hash of each partition or chunk to prove its integrity. Each file availability grows organically with the interested in that file, because more copies will exist in the network. Other examples of this type of system are: KaZaA[6], BitTorrent[7] and Freenet(**?**).

The second category is that of systems that fit into the domain of Public Computing, where users share their idle computer cycles; this can be done by starting or resuming a computing process when the user is not performing any task that is relevant for him/her, or by establishing the tasks as low priority processes, so it does not affect the user experience. One way of doing this is using a screen saver, so the shift to an idle state is obvious to the machine. These systems are possible because we can divide bigger computational jobs into smaller tasks that can run independently and in parallel, again this is known as the "bag-of-tasks" model of distributed computing. Several systems using this currently are Folding@Home, Genome@Home(**?**) and SETI@Home(**?**)(**?**), all BOINC(**?**) based. However these systems work in a one way direction: volunteers to the network do not have the possibility to use the network for its own use; nuBOINC(**?**), enables contributers to take advantage of the network by adding extensions to the platform that enable every user to submit jobs, adding more flexibility towards the goal in which the shared computer cycles are used.

Another interesting research on this field is moving the logic necessary for processing some data, alongside the data, this is known as Gridlet(**?**)(**?**), a unit of workload. This approach enables a more dynamic use of volunteer resource systems with the possibility of: having different goals for the same Grid, optimize the resources available of one machine by gathering different type of tasks in one machine, reduce the cost to start using a Grid for distributed computation.

[5] http://napster.com - seen on December 2013
[6] http://www.kazaa.com/
[7] http://www.bittorrent.com/

2.2.3 Peer-to-Peer Networks and Architectures

Efficient resource discovery mechanisms are fundamental for a distributed system success, such as grid computing, cycle sharing or web application's infrastructures(**?**), although in the centralized model, by keeping data bounded inside a data center, we have a stable and scalable way for resource discovery, this does not happen in a P2P network, where peers churn rate can vary greatly, there is no way to start new machines on demand for high periods of activity, the machines present are heterogeneous and so is their Internet connectivity, creating an unstable and unreliable environment. To overcome these challenges, several researches have been made to optimize how data is organized across all the nodes, improving the performance, stability and the availability of resources. The following paragraphs will describe the current state of the art P2P organizations, typically categorized in P2P literature as Unstructured or Structured(**?**), illustrated in Figure **??**.

2.2.3.1 Unstructured

We call 'Unstructured' to a P2P system that doesn't require or define any constraint for the placement of data, these include Napster, Kazaa and Gnutella, famous for its file sharing capabilities, where nodes can share their local files directly, without storing the file in any specific Node. There is however a 'caveat' in the Unstructured networks, by not having an inherent way of indexing the data present in the network, performing a lookup results of the cost of asking several nodes the whereabouts of a specific file or chunk of the file, creating a huge performance impact with an increasing number of nodes.

In order to calibrate the performance, Unstructured P2P networks offer several degrees of decentralization, one example is the evolution from Gnutella 0.4(**?**) to Gnutella 0.6 (**?**)(**?**), which added the concept of super nodes, entities responsible for storing the lookup tables for the files in parts of the network they are responsible for, increasing the performance, but adding centralized, single points of failure.

Unstructured networks are classified(**?**) in two types: deterministic and non-deterministic, defining that in a deterministic system, we can calculate before hand the number of hops needed to perform a lookup, knowing the predefined bounds, this includes systems such as Napster and BitTorrent(**?**), in which the file transfers are decentralized, the object lookup re-

mains centralized, keeping the data for the lookup tables stored in one place, which can be gathered by one of two ways: (i) peers inform directly the index server the files they have; or (ii) the index server performs a crawling in the network, just like a common web search engine, this gives this network a complexity of O(1) to perform a search, however systems like Gnutella 0.6, which added the super node concept, remain non deterministic because it's required to execute a query flood across all the super nodes to perform the search.

2.2.3.2 Structured with Distributed Hash Tables

Structured P2P networks have an implicit way of allocating nodes for files and replicas storage, without the need of having any specie of centralized system for indexing, this is done by taking the properties of a cryptographic hash function (**?**)(**?**)(**?**), such as SHA-1(**?**), which applies a transformation to any set of data with a uniform distribution of possibilities, creating an index with O(log(n)) peers, where the hash of the file represents the key and gives a reference to the position of the file in the network.

DHT's such as Chord(**?**), Pastry(**?**) and Tapestry(**?**), use a similar strategy, mapping the nodes present in the network inside an hash ring, where each node becomes responsible for a segment of the hash ring, leveraging the responsibility to forward messages across the ring to its 'fingers'(nodes that it knows the whereabouts). Kademlia(**?**) organizes its nodes in a balanced binary tree, using XOR as a metric to perform the searches, while CAN(**?**) introduced and a several dimension indexing system, in which a new node joining the network, will split the space with another node that has the most to leverage.

Evaluating the DHT Structured P2P networks raises identifiable issues, that result as the trade-off of not having an centralized infrastructure, responsible for railing new nodes or storing the meta-data, these are: (i) generation of unique node-ids is not easy achievable, we need always to verify that the node-id generated does not exist, in order to avoid collisions; (ii) the routing table is partitioned across the nodes, increasing the lookup time as it scales.

Table **??**, showcases a comparison of the studied DHT algorithms.

2.2.3.3 Structured without Distributed Hash Tables

Mercury(**?**), a structured P2P network that uses a non DHT model, was designed to enable range queries over several attributes that data can be dimensioned on, which is desired on searches over keywords in several documents of text. Mercury design offers an explicit load balancing without the use of cryptographic hash functions, organizing the data in a circular way, named 'attribute hubs'.

2.2.3.4 Fault Tolerance, Load Balancing, Assurance and Trust

Volunteer resource sharing means that we no longer have our computational infrastructure confined in a well monitored place, introducing new challenges that we have to address (**?**) to maintain the system running with the minimum service quality. These issues can be: scalability, fault tolerance, persistence, availability and security(**?**) of the data and that the system doesn't get compromised. This part of the document serves to describe the techniques implemented in previous non centralized systems to address this issues.

2.2.3.5 Fault Tolerance, Persistence and Availability

are one of the key challenges in P2P community networks, due to it's churn uncertainty, making the system unable to assume the availability of Node storing a certain group of files. Previous P2P systems offer a Fault Tolerance and Persistence by creating file replicas, across several Nodes in the network, one example is PAST(**?**)(**?**), a system that uses PASTRY routing algorithm, to determine which nodes are responsible to store a certain file, creating several different hashes which corresponds to different Nodes, guaranteeing an even distribution of files across all the nodes in the network. DynamoDB(**?**), a database created by Amazon to provided an scalable NOSQL solution, uses a storage algorithm, inspired by the Chord routing algorithm, in which stores file replicas in the consequent Nodes, in order to guarantee easy lookup if one of the Nodes goes down.

The strategy presented by the authors of PAST to provide high availability, is an intelligent Node system, that use a probabilistic model, able to verify if there is an high request for a file, deciding to keep a copy and avoiding to overload the standard Node with every request that is made.

2.2.3.6 Load Balancing

in an optimal state, can be defined as having each node sharing roughly 1/N of the total load inside the network, if a Node has a significantly hight load compared with the optimal distribution, we call it a 'heavy' node. There has been some research to find a optimal way to balance the load inside a P2P network, namely:

- Power of Two Choices(**?**) - Uses multiple hash functions to calculate different locations for an object, opts to store it in the least loaded node, where the other Nodes store a pointer. This approach is very simple, however it adds a lot of overhead when inserting data, however there is a proposed alternative of not using the pointers, which has the trade-off of increasing the message overhead at search.
- Virtual Servers(**?**) - Presents the concept of virtualizing the Node entity to easy transfer it amongst the machines present in the P2P network. It uses two approaches, 'one-to-one', where nodes contact other Nodes inside the network with the expectation of being able to trade some of the load, shifting a virtual server, or an 'one-to-many/many-to-many' in which a directory of load per node is built, so that a node can make a query in order to find it's perfect match to distribute his load. Virtual Servers approach has the major issue of adding a extra amount of work to maintain the finger tables in each node.
- Thermal-Dissipation-based Approach(**?**) - Inspired by the heat expansion process, this algorithm shifts nodes position inside the hash ring windows of load responsibility, in a way that the load will implicitly flow from a node to it's close peers.
- Simple Address-Space and Item Balancing(**?**) - It is an iteration over the virtual servers, by assigning several virtual nodes to each physical node, where only one of which is active at a time and this is only changed if having a different nodeId distribution in the network brings a more load balanced hash ring

S. Rieche, H. Niedermayer, S. Götz and K. Wehrle from the University of Tübingen, made a study comparing this different approaches in a scenario using the CHORD routing algorithm, using a SHA-1 as the hashing function, with 4096 nodes and 100.000 to 1.000.000 documents and executing up to 25 runs per test, the results can be observed in the Figure **??**

2.2.3.7 Assurance and Trust

in a P2P network is an interesting challenge due to the lack of control over the machines that are willing to share with their resources, in order to achieve it, several strategies have been developed to maintain the integrity of the data using Cryptography, Reputation modeling schemes based on it's node previous record and also economic models, that resemble our own economy, but to share and trade computational resources.

Starting with the Cryptographic techniques, storage systems such as PAST give the option to the user to store encrypted content, disabling any other user, that does not have the encryption key, to have access to the content itself, this is a technique that comes from the Client-Server model, adapted to P2P environment, however, other cryptography technique benefits such as user authorization and identity, cannot be directly replicated into a P2P network without having a centralized authority to issue this validations, one of the alternatives is using distributed signature strategy, known as Threshold Cryptography (**?**), where an access is granted if validated if several peers (a threshold), validates it's access, one implementation of Threshold Cryptography can be see in a P2P social network(**?**) in order to guarantee privacy over the contents inside the network.

Trust in a P2P system, as mentioned, is fundamental to it's well behaved functioning, not only in terms of data privacy, but also in giving the deserved resources to the executions that mostly need them, avoiding misbehaved peer intentions that can be a result of an Attack to jeopardize the network, one example is the known Sybil attack(**?**). To achieve a fair trust sharing system, several metrics for a reputation mechanism have been developed (**?**), these can be seen in Table **??**.

Incentives for sharing resources(**?**) can in the form of money rewards, greater speed access(used in Napster and some bittorrent networks) or it can be converted to a interchangeable rate to trade for more access to resources, giving the birth of economic models(**?**)(**?**), that model the traded resources as a currency in which a peer has to trade in order to use the network.

2.3 Resource sharing using the Web platform

One of the main focuses with the proposed work, is to take advantage of the more recent developments of the Web platform to make the intended design viable (presented in section 4), the system depends on very lower level components such as:

- High dynamic runtime for ongoing updates to the platform and specific assets for job execution
- Close-to-native performance for highly CPU-bound jobs
- Peer-to-peer interconnectivity
- Scalable storage and fast indexing

Therefore, we present in this section the relevant components present or undergoing a development process for the Web platform, such as: Javascript, Emscripten, IndexedDB, WebRTC and HTTP2.0. These will coexist as key enablers for the necessary features to such a distributed shared resource system:

2.3.1 Web Platform

Since the introduction of AJAX(**?**), the web has evolved into a new paradigm where it left being a place of static pages, known as Web 1.0. Nowadays, we can have rich web applications with degrees of interaction and levels of performance close to a native application. The programming languages that power the Web Platform, in special HTML, CSS and JavaScript(**?**), have been subject to several changes, enabling 'realtime' data transfers and fluid navigations through content. Javascript, an interpreted language with an high dynamic runtime, has proven to be the right candidate for a modular Web Platform, enabling applications to evolve continuously over time, by simply changing the pieces that were updated.

Emscripten(**?**), a LLVM(Low Level Virtual Machine) to JavaScript compiler, enabled native performance on Web apps by compiling any language that can be converted to LLVM bytecode, for example C/C++, into JavaScript. This tool enabled native game speed on the

browser, where two of the major examples are the project Codename: "BananaBread"[8] and "Epic Citadel"[9], in which Mozilla used Ecmascripten to port the entire Unreal Engine 3 to JavaScript. In Figure **??**, we can see a comparison of the performance of several algorithms, running on Dalvik, Android Java runtime, asm.js, the subset of Javascript that the code in C/C++ is transformed into when compiled with Emscripten and Native, the same C/C++ but running on a native environment. The results are very interesting, specially in the first test, where asm.js outperforms native. The explanation for this is due to the fact that BinaryTrees use a significant amount of 'malloc' invocations, which is an expensive system call, where in asm.js, the code uses typed arrays, using 'machine memory', which is flat allocated in the beginning of the execution for the entire run.

WebRTC(**?**), a technology being developed by Google, Mozilla and Opera, with the goal of enabling Real-Time Communications in the browser via a JavaScript API. WebRTC brings to the browser the possibility of peer-to-peer interoperability. Peers perform their handshake through a 'Signaling Server'. The signaling server will exchange the 'ICE(Interactive Connectivity Establishment) candidates' of each peer as this serves as an invite so a data-channel can be opened, a visualization of this process can be seen in Figure **??**. Since most of the browsers sit behind NAT, there is another server, named 'Turn'(Relay), which tells to each browser their public IP in the network. WebRTC, although being built with the goal of real-time voice and video communications, has also been shown as a viable technology do distribute content, as seen in PeerCDN and SwarmCDN(**?**).

'level.js' offers an efficient way to store larger amounts of data in the browser machine persistent storage, its implementation works as an abstraction on top of the leveldown API on top of IndexedDB(**?**), which in turn is implemented on top of the LevelDB(**?**), an open source on-disk key-value store inspired by Google BigTable. IndexedDB is an API for client-side storage of significant amounts of structured data and for high performance searches on this data using indexes. Since 'level.js' runs on the browser, we have an efficient way to storage data and quickly retrieve it.

One of the latest improvements being built for the Web Platform is the new HTTP spec, **HTTP2.0**(**?**), this next standard after HTTP1.1 which aims to improve performance towards a

[8]Mozilla, BananaBread, URL: https://developer.mozilla.org/en/demos/detail/bananabread, seen in December 2013

[9]Mozilla, Epic Citadel, URL: http://www.unrealengine.com/html5/, seen in December 2013

more realtime oriented web, while being retrocompatible at the same time. Several advancements in this new spec are:

- Parallel requests - HTTP1.1 was limited by a max of 6 parallel requests per origin and taking into account that the mean number of assets is around one hundred when loading an webapp, it means that transfers get queued and slowed down. In order to overcome this, we could distribute the assets through several origins in order to increase the throughput. However this optimization backfired when in mobile, since there was a lot of signaling traffic in TCP layer, starving the user connection. HTTP2.0 no longer has this constraint.
- Diff updates - One of the web developer favorites has been concatenating their javascript files so the response payload decreases, however, in modern webapps, most of the time, we do not want the user to download the entire webapp again, but only some lines of code referring to the latest update. With diff updates, the browser will only receive what has been changed.
- Prioritization and flow control - Different webapp assets have different weights in terms of user experience, with HTTP2.0, the developer can set priorities so the assets arrive by order. A simple flow control example can be seen on Figure **??**, where the headers of the file gain priority as soon as they are ready, and get transfered immediately.
- Binary framing - In HTTP2.0, binary framing is introduced with the goal of creating more performant HTTP parsers and encapsulating different frames as seen on Figure **??**, so they can be send in an independent way.
- HTTP headers compression - HTTP2.0 introduces an optimization with headers compression(**?**) that can go to a minimum of 8 bytes in identical requests, against the 800 bytes in HTTP1.1. This is possible because of the state of the connection is maintained, so if a identical requests is made, changing just one of the resources (for example path:/user/a to path:/user/b), the client only has to send that change in the request.
- Retrocompatibility - HTTP2.0 respects the common headers defined by HTTP1.1, it doesn't include any change in the semantics.

2.3.2 Previous attempts on cycle sharing through web platform

The first research of browser-based distributed cycle sharing was performed by Juan-J. Merelo, et. al., which introduced a Distributed Computation on Ruby on Rails framework(**?**). The system used a client-server architecture in which clients, using a browser would connect to a endpoint, where they would download the jobs to be executed and sent back the results. In order to increase the performance of this system, a new system(**?**) of browser-based distributed cycle sharing was creating using Node.js as a backend for very intensive Input/Output operations(**?**), with the goal of increased efficiency, this new system uses normal webpages (blogs, news sites, social networks) to host the client code that will connect with the backend in order to retrieve and execute the jobs, while the user is using the webpage, this concept is known as parasitic computing(**?**), where the user gets to contribute with his resources without having to know exactly how, however since it is Javascript code running on the client, any user has access to what is being processed and evaluate if it presents any risk to the machine.

2.4 Analysis and discussion

The related work presented was researched with the goal of deepen the knowledge about current strategies for resource sharing, as we intend to present a new one using the Web Platform. The concept of Gridlet, akin to those seen as well in state of the art databases such as Joyent's Manta,[10] which bring the computation to/with the data, reducing the possibility of a network bottleneck and increases the flexibility to use the platform for new type of jobs, will very important. To enable this new Cloud platform on using browsers, it is important to understand how to elastically scale storage and job execution, as in (**?**), but in peer-to-peer networks: therefore a study of the current algorithms and its capabilities was needed. Lastly, browsing the web is almost as old as the Internet itself, however on the last few years, we are seeing the Web Platform rapidly changing, and enabling new possibilities with peer-to-peer technology e.g. WebRTC; otherwise, it would not be possible to create browserCloud.js.

[10] http://www.joyent.com/products/manta - seen in December 2013

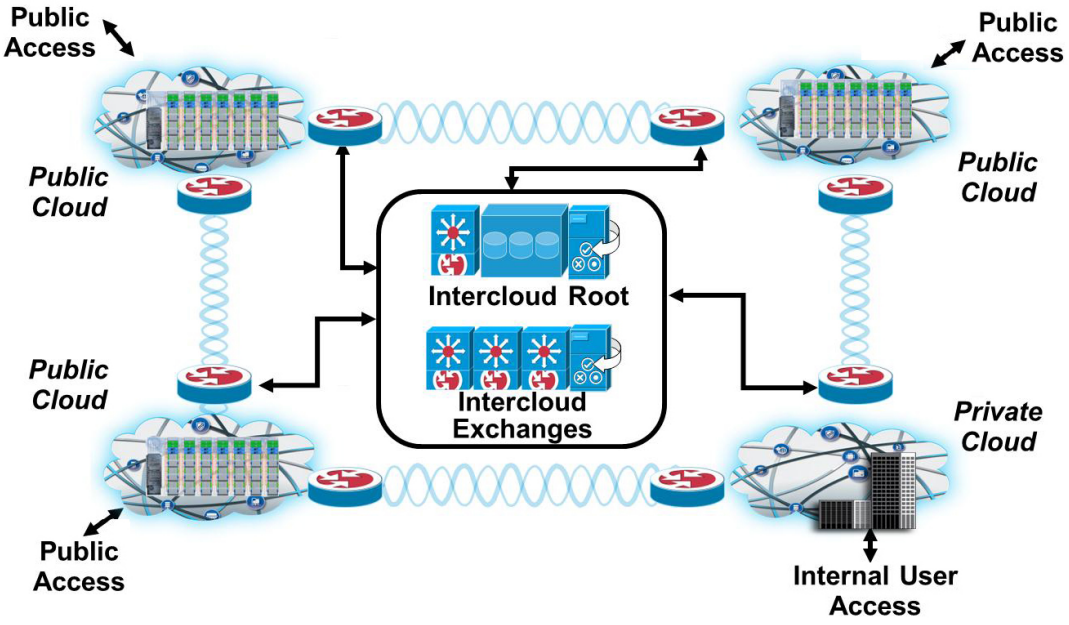


Figure 2.1: IEEE Intercloud Testbed Architecture

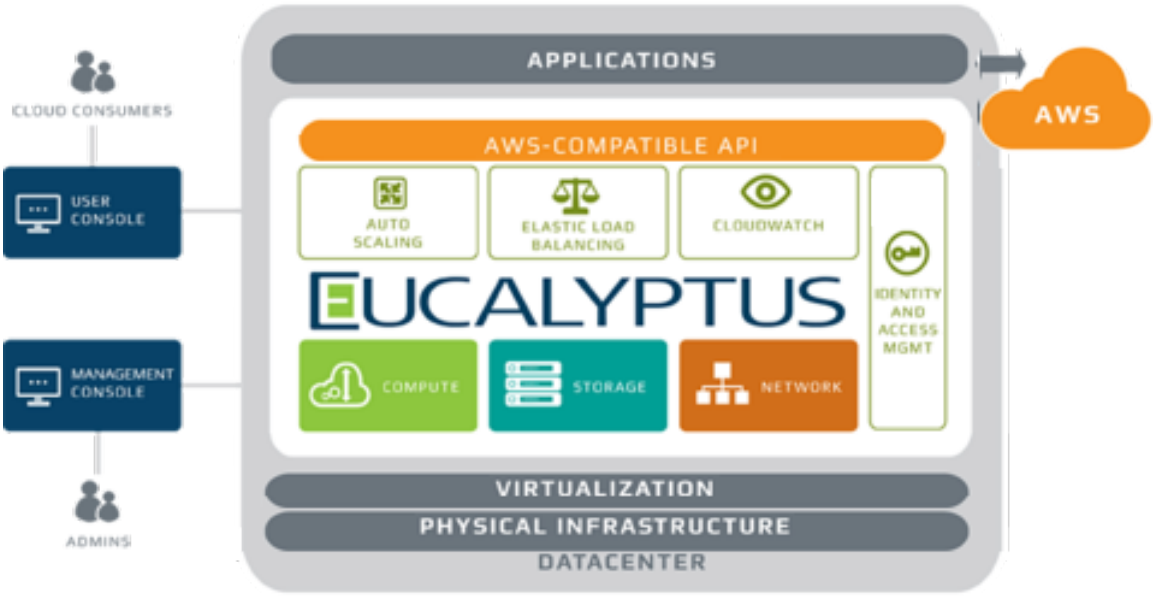


Figure 2.2: Eucalyptus Architecture

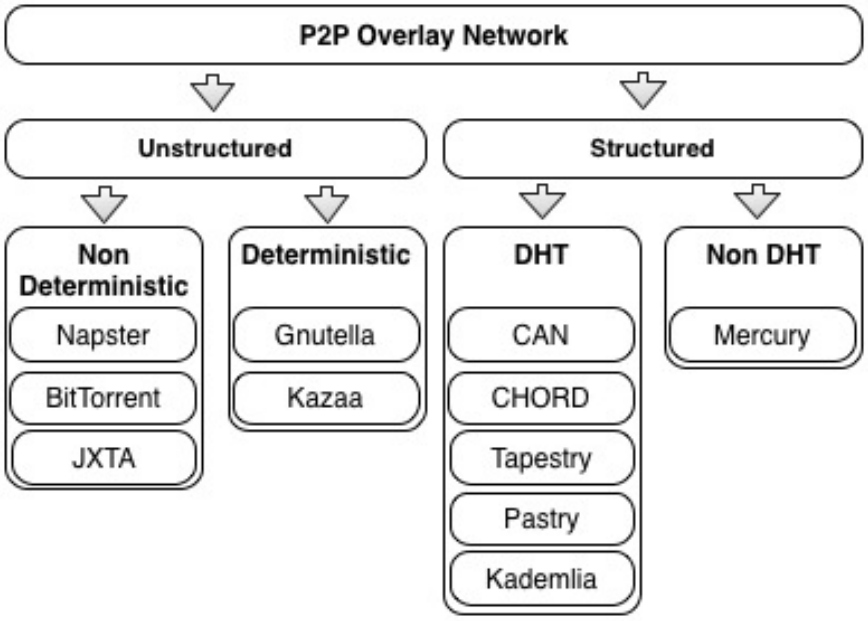


Figure 2.3: Different types of P2P Overlay networks organizations

P2P Algorithym	**Overlay Structure**	**Lookup Protocol**	**Networking parameter**	**Routing table size**	**Routing complexity**	**Join/leave overhead**
Chord	1 dimension, hash ring	Matching key and NodeID	n=number of nodes in the network	O(log(n))	O(log(n))	$O(\log(n)^2)$
Pastry	Plaxton style mesh structure	Matching key and prefix in NodeID	n= number of nodes in the network, b=base of identifier	$O(\log_b(n)$	O(b $\log_b(n)$ + b)	g
CAN	d-dimensional ID Space	Key value pair map to a point P in D-dimensional space	n=number of nodes in the network, d=number of dimensions	0(2d)	$O(d\ n^{1/2})$	O(2d)
Tapestry	Plaxton style mesh structure	Matching suffix in NodeID	n=number of nodes in the network, b=base of the identifier	$O(\log_b(n))$	O(b $\log_b$ (n)+b)	O(log(n))
Kademlia	Binary tree	XOR metric	n=number of nodes, m=number of bits(prefix)	O(log(n))	$O(\log_2(n))$	not stable

Table 2.3: Summary of complexity of structured P2P systems

Reputation Systems		
Information Gathering	Scoring and Ranking	Response
Identity Scheme Info. Sources Info. Aggregation Stranger Policy	Good vs. Bad Behavior Quantity vs. Quality Time-dependence Selection Threshold Peer Selection	Incentives Punishment

Table 2.4: Reputation system components and metric

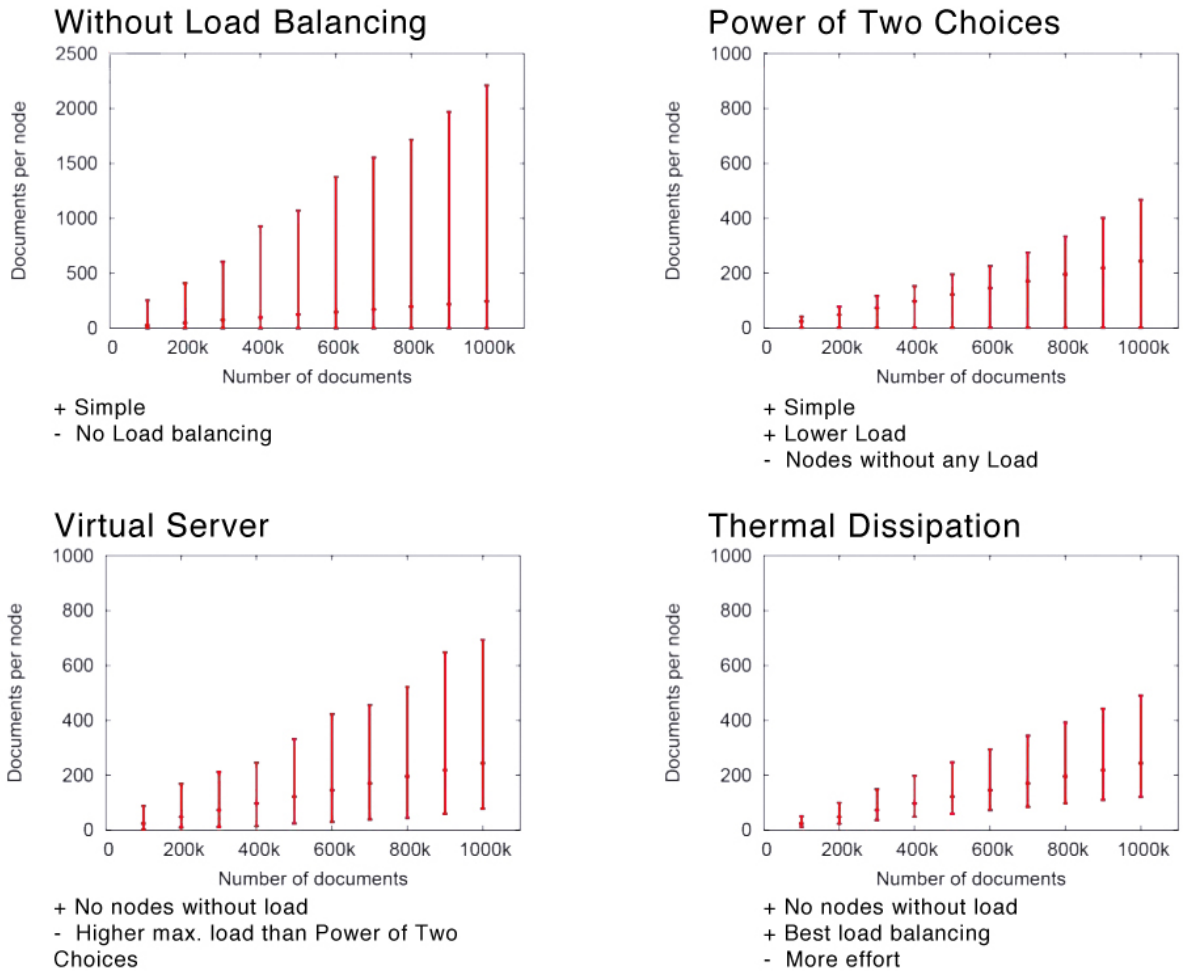


Figure 2.4: Load balancing approaches comparison

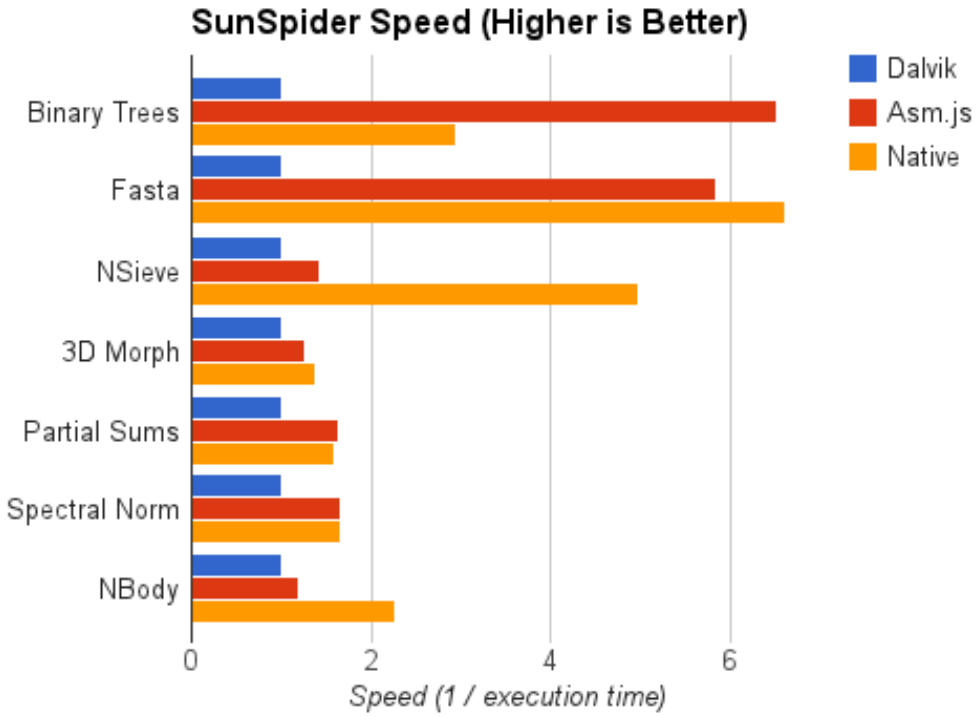


Figure 2.5: Dalvik vs. ASM.js vs. Native performance

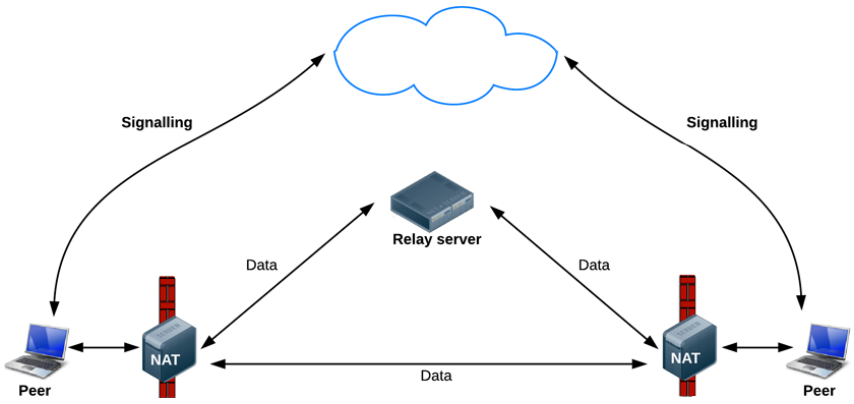


Figure 2.6: Example of a WebRTC session initiation

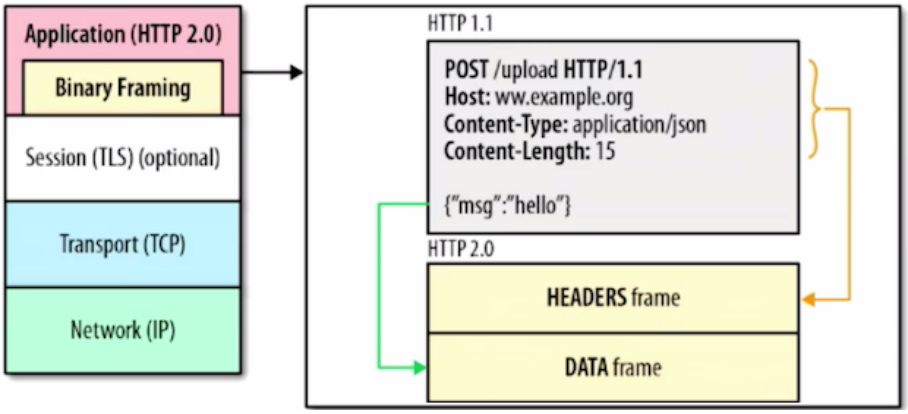


Figure 2.7: HTTP2.0 Binary framing

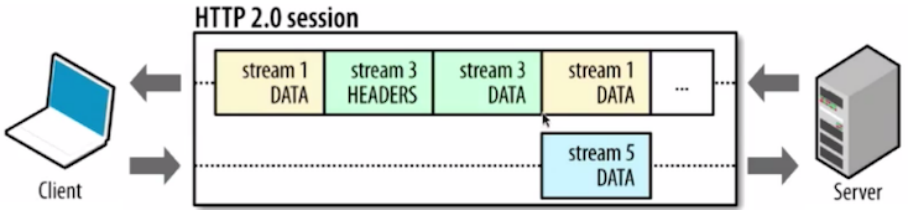


Figure 2.8: Example of an HTTP2.0 dataflow

3 Architecture

"Some fantastic quote" – *From some fantastic person*

browserCloud.js proposes a mechanism to find, gather and utilize idle resources present in a P2P overlay network, in which its participants will be joining and connecting to each other through a rendezvous point, as represented in Figure **??**. For a given peer, all that the peer needs to know is that once part of this network, it can submit a job which will be partitioned and distributed across a number of peers available, being responsible for later aggregating the results and delivering them to the user which summoned that job. The user doesn't need to understand how the network is organizer or which peers it is directly connected too, that complexity is abstracted by browserCloud.js.

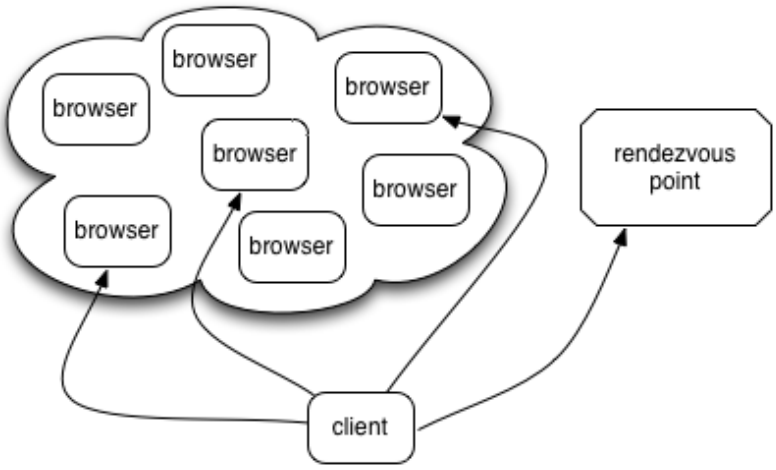


Figure 3.1: browserCloud.js Overview

A pratical use case for browserCloud.js is high CPU bound jobs and capable to run in parallel, e.g: image processing, video compressing, data manipulation, map and reduce functions, etc. These parallel tasks are divided by the peers available in the network, leveraging the parallelism to obtain a speed up.

This chapter describes the architecture designed for the browserCloud.js system. browserCloud.js was designed with with the Unix philosophy, that is, subtracting the unnecessary from a subsystem until it is constructed to perform one thing and one thing well, building more cohese abstractions through composition.

browserCloud.js was architectured to meet the following requirements:

- **Membership management** - The system has to enable peers to join and leave a current network of browserCloud.js peers or a subset of it. A peer should only have the knowledge of a small of other peers in the network and be available to rail in any other peer that wants to be part of the P2P network.
- **Message routing** - Peers must have a way to communicate with every other peer in the network without the necessity of contacting a centralized service to do so. Messages should be routed between peers, having each peer knowing a subset of the network, guaranteeing in full coverage in this manner.
- **Job scheduling and results aggregation** - The discovery of computational resources must be performed using a distributed fashing, peers interact between each other to send tasks and retrieve the results for the peer executing the job.
- **Support dynamic runtime** - Provide flexibility for jobs being executed. This is delivered thanks to the dynamic runtime offered by by peers in browserCloud.js due to the fact that they are standard compliant web browsers and Javascript is the language used.
- **Reduced entrance cost to enable greater adoption** - Designed simple APIs, abstracting the complexity in favor of greater extendability.

The overview of the network architecture can be seen in Figure **??**.

3.1 Distributed Architecture

3.1.1 Entities

There are two different kind of actors in the system:

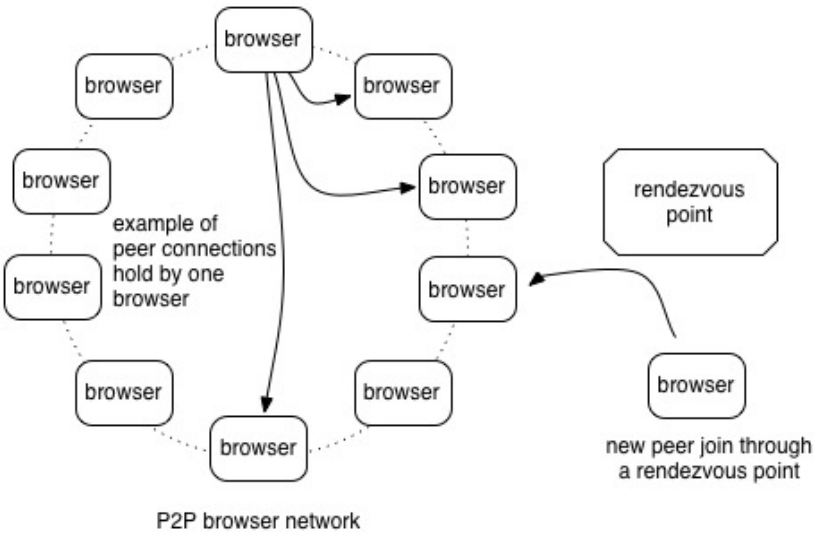


Figure 3.2: browserCloud.js Network Architecture Overview

- browser - The points on our network that will be able to issue jobs, execute tasks and route messages.
- rendezvous point - The only centralized component in this architecture, its purpose is for the clients to have a way to connect to the overlay network.

3.1.2 Interaction Protocols

In a browserCloud.js infrastructure, we have three main interaction patterns, the first being when a peer joins or leaves the network, which also we can call membership management, something that in traditionally P2P networks would simply mean an exchange of a pair IP:Port, but in a P2P browser network, a RTCPeerConnection has to be established and kept alive, meaning that an handshaking protocol must be performed. The second pattern is message routing between peers, this has been designed with inspiration on the Chord routing algorithm, studied on the related work. The third interaction demonstrates how to levarage the computer cycles available in the network to process CPU bound jobs.

3.1.2.1 Peer joins and leaves

A peer join compromisses of the following steps:

- **1 - Registration** - When a peer is ready to join the network, it performs the registration action to the custom browserCloud.js signalling server, the server replies with a confirmation and a unique ID for this peer to occupy in the network. This enables the signalling server, which holds the meta data of the current state in the network, to pick the ID in the ID interval that might be less occupied. We can observe this interaction in Figure **??**.
- **2 - New peer available** - As peers join the network, other peers present need to be notified to establish or update their connections to the new best candidates, so that the routing of messages (explained in the next section), remains efficient. For each peer join, a notification of a finger update can be sent to 1 or more peers present, as seen in Figure **??**.
- **3 - Connection establishment between two peers** - In order to establish a connection between two peers, once there is an interest for these to connect, for e.g, in the case of a finger update event. There are two substeps, the first being the SDP offer creation through a technique called "hole punching", where a browser uses one of the WebRTC API to traverse through NAT to obtain its public IP, which is crucial information when two browsers need to establish a direction connection, Figure **??**. The second step is the exchange of these SDP offers between browsers and that has to be performed by a centralized service, in browserCloud.js we developed a custom signalling server that handles that part, as seen in Figure **??**

A peer leave is a simplier and organic process, once a peer leaves the network, the RTCPeerConnections objects are destroied, notifying automatically the peers that have to update their finger tables that they should request the signalling server to update the metadata of the state of the network and therefore, issuing new finger-update messages.

The meta state of the network is always hold in memory by the signalling server, there is no need to keep this state persistence because it can be easily reconstructed, in the event of the signalling server failing, a new instance can be spawn and the peers simply have to register again, but this time with their current IDs.

3.1.2.2 Message routing

For message routing, we've designed an adaptation of the Chord routing algorithm, a P2P Structured Overlay network, a DHT studied in the related work, with the goal of keeping an

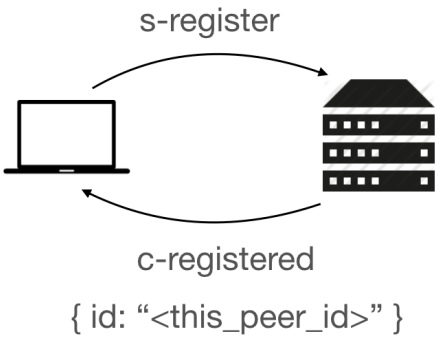


Figure 3.3: Registration of a peer, signaling itself as available to be part of the P2P network

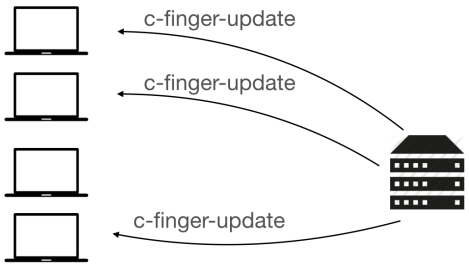


Figure 3.4: A peer is notified to update his finger table

efficiency routing and resource lookup with the increase of the number of peers in the network.

The ID namespace available in our DHT consists of 48 bit IDs (Figure **??**),this decision was made due to the fact that Javascript only supports 53 bit numbers, to support a greater variaty of IDs, we would have to resort to a big integer third party library, adding unnecessary consuption of computing resources. However, for demonstration purposes, we will explain using a 3 bit ID namespace.

In Figure **??**, we a DHT composed of 4 different peers, with IDs 0, 1, 3 and 6. Each one of these peers will be responsible for a segment of the DHT, in another words what this means is that every message that is destined to their segment, will be delivered to respective peer responsabile. A peer is responsible for a segment of IDs greater than the peer that is its predecessor and lesser or equal than its own ID, represented in Figure **??**. When a peer enters in the network, its ID is generated through a crop of a SHA-1 hash from a random generated number, creating a natural uniform distribution.

In order for messages to find its righteous destination, each peer has to know at minimum the peer that is next to it on the DHT, also called ”successor” (Figure **??**). Messages will be

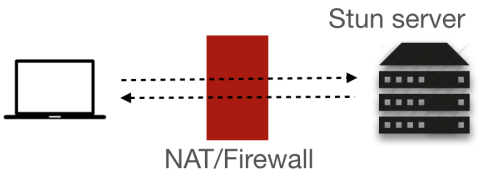


Figure 3.5: Hole punching through NAT to obtain a public IP and create a SDP offer

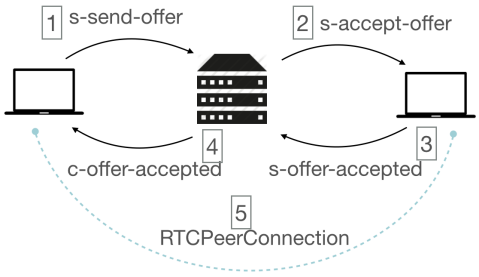


Figure 3.6: Establishment of a RTCPeerConnection through the custom Signalling Server

forward until they reach the peer which compromisses the responsability of being responsible for that message ID.

However, as specified earlier in the document, we want to achieve a good and stable efficiency when it comes to routing messages inside the DHT as the network grows. To achieve that, we introduce fingers to our peers. A finger is a direct connection to another peer in the network(Figure **??**), that was picked following a specific distribution, each peer will have 1 to N fingers, where N is the number of bits of the IDs (for this example, N = 3). A finger is always the peer responsible for the "start" value of the interval(see Figure **??** for reference and formula) and a message will be routed to that finger if it falls inside the interval.

The number of fingers and the fingers we use for a given instance of browserCloud.js are configurable, the reason behind this design decision was that RTCPeerConnections have a significant memory cost, so we have to be considerate in the number of data channels we keep open. In order to give greater flexibility to the developer, we've let the option of picking how many rows of the finger table will be filled by the developer creating a browserCloud.js application, this is also perfect since WebRTC is still a working draft and there might be good developments in resource consumption.

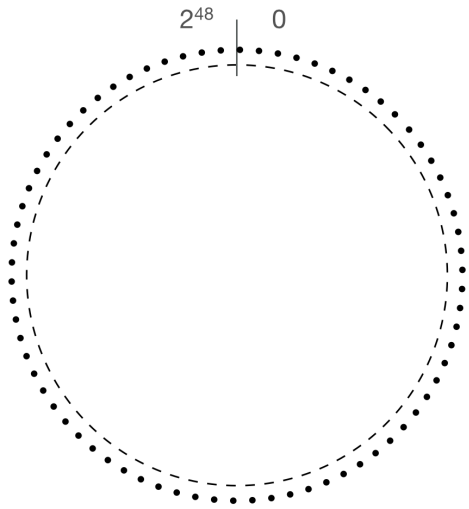


Figure 3.7: How the ID namespace is visualized in the DHT

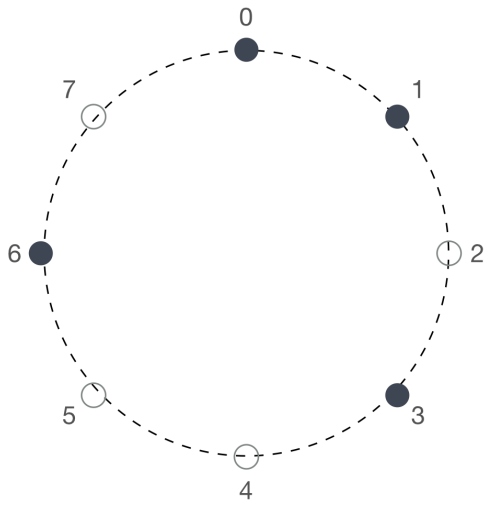


Figure 3.8: Example of a DHT with 4 peers for case study

3.1.2.3 Filling the finger table for optimal minimum number of hops

********** ******** EXPLAIN HERE WHAT CAN BE THE BEST STRATEGY FOR FINGER PICKING ****** **********

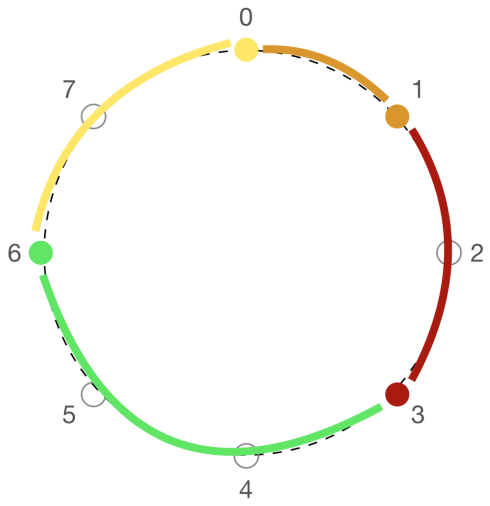


Figure 3.9: Responsability interval for each Peer

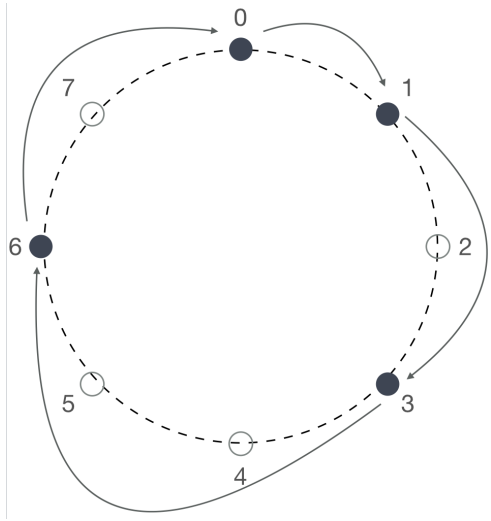


Figure 3.10: Minimum number of connections for messages to be routed properly

3.2 Resource Management

Leveraging the browser's dynamic runtime was a feature we pursue from the beginning of the design for browserCloud.js. A job is divided into individual tasks that are a composition of the function to be executed plus the data which should serve as input for that task, creating a transportable gridlet that can be migrated between browsers and executed by its final destination. A job execution is performed using the algorithm as follows:

- 1 - Select in how many units we want to divide a job.

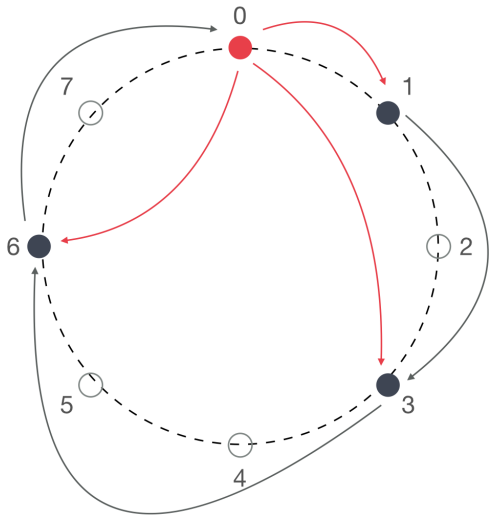
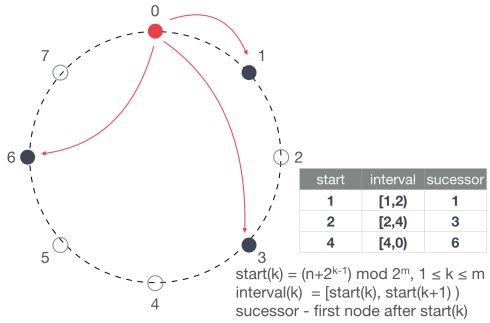


Figure 3.11: Example of peer with ID = 0 fingers



start	interval	sucessor
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

Figure 3.12: Peer with ID=0 finger table

- 2 - Select how many browsers we want to distribute the job to.
- 3 - Query the network for browser available (e.g. that are not performing other jobs at the moment).
- 4 - Compose the several units (gridlets) with with task plus data partition.
- 5 - Send this gridlets to the network to be routed to the browser that is going to execute them.
- 6 - browser compute the results and send them back to the job issuer.
- 7 - the browser that submited the job gathers all the tasks results and constructs the job result.

3.3 Architecture of the Software stack

When it comes to software, we divided our browser application appliance into three separate and fundamental components, namely: Communication layer, Service router and Job scheduler, leaving also the opportunity for these to be extended. We can observe a overview of this architecture in Figure **??**.

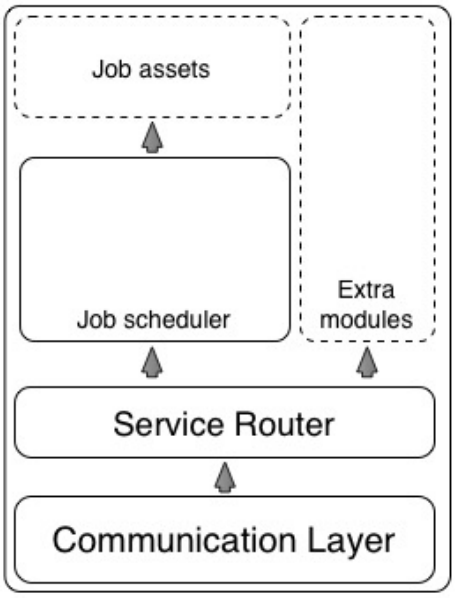


Figure 3.13: Software layers at the peer level

3.3.1 Communication layer

The communication layer is responsible for routing messages between peers and establish a connection with the rendezvous point to perform a peer join/leave. This means that the communication layer:

- Holds the connections with other peers.
- Performs the necessarity logic for efficient routing.
- Keeps the peer connected to the network by updating its routing table as necessary

3.3.2 Service router

The Service router establishes a protocol for modules like the job scheduler to interact with the network of peers, it uses an event driven model, where modules can register listeners to events that happen on the network (such as a specific reception of a message) and react to it. It also offers the necessary API calls for the modules to send messages to the network.

Service router offers extensibility to browserCloud.js, similar to Job scheduler, other modules can be implemented to interact with the already established P2P network.

3.3.3 Job scheduler

The Job scheduler benefits the API of the Service router to implement its logic, this means that although a job scheduler was implemented to fit our design purposes, it could easily be replaced by another job scheduler with different offers and guarantees.

A job consists in the partition of tasks which are enriched with data and sent to other peers to be executed, this tasks, which can be represented as functions (job assets), can be defined in runtime, therefore giving a greater flexibility to the developer that is using this system to run the distributed job they want. We can describe the work performed to schedule a job, by the following algorithm:

- 1. A user submits a job
- 2. The job is divided in smaller computing units, called tasks, each task compromisses of a segment of the data that is going to be processed and the transformation which is going to be applied, that is, a function.
- 3. These task and data are created
- 4. The peer will request the network for other peers availability, the user has the capability to specify how many peers should be used to process this job. This option is given since different jobs might benefit of more or less partition, depending on the data set.
- 5. The peer who submited the job (the peer that is controlled by the user submitting the job) will receive the individual results for each task as they are ready and transmited. Once all of the results are received, they are aggregated and delivered to the user.

3.4 API design

For the user of browserCloud.js, a simple API was created to perform: peer join, message listening and job scheduling as demonstrated by the following code (which should be interpreted as pseudo-code since the API might change with the release of new versions):

Peer join

```
// browserCloud.js browser module name is called webrtc-explorer.

var Explorer = require('webrtc-explorer');

var config = {
    signalingURL: '<signalling server URL>'
};

var peer = new Explorer(config);

peer.events.on('registered', function(data) {
    console.log('registered with Id:', data.peerId);
});

peer.events.on('ready', function() {
    console.log('ready to send messages');
});

peer.register();
```

Listen for messages

```
// The only action that has to be performed is listen for the message event
peer.events.on('message', function(envelope) {
    console.log(envelope);
});
```

Execute a job

```
var browserProcess = require('webrtc-explorer-browser-process');

var config = {
    signalingURL: 'http://localhost:9000'
};

// Make this browser available to execute tasks and also prepared to issue jobs to the network
```

```
browserProcess.participate(config);

var start = function() {
    var data = [0,1,2,3,4,5,6,7,8,9,10];  // simple data input
    var task = function(a) {return a+1;}; // e.g of a task (
    var nPeers = 2; // number of peers we are requesting from the network to execute our job

    browserProcess.execute(data, task, nPeers, function done(result){
        console.log('Received the final result: ', result);
    });
};
```

3.5 Testing framework

TODO***** requirements: 1. spawn browsers 2. orchestrate them 3. collect their events 4. create a timeline with the events 5. assess if it happen as expected

3.6 Summary

In this section, we covered the network topology of browserCloud.js, which and how interactions are performed and how a developer can use this system.

4 Implementation

"Tiny modules built on other tiny modules to make tiny powerful high level abstractions". – *James Halliday (substack), founder of browserling, prolific Node.js developer*

During the process of developing browserCloud.js, several attempts were created following the Agile methodology, rapidly creating working prototypes and iterating over them. This led to the creation of several open source modules, MIT licensed, having been downloaded in the order of dozens of thousand times until the creation of this document.

Every code artifact was developed following the Unix philosophy, every module attempts to do at most one thing and one thing well, creating small, maintainable and powerful abstractions.

In this section, we describe the implementation details of the final code artifacts that compose the browserCloud.js and the collaterals designed and created that although not projected in the beginning, were needed in order to collect the data we were looking to study.

4.1 Browser module

The browser module is the agent that sits inside our browser nodes, implementing all the communication protocols designed for the browserCloud.js platform and exposing a developer API to send and receive messages.

Essentially it is broken down into 4 components

- channel manager - a code artefact responsible to levarage the websockets connection with the signalling server and abstracts the necessary work to open new RTCPeerConnections with other peers.
- finger table manager - where the information about a specific peer finger table lives.

- router - the routing logic to deliver the messages on the most efficient way. It uses finger table manager to understand what is the most efficient way.
- interface - developer exposed interface.

The browser module exposes a factory method, meanthing that a developer can instantiate several instances inside a browser.

There are two technologies used in this module, other than the raw Javascript APIs that the browser offers, these are:

- browserify - enables the development of browser Javascript code in a modular fashion using the CommonJS standard, such as Node.js does, this means we can require('modules') and create concat and minify our Browser module, so that it can be loaded inside a webpage through a normal ¡script¿ tag.
- socket.io - socket.io is the most reliable and famous open source implementation of WebSocket API for the browser..

4.2 Signalling server

The signalling server offers two Web APIs, one being a WebSockets API and the other a RESTful API. The design decision behind these two APIs was mainly because since the network evolves with time, we needed a way to be able to push, on demand, new information to browsers, for example, when a new peer needs to be railed in, or when the Signalling Server acts as a rendezvous point for SDP data exchange between browsers so they can establish a RTCPeerConnection.

The second API, RESTful, is used to instruct the server or to collect analytics data from it remotely.

4.3 Key learnings from the earlier iterations - webrtc-ring

During one of the earlier iterations, we've developed a prototype of webrtc-explorer, called webrtc-ring, which although very similiar in routing strategy, each peer only knew about its

successor, in another words, each peer only had access to one finger, this originated a system with the following properties:

- overlay structure - 1 dimension Hash Ring
- lookup protocol - Matching key and NodeID
- network parameters - Number of Nodes in the network
- routing table size - 1
- routing complexity - O(log(N))
- join/leave overhead - 2

During the development, we performed tests to evaluate the capacity of the system to distribute work, later discussed on the Evaluation section. We learned that due to the single thread nature of Javascript, running message routing inside the same process that would be use to perform CPU bound tasks could be highly disavantageous for browserCloud.js performance. To overcome this, we introduced Web Workers to the system, independent threads inside the browser to separate communication from CPU bound tasks.

4.4 Implementation collaterals

During the development of the browserCloud.js system, several tools were built in order to properly test components of the system, or to perform performance analytics. These needed to be designed from the ground up due to browserCloud.js peculiar nature.

4.4.1 Testing browserCloud.js

There are a panoply of excellent browser testing frameworks and services available today, however their focus is on testing browser implementations (CSS, HTML and JavaScript) and user interactions of the apps their are testing (clicks, mouse movements, what the user sees).

When it comes to testing to test a decentralized browser app or library, the focus stops being how a browser implements a specific behaviour, but how the decentralized network handles

node joins and leaves and if nodes are effectively communicating between each other. In this scenario, we have several events that the server never sees or that the server never instructs the clients to do, so we need to create a new way to coordinate the browser joins and leaves and also how they interact between each other remotely and this is were piri-piri comes into play.

The specific set of problems piri-piri tries to solve:

- browser times X, where 1¡=X¡=virtually unlimited - Most browser testing frameworks only let you launch a couple of browsers at a time, piri-piri aims to launch several browsers and/or tabs to load a webpage, in a local or distributed fashion.
- instruct browsers on demand - Since there is a ton of stuff happening on browser decentralized apps, we can't just write a script to test and listen to events that happens in a single browser, there are triggers coming from all of them.
- gather information and evaluate the state as a whole - collect the events and data generated by each browser and assess if the order was correct with pseudo external consistency

4.4.2 Visualize the network state

Using D3JS, a API library that works as a thin veneer on top of SVG, we've developed an application that grabs the state of the browserCloud.js network and shows a live graphical representation, as seen on Figure **??**, where each node is represented by a dot and its ID and the arcs being the connections established between the nodes in the network.

4.4.3 Simulate a browserCloud.js network

In addition to the visualizer application, one simulator application was developed, were not only a graphical representation is generated, but also, it gives the developer a way to create a new virtual network, without any real peers. We've the option to pick the number of peers we want present and how many and which fingers will be used, so we can analyse different distribution paths and optimize for number of hops between any two peers in the network.

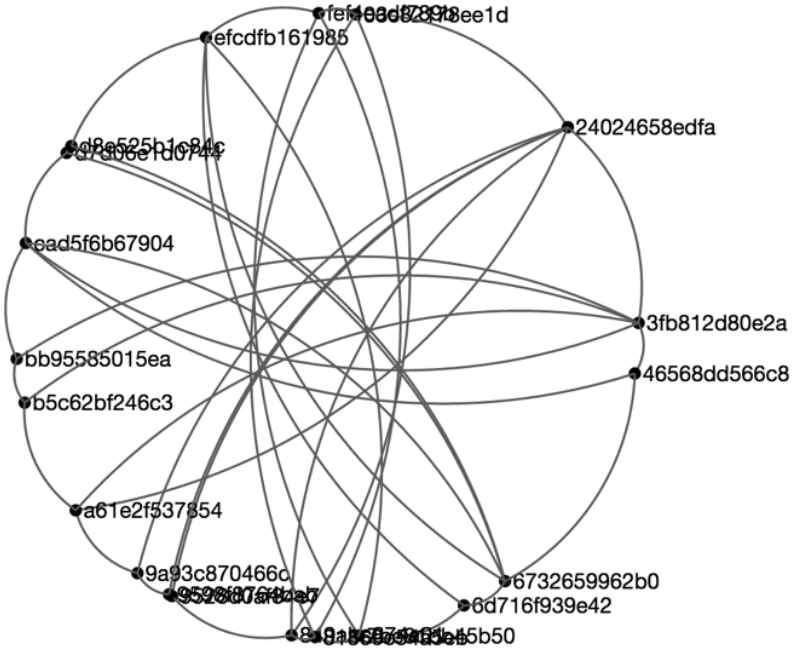


Figure 4.1: Visualization of a browserCloud.js network

4.4.4 Ray Tracing module

To perform the parallel CPU bound tests, we've developed a module that works in Node.js and in the browser to perform Ray Tracing Tasks. The module works in a synchronous fashion so it performs faster, in another words, there are not techniques to make que module asynchronous that would create an overhead for the processing, in order to avoid stopping javascript event loop, the ray tracing task has to be ran in a sub process.

This module performs the interpretation of a scene designed in CSS, division of the scene in multiple parts (tasks) and reconstruction of the ray traced scene when every task is completed.

4.5 Summary

5 Evaluation

"ANOTHER FANTASTIC QUOTE"– *FROM SOMEONE FANTASTIC*

In this section we evaluate browserCloud.js via real executions on top of increasing number of browsers executing locally, to assess the limits of current Javascript engines on typical desktop machines, and with micro-benchmarks to determine the speedups that can be achieved in distributed executions with one browser per individual desktop machine.

This, in order to assess the potential of the proposed system, we have built a ray-tracing application, adapted from algorithms available, written in full vanilla JavaScript, that can be run on any typical modern browser engine. This algorithm allows us to stress-test the CPU, and the possibility to obtain advantages through processing parallelism. We need this to understand whether the expected speeds up resulting from distributing the tasks through the browserCloud.js peers network, are not hindered by loosing efficiency due to message routing on the overlay Network.

When distributing a job through a multiple node network, one of the aspects we observed was that we can influence overall efficiency by adjusting how much resources we are going to take from the network to process the job, in this case, how much browsers. We also can influence it by deciding how much fine-grained each task it will be: the smaller the computation unit, the more we can distribute tasks through the network, with a natural trade-off of adding more task generation and messaging overhead, with diminishing returns when more and more, and smaller tasks are created.

The standard ray-tracing job using the algorithm developed, running in a single browser takes as median 2361.434s to complete (around 40 minutes). As we can see in Figures **??** and **??**, our system excels in delivering faster results by dividing the job up to 2500 computational units (or tasks) and requesting from the browsers available in the network to compute those (i.e., a rectangle of the resulting output image). This is expected as ray-tracing is a known case of an

embarrassingly parallel application.

One fact interesting to note is that we obtained much better results by reducing the granularity of which ray-tracing job was divided into, as we can see on Figures **??** and **??**. This happens due to two factors: a) the first is that since we have a lower number of tasks to be run by other browsers, we reduce the message routing overhead between nodes (i.e., resource discovery does not take so long); b) the second factor is that since this system was tested using a single machine and a networked simulated delay. When the number of tasks is too large, the workers in the browser are in fact competing for CPU resources (to execute tasks and to forward messages among them). This creates a scenario, where more nodes/workers actually make the system slower, since this is a much more strict and resource constrained scenario, than a real example with browsers executing in different machines.

In a real world example, the actual execution time would be bounded by:

$$\textbf{jobTime} = slowestDelayFromResourceDiscovery + timeOfExecutingSlowestTask + slowestDelayFromResultReply(1)$$

with full parallelism, where in our test scenario we have:

$$\textbf{jobTime} = \sum DelayFromResourceDiscovery + (TimeOfExecuting\_N\_Tasks\_on\_M\_Resources) + \sum DelayFromResultReply(2)$$

where N=2500 and M=8 hardware threads, therefore contention for CPU becomes higher with more nodes (browsers) as more messaging is taking place, besides the parallelized computation.

In a real world scenario, with more browsers from more machines, the total execution time (makespan) of a ray-tracing job would be closer to that described by Equation 1. It would be influenced by the maximum round trip time between any two nodes (so that the information for every task can be received and processed by another node), plus the time it would take to execute the most of CPU intensive task (e.g., the rectangle in the frame that has the more complex geometry and light reflections to be processed). Figures **??** and **??** show what is the

average task length and RTT between any two nodes, being the maximum for the first 61ms and the second 11174ms, creating a total of 11235ms (or 11.296s overall). This is a significant increase of efficiency, comparing to the sequential execution (a speedup of about 209 times fold) and also to the previous single-machine experiments.

It is important to note that in Figure **??**, we can see several task execution lengths due to the complexity of each task, with more or less light reflections. With this microbenchmark we see that the execution time of each task, without any resource contention (1 node = 1 browser per machine), the task duration has an even lower upper bound (lower than 5s). This would entail the upper bound of total task execution time to be under 5061 ms (around just 5s), with a theoretical speedup of about 466 times (take into account that we would be using 2500 nodes then, so speedups are not perfectly linear due to communication overhead, as expected).

There are also some other performance bottlenecks we noticed that arise from the single threaded nature of JavaScript engines. These aspects are considered in our performance evaluation, such as:

- Logging - Since V8 runs in a single thread, any synchronous operation will block the event loop and add delay to the whole processing, although these logs are essential for us to assess the efficiency of the infrastructure, they are not vital to the job.
- Delay added - One technique we used to simulate the network delay is to use the 'setTimeout' native function of V8's JavaScript implementation, since this function is unable to receive floating millisecond values. Moreover, since 'setTimeout' does not necessarily guarantee that the function will be executed in X amount of milliseconds, due to the nature of the event loop, there is always an extra delay added implicitly to the operation in the order of 1 to 3 ms.
- Tasks can not be sent in parallel - A node has to send each individual computing unit sequentially and independently, meaning that if we divide a job into 2000 for e.g, each task will have to wait for the previous to be sent.

These bottlenecks were studied and will be tackled in future work, one of the solutions proposed is to use Service Workers(**?**) for full multithreaded operation inside browserCloud.js.

Summary

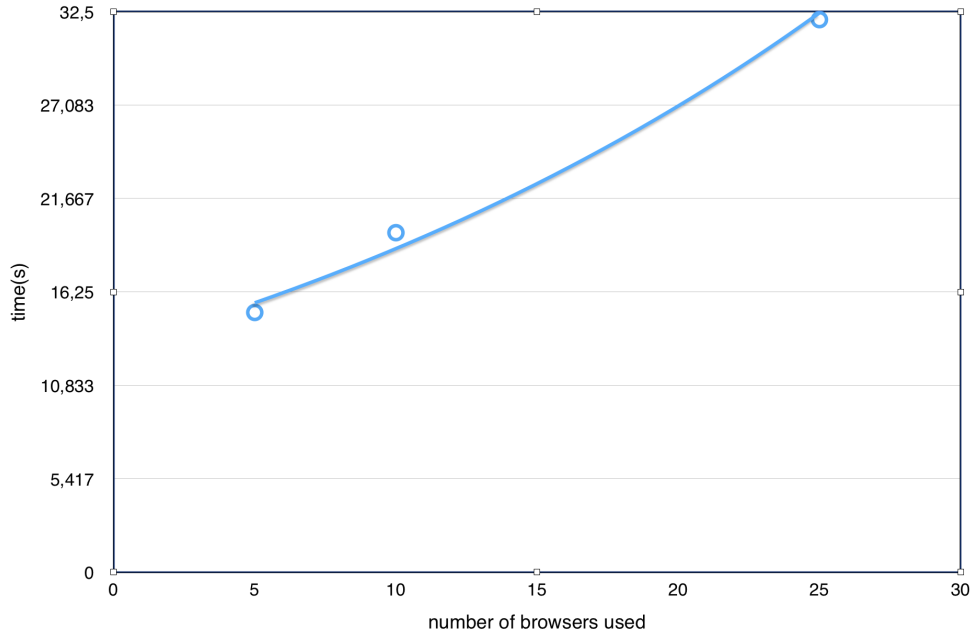


Figure 5.1: Time elapsed on a ray-tracing job divided in 2500 computing units

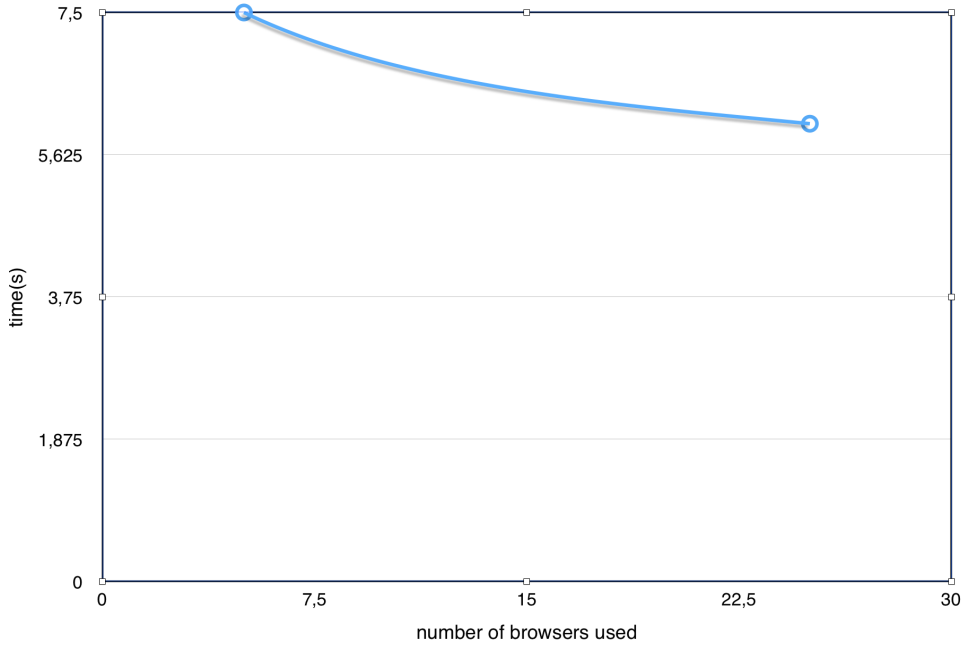


Figure 5.2: Time elapsed on a ray-tracing job divided in 25 computing units

Figure 5.3: Time elapsed on a ray-tracing job divided in 2500 computing units (with induced web RTT delay)

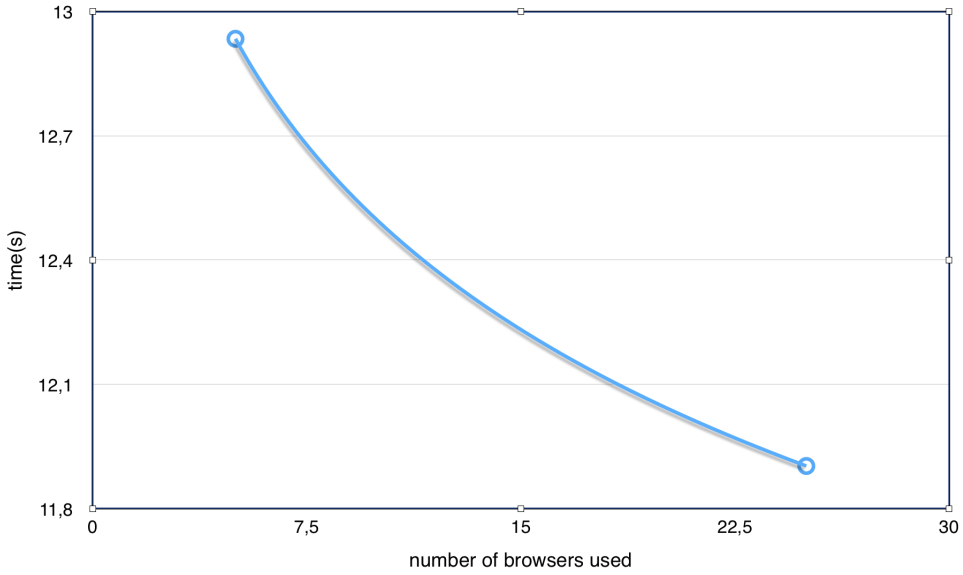


Figure 5.4: Time elapsed on a ray-tracing job divided in 25 computing units (with induced web RTT delay)

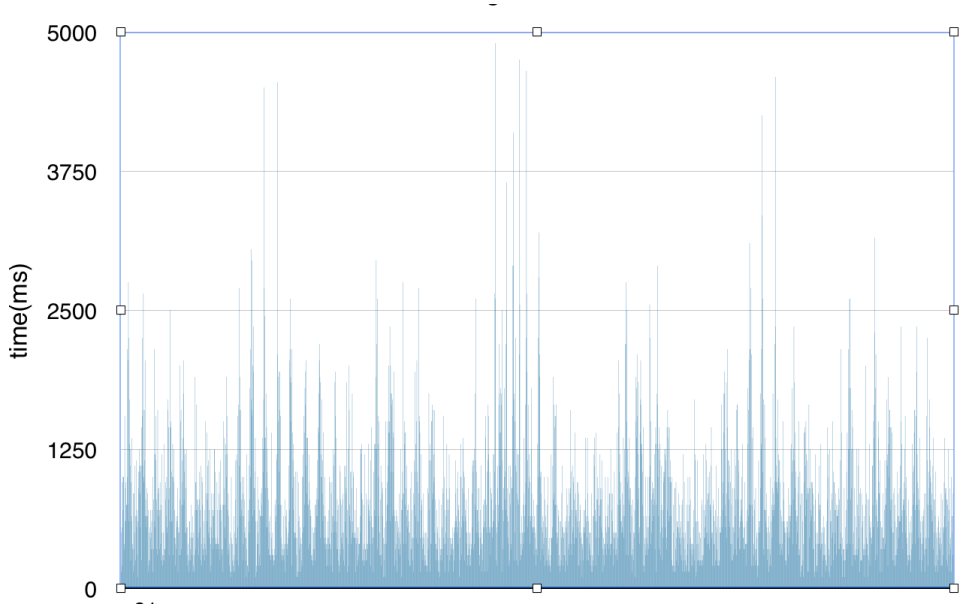


Figure 5.5: Average time for a task execution for a job fragmented in 2500 computing units

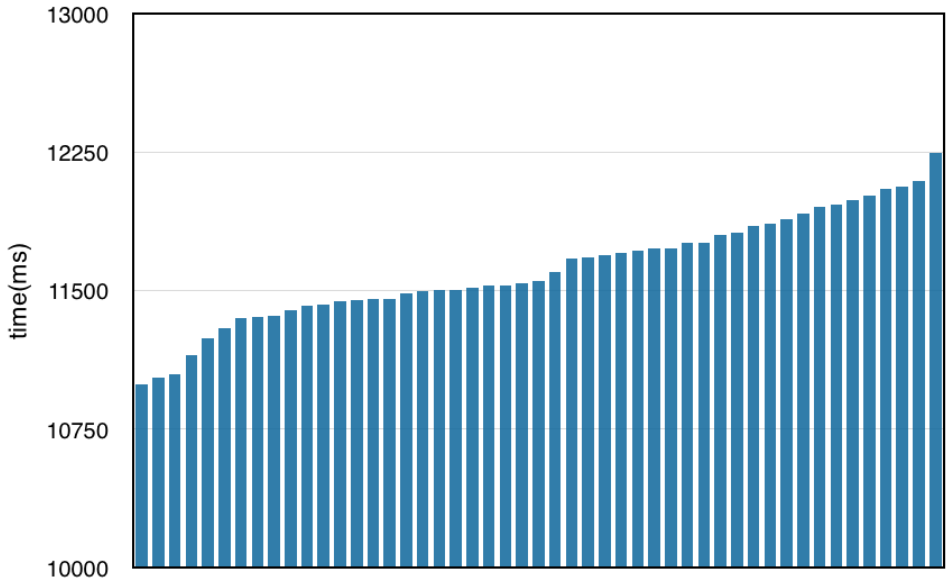


Figure 5.6: Average Round Trip Time between an two nodes in a 10 browser network

6 Conclusion

"The last mile is always the most difficult, and (looking backwards) the best" –
Miguel Mira Da Silva, professor at IST

We end this article, making an overview and summing up all the primary aspects of the proposed work and how it relates to what has been researched so far, presenting also some concluding remarks. People sharing resources is one of the oldest sociological behaviors in human history, however although some known attempts as SETI@HOME (even if extended with nuBOINC) have enabled that for our computer machinery, the level of friction that has to be made in order for a user to join, has been significantly high to cause a great user adoption. On the other hand, Open Cloud stacks have been evolving, providing nowadays the most reliable and distributed systems performance, having a bigger adoption even if the resources are geographically more distant or expensive. The proposed work is an exercise to strive towards a federated community cloud that will enable its users to share effectively their resources, giving developers a reliable and efficient way to store and process data for their applications, with an API that is familiar to the centralized Cloud model.

7 Future work

what do you see