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 in P2P Networks, Cloud computing and Mobile Applications

BSc in Communication Networks

Examination Committee

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

March 2015

Acknowledgements

WHO YOU ARE THANKFUL TO :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, that use volunteered shared resources and not only institution-wide data centers as before, creating the concept 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 applications. This work explores current ways of making an interopable way of end user machines to communicate, using new Web technologies, 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 IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS IGUAL AO ABSTRACT MAS EM PORTUGUÊS 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, Tolerância à faltas, Mecanismo de reputação, Nuvem comunitária

Keywords

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

Index

List of Figures

List of Tables

1 Introduction

"Your system can fail no matter how well you thought you tested it... what users will not tolerate is losing their data". – [1]

Today, in the information communications technology landscape, with the introduction of social networks, search engines, Internet of things, which eventually will drive home and vehicle automation, user data has been growing at a large pace. The store, transfer, processing and analysis of all this data brings considerable new knowledge 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.

Another challenges, is the fact that typically today, user generated data is controlled by some entity, company or organization, which holds the right to keep this information private, exploiting user data for their own goals and business. 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 need big amounts of CPU cycles for processing, analysis and inference.

Currently addressing this issues in part, Cloud computing has revolutionized the computing landscape mainly 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.

[1] *Lehene C. HStack, http://hstack.org/why-were-using-hbase-part-2*

- 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.

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 novelty in the Distributed Systems research area. However, existing models have been developed to follow the client-server model, transporting 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.

To accomplish this, we propose a new approach to abandon the classic centralized Cloud Computing paradigm, towards a common, dynamic, and privacy-aware cloud infrastructure. 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 (**?**)

Finally, browserCloud.js has the possibility to grow organically with the number of users. The management of these resources is done by an RESTful API, enabling desktop and mobile apps to use the resources available in a way that's familiar to developers.

1.1 Overview

1.2 Problem Statement

1.3 Extended motivation and Roadmap

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 local storage module that offers persistence and availability, using browser storage for fast 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.
- A client API, RESTful, so it is easy to develop applications for Desktops and mobile platforms without having to change the codebase or building a new SDK
- A command line interface for access like 'mountable' partition to the storage in browserCloud.js, able to dispatch jobs in a very Unix way, by piping the results from one task to another task.

These components are fully described in section 4. 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 as a Cloud provider, comparing to existing centralized Cloud systems.

1.4 Research Proposal

1.5 Contributions

1.6 Publications

1.7 Structure of the thesis

Document roadmap: We start by describing the objectives of our solution in Section 2, and then, in Section 3 we present 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 3.1), Volunteered resource sharing (at 3.2) and Resource sharing using the Web platform (at 3.3). In Section 4, we present the proposed architecture and respective software stack, moving to the system evaluation present on Section 5.

2 Related Work

No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be. – *Isaac Asimov, writer and scientist (1919 - 1992)*

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.

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

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 2.1. 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:

"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 2.2, 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.

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.

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 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 2.1:

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

- 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.

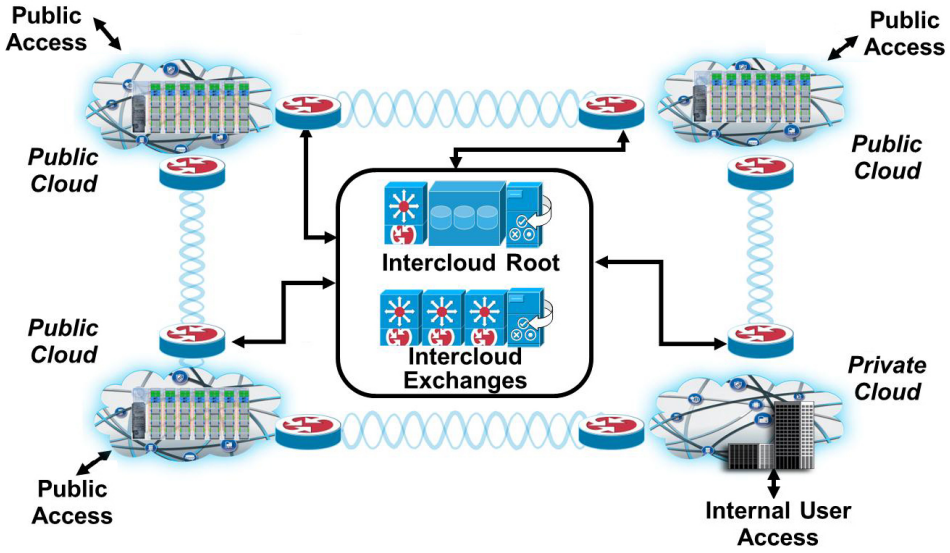

Figure 2.1: IEEE Intercloud Testbed Architecture

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 2.2. 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.

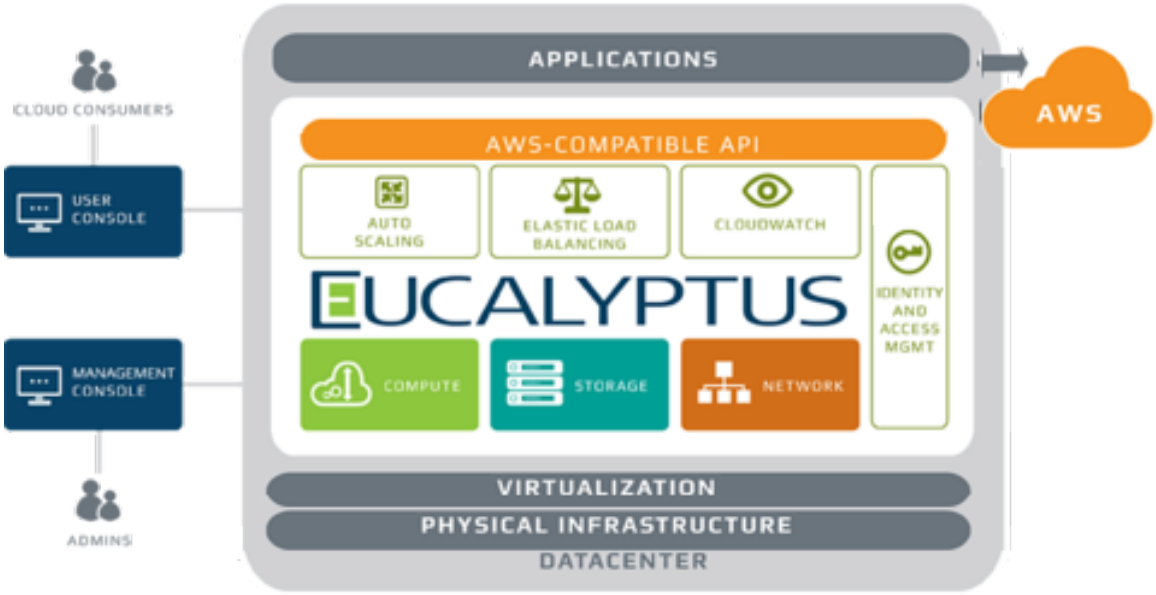

Figure 2.2: Eucalyptus Architecture

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 optimization (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-

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

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.

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

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

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.

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 2.3.

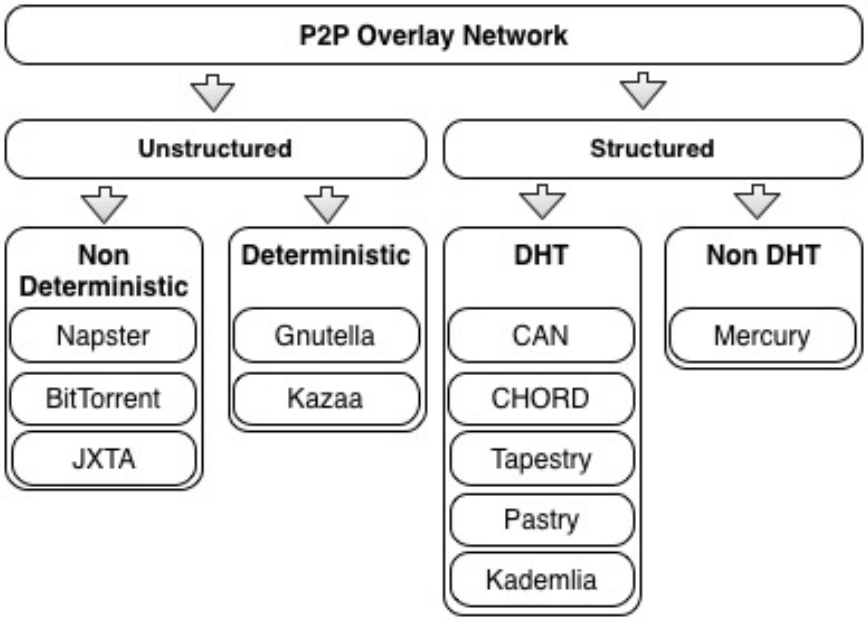

Figure 2.3: Different types of P2P Overlay networks organizations

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 remains 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 2.3, showcases a comparison of the studied DHT algorithms.

P2P system	**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) |	O(log(n))	
CAN	d-dimensional ID Space	Key value pair map to a point P in the D-dimensional space	n= number of nodes in the network, d=number of dimensions	O(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 different bits (prefix)	O(log(n))	$O(\log_2(n))$ not stable	

Table 2.3: Summary of complexity of structured P2P systems

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'.

2.2.4 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.

2.2.4.1 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.

2.2.4.2 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 distri-

bution, 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 2.4

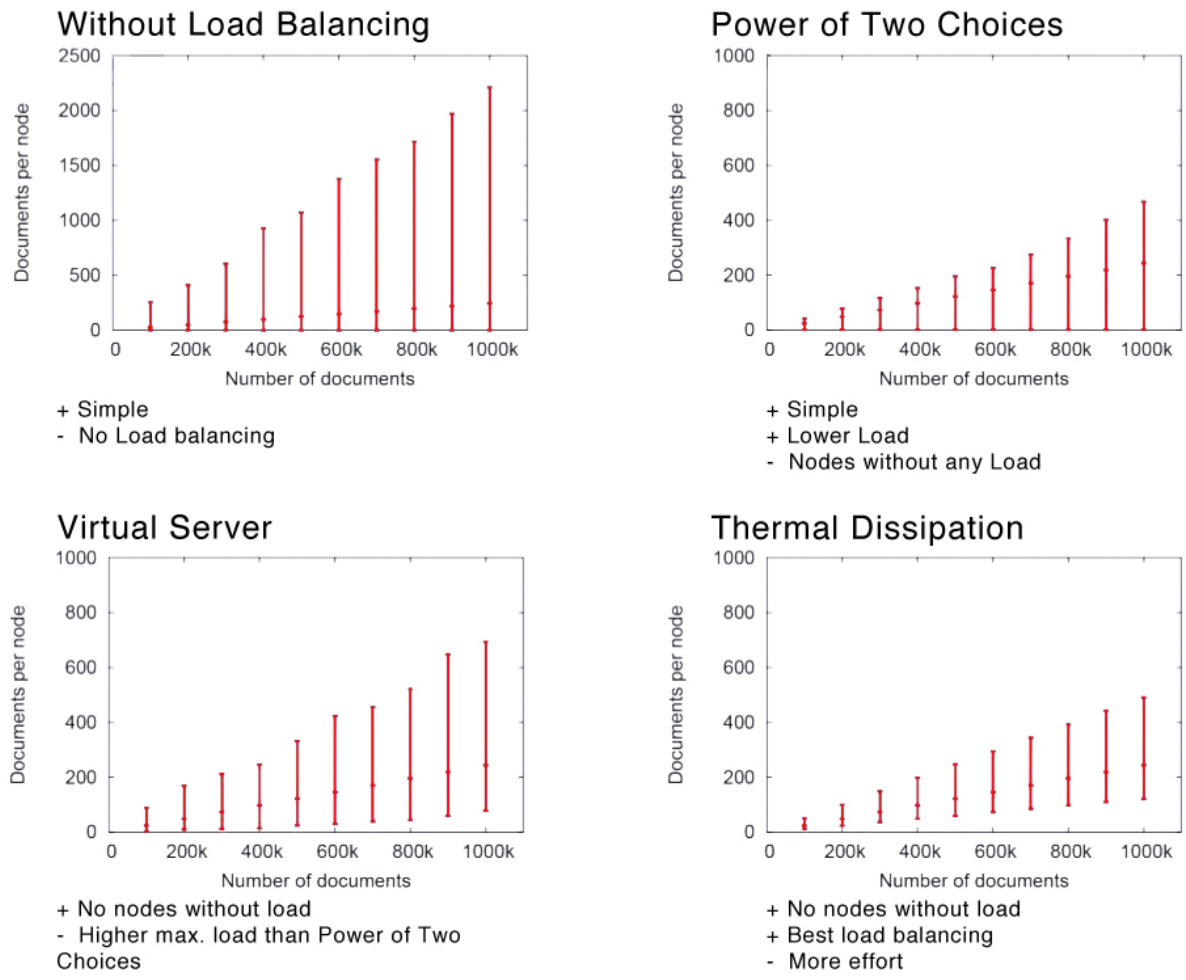

Figure 2.4: Load balancing approaches comparison

2.2.4.3 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 encryp-

tion 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 2.4.

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

Table 2.4: Reputation system components and metric

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 2.5, 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.

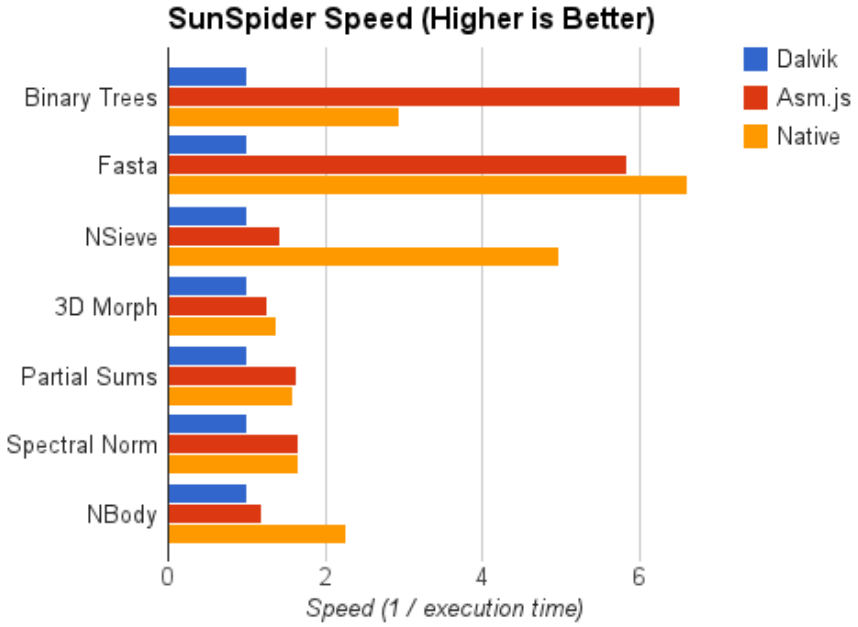

Figure 2.5: Dalvik vs. ASM.js vs. Native performance

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

[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

be opened, a visualization of this process can be seen in Figure 2.6. 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(**?**).

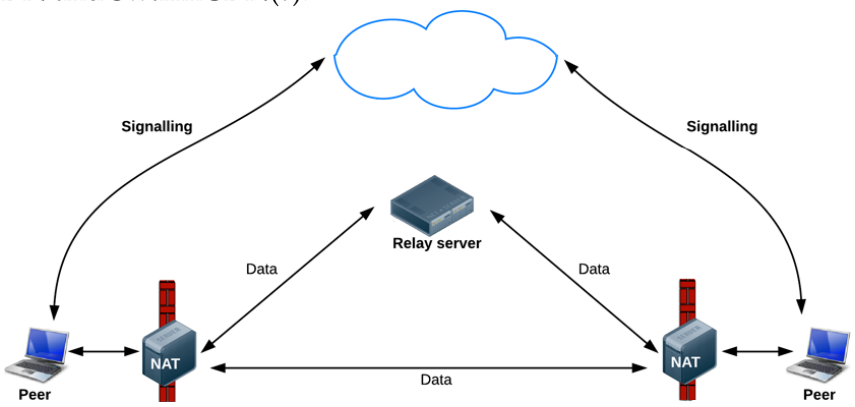

Figure 2.6: Example of a WebRTC session initiation

'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 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 2.8, 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 2.7, 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.

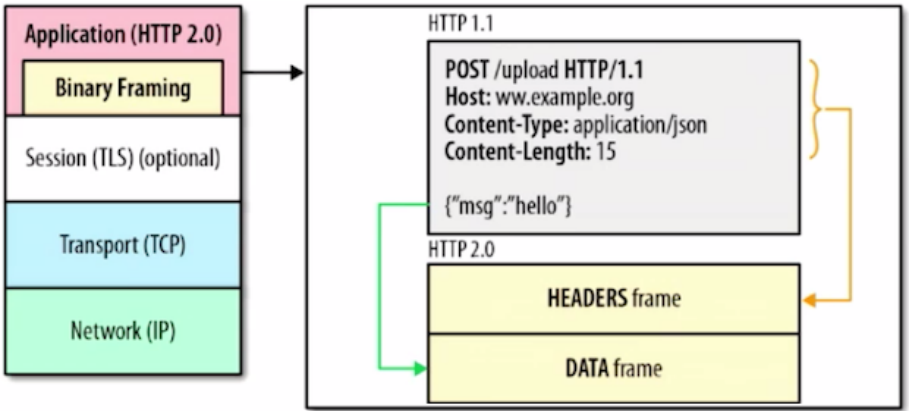

Figure 2.7: HTTP2.0 Binary framing

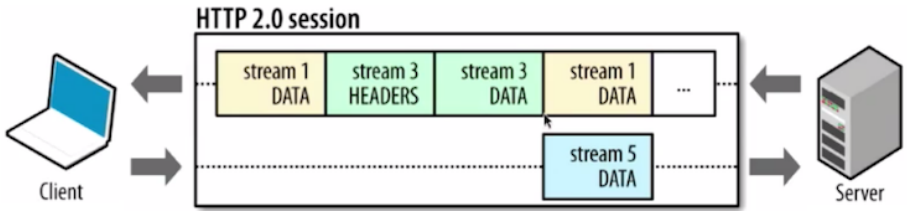

Figure 2.8: Example of an HTTP2.0 dataflow

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

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

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.

Summary

3 Architecture

"The greatest pleasure in life is doing what people say you cannot do." – *Walter Bagehot (British political Analyst, Economist and Editor, one of the most influential journalists of the mid-Victorian period.1826-1877)*

In this section we describe our proposed architecture for the remaining implementation work. The software stack is composed by several subsystems that have one specific goal, exposing a well known API, this way the subsystems become interchangeable. These subsystems include:

- **Communication service** - Responsible for routing messages between nodes in the DHT.
- **Service router** - Processes the messages that have as destiny the its node, the goal is to call the right service (storage, reputation mechanism, job execution, etc) accordingly. One other key aspect is the ability to attach new services during the runtime.
- **Storage service** - Stores any data that requires persistence in the network, such as job logs, reputation logs and file meta data and chunks.
- **Job coordination** - A subsystem responsible for coordinating jobs requested by the client, keeping state and assuring its completion.
- **Job execution** - Execution of jobs, gathering all the necessary assets (image processors, sound wave manipulators, etc) to complete the job.
- **Reputation Mechanism** - Validate user behavior and rights to take different responsibilities in the network.
- **Client API and CLI** - In order to interact with the network, we offer an API and a CLI with Unix type instructions and familiar web cloud instructions with which developers are familiar.

- **Rendezvous points** - The only centralized component in this architecture, its purpose is for the clients to have a way to connect to the overlay network.

In the following section we present the proposed components of the architecture using a 'bottom up' approach, starting with the software architecture of each node in Section 4.1, moving into how the network is structured and how the nodes can join the network, described in Section 4.2, ending with a specification of how to interact with browserCloud.js, thorough a RESTful API endpoint or using a Command Line Interface (CLI). We present how these components are connected with each other in Figure 3.1.

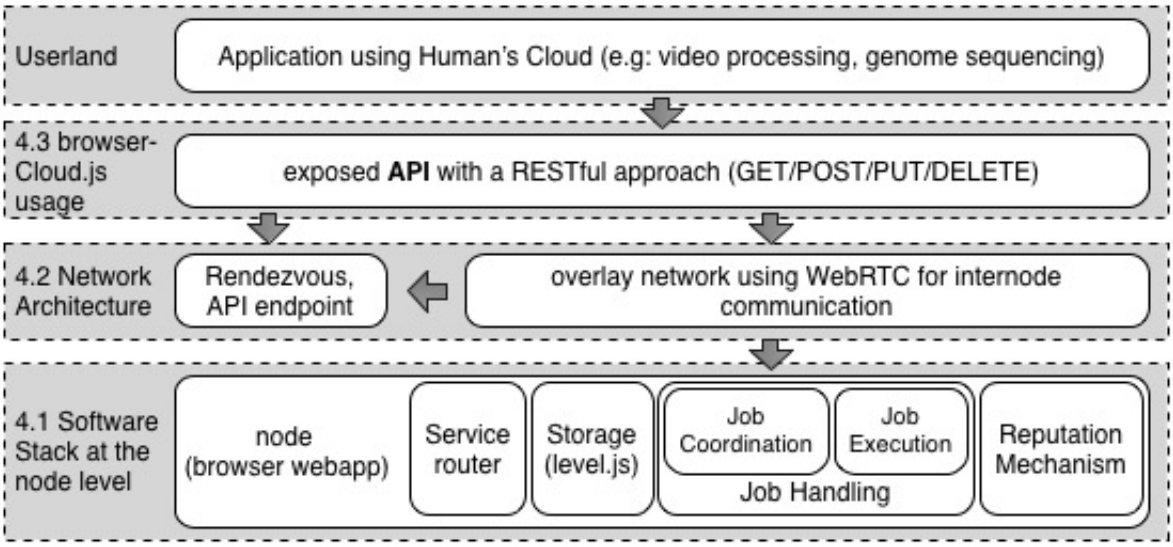

Figure 3.1: browserCloud.js overall architecture

3.1 Software Architecture at the node level

At the node level, we divide the application into two fundamental services and three pluggable components, with the possibility for expansion, thanks to Javascript dynamic runtime, we can find this structure in Figure 3.2.

In the communication layer, we find the DHT logic implemented to effectively propagate messages. One of the main goals with component is to be modular, so we can switch between

different DHT algorithms and topologies if necessary, without affecting the rest of the application functionality.

Next, we have the Service Routing layer, this service is responsible to guide the message to the right component, enabling the architecture to be more modular, plugging in more components as it is needed. For example, when a node ascends and needs the storage component to fulfill his responsibility.

Last, we have the components, individual modules that do one thing and one thing well. Currently, we present the Storage module, responsible for holding the data; the Job Scheduler, responsible to orchestrate jobs issued by the users; the assets needed to execute the jobs and finally; the job executor, the module that will execute the jobs in a separate process using webworkers.

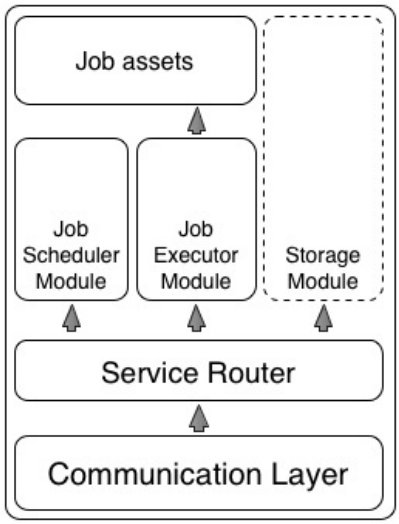

Figure 3.2: browserCloud.js Node

3.1.1 Storage -

browserCloud.js storage takes place in what it is named, the "Ascended node ring", these nodes have an higher reliability, making the storage system more stable, without the need of constantly burning computer cycles to maintain the files replica level.

Data stored in nodes can be:

- File metadata (name of the file, size, location of the chunks, chunks hash).

- File chunks.
- Directories metadata - this way, bcls can be more efficient.
- Job information (state, issuer, workflow).
- Reputation log.

We classify storage nodes into two types: 1) the 'sKeeper', responsible for holding the metadata of the file and hashing each chunk to identify the 'sHolder'; 2) 'sHolder' nodes responsible to store the chunk into their system. This approach mitigates the possibility of having an highly unbalanced storage distribution, dividing each file in equal chunks across several nodes. As we can see in Figure 3.3, each chunk gets hashed more than one time with a different hash function, its purpose being to identify several Nodes that will be responsible to store a replica of the chunk. Also, in order to increase the fault tolerance of the system, we replicate the 'sKeeper' responsibility in the two following nodes in the hashring of the DHT, so if one of these fails, another is assigned.

Figure 3.3: A file partitioned in several chunks, each with its corresponding hashes that correspond to nodeIds

In Figure 3.4, we can find the 'sKeeper' and 'sHolder' relationship. Only the sKeeper performs the chunk hashing and stores the information in the file lookup table. This happens one single time for each chunk, reducing several network hops per message on the consequent searches.

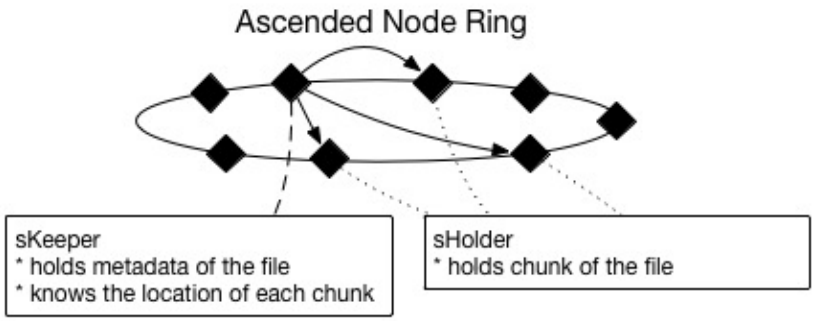

Figure 3.4: Representation of the Node responsible for the file(sKeeper) and it's individual chunk holders(sHolders)

Each stored file is chunked as soon as it enters the network, thus mitigating the risk that would be present if we were transferring files with considerable sizes all at once, starving the network and the node's heap. The only point where the file gets assembled together again is when it leaves the network and sent to the user, and even this could be made to perform chunk transfer in parallel to the client directly.

browserCloud.js adapts the Load Balancing virtual server's method, by using the same strategy of global load, but by transferring files between sHolders and not an entire virtual server, updating the respective sKeeper accordingly. Files are stored as objects in a indexedDB type storage, provided by the leveljs module.

3.1.2 Distributed Job Scheduling -

Job coordination is one of the main challenges in a completely distributed environment, in a sustainable and scalable way. Traditionally in the client-server model, we have the possibility to select one of the nodes to be the job coordinator. To implement this in a P2P network, we take advantage of the DHT, to select randomly one of the ascended nodes to be the 'jKeeper', the node responsible for coordinating the job in an environment such as a P2P network.

The 'jKeeper' is responsible for contacting the 'sKeepers' of each individual file, and coordinate them to command each of 'sHolder' to perform the desired computation on the file. All the steps during the computation are journaled in the Job log, stored with the coordinator, and replicated in the two following nodes as Fault Tolerance measure.

All the coordination takes place in the ascended Node ring; however, in order to take advantage of the normal node ring resources, 'sHolders' are allowed to offload the computation to process this job in the 'normal hash ring'. This is done by sending a probe, asking for 'volunteers' for a job; when the threshold required is met, the orchestration starts, where the 'sHolder' transfers the data and the assets necessary for processing it.

An example in pseudo-code can be analyzed below:

Client pseudo-code

```
var jobId = randomUniqueIdGenerator();
sendJob(jobId, job); // job object includes the files names being manipulated+assets
```

jKeeper pseudo-code

```
var jobLog = createNewJobLog();
replicateJob(jobLog); // each job replica holder will ping the jKeeper to make sure progress
// is made, if the node fails, other will assume its role
job.sKeepersList.forEach(function (sKeeper){
  commandJobExecution(job, sKeeper, statusReport);
  function statusReport(status){
    jobLog.log(status);
  }
}); //Job is complete
```

sKeeper pseudo-code

```
var sHolders = this.getsHolders(job.filename);
sHolders.forEach(function(sHolder){
  commandTaskExecution(sHolder, taskReport);
  function taskReport(status){
    reportBack(status); // report to jKeeper
  }
});
```

sHolder pseudo-code

```
if(smallTask && available) {
  doIt(task, taskReport);
} else {
  requestNormalNodesToExecute(task, taskReport)
}
function taskReport(status){
  reportBack(status); // report to sKeeper
}
```

3.1.3 Reputation Mechanism -

The reputation mechanism present will enable the network to identify the nodes that show more availability, and that have the necessary means to ascend and take a more important role. In order to evaluate each node, we define several metrics, these are: uptime, number of job completions, network throughput and computational resources (CPU) available, being the uptime, the most important, to assure stability. The reputation metric is calculated as follows:

$$\textbf{reputation} = \alpha * log(uptime) + \beta * log(jobcompletions) + \gamma * log(networkthroughput) + \delta * log(CPU)$$

where:

$$\alpha + \beta + \gamma + \delta = 1;$$
$$\alpha > \beta + \gamma + \delta;$$

We chose to normalize the metrics and give more importance to the uptime of the system, because this is the one metric allowing a more stable network for storage.

The reputation of each node is stored with its node identifier on the 'ascended hash ring'. Each time a job is completed successfully, this score gets updated and in case it reaches the required level to ascend, the jKeeper that was updating this score will enable and deploy the remaining features (storage and job schedule module) it needed to join the ascended group.

3.2 Network Architecture Details

We designed browserCLoud.js network architecture as shown in Figure 3.5, being composed by 4 different elements, being:

- Normal Nodes - We refer to normal node as the node responsible for only computation task in our network.
- Ascended Nodes - These nodes are responsible for both computation, storage and job coordination.

- API endpoints - The rendezvous points for the browsers to download the necessary code to enter in the browserCloud.js network. These API endpoints are also responsible for establishing the connection from a client to a browser.
- Clients that use browserCloud.js - Applications using the RESTful API to interact with browserCloud.js

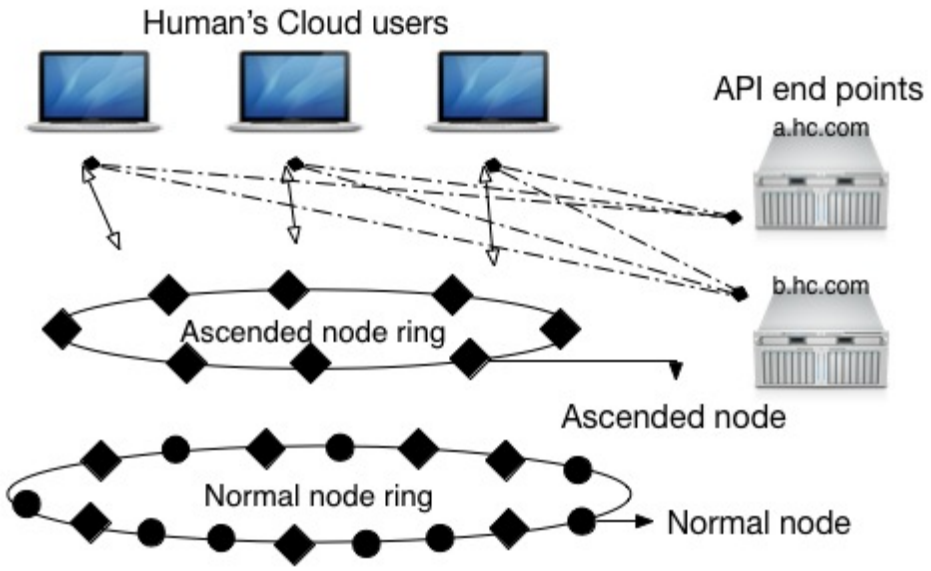

Figure 3.5: browserCloud.js network architecture

3.2.1 Structured Overlay Network -

Nodes are divided into two Chord DHTs with the purpose of separating the nodes with storage and job coordination responsibility from the ones with only computing responsibility. The reason behind this decision is due to the high churn rate in a P2P network. Keeping the files in nodes that have proven to be more trustworthy for staying longer in the network makes the system more robust, keeping the file replica level stable. This also reduces the message overhead that would require to keep the replica level in a more inconsistent environment. Nevertheless, the more volatile nodes are perfect for short computing operations, till they proven to be trustworthy to 'ascend' in the network.

3.2.2 Rendezvous points -

Since browsers cannot effectively have a static IP nor have a DNS record updated on demand pointing to themselves, we designed the API endpoints as the rendezvous between browsers and clients, so the connection can be established.

3.3 browserCloud.js usage

The client API goal is to be familiar to experienced developers using cloud providers today and at the same time, respect the Unix philosophy, where the job work flows are composed by a stream of assets, that do one thing and one thing well. It will support CRUD operations through a REST with JSON API, a filesystem-like interface (directories and objects), where user is identified by a username and it maps to the paths shown in Table 3.1:

Path	Description
/:username/jobs/	where new jobs can be inserted by the user
/:username/jobs-reports/	Job status. The user is only able to read and delete the records, they are created by the system.
/:username/home/	Private user store, only place where the user has write access
/:username/reports/	Usage and Access log reports.
/nodes/	Registry of all the nodes in the network with their metadata, the user can only read this folder.

Table 3.1: Directories representation inside the network

3.3.1 Application Programming Interface (API) -

We provide a REST API for the developer to use. The reason behind this design decision is to create a familiar interface to the majority of web developers. The server replying to these requests defined in Table 3.2, can be a public or a private proxy, in the user machine, behind a company firewall or as a public service available to the community. Thus, it remains portable and does not lock in the user to a provider.

Directories	
Action: PutDirectory	PUT /:username/home/[:directory]/:directory
Action: ListDirectory	GET /:username/home/[:directory]/:directory
Action: DeleteDirectory	DELETE /:username/home/[:directory]/:directory
Files	
Action: PutFile	PUT /:username/home/[:directory]/:filename
Action: GetFile	GET /:username/home/[:directory]/:filename
Action: DeleteFile	DELETE /:username/home/[:directory]/:filename
Note: PutFile and GetFile, the body of the request and response respectively is the file	
Jobs	
Action: CreateJob	PUT /:username/jobs
Action: CancelJob	DELETE /:username/jobs/:jobId
Action: ListJobs	GET /:username/jobs
Action: GetJob	GET /:username/home/jobs/:jobId
Action: GetJobOutput	GET /:username/jobs-reports/:jobId
Note: Create Job, several arguments are passed, most importantly, an array named "phases" that includes the orders with assets necessary in order to execute the job(e.g. 'grep -ci' or if its a new asset, it should be a JS object with a closure.)	

Table 3.2: browserCloud.js REST API Draft

3.3.2 Command Line Interface (CLI) -

We are also including a CLI[1] tool to enable quickly bash scripts for computation jobs and file storage, this CLI uses in the background the API defined in Table 3.2. For example, if we are looking for video transcoding: "hcjob create /path/to/file — ffmpeg — /path/to/out.webm". The rest of the commands are:

- **$ bcls** - List files in a directory
- **$ bcget** - Get an object stored

[1]CLI - Command Line Interface

- **$ bcput** - Store an object
- **$ bcjob** - Initialize a job

Summary

4 Implementation

"Keep it simple, stupid" K-I-S-S, is an acronym as a design principle noted by the U.S. Navy in 1960. The KISS principle states that most systems work best if they are kept simple rather than made complex; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided. – *Kelly Johnson, aircraft engineer (1910 - 1990)*

We've done a lot of things

Summary

5 Evaluation

"Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted"– *Albert Einstein*

The proposed system will be evaluated with the goal of comparing to existing centralized non P2P cloud and P2P distributed voluntary job computing and storage. The desired outcome of this analysis will be to produce a quantifiable set of comparable metrics to other systems, taking into account issues such as: scalability, resilience, availability, processing power and latency. In the end, we expect to have an assessment, "a new type of application/solution", mentioning the advantages present using browserCloud.js in comparison with the other systems, presented in table 5.1.

5.1 Qualitative Evaluation of data consistency, availability and partition tolerance

In browserCloud.js files are immutable, any operation that involves data manipulation will create a new file with the changes. This means that once the file is ready to be read, it is by default, the most current version. Any transactional semantics of writing(PUT) and deleting(DELETE) files are maintained on the application, developed by a user, using browserCloud.js platform.

In this evaluation, our target is to test how the system behaves in different conditions, in order to assess the data consistency semantics possible (Eventual Consistency, Monotonic Read Consistency, Immediate Consistency, etc).

Another point that we want to evaluate how tolerant is browserCloud.js is for data partition, taking into account we are limited by the available storage that the browser provides and enables access to, and of course, the need to have smaller chunks to be quickly transfered when churn rate is high.

These tests will be executed in two different stages: the first one, “in lab”, will be a controlled P2P environment, where different browsers and computers will be used for tests, in order to evaluate and calculate the factors that are used to calculate values such as: reputation, threshold to ascend one Node, and block size.

After realizing how the system can perform best, a “field” trial will be executed, this will be executed by approaching volunteers that might want to contribute to the experiment, loading the code into their browser so real world tests can be performed.

5.2 Quantitative Evaluation of system performance when executing jobs and storing/fetching data

One of the key factors for an App ready cloud platform is its latency; storing and fetching data has to be rapid enough that it does not limit the performance of the applications using browserCloud.js. The system performance varies depending on its usage; in order to evaluate it correctly, tests will be performed, changing several factors that will impact latency:

System evaluation parameters:

- Churn Rate - Varying the churn rate will create instability in the availability and readability of the overall computing power of the platform, creating scenarios where job tasks have to be resent to another node to be completed, adding delay to the estimated time of the job. We can also loose the point of contact to the network, which requires need to reconnect again, adding more time for any request.
- Number of nodes - The greater the network, the more distributed can be its load, which favors faster request handling by the node; however, as the network grows, the number of messages traded between the nodes grows and these have impact as well.
- Number of parallel connections performing requests - With this variance, we want to make sure how much traffic and load the system is able to cope with.
- Number of requests - This is related with the number of parallel connections, however, more focused on number per each application using browserCloud.js API.
- Number of jobs running - Jobs manipulate data and consume the processing power of the system, which also influences latency.

- Different volumes of file storage - Different types of data have different needs, serving big volumes of data is an harder task because it must be served by several nodes

The tuning of this system evaluation parameters will enable us to asses how this system behaves, using the following metrics of evaluation, comparable to other Cloud systems:

System performance metrics:

- Resource utilization - How much, in percentage, is possible to utilize from all the potentially available resources using this approach.
- Load Balancing - If the load is well distributed among all the nodes or if it is imbalanced.
- Cost - What is the cost associated with the set up of browserCloud.js and what are the costs of using it, regarding CPU, memory and bandwidth usage overhead.
- Execution speed - Time necessary to perform a job with different levels of complexity, and comparative speedups.

The tests will be executed in two different phases: 1) the first in a controlled environment, being able to modify the churn rate on demand and evaluation its behavior, these tests will be essential to evaluate and tune browserCloud.js to the 2) real world tests, using pure voluntary browsers, which are a non controlled environment. These tests will permit us to assess quantifiable results. Both regarding adequate operation and stress testing in small scale, as well as efficiency and scalability in the large. They will also be compared with other cloud like platforms.

5.3 Envisioned final comparative analysis

Once the evaluation of each component it i s done, we envisioned a one by one comparison between browserCloud.js and some of the most used, or known systems for distributed computing jobs and storage to pinpoint which excels the best for each type of task with the respective relative trade offs, classifying with a numeric value in the range [-1,1] for each of the respective dimensions of comparison: **Q - Quality of results/functional compliance**, **P - Performance** , **C - Cost**. In Table 5.1, we can see an example.

Type of Task	AWS	SETI@Home	community-lab	browserCloud.js	Obs.
Genome Sequencing	[Q,P,C]	[Q,P,C]	[Q,P,C]	[Q,P,C]	...
Photo Storage	[Q,P,C]	[Q,P,C]	[Q,P,C]	[Q,P,C]	...
Realtime Application	[Q,P,C]	[Q,P,C]	[Q,P,C]	[Q,P,C]	...
...	...	...	...	...	...

Table 5.1: Outline for summary comparison table of browserCloud.js against other Cloud and distributed computing platforms.

Summary

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

Bibliography

Afify, Y. (2008). *Access Control in a Peer-to-peer Social Network*. Ph. D. thesis, ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE.

Anderson, B. D. P., J. Cobb, E. Korpela, & M. Lebofsky (2002). SETI@Home, an Experiment in Public-Resource Computing. *45*(11).

Anderson, D. (2004). Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings.*

Armbrust, M., I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, & A. Rabkin (2010, April). A view of cloud computing. *Communications of the ACM 53*(4), 50.

Bakhtiari, S. & J. Pieprzyk (1995). Cryptographic hash functions: A survey." Centre for Computer Security Research, Department of Computer Science. pp. 1–26.

Barabási, a. L., V. W. Freeh, H. Jeong, & J. B. Brockman (2001, August). Parasitic computing. *Nature 412*(6850), 894–7.

Barraca, J. a. P., A. Matos, & R. L. Aguiar (2011, April). User Centric Community Clouds. *Wireless Personal Communications 58*(1), 31–48.

Bharambe, A. R., M. Agrawal, & S. Seshan. Mercury : Supporting Scalable Multi-Attribute Range Queries. pp. 353–366.

Byers, J., J. Considine, & M. Mitzenmacher (2003). Simple Load Balancing for Distributed Hash Tables. In M. Frans Kaashoek ; Ion Stoica (Ed.), *Peer-to-Peer Systems II*, pp. 80–88. Springer Berlin Heidelberg.

Clarke, I., O. Sandberg, B. Wiley, & T. Hong (2001). Freenet: A distributed anonymous information storage and retrieval system. In H. Federrath (Ed.), *Designing Privacy Enhancing . . .* , pp. 46–66. Springer Berlin Heidelberg.

Cohen, B. (2009). The BitTorrent Protocol Specification.

Costa, F., J. Silva, L. Veiga, & P. Ferreira (2012). Large-scale volunteer computing over the Internet. *Internet Services and Applications*, 1–18.

D. Eastlake, 3rd Motorola; P. Jones Systems, C. (2001). RFC 3174 US Secure Hash Algorithm 1 (SHA1).

Decandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo : Amazon's Highly Available Key-value Store. pp. 205–220.

Definition, P. (2003). The Gnutella Protocol Specification v0 . 4. *Solutions*, 1–8.

Desmedt, Y. & Y. Frankel (1990). Threshold cryptosystems. *Advances in Cryptology—CRYPTO'89 ...*.

Douceur, J. R. (2002). The Sybil Attack. In P. D. Druschel@cs.rice.edu & A. R. Antr@microsoft.com (Eds.), *Peer-to-Peer Systems*, pp. 251–260. Springer Berlin Heidelberg.

Duda, J. & W. Dłubacz (2013). Distributed evolutionary computing system based on web browsers with javascript. *Applied Parallel and Scientific Computing*.

Ecma, S. (2009). ECMA-262 ECMAScript Language Specification.

Filipe, P. & G. Oliveira (2011). Gridlet Economics : Resource Management Models and Policies for Cycle-Sharing Systems.

Golle, P., K. Leyton-brown, I. Mironov, & M. Lillibridge (2001). Incentives for Sharing in Peer-to-Peer Networks. pp. 75–87.

Handley, M. & R. Karp (2001). A Scalable Content-Addressable Network. In *SIGCOMM '01 Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Volume 21, pp. 161–172.

Ian Hickson (2013). WebRTC 1.0: Real-time Communication Between Browsers.

Jeffrey Dean; Sanjay Ghemawat. LevelDB.

Karger, D., T. Leightonl, D. Lewinl, E. Lehman, & R. Panigrahy (1997). Consistent Hashing and Random Trees : Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web *. In *STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663.

Karger, D. R. & M. Ruhl (2004). Simple efficient load balancing algorithms for peer-to-peer systems. *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA '04*, 36.

Koloniari, G. & E. Pitoura (2005). Peer-to-Peer Management of XML Data : Issues and Research Challenges. *34*(2), 6–17.

Korpela, E. & D. Werthimer (2001). SETI@Home, Massively Distributed Computing for SETI. *Computing in Science & Engineering*, 78–83.

Larson, S., C. Snow, & M. Shirts (2002). Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology.

Marti, S. & H. Garcia-molina (2006, March). Taxonomy of Trust : Categorizing P2P Reputation Systems. *Computer Networks* (April 2005), 472–484.

Maymounkov, P. & D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric.

Merelo, J.-j., A. Mora-garcía, J. Lupión, & F. Tricas (2007). Browser-based Distributed Evolutionary Computation : Performance and Scaling Behavior Categories and Subject Descriptors. pp. 2851–2858.

Milojicic, D. S., V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu, & J. I. M. Pruyne (2003). Peer-to-Peer Computing. Technical report.

Navarro, L. Experimental research on community networks. Technical report.

Nurmi, D., R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, & D. Zagorodnov (2009). The Eucalyptus Open-Source Cloud-Computing System. *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 124–131.

Paulson, L. (2005, October). Building rich web applications with Ajax. *Computer 38*(10), 14–17.

Preneel, B. (1999). The State of Cryptographic Hash Functions. pp. 158–182.

Ranjan, R., A. Harwood, & R. Buyya (2006). A study on peer-to-peer based discovery of grid resource information. *Australia, Technical Report GRIDS*, 1–36.

Rao, A., K. Lakshminarayanan, S. Surana, & R. Karp (2003). Load Balancing in Structured P2P Systems. *0225660*, 68–79.

Rieche, S., L. Petrak, & K. Wehrle. A thermal-dissipation-based approach for balancing data load in distributed hash tables. *29th Annual IEEE International Conference on Local Computer Networks*, 15–23.

Ripeanu, M. (2002). Peer-to-peer architecture case study: Gnutella network. *Proceedings First International Conference on Peer-to-Peer Computing*, 99–100.

Rodrigues, R. & P. Ferreira. GiGi : An Ocean of Gridlets on a " Grid-for-the-Masses ".

Rowstron, A. & P. Druschel (2001a). PAST A large-scale , persistent peer-to-peer storage utility. *Proceedings of the eighteenth ACM symposium on Operating systems principles - SOSP '01*, 75–80.

Rowstron, A. & P. Druschel (2001b). Pastry : Scalable , Decentralized Object Location , and Routing for Large-Scale Peer-to-Peer Systems. In Rachid Guerraoui (Ed.), *Middleware 2001*, pp. 329–350. Springer Berlin Heidelberg.

Ruellan, H. & R. Peon (2013). HPACK-Header Compression for HTTP/2.0. *draft-ietf-httpbis-header-compression-04 (work in progress)* (c), 1–57.

Shirky., C. Clay shirky's writings about the internet. In http://www.shirky.com/.

Silva, J. a. N., L. Veiga, & P. Ferreira (2008, October). nuBOINC: BOINC Extensions for Community Cycle Sharing. In *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pp. 248–253. IEEE.

Silva, J. a. N., L. Veiga, & P. Ferreira (2011, August). A2HA—automatic and adaptive host allocation in utility computing for bag-of-tasks. *Journal of Internet Services and Applications 2*(2), 171–185.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, H. B. Ý, & H. Balakrishnan (2001). Chord : A Scalable Peer-to-peer Lookup Service for Internet. pp. 149–160.

T. Klingberg, R. M. (2002). RFC - Gnutella 0.6 Protocol Specification.

Thomson, M. & A. Melnikov (2013). Hypertext Transfer Protocol version 2.0 draft-ietf-httpbis-http2-09.

Tilkov, S. & S. V. Verivue (2010). Node.js : Using JavaScript to Build High-Performance Network Programs.

Vishnumurthy, V., S. Chandrakumar, & G. Emin (2003). Karma: A secure economic framework for peer-to-peer resource sharing.

Vogt, C., M. Werner, & T. Schmidt (2013). Leveraging WebRTC for P2P Content Distribution in Web Browsers. *21st IEEE Internanicoal Conference*.

W3C (2013). Indexed Database.

Wallach, D. S. (2003). A Survey of Peer-to-Peer Security Issues. In M. O. Mitsu@abelard.flet.keio.ac.jp (Ed.), *Software Security — Theories and Systems*, pp. 42–57.

Zakai, A. (2011). Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM.

Zhao, B. Y., J. Kubiatowicz, & A. D. Joseph (2001). Tapestry : An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report April, University of California, Berkeley,.