browserCloud.js

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

David Dias, david.dias@computer.org

Lisbon Tech, University of Lisbon

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's familiar to those used to develop applications for the Cloud, but with resources provided by a community and not by a company or institution.

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

1 Introduction

"An application is peer-to-peer if it aggregates resources at the networks 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 [35]

2 Objectives

3 Related Work

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

3.1 Cloud computing and Open Source Cloud Platforms

3.2 Volunteered resource sharing

3.2.1 Hybrid and Community Clouds

3.2.2 Cycle and Storage Sharing, using Volunteer Computing Systems

3.2.3 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 infrastructures[28], 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 this challenges, several researches have been made in order 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[25], illustrated in Figure 1.

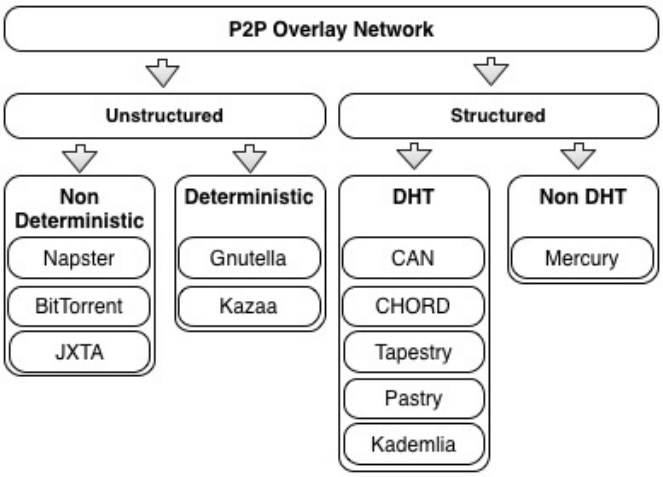

Fig. 1. Different types of P2P Overlay networks organizations

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 it's 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[9] to Gnutella 0.6 [37][31], 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[28] 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[6], 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.

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 [2][19][27], such as SHA-1[7], 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[36], Pastry[33] and Tapestry[45], 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 his 'fingers'(nodes that it knows the whereabouts). Kademlia[23] organizes it's nodes in a balanced binary tree, using XOR as a metric to perform the searches, while CAN[16] 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 gener-

ated doesn't exist, in order to avoid collisions; (ii) the routing table is partitioned across the nodes, increasing the lookup time as it scales.

Table 1, 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 1. Summary of complexity of structured P2P systems

Structured without without Distributed Hash Tables - Mercury[4], a structured P2P network that uses a non DHT model, was design 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'.

3.2.4 Fault Tolerance, Load Balancing, Assurance and Trust - Volunteer resource sharing means that we no longer have our computational infrastructure in a confined and well monitored place, this introducing new challenges that we have to address [21] to maintain the system running with the minimum service quality, this issues can be: scalability, fault tolerance, persistence, availability

and security[43] 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.

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[**?**][32], 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[8], 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.

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[5] - 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[29] - 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[30] - 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[20] - It's 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. Gtz and K. Wehrle from the University of Tbingen, 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

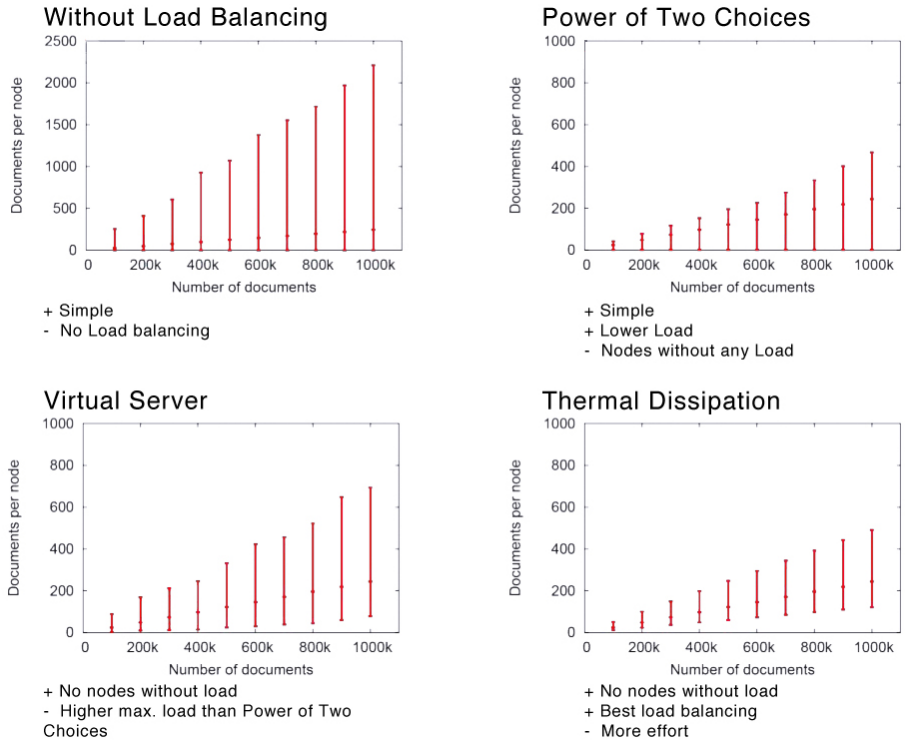

Fig. 2. Load balancing approaches comparison

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 [10], 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[1] 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[11]. To achieve a fair trust sharing system, several metrics for a reputation mechanism have been developed [22], these can be seen in Table 2.

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. Reputation system components and metric

Incentives for sharing resources[15] 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[14][40], that model the traded resources as a currency in which a peer has to trade in order to use the network.

3.3 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:

3.3.1 Web Platform Since the introduction of AJAX[26], 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[13], 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[44], 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"[1] and "Epic Citadel"[2], in which Mozilla used Ecmascripten to port the entire Unreal Engine 3 to JavaScript. In Figure 3, 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[17], 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 4. Since most of the browsers sit behind NAT, there is another server, named 'Turn'(Relay), which

[1] Mozilla, BananaBread, URL: https://developer.mozilla.org/en/demos/detail/bananabread, seen in December 2013
[2] Mozilla, Epic Citadel, URL: http://www.unrealengine.com/html5/, seen in December 2013

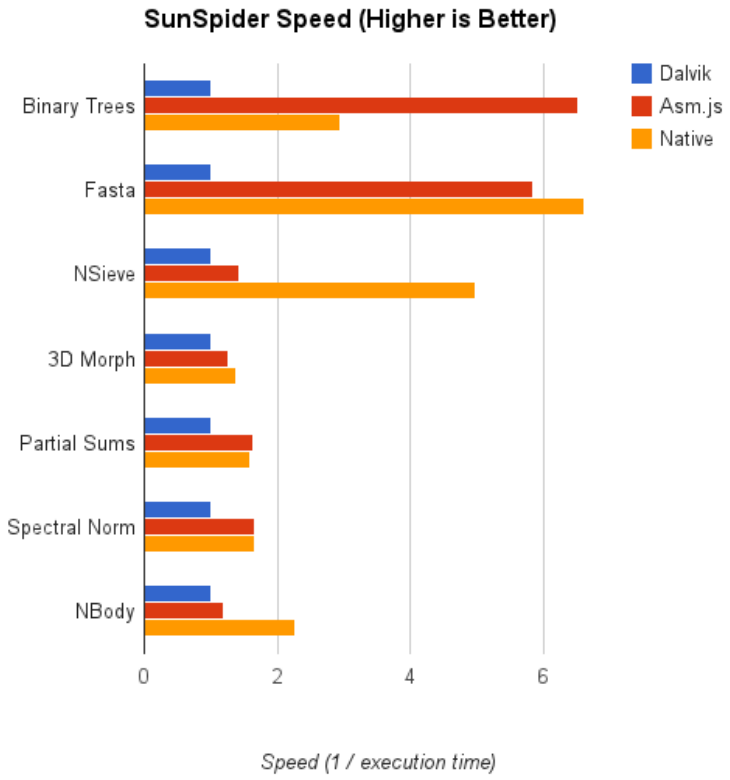

Fig. 3. Dalvik vs. ASM.js vs. Native performance

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[41].

'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[42], which in turn is implemented on top of the LevelDB[18], 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.

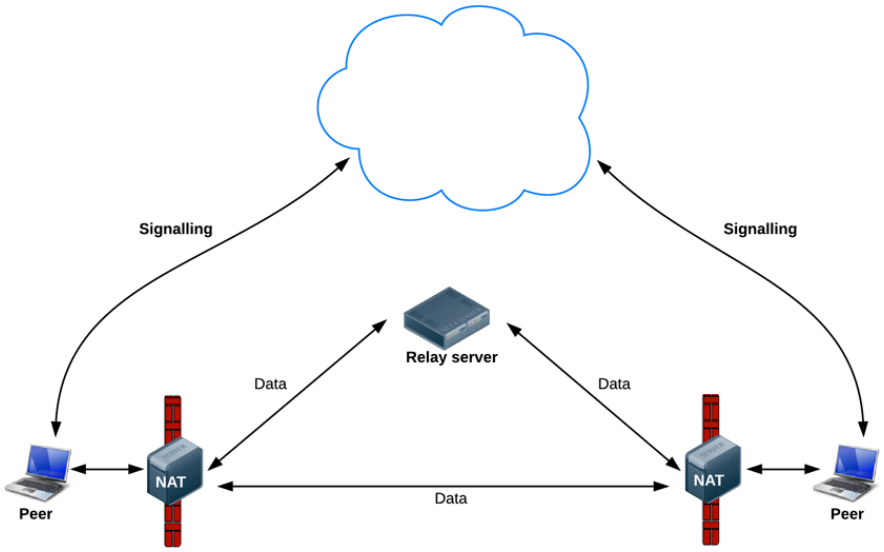

Fig. 4. Example of a WebRTC session initiation

One of the latest improvements being built for the Web Platform is the new HTTP spec, **HTTP2.0**[38], this next standard after HTTP1.1 which aims to improve performance towards a more realtime oriented web, while being retro-compatible 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 6, 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 5, so they can be send in an independent way.
– HTTP headers compression - HTTP2.0 introduces an optimization with headers compression[34] 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.

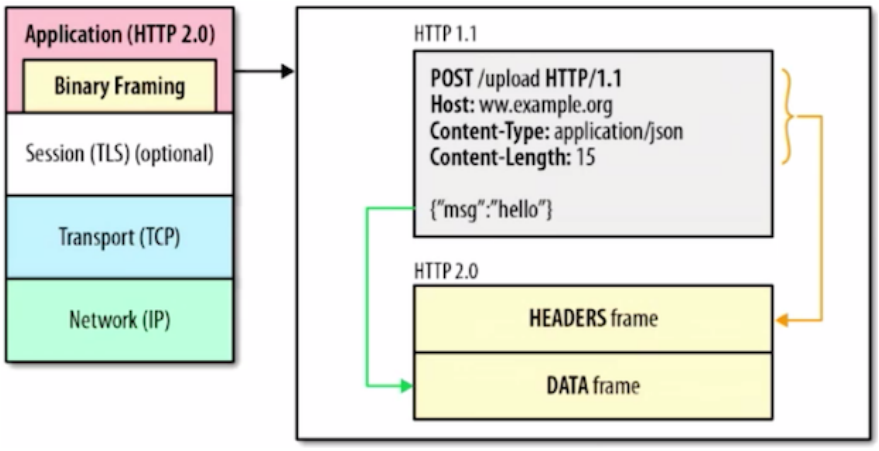

Fig. 5. HTTP2.0 Binary framing

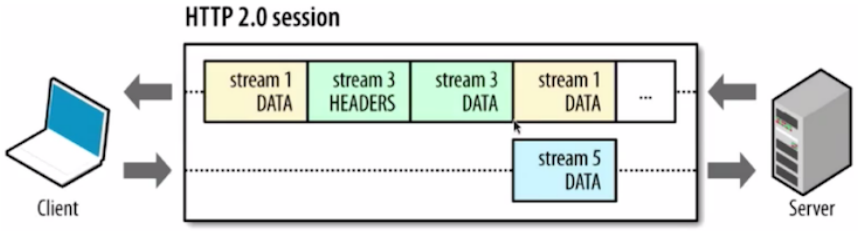

Fig. 6. Example of an HTTP2.0 dataflow

3.3.2 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[24]. 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[12] of browser-based distributed cycle sharing was creating using Node.js as a backend for very intensive Input/Output operations[39], 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[3], where the user gets to contribute with his resources without having to know exactly how, however since it's 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.

4 Analysis and discussion of the architecture

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 right 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 that 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.

We can observe how this subsystems interconnect with each other by observing Figure 7

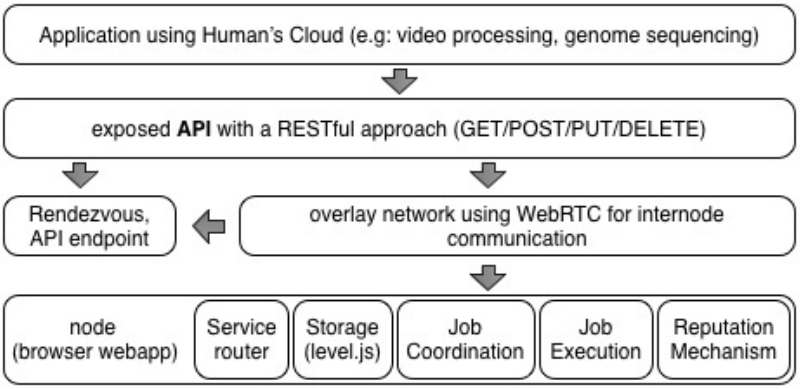

Fig. 7. browserCloud.js Software Stack

4.1 Network Architecture

Network architecture can be seen on Figure 8). Nodes (volunteer computers), are divided into two Chord DHTs with the purpose of separating the nodes with storage 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 have proven to be more trustworthy for staying longer in the network makes the system more robust by 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.

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.

4.2 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 9.

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 algorithm 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.

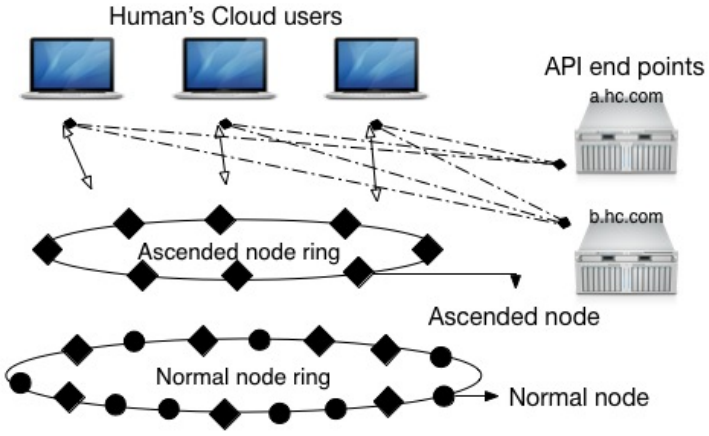

Fig. 8. browserCloud.js network architecture

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.

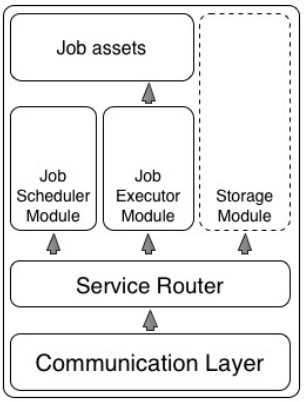

Fig. 9. browserCloud.js Node

4.3 Storage

browserCloud.js storage happens 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 10, 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, so if one of these fails, another is assigned.

Fig. 10. A file partitioned in several chunks, each with its corresponding hashes that correspond to nodeIds

In Figure 11, 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.

Each store 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

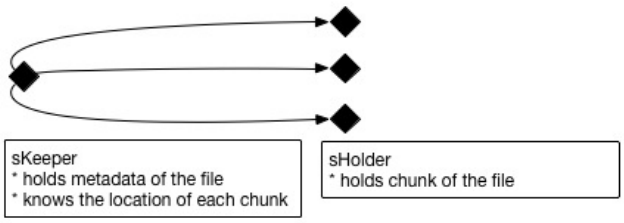

Fig. 11. Representation of the Node responsible for the file(sKeeper) and it's individual chunk holders(sHolders)

are storage as objects in a indexedDB type storage, provided by the leveljs module.

4.4 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 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 for 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

```
replicateJob(); // 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){
    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
}
```

4.5 Reputation Mechanism

The reputation mechanism present will enable the network to identify the nodes that show more availability and 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 completely successfully, this score gets updated and in case it reaches the required level to ascend, the jKeeper that was updating thus score will enable and deploy the remaining features (storage and job schedule module) it needed to join the ascended group.

4.6 Client API and CLI

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,

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. Directories representation inside the network

a filesystem-like interface (directories and objects), where user is identified by a username and it maps to the paths shown in Table 3:

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 4, 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 4. browserCloud.js REST API Draft

We are also including a CLI[3] tool to enable quickly bash scripts for computation jobs and file storage, this CLI uses in the background the API defined in Table 4. 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
- **$ bcput** - Store an object
- **$ bcjob** - Initialize a job

5 Evaluation

The proposed system will be evaluated having into consideration the several dimensions that will define its ability to provide a reliable storage, following the CAP theorem(Consistency , Availability and Partition Tolerant), adding also as criteria, its latency and speed while performing distributed jobs over the stored data.

5.1 Evaluation of the data consistency, availability and partition tolerance

Data consistency will be implicitly granted by having an put/delete system only, since an update to a file means rewriting the file again, a new ID will be created and by doing so, a new sKeeper is elected, avoiding a distributed lock or conflict over the same id of file. Never the less, availability and partition tolerance must be evaluated, having in consideration the hostility of environment, we want to prove that Nodes considered trustworthy(ascended), are in fact enough to guaranteed the necessary availability for any given file at any given time and that the replica number can keep stable without generating too much message overhead. To prove this, resilience will be put into test by varying the churn rate with several usage scenarios, for example, a surge.

This 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.

[3] CLI - Command Line Interface

5.2 Evaluation of latency when storing, fetching and jobs execution

To prove that the system is app development ready, we need to optimize and verify how much performant it behaves, this will enable for it's users to know before hand how it stacks to current cloud systems using a more traditional approach. To evaluate and identify how the system behaves, several tests will be executed during the "in lab" and "field" experiments, varying the load, the complexity of the job, the churn rate and the number of nodes.

6 Conclusions

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 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, in 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 work towards a federated community cloud that will enable its users to share effectively their resources, giving the developers a reliable and efficient way to store and process data for their applications, with an API thats familiar to the centralized Cloud model.

References

1. Youssef Afify. *Access Control in a Peer-to-peer Social Network*. PhD thesis, ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE, 2008.
2. S Bakhtiari and J Pieprzyk. Cryptographic hash functions: A survey." Centre for Computer Security Research, Department of Computer Science. pages 1–26, 1995.
3. a L Barabási, V W Freeh, H Jeong, and J B Brockman. Parasitic computing. *Nature*, 412(6850):894–7, August 2001.
4. Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury : Supporting Scalable Multi-Attribute Range Queries. pages 353–366.
5. John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In M. Frans Kaashoek ; Ion Stoica, editor, *Peer-to-Peer Systems II*, pages 80–88. Springer Berlin Heidelberg, 2003.
6. Bram Cohen. The BitTorrent Protocol Specification, 2009.
7. Cisco D. Eastlake, 3rd Motorola; P. Jones Systems. RFC 3174 US Secure Hash Algorithm 1 (SHA1), 2001.
8. Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : Amazons Highly Available Key-value Store. pages 205–220, 2007.

9. Protocol Definition. The Gnutella Protocol Specification v0 . 4. *Solutions*, pages 1–8, 2003.
10. Y Desmedt and Y Frankel. Threshold cryptosystems. *Advances in Cryptology-CRYPTO'89 ...*, 1990.
11. John R Douceur. The Sybil Attack. In Peter Drusch Druschel@cs.rice.edu and Antony Rowstron Antr@microsoft.com, editors, *Peer-to-Peer Systems*, pages 251–260. Springer Berlin Heidelberg, 2002.
12. Jerzy Duda and W Dubacz. Distributed evolutionary computing system based on web browsers with javascript. *Applied Parallel and Scientific Computing*, 2013.
13. S Ecma. ECMA-262 ECMAScript Language Specification, 2009.
14. Pedro Filipe and Goldschmidt Oliveira. Gridlet Economics : Modelo e Políticas de Gestão de Recursos num Sistema para Partilha de Ciclos Gridlet Economics : Resource Management Models and Policies for Cycle-Sharing Systems Pedro Filipe Goldschmidt Oliveira Dissertação para a obtenção do Grau de. 2011.
15. Philippe Golle, Kevin Leyton-brown, Ilya Mironov, and Mark Lillibridge. Incentives for Sharing in Peer-to-Peer Networks. pages 75–87, 2001.
16. Mark Handley and Richard Karp. A Scalable Content-Addressable Network. In *SIGCOMM '01 Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 21, pages 161–172, 2001.
17. Ian Hickson. WebRTC 1.0: Real-time Communication Between Browsers, 2013.
18. Jeffrey Dean; Sanjay Ghemawat. LevelDB.
19. David Karger, Tom Leightonl, Daniel Lewinl, Eric Lehman, and Rina Panigrahy. 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*, pages 654–663, 1997.
20. David R. Karger and Matthias Ruhl. 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*, page 36, 2004.
21. Georgia Koloniari and Evaggelia Pitoura. Peer-to-Peer Management of XML Data : Issues and Research Challenges. 34(2):6–17, 2005.
22. Sergio Marti and Hector Garcia-molina. Taxonomy of Trust : Categorizing P2P Reputation Systems. *Computer Networks*, (April 2005):472–484, March 2006.
23. Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric.
24. Juan-j Merelo, Antonio Mora-garcía, Juan Lupión, and Fernando Tricas. Browser-based Distributed Evolutionary Computation : Performance and Scaling Behavior Categories and Subject Descriptors. pages 2851–2858, 2007.
25. Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu, and J I M Pruyne. Peer-to-Peer Computing. Technical report, 2003.
26. L.D. Paulson. Building rich web applications with Ajax. *Computer*, 38(10):14–17, October 2005.
27. Bart Preneel. The State of Cryptographic Hash Functions. pages 158–182, 1999.
28. Rajiv Ranjan, Aaron Harwood, and Rajkumar Buyya. A study on peer-to-peer based discovery of grid resource information. *..., Australia, Technical Report GRIDS ...*, pages 1–36, 2006.
29. Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, and Richard Karp. Load Balancing in Structured P2P Systems. 0225660:68–79, 2003.

30. S. Rieche, L. Petrak, and K. Wehrle. A thermal-dissipation-based approach for balancing data load in distributed hash tables. *29th Annual IEEE International Conference on Local Computer Networks*, pages 15–23.
31. M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. *Proceedings First International Conference on Peer-to-Peer Computing*, pages 99–100, 2002.
32. Antony Rowstron and Peter Druschel. PAST A large-scale , persistent peer-to-peer storage utility. *Proceedings of the eighteenth ACM symposium on Operating systems principles - SOSP '01*, pages 75–80, 2001.
33. Antony Rowstron and Peter Druschel. Pastry : Scalable , Decentralized Object Location , and Routing for Large-Scale Peer-to-Peer Systems. In Rachid Guerraoui, editor, *Middleware 2001*, pages 329–350. Springer Berlin Heidelberg, 2001.
34. H Ruellan and R. Peon. HPACK-Header Compression for HTTP/2.0. *draft-ietf-httpbis-header-compression-04 (work in ...*, (c):1–57, 2013.
35. C. Shirky. Clay shirkys writings about the internet. In http://www.shirky.com/.
36. Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, Hari Balakrishnan Ý, and Hari Balakrishnan. Chord : A Scalable Peer-to-peer Lookup Service for Internet. pages 149–160, 2001.
37. R. Manfredi T. Klingberg. RFC - Gnutella 0.6 Protocol Specification, 2002.
38. M Thomson and A Melnikov. Hypertext Transfer Protocol version 2.0 draft-ietf-httpbis-http2-09. 2013.
39. Stefan Tilkov and Steve Vinoski Verivue. Node.js : Using JavaScript to Build High-Performance Network Programs. 2010.
40. Vivek Vishnumurthy, Sangeeth Chandrakumar, and G Emin. Karma: A secure economic framework for peer-to-peer resource sharing. 2003.
41. Christian Vogt, MJ Werner, and TC Schmidt. Leveraging WebRTC for P2P Content Distribution in Web Browsers. *21st IEEE Intern. Conf. ...*, 2013.
42. W3C. Indexed Database, 2013.
43. Dan S Wallach. A Survey of Peer-to-Peer Security Issues. In Mitsuhiro Okada Mitsu@abelard.flet.keio.ac.jp, editor, *Software Security Theories and Systems*, pages 42–57. 2003.
44. Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011.
45. Ben Y Zhao, John Kubiatowicz, and Anthony D Joseph. Tapestry : An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report April, University of California, Berkeley,, 2001.