browserCloud.js - A distributed computing fabric powered by a P2P Overlay Network on top of the Web Platform

David Dias
INESC-ID Lisboa
ULisboa / Instituto Superior Técnico
mail@daviddias.me

Luís Veiga
INESC-ID Lisboa
ULisboa / Instituto Superior Técnico
luis.veiga@inesc-id.pt

Abstract

Grid Computing fundamental basis is to use idle resources in order to maximize their efficiency. This approach quickly grew into non Grid environments, leveraging volunteered shared resources and giving the birth of Public Computing. Today, we face the challenge of how to create a simple and effective way for people to participate in such community efforts and even more importantly, how to enable developers and researchers to use and provide these resources for their applications. This paper explores and proposes a novel way to enable end user machines to communicate using bleeding edge P2P Web technologies such as WebRTC.

ACM Reference format:
David Dias and Luís Veiga. 2018. browserCloud.js - A distributed computing fabric powered by a P2P Overlay Network on top of the Web Platform. In *Proceedings of ACM SAC Conference, Pau, France, April 9-13, 2018 (SAC'18),* 10 pages.
https://doi.org/10.1145/3167132.3167366

1 Introduction

User generated data has been growing at a large pace with the proliferation of social networks, search engines, Internet of Things. All of these applications require huge amounts of storage and computing power to process the harvested user data useful. Cloud Computing tries to answer the demand for storage and computing power, revolutionizing the landscape with key advantages to developers/users over pre-existing computing paradigms, the main reasons are:

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

Grid computing had offered before a solution for high CPU bound computations, however it had 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" transforming computing into a utility.

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

There is a large untapped source of volunteered shared resources that can be used as a cheaper alternative to large computing platforms.

We have identified several issues with current solutions, these are:

- Typical resource sharing networks do not offer an interface for a user to act as a consumer and contributor at the same time.
- Interoperability is not a priority.
- There is a high level of entrance cost for a user to contribute to one resource sharing network.
- Load balancing strategies for volunteer computing networks are based on centralized control, often not using the resources available efficiently and effectively.
- Centralized Computing platforms have scalability problems as the network and resource usage grows.

To solve these issues, we propose a new approach that abandons the classic centralized Cloud Computing paradigm to a fully decentralized architecture, federating freely ad-hoc distributed and heterogeneous resources, with direct

SAC'18, April 9-13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167366

resource usage and progress report. This work aims to address extending the Web Platform with technologies such as WebRTC, Emscripten, Javascript and IndexedDB to create a structured peer-to-peer overlay network, federating standard Web Browsers into a geo-distributed cloud infrastructure.[1]

We start by presenting in Section. 2, the state of the art for the technologies and areas of study relevant for the proposed work, which are: Cloud computing and Open Source Cloud Platforms (at 2.1), Volunteered Resource Sharing (at 2.2) and Resource sharing using the Web platform (at 2.3). In Section 3 we present the architecture and respective software stack, moving to Implementation details in Section 4 and system evaluation present on Section 5. Section 6 concludes the paper.

2 Related Work

The lack of applications portability in Cloud Computing has been identified as a major issue by growing companies, known as 'lock-in syndrome', 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.

OpenStack is an open source cloud computing platform initiative founded by Rackspace Hosting and NASA. It has grown to be *de facto* standard of massively scalable open source cloud operating systems. 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'. Other solutions are:

- **Eucalyptus** - 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.
- **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.
- **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.

One interesting aspect that we want to remark is that the more recent solutions look for interoperability through abstraction and not by enforcing a specif stack.

Another trend in Cloud Computing are the Community Clouds, where computing resources might be shared and traded through the available network or through a Community Network, where individuals can build their own data links, this is also known as "bottom-up networking". CONFINE [6] 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,[2] a federation between guifi.net, AWMN and FunkFeuer (community network from Vienna and Graz, Austria).

Volunteered resource sharing networks enable the cooperation between individuals to solve higher degree computational problems, by sharing idle resources that otherwise would be wasted. 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 [3]. 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.

In order to increase the flexibility of the jobs executed by the volunteered resources, the concept of Gridlet [2] [11] appears as an unit of workload, combining the data with the logic needed to perform the computation in one package.

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, 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, using JavaScript [4], 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. **HTTP2.0** [9] also plays a important role towards this goal with differential updates, binary framing and prioritization of data frames.
- Close-to-native performance for highly CPU-bound jobs. This is achieve through **Emscripten** [12], 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.
- Peer-to-peer interconnectivity with **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.
- Scalable storage and fast indexing with **'level.js'**, 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

[1] The code for this project is MIT Licensed and available at: https://github.com/diasdavid/webrtc-explorer

[2] http://community-lab.org/

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.

Previous attempts at cycle sharing through web platform: The first research on browser-based distributed cycle sharing was performed by Juan-J. Merelo et. al., which introduced a Distributed Computation on Ruby on Rails framework [5]. 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 send back the results. In order to increase the performance of this system, a new system [3] of browser-based distributed cycle sharing was created using Node.js as a backend for very intensive Input/Output operations [10], 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 [1], 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.

Analysis and discussion:

The concept of Gridlet, akin to those seen as well in state-of-the-art databases such as Joyent's Manta,[3] which bring the computation to/with the data, reducing the possibility of a network bottleneck and increases the flexibility to use the platform for new type of jobs, will very important. To enable this new Cloud platform on using browsers, it is important to understand how to elastically scale storage and job execution, as in [7], 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, in 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.

3 Architecture

browserCloud.js proposes a mechanism to find, gather and utilize idle resources through a P2P overlay network. Participants will be joining and connecting to each other through a rendezvous point, as represented in Figure 1. Any Peer that joins the network can submit a job which will be partitioned and distributed across peers available. Once a Peer completes its task, it is sends back the results to the issue. The user does not need to understand how the network is organized or which peers it is directly connected too, that complexity is abstracted by browserCloud.js.

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

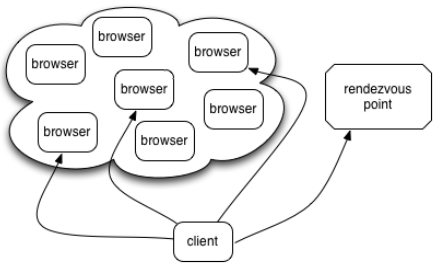

Figure 1. browserCloud.js Overview

browserCloud.js was architectured to meet the following requirements:

- **Membership management** - The system has to enable peers to join and leave a current network of browserCloud.js peers or a subset of it.
- **Message routing** - Messages are be routed between peers, having each peer knowing a subset of the network, guaranteeing in full coverage in this manner.
- **Job scheduling and results aggregation** - The discovery of computational resources must be performed using a distributed approach, peers interact between each other to send tasks and retrieve the results for the peer executing the job.
- **Support dynamic runtime** - Provide flexibility for jobs being executed.
- **Reduced entrance cost to enable greater adoption** - Simple APIs design, abstracting the complexity in favor of greater extendability.
- **Enable integration and compliance tests** - Automate the process of verifying browserCloudjs integrity and functionality.

3.1 Distributed Architecture

The overview of the distributed architecture can be seen in Figure 2.

There are two different kinds of actors in the system:

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

In a browserCloud.js infrastructure, we have three main interaction patterns. The first being when a peer joins or leaves the network, membership management. In traditional P2P network, we could simply exchange IP:Port pairs, but in a P2P browser network, a RTCPeerConnection has to be established and kept alive, meaning that an handshaking protocol must be performed. The second pattern is message

[3] http://www.joyent.com/products/manta

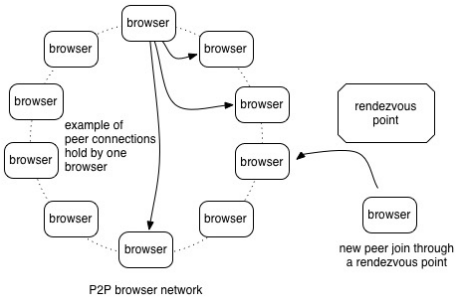

Figure 2. browserCloud.js Distributed Architecture Overview

routing between peers, this has been designed with inspiration on the Chord[8],routing algorithm, studied on the related work. The third interaction demonstrates how to levarage the computer cycles available in the network to process CPU bound jobs.

Peer joins and leaves A peer join compromisses of the following steps:

- **1 - Registration** - When a peer is ready to join the network, it performs the registration action to the custom browserCloud.js signalling server, the server replies with a confirmation and a unique ID for this peer to occupy in the network. This enables the signalling server, which holds the meta data of the current state in the network, to pick the ID in the ID interval that might be less occupied. We can observe this interaction in Figure 3.
- **2 - New peer available** - As peers join the network, the other peers get notified to establish or update their routing tables, keeping the message routing efficient (explained in the next subsection). For each peer join, a notification with a finger update can be sent to one or more peers present, as seen in Figure 4.
- **3 - Connection establishment between two peers** - In order to establish a connection between two peers, once there is an interest for these to connect, for e.g, in the case of a finger update event. There are two substeps, the first being the SDP offer creation through a step called "hole punching". A browser uses the WebRTC APIs to traverse through NAT to obtain its public IP, which is crucial information when two browsers need to establish a direction connection, Figure 5. The second step is the exchange of these SDP offers between browsers and that has to be performed by a centralized service; in browserCloud.js we developed a custom signalling server that handles that part, as seen in Figure 6.

A peer leave is a simpler and organic process, once a peer leaves the network, the RTCPeerConnections objects are closed and destroyed, notifying automatically the peers that have to update their routing tables.

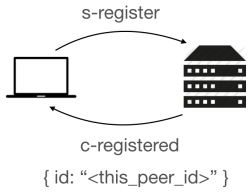

Figure 3. Registration of a peer, signaling itself as available to be part of the P2P network

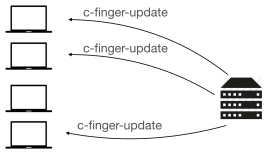

Figure 4. A peer is notified to update his finger table

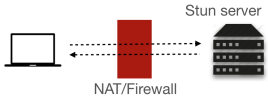

Figure 5. Hole punching through NAT to obtain a public IP and create a SDP offer

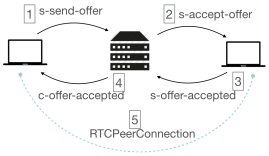

Figure 6. Establishment of a RTCPeerConnection through the custom Signalling Server

Message routing For message routing, we designed an adaptation of the Chord routing algorithm, a P2P Structured Overlay network, a DHT studied in the related work, with the goal of keeping an efficient routing and resource lookup with the increase of the number of peers in the network.

The ID namespace available in our DHT consists of 48 bit IDs (Figure 7). This decision was made due to the fact that Javascript only supports 53 bit numbers, to support a greater variety of IDs, we would have to resort to a big integer third party library, adding unnecessary consumption of computing resources. However, for demonstration purposes, we will explain using a 3 bit ID namespace.

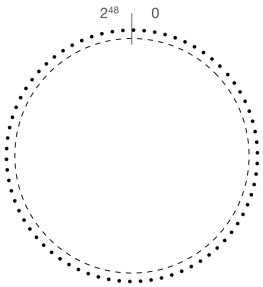

Figure 7. How the ID namespace is visualized in the DHT

In Figure 8, we have a DHT composed of 4 different peers, with IDs 0, 1, 3 and 6. Each one of these peers will be responsible for a segment of the DHT. In other words, what this means is that every message that is destined to their segment will be delivered to respective peer responsible. A peer is responsible for a segment of IDs greater than the peer that is its predecessor and lesser or equal than its own ID, represented in Figure 9. When a peer enters the network, its ID is generated through a crop of a SHA-1 hash from a random generated number, creating a natural uniform distribution.

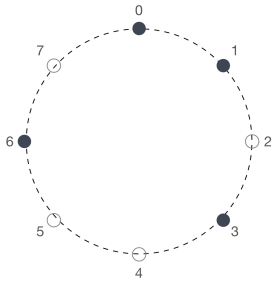

Figure 8. Example of a DHT with 4 peers for case study

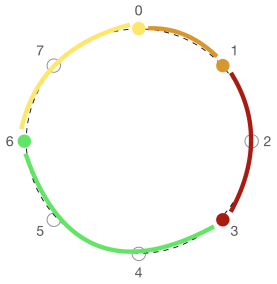

Figure 9. Responsability interval for each Peer

In order for messages to find its correct destination, each peer has to know at minimum the peer that is next to it on the DHT, also called "successor" (Figure 10). Messages will be forwarded until they reach the peer which compromisses the responsability of being responsible for that message ID.

However, as specified earlier in the document, we want to achieve a good and stable efficiency when it comes to routing

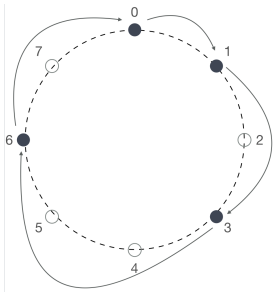

Figure 10. Minimum number of connections for messages to be routed properly

messages inside the DHT as the network grows. To achieve that, we introduce fingers in our peers, as we mentioned earlier. A finger is a direct connection to another peer in the network (Figure 11) that was picked following a specific distribution, each peer will have 1 to N fingers, where N is the number of bits of the IDs (for this example, N = 3). A finger is always the peer responsible for the "start" value of the interval (see Figure 12 for reference and formula) and a message will be routed to that finger if it falls inside the interval.

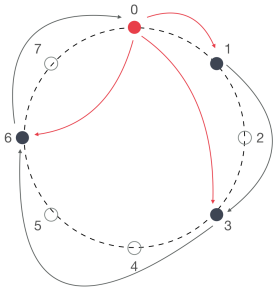

Figure 11. Example of peer with ID = 0 fingers

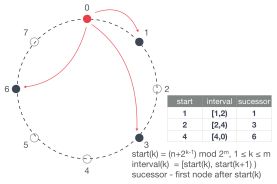

Figure 12. Peer with ID=0 finger table

The number of fingers and the fingers we use for a given instance of browserCloud.js are configurable. The reason behind this design decision was that RTCPeerConnections have a significant memory cost, so we have to be considerate in the number of data channels we keep open. In order to give greater flexibility to the developer, we allow the option of picking how many rows of the finger table will be filled by the developer creating a browserCloud.js application. This is

also perfect since WebRTC is still a working draft and there might be good developments in resource consumption.

3.2 Resource Management

Leveraging the browser's dynamic runtime was a feature we pursue from the beginning of the design for browserCloud.js.

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

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

3.3 Architecture of the Software stack

We can observe a overview of this architecture in Figure 13.

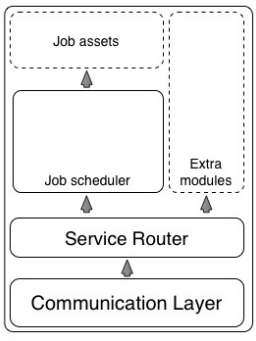

Figure 13. Software layers at the peer level

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

The Service router establishes a protocol for modules like the job scheduler to interact with the network of peers, it uses an event driven model, where modules can register listeners to events that happen on the network or send messages.

The Job scheduler benefits the API of the Service router to implement its logic.

3.4 API design

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

Peer join

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

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

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

var peer = new Explorer(config);

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

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

peer.register();
```

Listen for messages

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

Execute a job

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

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

// Make this browser available to execute tasks and also prepared to
// issue jobs to the network
browserProcess.participate(config);

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

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

3.5 Testing framework requirement

When it comes to testing to test a decentralized browser app or library, the focus stops being how a browser implements a specific behaviour, but how the decentralized network handles node joins and leaves, and whether nodes are effectively communicating between each other. For this scenario, we have a specific set of requirements for the framework, these are:

- Have N browsers available, where 1<=N<=virtually unlimited
- Serve a custom web page for the desired test

- Instruct browsers on demand
- Gather information and evaluate the state as a whole

In order to evaluate that a browserCloudjs instance is working as desired, we have designed the following workflow, which can also be seen in Figure 14:

- 1 - A Web Server is started by the Control Center, this endpoint will be serving the necessary static assets (e,g .html, .css and .js files) that will contain our browserCloudjs module, so that when a browser loads the page through this endpoints, has a way to run browserCloudjs.
- 2 - The number of required browsers for the test being executed, are spawned. In our example in Figure 14, we see that number is 2.
- 3 - Once the browser loads the web page containing the browserCloudjs module, the Control Center starts sending commands to each browser to execute.
- 4 - Since the messages and data transferred between browsers happens in a side channel, browsers report to the Control Center which events were triggered.
- 5 - Once all the commands were executed, the Control Center assesses the order in which these events happened and asserts if the behavior was the expected.

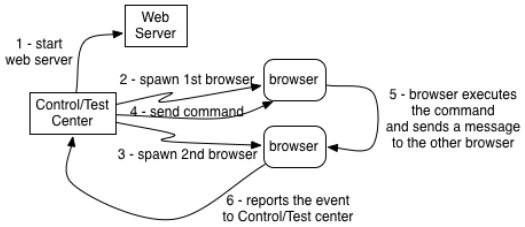

Figure 14. Normal execution of a browserCloudjs test

browserCloudjs tests are not linear, a message can be routed between any two browsers through several combinations, depending on the current size of the network and the respective IDs of those browsers, which will influence how their finger table looks like.

In Figure 15, we have an example of two browsers communicating between each other. We can see that some of the browsers between them will have the responsibility to forward the message, while others, will be idle.

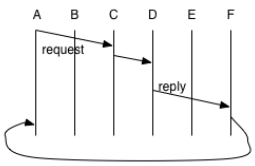

Figure 15. Possible timeline of events for an request from browser A to browser D and the consequent reply

4 Implementation Details

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

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

Essentially it is broken down into 4 components:

- channel manager - responsible to leverage the websockets connection with the signalling server and abstracts the necessary work to open new RTCPeerConnections with other peers.
- finger table manager - where the information about a specific peer finger table lives.
- router - the routing logic to deliver the messages on the most efficient way. It uses the finger table manager to understand what is the most efficient way to rout messages.
- interface - developer exposed interface.

The Signalling Server offers a HTTP and Web Sockets API and serves as a rendezvous point for SDP data exchange between browsers so they can establish a RTCPeerConnection.

The testing framework implementation, which we named "piri-piri", encapsulates the necessary logic described on section 3.5.

Using D3JS[4], we have developed an application that grabs the state of the browserCloud.js network and shows a live graphical representation, as seen on Figure 16, where each node is represented by a dot and its ID and the arcs being the connections established between the nodes in the network.

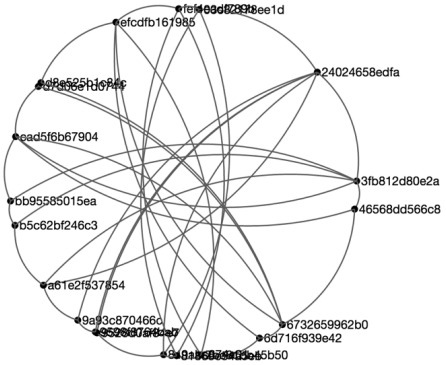

Figure 16. Visualization of a browserCloud.js network

To perform the parallel CPU bound tests, we have developed a module that works in Node.js and in the browser to perform Ray Tracing Tasks.

[4]http://d3js.org

5 Evaluation

In this chapter, we go through our qualitative and quantitative evaluation of browserCloudjs system, comparing it to our initial goals and expectations.

5.1 Qualitative assessment

In a qualitative perspective, browserCloudjs performs successfully the following:

- Efficient resource discovery through peer-to-peer routing over a structured overlay network, using a DHT.
- Remove the need for centralized indexes or points of control. There is still a need of a rendezvous point to enable new peer joins, however the data transmited, computed and stored inside the network is peers responsability.
- Enable every machine equiped with a WebRTC enabled browser to be part of a browserCloudjs instance. In 2013, the number of WebRTC capable devices already exceed one billion[5]
- Enable peers to both participate and contribute to a job and at the same time submiting and requesting the network to process their own.
- browserCloudjs' Job Scheduler is job agnostic, this means that different types of jobs can be executed on demand without any previous configuration or preparation.
- browserCloudjs solves the decentralized communication problem between browsers in a scalable way, giving the opportunity for new scenarios to be developed on top of it through its modular and pluggable approach.

5.2 Quantitative assessment

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

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

The setup used during the tests was a system running Chrome version 39 on a Intel Processor Code i7 2.3Ghz with 16Gb of RAM. The STUN server used was provided by Google.

Following our motivation to build browserCloudjs in the first place, that is, to provide a way to take advantage of the volunteer computing paradigm, using the idle resources available on user machines, leveraging the capabilities that offered to us by the Web Platform, we set ouselves with some goals to prove if our solution is viable, through:

- Measuring the time lapsed for a single browser to compute a CPU bound job and several browsers to compute that same job, but in parallel.
- Measuring the RTT time between any of two browsers in the network and evaluate as routing efficiency evolves with the increase in number of browser
- Assessing if there are significant speedups

We have performed tests in order to assess:

- Time elapsed during a distributed ray-tracing job, checking for how it changed when we increased the number of browsers and the level of granularity in which we divided the job. Seen in Figures 17 , 18 , 19 and 20.
- How much time each ray-tracing task takes. Seen in Figure 21.
- What is the average round trip time between any of two browsers in a 10 browser network. Observed in Figure 22.

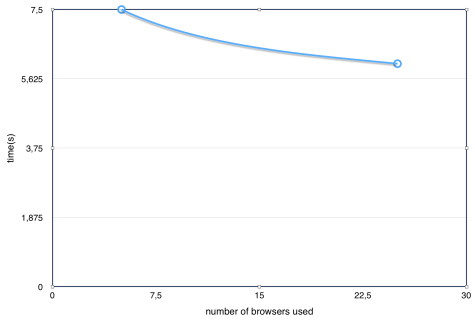

Figure 17. Time elapsed on a ray-tracing job divided in 25 computing units

The standard ray-tracing job using the algorithm developed, running in a single browser takes as median 23610.434ms to complete. As we can see in Figures 18 and 20, our system excels in delivering faster results by dividing the job up to 2500 computational units (or tasks) and requesting from the browsers available in the network to compute those (i.e., a rectangle of the resulting output image). This is expected as ray-tracing is a known case of an embarrassingly parallel application.

One fact interesting to note is that we obtained much better results by reducing the granularity of which ray-tracing job was divided into, as we can see on Figures 17 and 19. This happens due to two factors: a) the first is that since we

[5]Google I/O presentation - https://bloggeek.me/webrtc-next-billion/

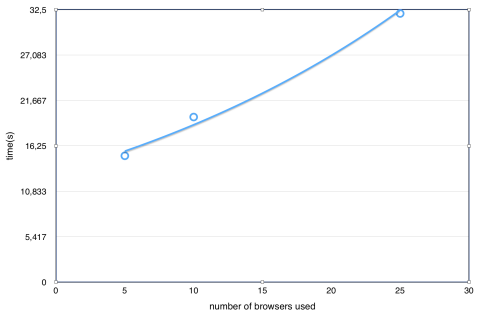

Figure 18. Time elapsed on a ray-tracing job divided in 2500 computing units

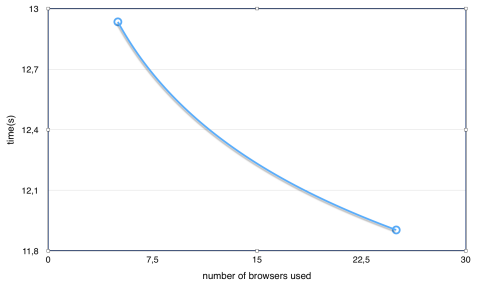

Figure 19. Time elapsed on a ray-tracing job divided in 25 computing units (with induced web RTT delay)

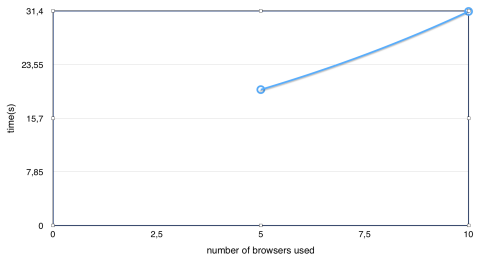

Figure 20. Time elapsed on a ray-tracing job divided in 2500 computing units (with induced web RTT delay)

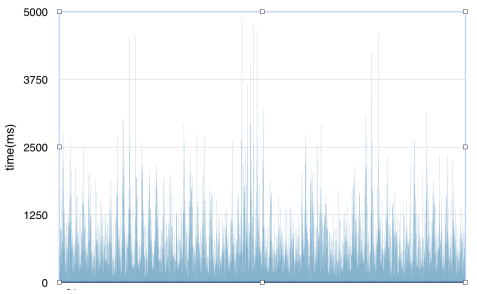

Figure 21. Average time for a task execution for a job fragmented in 2500 computing units

have a lower number of tasks to be run by other browsers, we reduce the message routing overhead between nodes

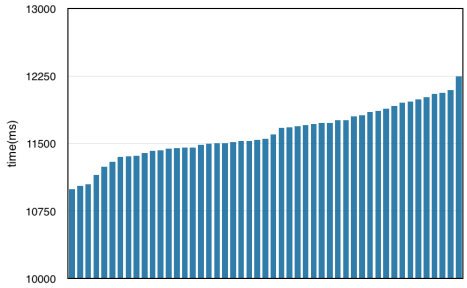

Figure 22. Average Round Trip Time between an two nodes in a 10 browser network

(i.e., resource discovery does not take so long); b) the second factor is that since this system was tested using a single machine and a networked simulated delay. When the number of tasks is too large, the workers in the browser are in fact competing for CPU resources (to execute tasks and to forward messages among them). This creates a scenario, where more nodes/workers actually make the system slower, since this is a much more strict and resource constrained scenario, than a real example with browsers executing in different machines.

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

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

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

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

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

In a real world scenario, with more browsers from more machines, the total execution time (makespan) of a ray-tracing job would be closer to that described by Equation 1. It would be influenced by the maximum round trip time between any two nodes (so that the information for every task can be received and processed by another node), plus the time it would take to execute the most of CPU intensive task (e.g., the rectangle in the frame that has the more complex geometry and light reflections to be processed). Figures 21 and 22 show what is the average task length and RTT between any two nodes, being the maximum for the first 61ms and the second 11174ms, creating a total of 11235ms (or 11.296s overall). This is a significant increase of efficiency, comparing to the sequential execution and also to the previous single-machine experiments.

It is important to note that in Figure 21, we can see several task execution lengths due to the complexity of each task, with more or less light reflections. With this microbenchmark we see that the execution time of each task, without any resource contention (1 node = 1 browser per machine), the task duration has an even lower upper bound (lower than 5s). This would entail the upper bound of total task execution time to be under 5061 ms (around just 5s), with a theoretical speedup of about 4.6 times (take into account that we would be using 2500 nodes then, so speedups are not perfectly linear due to communication overhead, as expected).

As we have discussed in the previous subsections, we did managed to reach significant speedup between 2 and close to 5 times for our experiment, using only volunteer resources, that is a reduction between 50% and 76%.

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

6 Conclusions

We end this report by making an overview and summing up all the primary aspects, from proposed work, contributions, state of the art, definition of the architecture, implementation of the respective architecture and evaluation, moving to what were the major breakthroughs and ending with concluding remarks and future work.

browserCloud.js was an exercise to create a distributed computing fabric, enabling its users to effectively share their resources, giving developers a reliable and efficient way to store and process data for their applications.

The system was designed to be native to the Web Platform, the most ubiquitous runtime. There were two reasons behind this decision, the first being longevity. The Web Platform, even though it is quite popular, is still an emerging platform, meaning that our assumptions of ubiquity will prevail. The second reason was developer adoption. JavaScript is the "lingua franca" of the web, meaning that it will be common for a developer to know how to code with JavaScript's APIs. What is more, since browserCloudjs was built in JavaScript, developers will know inherently how to use the platform.

Going after a decentralized model was also something we saw as a potential key factor for the browserCloudjs platform, structured peer-2-peer networks scale well with demand, while centralized networks have a number of significant challenges once a certain threshold of users is reached. WebRTC, the technology enabling browsers to communicate in a peer-2-peer way, is in great part responsible for this platform success.

With browserCloudjs, we achieved in bulk mainly two great milestones:

- **The first browser based DHT** - browserCloudjs offers for the first time in browser history a fully functional DHT, performing resource decentralized resource discovery on the browser.
- **The first peer-2-peer browser computing platform** - the research of using browsers to leverage the idle computer cycles have been in the literature for a while, however, always following the centralized/BOINC model. browserCloudjs offers the first peer-2-peer browser computing framework with proven speedups.

We have found this paper to be a source of hard work and enthusiasm, a great opportunity to research and interact with bleeding edge technologies and also interact with the developer communities that are pushing the web forward.

References

[1] Albert-Laszlo Barabasi, Vincent W Freeh, Hawoong Jeong, and Jay B Brockman. Parasitic computing. *Nature*, 412(6850):894–897, 2001.

[2] Fernando Costa, Joao Nuno Silva, Luís Veiga, and Paulo Ferreira. Large-scale volunteer computing over the internet. *Journal of Internet Services and Applications*, 3(3):329–346, 2012.

[3] Jerzy Duda and W Dtubacz. Distributed evolutionary computing system based on web browsers with javascript. *Applied Parallel and Scientific Computing*, 2013.

[4] S Ecma. ECMA-262 ECMAScript Language Specification, 2009.

[5] 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.

[6] Leandro Navarro. Experimental research on community networks. Technical report, 2012.

[7] João Nuno Silva, Luís Veiga, and Paulo Ferreira. A2ha - automatic and adaptive host allocation in utility computing for bag-of-tasks. *Journal of Internet Services and Applications*, 2(2):171–185, 2011.

[8] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, Hari Balakrishnan Y, and Hari Balakrishnan. Chord : A Scalable Peer-to-peer Lookup Service for Internet. pages 149–160, 2001.

[9] M Thomson and A Melnikov. Hypertext Transfer Protocol version 2.0 draft-ietf-httpbis-http2-09. 2013.

[10] Stefan Tilkov and Steve Vinoski Verivue. Node.js : Using JavaScript to Build High-Performance Network Programs. 2010.

[11] Luís Veiga, Rodrigo Rodrigues, and Paulo Ferreira. Gigi: An ocean of gridlets on a" grid-for-the-masses". In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 783–788. IEEE, 2007.

[12] 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.