university of
groningen

faculty of science
and engineering

Highly Distributed In-Browser Computing

Bachelor's Thesis

July 2019

Authors
George Argyrousis
Tolga Parlan

Primary Supervisor
Alexander Lazovik

Secondary Supervisor
Frank Blaauw

CONTENTS

ABSTRACT

Browser based distributed systems constitute a promising field which has been subject to exploration for scientific and commercial usage. We inspect an array of such projects and identify a lack of successful and popular projects, in stark contrast to a large group of volunteer computing projects such as BOINC which can attract significant user-bases. This paper reasons about the social and technical problems commonly encountered in previous projects, why they fail to tap into broad computing resources of the internet community and discusses solutions which are implemented into a novel framework for browser based distributed computing.

We created an easy to use Javascript based Distributed System which runs on any Browser. We discuss how it can be integrated with any available JavaScript codebase and algorithm in order to shift heavy computations to volunteer and commercial users. Features which can play an important role for an internet facing system such as Fault Tolerance, Load-Balancing, Multi-Threading and Streams are explained and the implementations are discussed in detail.

We gather data to evaluate the scalability of the system and show our evaluation findings in terms of how our system scales according to various loads of computational intensity as well as different volumes of data being distributed. The findings suggest that our system can scale successfully under certain assumptions and conditions but faces limitations as the connected clients increase. Further research is warranted for a full understanding of the current limitations and for a path to future development.

1 INTRODUCTION

Distributing a large pool of computing tasks to many computers across the internet and letting them run the calculations is an idea which goes back to successful projects such as BOINC [2] from 2002, distributed.net[1] from 1997 and Great Internet Mersenne Prime Search[2] from 1996. Such projects have been usually termed Volunteer Computing since the participants would actively volunteer to run tasks on their machines, usually via signing up on a website and downloading a purpose-made program. Later sections present the volunteer structures of such projects, the incentive structures they employ, and their limitations.

The modern web relies increasingly on Javascript, which is used almost ubiquitously in the client-side of any website and often has the purpose of making websites more reactive to user actions by offloading some application logic to the client's browser. This approach is increasingly popular due to user computing devices getting ever more powerful, and serious innovations in Javascript engines such as Chrome V8 Engine[3] and related technologies making it ever more sensible to invest development time into shifting computational tasks to the client-side. This shift to running more computations on the browsers and developing efficient Javascript environments has been exposing a considerable amount of computing power to the website owners in the form of their users' devices since all the users visiting a page run code served to their browsers, and the exposed computing power can grow to serious proportions for a relatively popular website. Javascript code running on modern browsers are broadly considered safer than ever due to extensive sandboxing[4] and formally there is no limitation on what tasks client-side scripts can work on, opening up the opportunity to use client devices for large calculations which would normally require a large grid setup.

In the past, multiple distributed systems with the aim to utilize the exposed computing power from browsers with diverse aims have been proposed and implemented. Such systems were typically **Browser Based Volunteer Computing** projects [11], attempting to harness computing cycles from volunteer devices who would browse on a particular website to run the served code and support a large computational task. However none of the proposed distributed systems, some of which will be discussed in detail as well, has achieved wide-spread popularity. Later in the paper, we discuss possible causes for the lack

1 `distributed.net`
2 `https://www.mersenne.org/`
3 `https://v8.dev/`
4 `https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md`

of general use of distributed systems working over browsers, and introduce our improvements in the form of an open source Javascript package to build browser distributed computing architectures upon.

1.1 HIGHLY DISTRIBUTED IN-BROWSER COMPUTING

Our project aims to develop a powerful, simple and highly customizable open-source tool which can be used as a basis for Browser Based Volunteer Computing as well Gray Computing projects to unearth the aforementioned underused computing potential of billions of devices that access the internet via web browsers. Our solution intends to be more easily adaptable to different use cases than the existing projects and therefore be a better choice for the unpredictable and shifting use cases of real life agents, while being scalable enough for real life applications. We envision that harnessing this processing power, with the right tools, can create a serious alternative revenue stream for many websites, nullifying or decreasing the need for internet advertisement, while serving useful scientific, social and commercial purposes.

We have implemented an npm package[5] which has an easy to use and highly customizable interface for programmers. The package is intended for trivial integration with any NodeJS[6] based server, and work beside any other server implementation. We believe that our unopinionated approach which allows other implementers to run their own tasks in their own way with little need to adapt the architecture or logic of their codebase to accommodate our package is a serious advantage over other similar projects, and can help to resolve the relative lack of real life popularity of past browser based distributed computing projects. A detailed discussion of our principles can be found in the Architecture chapter.

The source code for the in-browser computation platform is released as open source software[7] and licensed under the MIT license.

1.2 PAPER STRUCTURE

The remainder of the document will be structured as follows. In Chapter 2 we give a summary of related work, explaining how we build upon the design and technology choices of projects similar to ours. The chapter further discusses survey papers which have examined many past projects, and goes on to explain in which ways we have attempted to differ in our project in order to produce a solution that can be adopted easily by people for diverse tasks. In Chapter 3 we summarise our code architecture, and talk about the main components of our program in more detail. Chapter 4 focuses on evaluating different types of tasks and their performance on our system. Finally in Chapter 5 we present our conclusions and in Chapter 6 we present ideas that can be considered to improve the project in the future.

5 `https://www.npmjs.com/`
6 `https://nodejs.org/`
7 `https://github.com/rug-ds-lab/bsc-2019-in-browser-computing`

GROUP WORK

This project is a collaboration between Tolga Parlan and George Argyrousis. The group members worked jointly on chapters Introduction, Architecture, Conclusion and Future Work as well as the Abstract. Related Work chapter belongs to Tolga Parlan and the Evaluation belongs to George Argyrousis. The presented program is coded together and most of the benchmarking is also completed as a group effort. The metrics about Web-Worker Performance is the work of George Argyrousis.

2 RELATED WORK

The concept of a distributed system that uses previously unknown computers as components through running code on their web browsers and communicates with them over internet is not new. There have been many attempts at creating such distributed networks, applying different approaches, architectures and aims. Here we give a quick breakdown of the past projects that this project has directly or indirectly benefited from. We also attempt an analysis over why browser based distributed computing projects have not caught on popularity-wise, and speculate about solutions our architecture may provide.

We follow the three generation paradigm proposed in Fabisiak and Danilecki (2017)[11] while surveying the existing Browser Based Voluntary Computing systems. They describe the first generation as the Java applets, which used to be a common solution when Javascript performance deficit was extreme, being at around 10 to 100 times slower according to the authors. However Java applets provide an unsatisfactory user experience, since they function similarly to an additional program running on the volunteer's computer, with slow initialization times and disturbing pop-up windows. The second generation has mostly used Javascript to overcome such problems, albeit still suffering from performance deficits and lack of multi-threading support on browsers. We concern ourselves mainly with what they define as the **Third Generation**. The third generation is distinct from the previous ones because of the new Javascript features it can benefit from such as a fast compiler, thread support (via WebWorkers) and WebSockets. These technologies significantly increase the feasibility of such projects due to improvements in performance and user experience. We have used a similar technology stack to the Third Generation projects with similar aims, thus they are the most relevant examples we can use to examine how to improve current systems.

2.1 SIMILAR PROJECTS

This section aims to provide a summary of similar projects which has given us inspiration for our architecture. Therefore we present three distinct projects, and argue about the successful and unsuccessful design choices they have made, and how we have tried to design our project accordingly.

MRJS [17]

MRJS sets to discover the feasibility of implementing Google's popular and influential Map-Reduce architecture[7] while using web browsers as the worker components. The Map-Reduce paradigm is useful in a broad range of computational tasks, and has the side advantage that the it presents a very high level and intuitive interface for the programmers to interact with, hiding away details that would require experience with distributed systems. The system breaks down given work into chunks before distributing it to clients, which includes the Javascript code to run, starting and ending indices, and a data URL. They have a **Job Server** which schedules the chunks and organizes the results. They also use simple majority voting to detect possibly rogue clients. The communications with the workers rely on plain HTTP requests. They have implemented a proxy server for data transfers and their clients use WebWorkers for moving computations to the background. The authors find that implementing a Map-Reduce platform over the internet does not always have the same performance capacity as the original system, especially in jobs that require too much data transfer. The authors conclude, after their benchmarks, more computationally intensive jobs could take better advantage of such a system and classic examples of Map-Reduce, such as counting words in a large text, are not suitable for MRJS.

QMachine [19]

QMachine is another system targeting scientific applications. It consists of three different components: **browsers** which do the actual computations and submit new computations, an **API Server** which receives computation requests from the browsers and distributes these computations to other available browsers and a **Web Server**, which is needed to serve the initial code to the browser clients. The API Server should be installed and run individually by the user who wants to create a network for their own purposes, and it can use only a given set of databases. The authors assume that QM will be used by closed scientific groups who will then invite only trusted volunteers, thus the only security feature involved is verifying that the participants are indeed the registered trusted volunteers. The communication between the API server and the browsers use plain AJAX requests to submit jobs to the server, and to poll the server for new jobs. This polling approach inherently has worse scalability than WebSockets. Lastly, The QM class written by the authors can be used outside of a browser as well, using NodeJS.

Unfortunately QMachine has not become a popular platform. The authors report that their system received more than 2.2 million API calls from 87 countries during the initial 12 months period it was made public. However, this number is less impressive when one looks at the number of distinct users, which is reported to be 2100 separate IP addresses. We suspect that this outcome was effected by an array of reasons. Firstly, it is not clear how this system would generalize to open internet, given its lack of any sabotage-tolerance features. In an environment where not everyone is a trusted volunteer as assumed by the system, sending wrongly calculated results back would easily sabotage the computation. Distributing computational tasks makes sense practical only if there is an abundance of computing nodes. However a system in which only trusted volunteers can participate is severely limited in scalability. Furthermore, the system design allows decentralized job submissions, giving the any connected rouge browser to opportunity to eat up system resources by submitting very large jobs. Also the system has little flexibility in how the computed data is later on stored or handled. The computed data has to be stored on a database, and the API Server owner needs to use one of the database systems already supported by the QMachine. This imposes an unnecessary limitation for what sort of use cases this system can support.

MLitB [14]

MLtiB, Machine Learning in the Browser, is a prototype Machine Learning system that is intended to provide an easy platform for researchers to run their machine learning jobs, using the computing power provided by volunteers. When users connect to the system, they interact with a User Interface on which they can create workers (which are implemented as WebWorkers) to perform various tasks. The system relies on the assumption that the contributors will voluntarily go to a web page intended for machine learning and create these workers, meanwhile understanding what they are doing and trusting the system. After the initial transfer of data (which is compressed to speed up the transfer), all communication happens through WebSockets. They have two servers, one for distributing data and another master server for coordinating the process. Both of them are written in NodeJS, and the authors justify this design decision with the event-driven and lightweight architecture of NodeJS servers: "Since the main computational load is carried by the clients, and not the server, a light-weight server that can handle many clients concurrently is all that is required by MLitB." They have been influenced by the Map-Reduce paradigm and use it for their machine learning algorithms.

Since the project was merely a prototype, it is not possible to talk about its popularity in certain terms. However one can point out several project characteristics which would create user retention problems. Having a dedicated user interface assumes that the users are willing and competent enough to create WebWorkers on their browsers for specific machine learning tasks. This could be a severe limitation on the user base, comparable to the demographics limitations on the projects such as BOINC, which required users to install a program on their computer. According to their own survey, about , 63% of BOINC users describe themselves as advanced, and 35% as intermediate computer users [6]. We advocate that an approach that runs the computing in the background without needing any user action is much more preferable if Browser Based Volunteer Computing systems want to tap into broader computing resources.

2.2 GOOD PRACTICES

From the inspected projects, we have adopted practices with the potential to improve our system. A discussion of such technological and architectural components is presented in this section.

Amongst the Third Generation projects, the use of WebWorkers is ubiquitous. WebWorkers allow multi-threading support in Browser environments, which is important for user experience. If the calculations were ran in the default Javascript thread, rest of the webpage would lose its responsiveness and running a distributed system would be much less desirable for websites with high user engagement. As a result our system mandates the use of WebWorkers. The heavy calculations are ran in a separate thread and does not damage page responsiveness. The concept of multi-threading in browsers is discussed in detail in the Multi-Threading section, while our implementation is presented in the Web-Workers section.

WebSockets are intended for fast bi-directional communication between a client and a server. MLtiB, amongst many projects, recognizes that data transfer can become a serious bottleneck in the system due to bandwidth and latency limitations and uses WebSockets as a solution. We have also opted to use WebSockets as the main communication channel, after an initial HTTP call loads the web page. The implementation in our system is discussed in the WebSockets section.

MRJS aims to give programmers control over parameters that might affect the overall system performance. The idea of hiding the distribution details from the programmer who is using the system, while giving optional access to parameters related to performance is a principle we have tried to follow in our program as well.

A great deal of distributed computing projects support Map-Reduce. It is notable for its ease of use even for programmers without any distributed computing experience since it hides low-level details of the system and allows for configuration through simple intuitive interfaces. It has successfully been deployed in large scales and many algorithms which inherently allow for parallelization can be expressed in terms of Map-Reduce[8]. Our system is also inspired by this paradigm, and as such we abstract away the mapping step, distributing the data pieces across workers who send back the results. We additionally expose an easy interface to reduce the mapped data.

The QMachine project presents useful ideas such as having a separate server running the distributed system independently of the web server. We think that this modularity is very important for easy integration into other projects and allows creative architectures such as one distributed network running the same jobs through several websites. We have designed our project with the same possibility of decoupling the web server serving the web pages, from the distributed system server serving the tasks.

Packaging the client code so that it can simply be a part of any web page is another useful idea from the QMachine authors which we have implemented. Serving javascript files as independent packages eases the process of embedding our system to an existing web page which may originally have been created without accounting for it.

MLitB authors provide arguments in favor of using NodeJS as the server in a distributed computing application where the server is mainly concerned with coordinating the clients and distributing tasks. The server in such a system ideally would not be busy with processor-bound tasks but IO-bound tasks to support communications. The event-driven asynchronous model which can serve many clients concurrently is a strength of NodeJS and we have adopted the technology in our project.

2.3 COMMON PROBLEMS

2.3.1 *User Retention*

A common problem that is introduced by using browsers as the computation medium for a distributed network is to convince the users to visit a website and stay there. This is a problem which earlier projects such as SETI@HOME or the GIMPS did not face, since in these projects volunteers download and install a program which then runs in the background. In contrast, in a browser the computation stops when a tab is closed and it is safe to assume that the common workflow of many casual web surfers does not include keeping a tab which does not serve a function open for hours. This section discusses ideas about how to solve this major problem.

According to the survey by Fabisiak & Danilecki (2017) [11] "the problem of recruiting the users is the most important challenge faced by browser-based platforms". One common theme amongst all the projects they have inspected is either a very low number of volunteers, or the numbers not being reported at all. In the paper the authors suggest that incentive systems employed by older generations of voluntary computing systems, such as reputation systems, virtual credits or real monetary awards based on computing power contributions could to some degree be adapted to the newer systems. However this suggestion in our opinion ignores one of the most important properties of browser based systems. Browser tabs are designed to be easily disposable and many users open and close plenty of tabs routinely. Convincing a user to keep a tab open for meaningful amounts of time and routinely come back is difficult and we suspect most projects fail at this, even when they have successfully attract a decent number of contributors initially such as QMachine.

Gray computing practices which rely on running computing tasks on websites without explicit user consent can be a solution to the aforementioned user retention problem. It has proven difficult to attract volunteers, who would spend meaningful amounts of time on a web page and come back frequently. A possible solution would be to embed such systems in already popular websites with which users engage for a long time. Thus we have striven to make our system easily adaptable into any existing website with adding only a small script to the client-side code and keeping the server implementation as simple as possible. Gray Computing section discusses this topic in detail.

2.3.2 *Adaptability*

We have observed that many projects in the field of Browser Based Volunteer Computing are done with one sort of calculation in mind. Apart from MLitB examined before, most other projects we have examined such as Zorrilla et al. (2013) [20] focus on social networking, Krupa et al. (2012) [12] limits their investigation only to web search, Duda & Dlubcaz (2012) [10] target only evaluationary algorithms and so forth. While the investigative value of these papers is indisputable, they are not really adaptable to many other purposes.

A similar problem that arises from this problem-centric approach is that the program architectures are not built with much modularity. This is a common theme that might be harming the adaptability of the projects to different technological environments. Trying to provide a Swiss army knife solution, which attempts to be very simple to use by providing as many as features as possible, is not necessarily the best software development approach. These decisions are understandable since many projects target scientists working in diverse fields who might need computing power to run their calculations on but not the understanding of the programming issues, and therefore the projects try to come up with an architecture that is as simple to use as possible. However, in the end, this often leads to systems that are rather complicated to adapt. For example a user of the QMachine system who does not want to store their processed data in one of the presented database options does not have a choice. Someone who would like to use their own servers cannot run the MLitB system since it requires its own data and machine learning servers to be used. As a solution we have designed our project as a package which is as agnostic as possible as far as the types of algorithms it can implement and the server architecture it can cohabit.

2.3.3 *Performance*

Langhans et al. (2013) [13], embed their Map-Reduce solution doing large calculations in the background while the user is busy playing a short browser game, and then survey the users. In their survey, "no user reported effects of the massive calculations in the background on the game application in the foreground. (...) Concerning allowing additional calculations in the background only 15% of the users raised doubts but none rejected the idea". They explain this by WebWorkers moving the calculations to a separate thread. Indeed we agree that the WebWorker technology makes running computational tasks much more feasible for websites which do not want the user experience to take a hit. A detailed discussion of multi-threading in browsers can be found in the Multi-Threading section while our implementation details can be read in the WebWorkers section.

2.4 GRAY COMPUTING

Pan et al. (2015) [16] set to investigate the economic feasibility of running background Javascript computations on browsers from the perspective of a website owner, taking into consideration the issues that arise with high heterogeneity in devices, non-uniform page view times, high computing tool volatility as well as the Byzantine nature of the distributed environment and harm that can be caused by malicious users. Their paper comments that the "line between what computational tasks should or should not be offloaded to the visitor's browser is not clear cut and creates a blurred boundary" which they name **Gray Computing**, referring to the possible ethical implications of running the said computing without explicit user consent. But aside from the ethical implications, they engage in a thorough investigation into whether running such tasks even makes sense over using more conventional services such as cloud computing providers. The findings are very optimistic, especially for a subset of tasks, and their paper lists empirical results on important issues such as handling malicious users, effectively allocating tasks to clients with unknown page view times, gray computing's possible impact on website performance and Javascript performance. We provide a summary below.

- The use of WebWorker technology makes sure that background gray computing tasks would not be easily discernible by the users, due to effective multi-threading. However they find that even without the WebWorkers the effect on the page performance is negligible, however in this case much less CPU power is used for the computations.

- They compute the security provided by a simple task duplication and majority voting scheme against malicious users and find that for large enough websites where it would be difficult to take over most of the computations, even duplicating tasks twice offers substantial protection. We expand on this concept in the Fault-Tolerance section and present the implementation of an improved algorithm in the Redundancy section.

- Cost effectiveness largely depends on the algorithm and how it is implemented. Algorithms that need to be fed large data chunks are typically not cost effective, due to the slow nature of data transfer and the costs associated with downloading data. However the authors point out that most cloud computing providers have a price model which charges for data downloads from the cloud, but not for data uploads to the cloud. This gives a substantial cost effectiveness benefit to algorithms that does not need much input but produces large outputs. Similarly, scenarios where the data would be served to the clients anyway, such as Face detection in a social network site which already serves the photos to the users, can be made distributed very effectively. However the paper comments that some popular Map-Reduce use cases such as counting words in large texts are not cost effective due to the large data transfers needed.

- Gray computing clients cannot be relied on to stay connected to the system for the whole duration of a large computation, therefore how the work has been divided into chunks becomes important. "Reducing the single task size assigned to the clients will increase the task completion ratio, but result in more task units. More task units means more cost on the requests to fetch data and return results. Therefore, there is a trade-off between using smaller tasks and larger tasks." The authors propose that an adaptive scheduler would achieve a higher task completion rate compared to uniformly sized chunks. We describe our own system of adaptive load balancing in the Architecture Chapter.

3 ARCHITECTURE

As part of the project, we have created a Javascript system which utilizes the good practices identified in the Related Works chapter and implements solutions for the common problems. Rest of the chapter introduces important terminology used in the system, touches the principles upon which the system was built, introduces technological concepts on a high level and lastly details the implementation of the actual program.

3.1 TERMINOLOGY

3.1.1 *User*

The User is defined as the person whose browser is being used as the computation component of a Distributed System built on top of our system. A client-side script will be executed by their browser when they visit a URL with an implementation of our framework. Upon having the script load, the user will automatically be registered in the server, thus donating resources to the system for as long as they remain connected.

3.1.2 *Creator*

A Creator is the person building their application on top of our system, interacting with it through the programming interface we have defined, by implementing the work function for the client-side and by defining two streams for providing raw data to the server and acting on the processed data from the server.

3.1.3 *Task*

A Task is characterised as one singular entry in the data set being provided to the server. A singular Task can be any JavaScript data type and it is defined by the Creator. The system was designed with Distributed Computation applications which consists of a large multitude of Tasks in mind. Each Task should be calculable independently from any other Task using the same Work Function and ideally each Task should take a similar amount of time to compute on the same computing environment. It is desirable to have Tasks which do not take more than a reasonable amount of time on an average computer in order to make sure disconnecting Users cause as little computation loss as possible.

3.1.4 *Load Function*

The Load Function is responsible for generating Tasks or supplying the already generated Tasks to the system. It needs to take the form of a Stream, which is discussed in Streams and Duplex Stream sections in detail.

3.1.5 *Work Function*

The Work Function is defined as the function that performs algorithmic steps on the Tasks created by the Load Function. It is called once on each piece of data generated by the load function. Since the computations are performed by the Users, the Work Function resides on the client-side, in a Web-Worker file. Multi-threading issues and Web-Workers are discussed in their respective sections.

3.1.6 *Distribution Function*

The distribution function is responsible for determining how many Tasks should be transferred to a User at any given time. It can take multiple forms depending on the needs of the application. Different kinds of Distribution Functions present in our system are discussed in the Load-Balancer section.

3.2 PRINCIPLES

The Principles section analyses all the assumptions and principles which lead to the architecture choices in the system.

Extensibility & Adaptability

Many algorithms implementing some sort of concurrent operation can be implemented with our system. Such algorithms would typically have a data-set that needs the same operation to be performed on each individual data piece. Existence of a broad range of algorithms that can use the system inevitably makes it less sensible to focus on the peculiarities of one algorithm. Instead a general solution which can be extended for any algorithm carries the potential to have more utility. Therefore the system aims to take care of the distribution of data and coordination of clients, while giving the Creators the ability to implement how to provide input and how to process the output, as well as how the data is processed by the clients. On the technology side, the system follows common and recognizable patterns used by other NodeJS packages such as synchronicity, event-based architecture and data piping through streams. Therefore the system can be relatively easily integrated into NodeJS server code-bases. If the Creator's server uses another programming language, a separate server with a different domain for the distributed computation can be implemented with ease. The client-side code can be part of most websites by adding the necessary scripts and Multi-Threading should allow the websites with a need for performant User Interface to also partake.

Scalability

A distributed system will naturally perform better with more computation nodes, barring the overhead. Therefore such a system should be able to effectively scale up to handle large numbers of worker nodes with minimum overhead. We design the system with as minimal general overhead and minimal computational load added by each additional client. The scalability of the system for different type of tasks is further discussed in the Evaluation chapter.

Maintainability

Our project culminates in an open source package and therefore in the future if enough interest is generated, outside parties may want to contribute to the development as well. Contribution from anyone would be most welcome. It is expected that using our package in real life would unearth many problems, potential performance improvements and useful features, and the contributions of willing outside parties can be an important asset to perfecting the code. Therefore we adhere to good software development practices with clearly defined modules, comment our code and architecture and implement adequate testing and continuous integration. These practices are meant to ease the maintenance for the authors, and also lower the barrier of entry for anyone who might wish to develop the project further.

3.3 CONCEPTS

3.3.1 *Streams*

Streams are a useful Computer Science concept which can provide an analogy for large data flows, such as reading a large file, processing it and outputting the processed data. Stream components can be added together to create a pipeline where data flows between each component, and the components do not need to know about pipeline step before or after them. An important use case of Streams is with handling data which would be too large to keep in the memory at once. If the data is instead processed through a Stream, only a small portion of it needs to be kept in the memory at any given time. [15] Distributed Computing systems such as this one can be designed as a Stream component which stands in between inputted raw data and outputted processed data.

3.3.2 *Fault Tolerance*

A great challenge for distributed computing applications is Byzantine Fault Tolerance. Even a system such as SETI@home which requires registration has in the past experienced users cheating on the results [1]. Manipulating the client-side code and sabotaging the distributed algorithms would be a relatively easy task for any malicious Client. One attacker can coordinate many clients and produce large amounts of invalid results for a calculation. This could be especially damaging since in a system open to public, it is not easy to blacklist users. Even banning specific IP addresses suspected of sabotage could easily be circumvented. In theory, it is impossible to achieve completely trusted results using this mode of calculation, unless a completely trusted machine was checking all the calculations. But obviously, a separate trusted machine running all the calculations makes distributing the calculations pointless. However, for a system with a relatively large client base adding redundancy to the calculations and making different users vote on a piece of result severely limits the error rates. Such a majority voting algorithm has already been in use by many of the most popular distributed computation architectures, such as BOINC and Hadoop [4].

We have inspected the **Spot Checking** idea examined by Sarmenta (2001) [18]. Spot Checking is introduced as an alternative for systems where the acceptable error rate is not too small. Unlike traditional majority voting, spot checking does not redo all the work objects several times, "but instead randomly gives a worker a spotter work object whose correct result is already known or will be known by checking it in some manner afterwards". The author demonstrates that while spot-checking combined with blacklisting of caught cheaters is effective, if blacklisting is not possible, it is not very difficult for an attacker to increase the error rate significantly. Therefore this technique would not be useful in our system.

Some researchers approach the problem in a totally different manner, and instead try to establish the trustworthiness of a user. Domingues et al. (2007) [9] devise a reputation system that calculates the reputation of users through invitations, in which users invite other users to participate in a volunteer computing project, and each user receives a reputation depending on the reputation of other users in their network. Such a system however is not suitable for a gray computing project, where the aim is not to limit the users to a small clique of people who are already interested in distributed computing projects by themselves, but instead to benefit from the computing power of the general public.

Brun et al. (2011) [4] introduces an improved idea, which is based on majority voting but provides an improved theoretical error rate, called **Iterative Redundancy**. The authors demonstrate that the iterative redundancy is more efficient than traditional majority voting, and it can achieve a desired system reliability by distributing fewer jobs than similar methods. Intuitively the idea comes down to copying the same task until the difference between the most popular result and all the other results are larger than a set number. This is distinct from the idea behind traditional majority voting, which sends a set number of copies and then accepts the majority vote. We have implemented this algorithm in our system to benefit from the slightly lower theoretical error rates, with the option to turn it off or set the redundancy requirement relatively low since many algorithms can still function even in the presence of erroneous results. The implementation details are explained in the Implementation Section.

3.3.3 *Load-Balancer*

The Load-Balancer is responsible for the size of tasks that end up being fetched for a User. Since a user's performance cannot be guaranteed to be constant, we need to constantly be computing the appropriate size of tasks that needs to be delivered to each independent User. Events such a user browsing away to a more computationally intensive tab or having a worse Wi-Fi connection than before can lead to slower task processing, thus ideally the Load Balancer should try to adapt and maintain a consistent processing time per chunk of tasks.

The task size is not in any way limited with our current implementation, allowing the User's computational performance and network speed to be the source of measurement. We intentionally decided to avoid having the latency being computed on the User's side. By doing so we could potentially be exposing a flaw in the system where a User could alter the latency manually and provide inaccurate results.

The latency of each User will be computed as a result of the time ts of sending the load to the Client, the time to compute the result of the Work Function tc and the time to receive the response back to the Server tr.

$$latency = ts + tc + tr$$

Adaptive Load

We will define all accessory information that needs to be computed in accordance to the Adaptive load. Below we will define the mathematical formulae that define the adaptive load at any given time. A client c has three internal attributes, the client's previous amount of tasks computed $c.ld$, the client's response time $c.lt$ and the client's send time $c.ls$ where $latency = c.lt - c.ls$.

First we will need to compute the average response time $f(c)$ of each connected client c per task. Then we will find the collective response time art for all c. Finally we will compute the average collective response time ava.

$$f(c) = \frac{c.lr - c.ls}{c.ld}$$

$$art = \sum_{i=0}^{c.length} f(c_i)$$

$$ava = \frac{art}{c}$$

Based on the available information, we can now compute the amount of tasks $g(c)$ that can be added to make the load match our system's average response time ava.

$$g(c) = \frac{(ava - f(c)) * c.ld}{ava} + c.ld$$

For example let's say that we have a client c with a response time of $c.lr - c.ls = 8669ms$ while computing $c.ld = 95$ tasks. Let's consider the system's $ava = 521ms$.

$$f(c) = \frac{8666}{95} = 91$$

$$g(c) = \frac{(512 - 91) * 95}{512} + 95$$

$$= 78 + 95$$

$$= 173$$

For this client, the distribution function deemed him fast enough to be able to compute an additional 78 tasks in order to meet the system's average *ava* metric. However, there is the possibility for the system to give a negative value in terms of how many additional tasks that need to be computed. For example a client with response time of $c.lr - c.ls = 7930ms$ while computing $c.ld = 13$ tasks. The system's $ava = 512ms$ as before.

$$f(c) = \frac{7930}{13} = 660$$

$$g(c) = \frac{(512 - 660) * 13}{512} + 13$$

$$= -4 + 13$$

$$= 9$$

If the overall outcome of $g(c) \leq 0$, the User is deemed incredibly slow compared to the system's *ava*. It is therefore beneficial to assign the same *c.ld* as before since decrements would not yield a better performance.

3.3.4 *Multi-Threading*

Javascript is a single-threaded programming language. The language itself offers no native way to implement concurrent programming and a common concern for JavaScript programmers is to avoid blocking the single thread. Not blocking the thread is important because browsers render pages by creating render events in intervals. However JavaScript event-loop model runs events only when the call stack is empty. A computationally intensive task which is occupying the call stack for long amounts of time can constitute a serious problem for the user experience because the page would not get rendered and thus it would become unresponsive to user actions. Some projects such as [3] tried to solve this issue by engineering non-blocking loops which would hand back the control of the thread after every iteration, thus allowing rendering. However Web-Workers[1] have mostly solved this problem, as they introduce proper multi-threading support for JavaScript written for browsers. Our implementation of Web-Workers is discussed in detail in the Implementation Section.

3.4 IMPLEMENTATION

3.4.1 *Client*

Upon visiting the URL being hosted on the Server, the browser will fetch and render the available HTML file that contains our Client package. Uncompressed JavaScript files can lead to an increase in loading times of the visited web page. Thus we decided to use WebPack[2] to make the Client file as small as possible.

The Client in our architecture is defined as the script responsible for retrieving chunks of steps fetched from the server that need to be executed. The client can be treated as a naive component in programming terms, and takes care of connecting to the server and passing on data provided from the server to the Work Function. The Client should be provided by the name of the Web-Worker file which will later on perform all the processing on the data sent by the server. The Client also needs to be provided a working socket to run the communications on.

Below is a simple example client.

```
new Client({
    /* Socket io instance with host:port */
    socket: io("localhost:3000"),
     /* WebWorker file location */
    workFile: "./work.js",
});
```

Leader Election

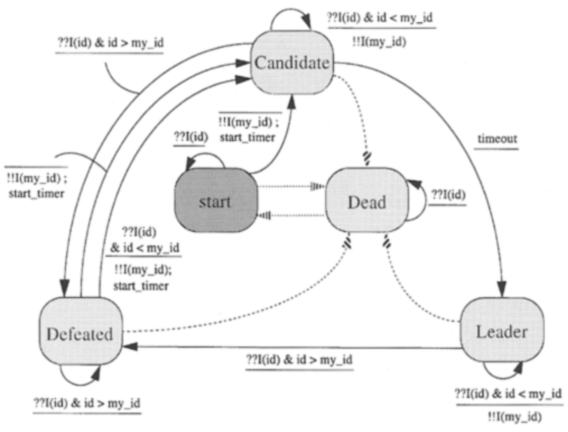
We have additionally implemented the option to block several tabs of the same website from opening their own separate connections and performing computations in their own instance of Web-Workers. A common internet surfing pattern with many modern browsers is to open many tabs while following links, and this may lead to a situation where a typical user opens many tabs all trying to perform the distributed calculation. This was due to performance concerns about running the same program in multiple tabs, which in our experience led to worse overall performance. The performance concerns about running the same program in multiple tabs is investigated further in the Evaluation Section.

Running the distributed calculation only in one of the tabs is equivalent to the **Leader Election** problem in the distributed computing field. The leader in this case would be the tab which got chosen to perform the calculations. Since tabs would not have a considerable advantage over each other, their priority is chosen randomly. Tabs constitute a dynamic environment because they can be closed or opened

1 https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
2 https://webpack.js.org/

unpredictably. This can also technically be considered a Byzantine environment since tabs do not have to follow the code written by us, however this was not a concern since there is no harm to be inflicted on the overall distributed system by manipulating the leader election process in a browser. It is possible for tabs to communicate with each other via broadcasting. BroadcastChannel API[3] allows Javascript in one tab to broadcast messages to all other tabs.

We have based our leader election on the algorithm described by Brunekreef et al (1993) [5]. They depict a protocol for dynamic leader election in broadcast networks, which describes our intention perfectly. In their protocol, the agents involved in the process can be in one of five distinct states, as *Leader*, *Candidate*, *Dead*, *Start* and *Defeated*. We have ignored *Dead* and *Start* since a dead tab is already closed thus irrelevant and a starting tab can immediately be appointed as a candidate. Their protocol is described as a Finite State Machine in the graph below, taken from their paper.



Intuitively, each tab becomes a *Candidate* at its inception. Tabs become *Candidate*s by broadcasting their ids and starting a timer which checks for a leader timeout. Upon receiving an id broadcast, all other tabs with larger ids broadcast theirs. If a *Candidate* or a *Leader* receives an id larger than its own, it becomes *Defeated*. If a *Defeated* tab receives an id smaller than its own, it announces candidacy. If the *Leader* times out (crashes), the *Candidate* which has defeated all the other tabs will become the new leader.

There is also another state transition from *Defeated* to *Candidate*, depicted without a guard for simplicity purposes. It is intended for a leader timeout. The need for this transition can be explained by an example. It is easy to postulate a situation with one Leader, one Candidate and several Defeated tabs. Suppose the Candidate tab is closed, followed by the Leader tab. Now without the timeout transition from Defeated to Candidate, there would never be a state change unless a new tab is opened.

We had faced practical issues while implementing the algorithm, which required several changes. The algorithm runs on the premise that constantly changing the leader is acceptable. However constantly changing the worker tab when another *Candidate* with a larger id connects is unacceptable due to the performance hit caused by reconnections and repeating calculations. This would be unnecessary

3 `https://developer.mozilla.org/en-US/docs/Web/API/BroadcastChannel`

in our system since tabs do not have a tangible performance difference, and the ids are assigned randomly. We added the additional state *Worker* to solve this issue. In the case where there is no *Worker* (which is detected by timeouts), the current *Leader* assumes the role of *Worker*. *Worker* tab is different in the sense that it does not participate in the leader election algorithm anymore but instead undertakes the actual calculations. The absence of a *Leader* triggers the *Defeated* to announce candidacy even if there were no *Candidate*s, thus the rest of the algorithm functions similarly to the situation where a *Leader* disconnects.

Another problem we encountered in our environment was that the system, in rare cases, produces two simultaneous worker tabs. This is due to the practically impossible assumption made by the authors, namely that "messages that are sent are received instantaneously by all processes, except the sending process". Unfortunately the Javascript event-loop architecture dictates that the messages sent between tabs are never instantly delivered, but the browser will wait until the current stack is empty until interrupting with the message. This has caused us to experiment with timeout intervals to find a time which is sufficiently long so that accidental timeouts caused by late delivered messages are rare enough while it does not take too long to choose a leader.

3.4.2 *WebWorkers*

WebWorkers[4] are a part of the client-side code, and require the Creator to create a separate file with the code which would run in the separate thread. The Creator should also provide the file's URL as an option to the client-side code while initializing the class. The implementation permits the creation of a single extra thread and communicates the data it receives from the Server with the thread. Processing the data is the responsibility of the code provided by the Creator. When the results are sent back to the main thread, the main thread communicates them to the Server. The data communication between the main and the worker thread is being facilitated utilising the Web-Worker API. Both sides can send messages using the *postMessage()* method and handle incoming messages with the *onMessage()* event handler method. Below is an example of a simple Web-Worker file which receives an array of data, runs the work function and posts the results back.

```
self.onmessage = (e) => {
  postMessage(e.data.map(workFunction));
}
```

The implementation does not communicate individual data pieces, but instead always an array, thus the need for using the *map()* function on the received data on the thread file. This was a result of experimentation we have run on the communication overhead between the threads, and is explained further in the Evaluation Chapter. It should

4 https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

be mentioned that WebWorkers are currently supported in every major desktop or mobile browser.[5]

3.4.3 *Server*

The Server is responsible for connection events related to the clients, as well as handling the data communications. The Server communicates with the connected Clients via the Socket.IO package, which is described in detail in the WebSockets Section. A connected client triggers the connection event in the Server, which adds it to the list of available clients and sends the initial data load if the Creator specified any. The Server also saves useful information which is later used by the Load-Balancer to calculate how many data pieces should be sent to a specific client. When instructed to send data to a Client, the Server saves the time. Similarly, when a result is received the response time and the count of data pieces is is saved.

3.4.4 *WebSockets*

Socket communications between the Server and the Client rely on the WebSocket technology[6]. WebSockets API however is low-level and not intended for direct use by most programmers. We have instead opted for the popular Socket.IO library[7]. Socket.IO provides an easy interface for WebSockets, and requires the inclusion of an extra file in the client-side code as well as a package on the server-side code. It has already implemented support for bidirectional disconnection support. Therefore both the Server and the User know whether each other are unresponsive since the library takes care of pinging. Furthermore, it supports fall-backs in case the user is using a browser which doesn't support WebSockets, providing communication via more traditional AJAX methods such as polling, through the same programming interface.

On the client side, the socket communications happen on the main thread and therefore it is important to avoid blocking the main thread any longer than what is necessary for data transfer. Socket.IO library is based on events and therefore avoids blocking the thread, due to the JavaScripT event-loop architecture detailed in the Multi-Threading Section.

The sockets listen for messages that can be sent either to the User or to the Server using the *socket.emit("tag", message)* method with the appropriate tag. In order to react upon receiving a message, socket.IO provides the *socket.on("tag", message)* method which is fired for messages with the appropriate tag. Upon a new connection the *socket.on('connection', ...)* event is fired and a disconnection is handled through *socket.on('disconnect')*.

Our system as a whole aims for modularity, and does not include the socket.IO class itself in either of the Client-side and Server-side packages. Instead the constructors expect that the Creator will supply the socket object. Since there can be multiple channels communicat-

5 https://caniuse.com/#search=webworker
6 https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
7 https://socket.io/

ing over the same socket for different purposes, our package uses the tag *data-distributedstream* for normal data communication, and the tag *initial-data-distributedstream* for the initial data load.

3.4.5 *Duplex Stream*

Streams[8] are one of the core parts of the NodeJS API, and they are intended for working with large data. NodeJS Streams come with build-in functionalities for piping data between different in a chain, and signaling to separate Streams in the pipe chain to stop or continue data production depending on the overall data processing speed, in order to prevent a module from working too fast and filling the memory.

The process of stopping data production along the pipe chain if a later part of the chain is not consuming data fast enough in order to keep memory consumption low is called Back-pressure[9] and it is a fundamental part of the NodeJS Stream architecture. Since the main class in our system inherits a Duplex Stream, which is a Stream that can be read from and be written into, we followed the requirements of implementing a Stream class which should take Back-pressure into account when producing data, thus distributing tasks to clients. If the data is being processed too fast for the next step in the piping chain to consume and therefore the buffer in between is full, the Stream architecture signals the Duplex Stream to temporarily stop producing and outputting data, by returning a false value from the function which pushes data into the output buffer. Our package in this case temporarily stops outputting data to the buffers and also sending jobs to the clients in order to prevent an accumulation of processed data which would increase memory consumption. When the piping chain can continue, Stream architecture signals this by invoking the _read function and causes our system to continue sending data to clients.

Piping data into our distributed stream and piping it out is the Creator's responsibility, and makes the system adaptable to many use cases. This also allows for a programming interface easily understandable for programmers already accustomed to NodeJS. Code for a simple use case in which the creator provides data from a stream, and then writes the processed results into a file would look like this:

```
dataStream // A Readable stream that inputs raw data
    .pipe(distributedStream)
    .pipe(eventstream.stringify()) // Stringify the results
    .pipe(fs.createWriteStream('results.txt'));
```

On the other hand, in a project where the results are used to modify the inputted data such as some machine learning applications where the results could be used to modify the model constantly, the Creator can provide their own duplex stream which creates data and absorbs the results to act accordingly:

```
mlModel // A Duplex Stream
    .pipe(distributedStream)
    .pipe(mlModel);
```

The Stream pipes out the results in the same order the input was piped in. This was made complicated by the fact that clients often dis-

8 https://nodejs.org/api/stream.html
9 https://nodejs.org/es/docs/guides/backpressuring-in-streams/

connect at random times and data they were supposed to process may end up being sent again to another client and be processed at a later time than data which was inputted after. As a time efficient solution, two buffers are implemented to store the order number for (1) data which are waiting to be processed, or (2) are processed and waiting to be piped out the Stream. These buffers are Red-black Trees since the data order in the buffers should be preserved while a great deal of insertion, removal and search operations are ran and Red-black Trees provide $O(log(n))$ time complexity for each of these operations. The buffers store only the data order instead of the entire data object, in order to speed up the comparisons inside the Red-black Tree. The data itself is kept in a Map[10] where the key is the input order.

3.4.6 *Redundancy*

We have implemented the **Iterative Redundancy** algorithm discussed in the Concepts Section to combat cheating as well as any technical problem which might cause false calculated results to be sent to the Server by Clients. When a Creator is initializing our *DistributedStream* class, they can choose to specify their *redundancy factor* as any number $r >= 1$. The redundancy factor is the difference the system is going to establish between the highest voted result and all the other results before accepting a data piece as reliably processed. So for instance if $r = 3$, and two malicious hosts returned some incorrect result, at least 5 votes should be cast that produces the correct result before it is accepted by the system.

A major deficiency of our redundancy system is the lack of protection against the same computer connecting through different WebSocket instances (which could be opening multiple tabs in practice), and banning clients that voted for losing results. While in theory implementing such barriers is not particularly useful because changing IP addresses is not a difficult process for a determined attacker, in practice it would make any attack relatively more difficult and contribute to security. We discuss possible extensions in this direction in the Future Work Chapter.

We have implemented a separate **Data** class to take care of the redundancy operations for each piece of data. When the data is initially sent to a client for processing (which is usually done in a batch with many other data pieces to minimize the network latency), the *addVoter()* function is called and the client is saved as a voter for the data. This is necessary to make sure the same data is not sent to the same client again in the future.

The system functions with the assumption that the vast majority of the clients will be honest, and thus data should initially be duplicated only as many times as it would minimally need. If $r = 3$, this would be 3 times. The function *shouldBeSent()* which determines whether a data piece currently needs to be sent to more clients, thus works with the assumption that all the current voters for a data piece will return the same result, even if they haven't responded with the results yet.

10 `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map`

The class function *doneWithProcessing()* returns true when the majority vote result is more popular than all the other results by at least the redundancy factor. When a data piece is done with processing, it's result is written into stream for the next part of the pipe to consume. As mentioned previously, the output is always in the same order as the input.

3.4.7 *Interaction Between Modules*

In order to further understand the Architecture we will explain the communication between our key components as they can be seen below in Figure 1. We will focus on explaining the interactions between the Load-Balancer, Server and Client as these components have the greatest impact and can directly alter the behaviour of our system.

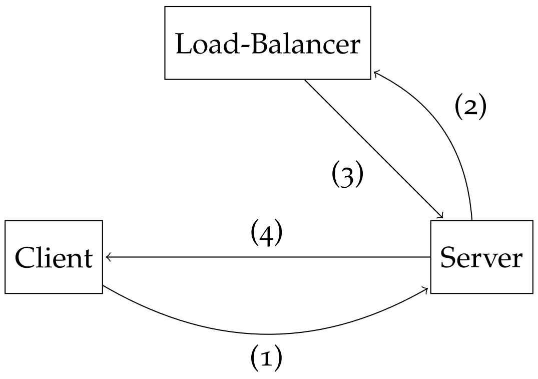


Figure 1: Architectural Interaction Model

Upon having the **Client** connect to the System the **Server** adds it to the list of all available Clients (1). The List of Clients contains relevant information about the current status of each Client. If a Client is not executing a task or is done executing a task the Server will assign a task to it.

Before assigning a task, the server will determine the amount *a* of tasks to send utilising the **Load-Balancer** (2)(3). Now that the correct size of tasks has been determined the Load Function will be send *a* tasks to the Connected Client).

The Client will receive (4) the assigned *a* tasks and perform the Work Function on that specific data-set. When the Client is done executing all assigned tasks, it will send a response to the Server at which point the Server will deem the Client free again.

The interaction will continue perpetually as long as the Load Function is generating data and there are connected Clients in the network. Below we have given a system sequence diagram that depicts the above mentioned procedure in greater detail.

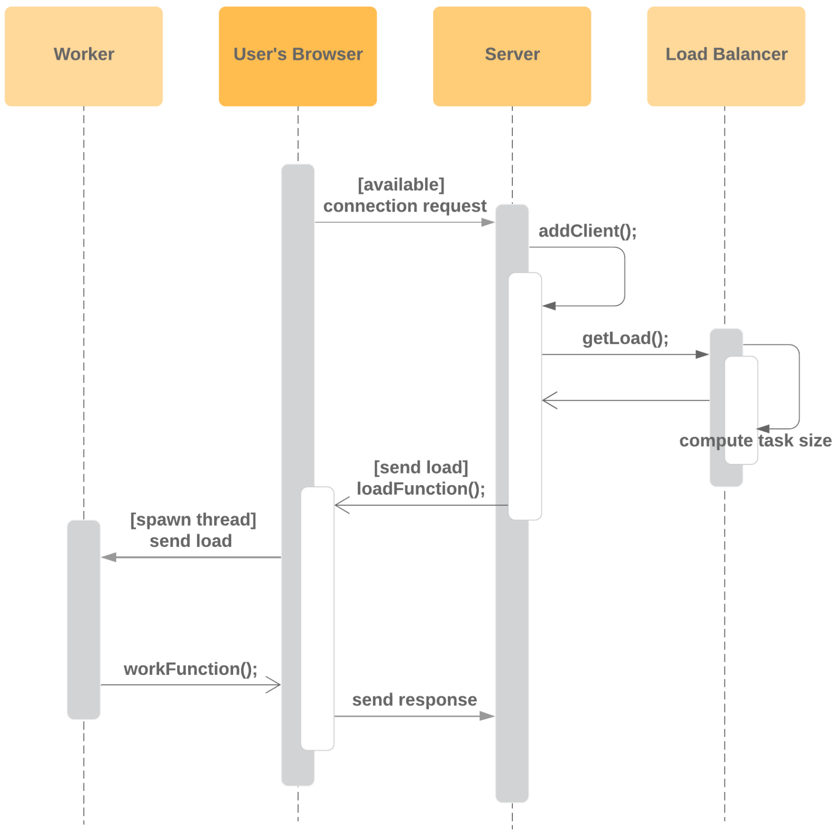


Figure 2: Architectural Sequence Diagram

In order to have our server support as many concurrent connections as possible with minimal system resources, we have implemented the aforementioned interaction sequence primarily using NodeJS events. The majority of the interaction takes place in a relative small code and simple section in which our modules listen to each other's events and react accordingly:

```
this.server = new Server({ socket, port, initialData})
.on("connection",
    this.loadBalancer.initializeClient.bind(this.
        loadBalancer)
).on("connection",
    this.clientManager.addClient.bind(this.clientManager)
).on("result",
    this.dataHandler.handleResult.bind(this.dataHandler)
).on("client-available",
    this._sendJob.bind(this)
);
```

4 EVALUATION

We performed two separate types of evaluation in our final implementation of our System. The first evaluation type is focused on testing the performance of one web-worker being spawned and communicating with the main-thread. The goal of this experiment is to show whether sending an Object from the Client to the Web-Worker and receiving it back should be considered as an additional computational cost in terms of time *(ms)* that we need to consider in our System.

The second type of evaluation is focused specifically in showing whether algorithms scale as more clients connect to the System. For this experiment we want to evaluate algorithms that are computationally heavy and algorithms that are not, along with various sizes of data being sent between the Server and the connected Clients. This experiment will show results that can indicate a specific configuration of the System in order for an algorithm to scale.

Finally we will test the performance of an algorithm being written in JavaScript and compare it to the C++ equivalent of that algorithm. By performing this experiment we can see if we can recommend alternative programmable languages that can be used and whether the alternate language gives better results overall.

4.1 WEB-WORKER PERFORMANCE

A critical aspect of running Web Workers is having fast communication between the Main thread and the Workers themselves. By having fast communication we can eliminate the additional variable of time that it takes to send information between the worker and the main thread.

In our implementation we use a Dedicated Worker to perform all the client side computations before sending them back to the server. However since there has been no previous research on the latency of sending and receiving a message from the main thread to the worker and back (e.g. Cycle), we decided to run performance tests that will give us the average Latency of a cycle.

Web-Worker Bench-marking Methodology

All tests were performed on a Mac-Book Air (13-inch, Early 2014) with 1.4 GHz Intel Core i5 and 8 GB 1600 MHz ram, using the Firefox 66.0.5 (64-bit) browser. We decided to use the Mozilla Firefox browser since it allows for serving the Web Workers directly from the file

System, instead of having the JavaScript files being fetched from the server.

Our focus is strictly on the latency. Therefore the worker script is simple and it does not have to compute any algorithmic operations.

```
/* The worker file './worker.js' */
self.onmessage = ({ data }) => {
    postMessage(data.trips === 0 ?
        /* If data.trips is 0 send termination request */
        'terminate'
    :
        /* Send the data back */
        data
    );
}
```

The benchmark script is responsible for calculating the total elapsed time of a Cycle.

```
/* Initialise worker */
const worker = new Worker('./worker.js');
let totalTime = 0,
    trips = 99;

/* Send first message to worker */
worker.postMessage({
    time: Date.now(),
    /* The Cycles we want to test the latency (e.g. 99) */
    trips,
});
/* Handler for receiving messages from worker */
worker.onmessage = ({ data }) => {
    const endingTimeMilis = Date.now();

    if (data === 'terminate') {
        worker.terminate();
        /* Latency (totalTime / data.trips) */
    } else {
        /* Elapsed time added to total time */
        totalTime +=  endingTimeMilis - data.time;

        worker.postMessage({
            time: Date.now(),
            trips: data.trips - 1,
        });
    }
}
```

Web-Worker Findings

We measured the average latency for performing a cycle with sending an object with two keys (i.e time, trips). We performed five 10x increments of cycles starting from 100:

$$n \in \{100, 1000, 10000, 100000, 1000000\}$$

cycle tests to validate the average latency of a cycle 3.

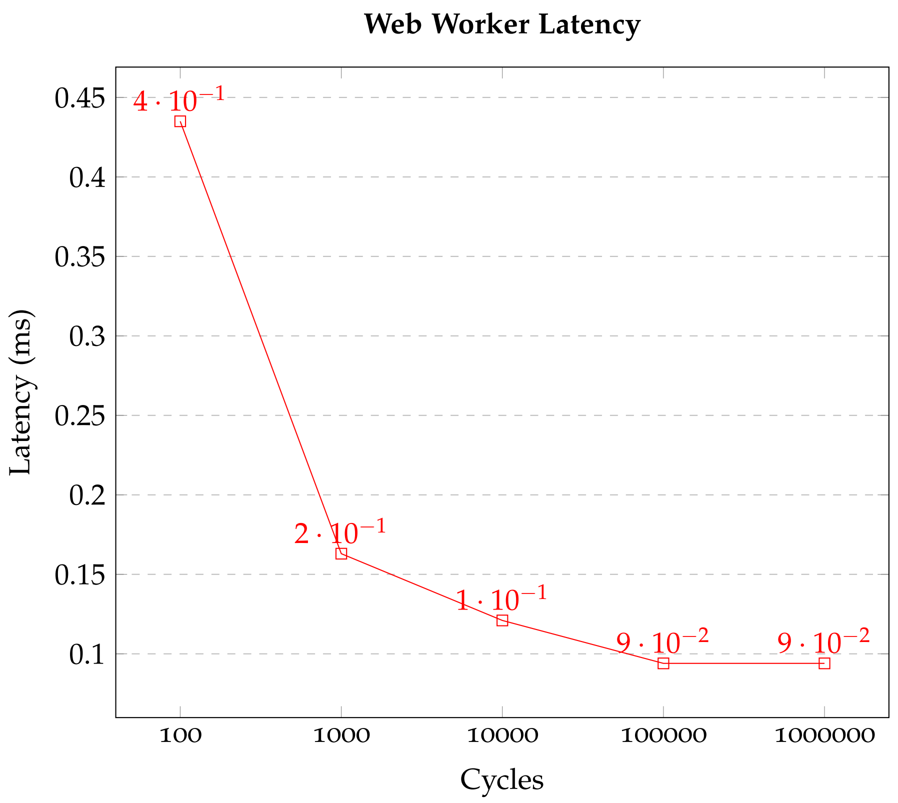


Figure 3: Average Latency to compute n cycles

The results indicate the average round trip to be dependent on the size of the trips performed. This is due to the initialisation of the thread having a range of values from $23ms - 233ms$ along with the seemingly random interference of a cycle taking $6ms - 32ms$ instead of the usual $0ms - 1ms$. From the data points plotted we can clearly see that as the size of the round trips increases the Latency decreases. While running the tests we expected the overall time to perform n cycles to grow proportionally as n grows larger.

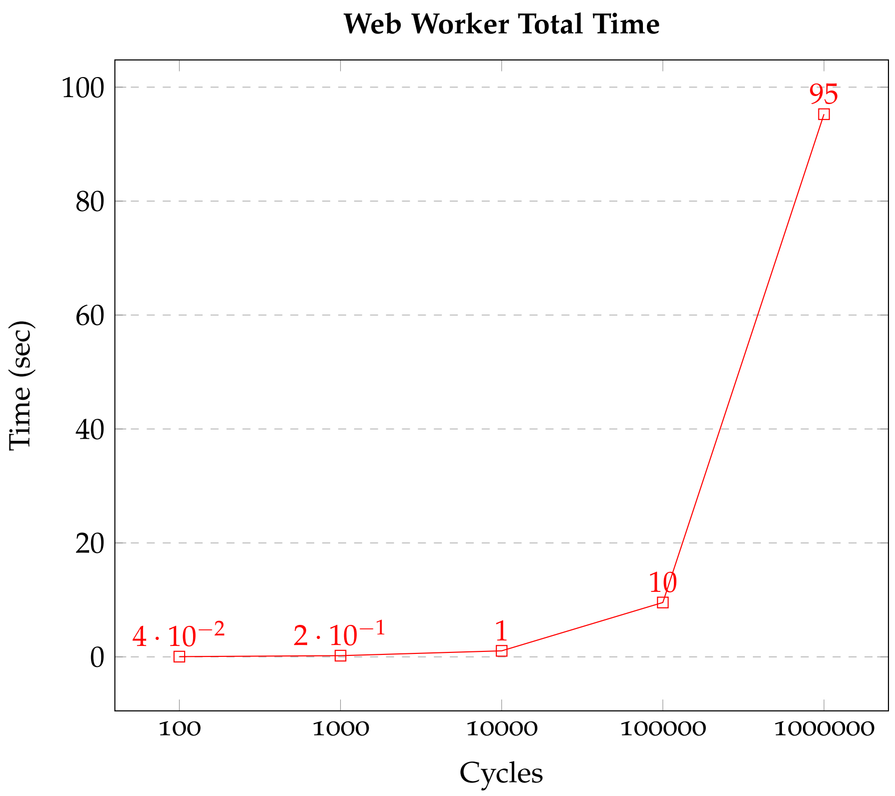


Figure 4: Total time to complete all n cycles

We can clearly see that there is an inverse relationship between Figure 3 and Figure 4. For Figure 3, as the amount of cycles are increasing the latency (ms) is decreasing. For Figure 4, as the cycles increase by 10 times, the time it takes to complete all of them, takes 10 times longer. Thus we can conclude that the communication latency of a Web Worker is fast $\tilde{0}.12ms$ and it can be treated as an invariant when measuring the performance of our client side script. If we choose to have multiple cycles between the workers and the main script, 10000 cycles or more would yield the best results in terms of average latency.

While running the tests with logging enabled, the results of each cycle severely impacted the performance of the computer overall. The browser threw errors referencing the immense load that we were putting it under just for the print statements, rendering it unusable until the task was fully completed (e.g. $n > 50000$ cycles). We have given clear instructions to not implement any printing in the worker file as it can be potentially dangerous for any Client connected.

4.2 DISTRIBUTED SYSTEM PERFORMANCE

It is crucial to measure if different computational intensities can scale as the our System grows with more connected browsers. We have successfully tested the SHA-256 hashing algorithm along with the Matrix Factorisation Algorithm. Each of the two algorithms that we are bench-marking has a different computational load as well as different amounts of data being distributed on the connected clients.

4.2.1 *Distributed System Bench-marking Methodology*

We used the Google Cloud platform [1] utilising the kubernetes clusters for all of our testing. Google cloud allows us to easily perform our experiments obtaining results that we would otherwise have to perform manually over physical computers. Each algorithm was executed on multiple connected browsers b on the Google Cloud platform:

$$b \in \{1, 2, 4, 8\}$$

For each of our connected browsers on the cloud platform we were using a vCPU (3.75 GB Memory) and a 30 GB total allocation of disc space.

In order to make the testing as concrete as possible we decided to eliminate aspects of the system that can introduce variance. Therefore we only performed the tests using the Chunk Distribution Type that is set to constantly distribute 400 Tasks. Additionally we made a small alteration to the core System functionality in order to only sent tasks after a specific amount of connected browsers has been reached. By doing so, we are able to have the correct total time of completion *(ms)* for all Distributed Tasks that were given by the Load-Function.

Environment difference

Since we are using the cloud platform we wanted to find if there is a big difference between running each algorithm on a local machine and on the Cloud.

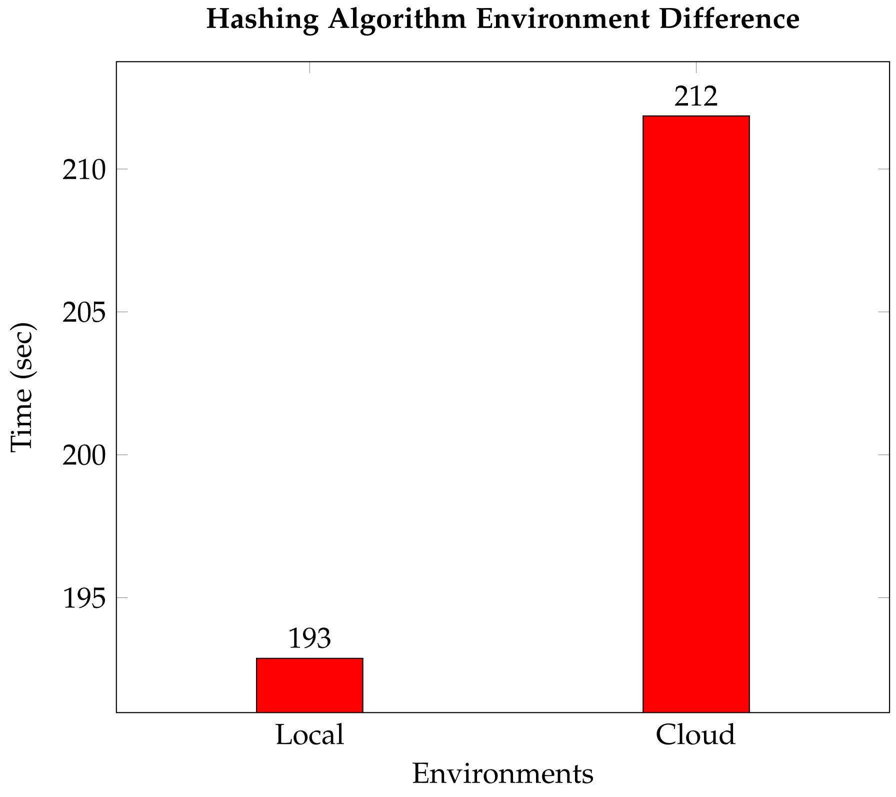


Figure 5: Completion of all Tasks for one connected client per environment

1 https://cloud.google.com

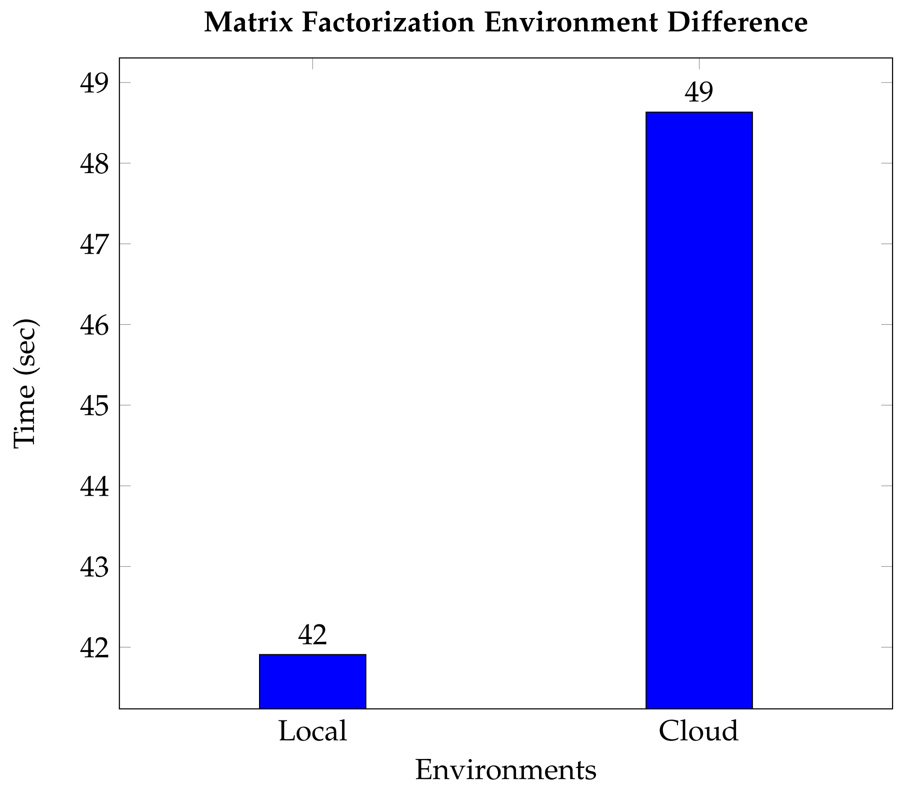


Figure 6: Completion of all Tasks for one connected client per environment

By running the Hashing Algorithm Figure 5 we see that there is a 19*ms* difference between the Cloud and the local environment whereas the Matrix Factorisation Algorithm Figure 6 has a difference of only 7*ms*. Our obtained results for the scalability of each algorithm will have this additional difference of 19*ms* and 7*ms* respectively that we could otherwise avoid by running the tests locally.

4.2.2 *Hashing Algorithm*

The first algorithm that we have implemented is the SHA-256 [2] hashing algorithm. Our entire data-set is comprised of 10000 strings with a ranging length from zero to ten characters long. We will make this algorithm computationally intensive by performing it 2000 times on each of the 400 Tasks that have been distributed as a Chunk from our Load-Balancer.

2 `https://www.movable-type.co.uk/scripts/sha256.html`

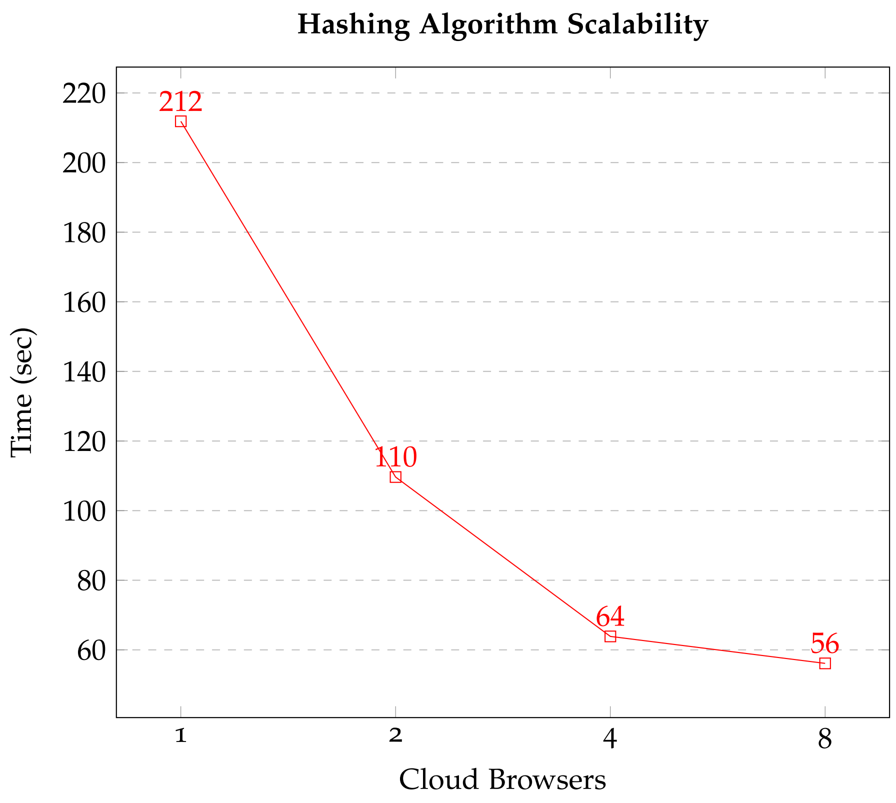


Figure 7: Total time to compute all results with 2000 hashings

As we can see from Figure 7, the hashing algorithm scales very well. As the number of connected cloud browsers increases the total time to complete the entire data-set decreases. We observer that for one connected client the completion time is 212*ms* which is almost four times higher than 56*ms* with eight connected browsers.

We observe almost 50% decrease going from one connected client to two and from two connected clients to four. However we can also see diminishing returns as the number of connected browsers increases from four connected browsers to five, with only 8*ms* of difference.

4.2.3 *Matrix Factorisation Algorithm*

The second algorithm that we have implemented is the Matrix factorisation. This algorithm has been converted from the programming language of C++ to WebAssembly [3]. The Matrix Factorisation differs from the Hashing algorithm in terms of data being sent over the network. There is a lot more data being distributed in one Task **??** and the computational complexity of the algorithm is far less intensive. Just by comparing the completion times on the Cloud environment we see that it takes 212*ms* for the hashing algorithm Figure 5 whereas the matrix factorisation algorithm Figure 7 only takes 49*ms* to complete.

3 `https://webassembly.org`

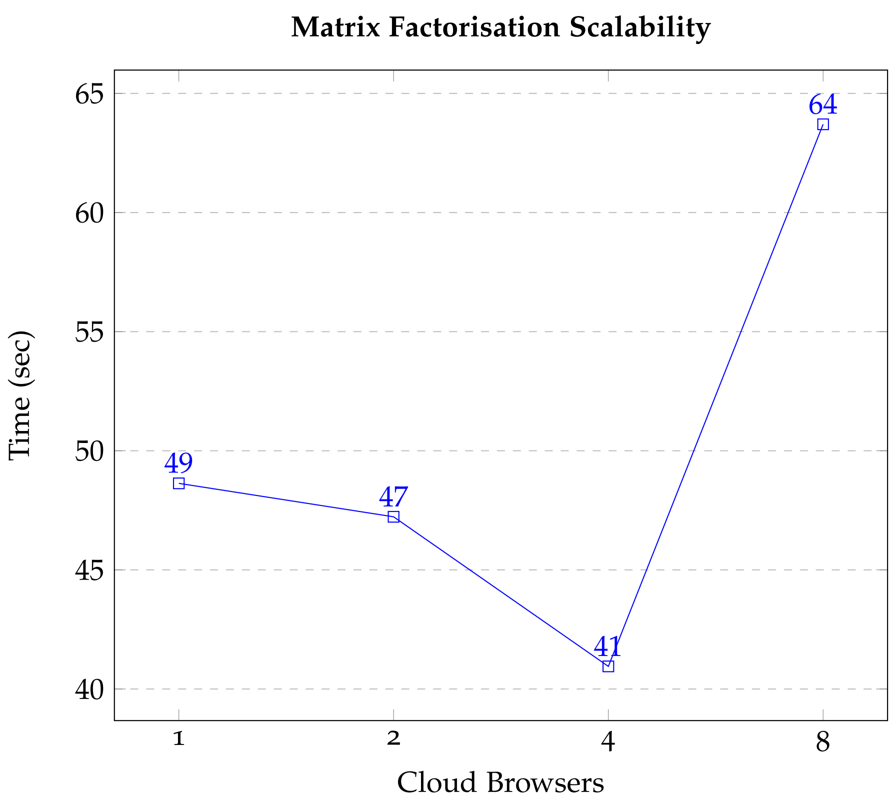


Figure 8: Total time to complete the algorithm with 400 iterations

The Matrix factorisation algorithm does not scale very well. Figure 6 shows very little performance increase by having two connected browsers where we gain a difference of just $2ms$ from having once connected browser. Where we see the most amount of improvement is with the four connected browsers where the performance improvement is $8ms$ compared to one browser. We see a performance decrease when there are 8 connected browsers. Surprisingly the total computational time is higher than just having one connected browser where the performance loss is $15ms$. This performance loss can be attributed to a network "bottleneck". The computational time of the algorithm is less than the time it takes to complete the tasks given to one client.

Matrix Factorisation Programming language

In our final benchmark we decided to compare the performance of the Matrix Factorisation Algorithm on different programming languages. We have two implementations of the same algorithm. One written in C++ and converted to WebAssembly and the second written in JavaScript. Since we saw that the Matrix Factorisation algorithm seems to have the best performance Figure 8 when four clients are connected, we decided to only compare the results between one connected client and four.

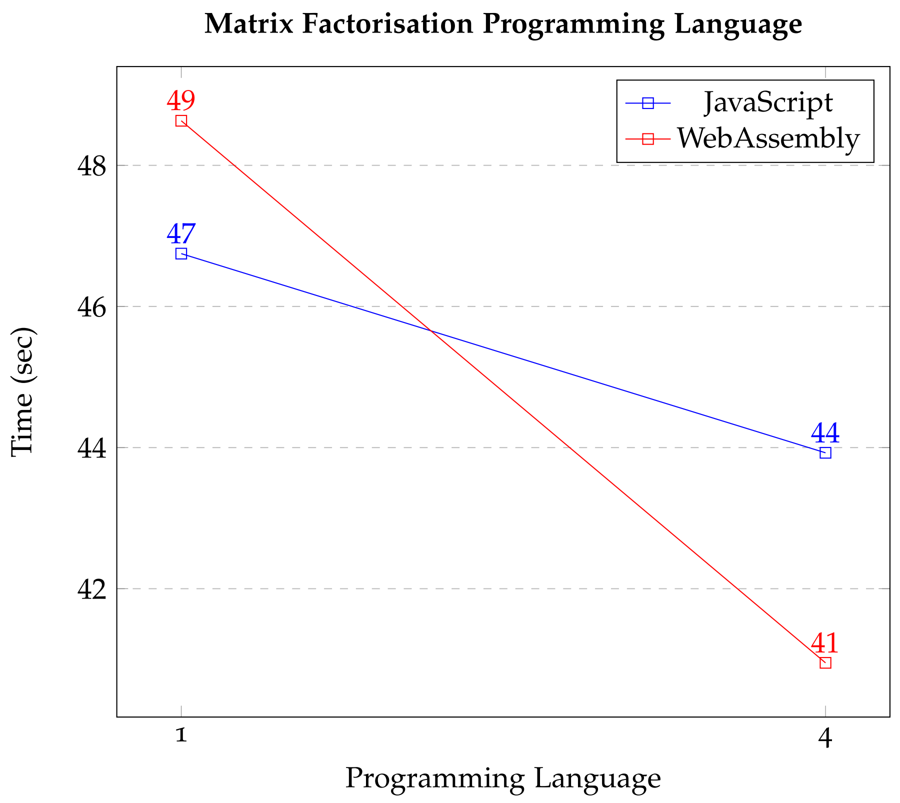


Figure 9: Total time to complete the algorithm with different programming languages

We can see on figure 9 that there is a performance increase with both algorithms when the number of connected browsers grows from one to four. WebAssembly seems to has the most performance increase of 8*ms*. JavaScript has a performance increase of only 3*ms*. Although WebAssembly seems to be having the worst performance 49*ms* with one connected client compared to JavaScript's 47*ms*, WebAssembly scaled better when four browsers were connected.

5 CONCLUSION

This paper has examined the concept of browser based distributed systems. It investigated multiple distinct browser based systems and put forth arguments for their lack of popularity and possible improvements. The ideas which were put forth as improvements, as well as recent technological advancements in browser JavaScript environments culminated in a new system built by the authors. The end result is a JavaScript package designed for distributed computing over browsers in a server-client topology.

In order to test the effectiveness of our system, multiple benchmarks were carried out. It was observed that computational load has a big impact on how an algorithm scales with the system. However, computational intensity alone, is not the only aspect introducing variance. Both of our benchmarks have a different base computational complexities after being executed on one browser. Furthermore, the total execution time includes the latency of sending and receiving the Tasks, which were not isolated. Since the less computationally intensive algorithm was sending significantly more data we expect copious amounts of latency to be introduced. Therefore we cannot concretely say that the System was performing optimally given the conditions of that specific benchmark.

Moreover we observed diminishing returns for both of our algorithmic scalability tests. The results indicate that after a specific amount of connected browsers, the performance gain is rendered to a minimum. After seeing consistent marginal increments in performance, the system will eventually reach its maximum amount of connected browsers that can achieve performance improvements. Therefore we can say that there is an indicative amount of maximum connected browsers that the System can have before yielding performance loss.

We do have promising results, however further benchmarks are needed before it can be ultimately concluded that our system is indeed scalable. We have not included a scenario where the same base computational intensity is introduced along with isolated latency. Further explorations about the relationship between distribution sizes and computational load would suggest how to provide ideal circumstances for an algorithm to reach its maximum distributed capabilities. To conclude, our research has shown that an algorithm can scale up to a specific number of connected browsers and yield an increase in performance.

6 FUTURE WORK

It is clear from the results that there is a possible limit on how many connected browsers can yield overall better performance with a server with fixed resources. Both of the algorithms we have tested reached a point of diminishing returns and even decrease in performance after a certain number of clients. Therefore a main focus point for future work can be in investigating solutions in this area. Increasing the overall efficiency of the system architecture should always be an aim, since it would lead to a decrease in the overhead introduced by each client and increase the maximum number of browsers which can efficiently contribute. However another solution can be investigating algorithms for detecting the maximum number of browsers which still brings an increase in the system performance. Future work can focus on dynamic detection of this number, since we do not know it before exceeding it, and therefore reaching an overall worse performance and limiting the connections to stay below that number in order to avoid a performance decrease.

Our project was limited to a single server architecture. However a practical way to improve scalability drastically, or even to achieve almost perfect scalability, can lie in the ability to run the task distribution on multiple servers. A major focus for future work can be an architecture with multiple server components, where new server components can be added or removed on demand to meet scalability needs. Such a system would benefit from devising an algorithm for detecting for peak scalability, as mentioned in the previous paragraph. Modern cloud computing platforms provide support for load balancing between multiple servers as well as dynamically starting and killing server instances. In order to automatize the server scalability this way, an interface can also be added through which the Server would communicate reaching peak scalability and demand more servers to be added.

The fault-tolerance system currently does not act on it if a User was found to be sending faulty results. This simplification was deemed acceptable because in theory any sort of banning can be circumvented by the User. The User can open a new socket connection or even change their IP address in order to change their identity in the eyes of the system. However banning users on the basis of such information would still introduce a practical barrier to cheating and can be useful in making it less feasible. Therefore an area of future work can be on investigating systems for banning cheating users.

WebGL[1] is a browser API for moving graphics calculations in browsers to the graphics card, similar to systems like OpenGL. We have not in-

1 `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`

vestigated the use of GPUs, which are inherently designed to have a high number of cores and can be suitable for parallel applications, mostly due to time constraints. Discovering the practical aspects of moving calculations to the GPU can be valuable.

BIBLIOGRAPHY

[1] David P Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, pages 17–19, 2003.

[2] David P Anderson. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[3] Fabio Boldrin, Chiara Taddia, and Gianluca Mazzini. Distributed computing through web browser. In *2007 IEEE 66th Vehicular Technology Conference*, pages 2020–2024. IEEE, 2007.

[4] Yuriy Brun, George Edwards, Jae Young Bang, and Nenad Medvidovic. Smart redundancy for distributed computation. In *2011 31st International Conference on Distributed Computing Systems*, pages 665–676. IEEE, 2011.

[5] Jacob Brunekreef, Joost-Pieter Katoen, Ron Koymans, and Sjouke Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157, 1996.

[6] Charles Cusack, Chris Martens, and Priyanshu Mutreja. Volunteer computing using casual games. In *Proceedings of Future Play 2006 International Conference on the Future of Game Design and Technology*, pages 1–8, 2006.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] Patricio Domingues, Bruno Sousa, and Luis Moura Silva. Sabotage-tolerance and trust management in desktop grid computing. *Future Generation Computer Systems*, 23(7):904–912, 2007.

[10] Jerzy Duda and Wojciech Dłubacz. Distributed evolutionary computing system based on web browsers with javascript. In *International Workshop on Applied Parallel Computing*, pages 183–191. Springer, 2012.

[11] Tomasz Fabisiak and Arkadiusz Danilecki. Browser-based harnessing of voluntary computational power. *Foundations of Computing and Decision Sciences*, 42(1):3–42, 2017.

[12] Tomasz Krupa, Przemyslaw Majewski, Bartosz Kowalczyk, and Wojciech Turek. On-demand web search using browser-based

volunteer computing. In *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 184–190. IEEE, 2012.

[13] Philipp Langhans, Christoph Wieser, and François Bry. Crowdsourcing mapreduce: Jsmapreduce. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 253–256. ACM, 2013.

[14] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. Mlitb: machine learning in the browser. *PeerJ Computer Science*, 1:e11, 2015.

[15] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[16] Yao Pan, Jules White, Yu Sun, and Jeff Gray. Gray computing: an analysis of computing with background javascript tasks. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 167–177. IEEE Press, 2015.

[17] Sandy Ryza and Tom Wall. Mrjs: A javascript mapreduce framework for web browsers. *URL http://www. cs. brown. edu/courses/csci2950-u/f11/papers/mrjs. pdf*, 2010.

[18] Luis FG Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 337–346. IEEE, 2001.

[19] Sean R Wilkinson and Jonas S Almeida. Qmachine: commodity supercomputing in web browsers. *BMC bioinformatics*, 15(1):176, 2014.

[20] Mikel Zorrilla, Angel Martin, Iñigo Tamayo, Naiara Aginako, and Igor G Olaizola. Web browser-based social distributed computing platform applied to image analysis. In *2013 International Conference on Cloud and Green Computing*, pages 389–396. IEEE, 2013.