MŰEGYETEM 1782

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Jánoky László Viktor

PARTICIPATORY COMPUTING PLATFORM IN A DYNAMIC WEB ENVIRONMENT

KONZULENS

Dr. Ekler Péter

BUDAPEST, 2016

Table of Contents

Abstract 4
1 Introduction 5
1.1 Overview 5
1.2 Related work 6
2 Problem statement 8
2.1 Conceptual problems 8
2.1.1 Client adaptation 8
2.1.2 Application creation 9
2.1.3 Setup and maintenance 9
2.2 Technological problems 9
2.2.1 Reliability 9
2.2.2 Consistency 10
2.2.3 Security 10
2.2.4 Scalability 11
3 The proposed solution 12
3.1 Basic concepts 12
3.1.1 Task 12
3.1.2 Task Description 13
3.1.3 Task Program 13
3.1.4 Task Data Part 13
3.1.5 Task Result 13
3.2 Participants and responsibilities 14
3.2.1 Clients 14
3.2.2 System owner 14
3.2.3 Application owner 14
3.3 Scope and typical deployment 14
3.3.1 Standalone deployment 14
3.3.2 Secondary deployment 15
3.4 High level architecture 15
3.4.1 Backbone technologies 16
3.4.2 System outline 17

3.4.3 Working modes 18
3.5 Components and technologies 19
3.5.1 Task Master 20
3.5.2 System Configuration 23
3.5.3 Centralized Task Group 24
3.5.4 Peer-to-Peer Task Group 25
4 Using the system 27
4.1 Use cases 27
4.1.1 Scientific uses 27
4.1.2 Commercial uses 28
5 Measurements 30
5.1 Goals 30
5.1.1 Proving feasibility 30
5.1.2 Examining behaviour 31
5.2 Measurement configuration 31
5.2.1 Hardware configuration 31
5.2.2 Software configuration 32
5.2.3 Executed tests and measured data 33
5.3 Results and evaluation 35
5.3.1 Feasibility test 35
5.3.2 Working mode comparison 37
6 Overview and future work 42
6.1 Summary 42
6.2 Future research directions 43
Bibliography 44
Appendix 46

Abstract

The concept of a building a distributed computing system from volunteer nodes has come up multiple times as the computing and networking technologies advanced. Some successful implementations of the concept were done, and these systems enabled some great scientific achievements to be made.

However, the concept did not get adopted by a wide audience, both the number of participants and the beneficiaries of these systems are dwarfed by the size of their potential target groups. Even the most successful of these systems, the SETI@home project, built on the BOINC platform, could only reach around 1.5 million users during its 17-year lifespan.

The reasons behind the low percentage of adaptation are mainly technical ones. Creating and running a distributed participatory computing system is nontrivial, participating in one requires deliberate intention and moderately complex steps from a user.

In the dissertation we propose a solution to build a platform which enables much larger adaptation by building on proven, widely used and supported technologies and by using a different approach for users to participate.

The system that we show in this research is integrated into the web, allowing users to participate simply by visiting a website. This way task givers don't need specialised infrastructure to facilitate the computing and it enables a much larger adaptation and different uses for the system.

This could possibly include scientific computational projects on a much larger scale or even commercial computing services provided by major websites, using the computational power of their visitors. Our solution could provide an alternative way of funding besides web based advertisements, with much less user experience deterioration.

1 Introduction

1.1 Overview

A participatory computing system is a distributed computing system where the participants are contributing processing power and storage to form a large, capable system.

A similar concept is volunteer computing, where the participants provides their resources voluntary, explicitly adding their resources the pool. However, in a participatory system, the intent from the participants is not important, it can be a secondary function of a system. For example, the main service could be a website, where visitors will also participate in the computing system.

The previous example points to one of the greatest weaknesses of volunteer systems, the clients have to have an explicit intention to contribute. In order to achieve this, they have to know about the system, they must know how to join and how to take the necessary steps for doing that. This severely limits the number of potential users, and requires considerable effort from the system owner to grow their platform.

In this dissertation we outline a participatory system, built on web technologies that requires no extra effort form the clients to participate. In practice joining can simply be done by visiting a specific website. This allows for a much larger reach than traditional volunteer systems. Instead of complicated user actions, a simple consent would be enough to join, much like the consent given when dealing with cookies, in accordance with the EU ePrivacy directive Article 5(3) [1].

We also investigate how accessibility can be improved from the perspective of the system owner. Showing a model where the owner of the system and the user (who wants to use the system for computational purposes) can be a different entity, and what benefits this separation has.

Ultimately, given a system like this, participatory computing could become a mainstream technology without requiring extra effort from the participants. Taping into the vast unused resources of web clients, previously unimaginable scales could be reached in computational resources.

In this paper we analyse what kind of obstacles does a web based participatory system has to face and we propose solutions how to overcome them. We prove the feasibility of the solutions by demonstrating and evaluating a working proof-of-concept system and we show measurements related to it.

1.2 Related work

As volunteer computing is a form a participatory computing, the advances of this field had a rather significant effect on our system. Some of the most notable and successful volunteer computing projects are built on the BOINC (Berkeley Open Infrastructure for Network Computing) platform [2].

BOINC is a software system with the goal to make it easy for scientists to create and operate volunteer computing projects. Supporting a wide range of possible applications (even with large computational, storage or communication requirements), the platform served and continuous to serve many successful projects. For example, SETI@home [3] or Folding@home [4] to name a few.

Our goal, that we show in this paper, is different from BOINC's, but some very interesting lessons can be learned about how it deals with common problems in public computing systems. For example, the handling of malicious clients or deviating numerical results because of different execution platforms, has some great insights and solutions.

In [5] the authors introduce project Bayanihan, a web based volunteer framework which relies on widely adopted technologies such as web browsers and Java. They created a flexible software framework which could be programmed in Java and joined by the clients using desktop Java applications or browser based Java applets. Bayanihan provides a framework for building and composing a volunteer system by extending the given components. This means there are multiple, specialized deployments of Bayanihan systems, each with their specialized goals and tools.

The main difference between this work and ours is that we intend to create the platform for executing tasks, without the need to write application specific modules. In our system multiple different tasks can co-exist on the same deployment, using the same resources, while executing completely different goals.

Several other projects were carried out with the intention of creating an easy-to-access, web based volunteer system. In [6] the authors suggest a system (POPCORN

Market) that could be used for global scale distributed computing. Their platform provides a market based mechanic for buying and selling CPU time. Just like Bayanihan, POPCORN Market also relies on Java from the client side.

In [7] the authors demonstrate their web based volunteer system; WebCom, which is also built on Java. One of the most interesting features of this system is its capability to scale dynamically. They achieved this feat by making it possible for clients to become masters, central components in their architecture, tending several clients. With clients becoming masters they could build a hierarchical structure of nodes, increasing the whole system size and capability.

In their study about the computational and storage potential of such systems [8] the authors concluded that not only processing power, but storage place can also be a major resource in a volunteer system.

Our system leverages both of these resources, but not necessarily in a direct way. Storage for instance can be used for storing and forwarding data related to the computation between clients.

From a non-technical standpoint, the need to ease the joining process from the perspective of the participant was already recognized in these past projects. Using the web as a platform to achieve this goal was considered several times. Ultimately the downfall of browser based Java applets (mainly because of past and perceived security issues), meant that the seamless client joining was not possible until recently.

2 Problem statement

In this section we discuss the conceptual and technological challenges the system must overcome if it's to succeed in reaching a large adaptation rate. Showing the sources of these problems and the reasons why they effect adaptation, we can design our system to mitigate their effects.

2.1 Conceptual problems

The problems that we are seeking to answer in this paper are not strictly bound to the distributed nature of participatory systems. We would like to answer the question; how to make participatory systems wide-spread, common tools. In this section we list some conceptual problems regarding large scale adaptation of these kind of systems.

2.1.1 Client adaptation

Comparing the number of participants in volunteer computing projects to the total number of internet connected PCs it is easy to see, that there are still vast computational resources left untouched by these systems.

Behind the lack of client adaptation there are several factors, one of them is the already stated user accessibility issue. For a user to join a traditional volunteer computing network the following conditions have to be true.

- They know about the system.
- They have an intention to join.
- They have the means and knowledge to join.
- They participate at least for a minimum amount of required time.

Making sure that these points are actually fulfilled is not an easy task. It requires a significant amount of resources to make the potential users aware of the system, persuade them to participate and teach them how to do it, if necessary.

2.1.2 Application creation

Another thing to consider is accessibility from the perspective of project owners running applications in a participatory system. Writing an application for a massively parallel system like this has its own limitations, mostly inherently parallel problems can be solved using this technique. If the original problem can't be parallelized well, the usage of the system won't provide much gain in terms of computation.

Most of the time the goal of the owner is very specific, requiring a custom application for each purpose, like in the case of SETI@home, where the goal is to analyse radio signals. The burden of creating this goal specific application also lies on the owner. At most what we can do is help them by providing a standardised set of tools and interfaces.

2.1.3 Setup and maintenance

Setting up and maintaining a volunteer system also have costs associated with it. Maintaining the infrastructure, providing tasks to clients, scheduling the task requires effort from the owner.

This effort consists of two main parts. One is providing the necessary computational resources for administrational purposes in the system, for example task trackers and result storage. The other part is ensuring that the system has enough clients to work reliably and profitable in terms of administrational overhead vs. gained performance.

2.2 Technological problems

The problem of building a large, distributed, participatory/volunteer computing system has been studied form a technical standpoint for a long time now. To the main problems such as reliability, scalability, consistency there are multiple proven solutions.

2.2.1 Reliability

Reliability problems arises from clients joining and leaving the system on their own will. When the number of clients is under a critical level, the operability of the system cannot be ensured without adding artificial clients.

The problems of clients leaving before finishing tasks can be solved by assigning tasks redundantly, monitoring their state and if necessary reassigning them.

Thanks to the law of large numbers, the unpredictable nature of client fluctuation decreases as the total number of clients increase. Statistical models can be deployed to predict large scale variations in time, improving task allocation and mitigating problems arising from client fluctuations.

Client profiling could also help with reliability and efficiency, assigning tasks to clients who are more likely to complete them helps minimizing the number of failed and reassigned jobs.

2.2.2 Consistency

Consistency problems can arise by having different clients coming to different conclusions on the same problem. This can be caused by an intentional attack or different hardware properties of different clients.

The effects of intentional attacks can be mitigated by detecting the malicious clients in the system and removing their calculations from the results pool. However, because of the nature of the system, if the attackers outnumber the valid clients it becomes possible to reach a false conclusion. This type of attack is very resource intensive from the perspective of the attacker, and becomes much harder to perform undetected as the scale of the system grows.

Ultimately attacks aimed at creating false results can be entirely mitigated by assigning authoritative clients, trusted clients who are guaranteed to return valid results. To do this for the entire data set however requires a significant amount of resources, somewhat invalidating the whole concept of using participatory computing.

2.2.3 Security

Besides false result attacks there are other security concerns regarding large scale participator systems. Security issues must be investigated from both the client's and the application owner's side.

On one hand the client software downloads and executes code from the internet on the user's machine, without them knowing what it exactly does. Therefore, the client software must ensure that the code is from a trusted source and that it does not have any harmful intent on the user's machine.

From the application owner’s perspective, the data and programs they send to the clients for execution cannot be expected to remain private. This for example rules out most of the cryptographic tasks where the owner wants to keep the data private. This could also have effects on commercial uses, like graphics rendering or the processing of any kind of sensitive data.

2.2.4 Scalability

In order for the system to be able to serve large number of clients it must be easily scalable. With the increased number of clients, the number of administrational tasks also rises, this is the main factor behind the scalability concern.

By utilizing some clients for administration purposes, the system can scale with the resources added. The inherently distributed nature of typical tasks executed also helps with scalability as task-bound operations can be separated from each other.

3 The proposed solution

To solve the previously shown issues we propose a system with accessibility as its main feature. This is considered from both the client’s and application owner’s perspective and allows for a much larger client adaptation.

From the client’s perspective accessibility is reached by placing the entire client software in the browser. Using the built-in tools provided by modern web browser we built a participatory client, which does not require any explicit action from the user when participating in the system. This allows for much easier and faster joining from the client’s side and eases the burden of providing guides and tutorials from the system owners side.

The application owner’s tasks are simplified by providing a standardized yet freely extendable interface for task creation and execution. The details of this mechanism are shown in the upcoming sections.

3.1 Basic concepts

The system is built around the following basic concepts in order to facilitate easier application creation and unified execution.

3.1.1 Task

The unit of work in the system is a task. A task is made of data and program executed on parts of this data. Because of the nature of distributed systems, a typical task is usually a very well parallelizable problem.

When adding a new task to the system, the owner can specify parameters about when is the task considered finished. For example, one of these parameters is the *confidence level*, which determines that at least how many different clients must execute each data part (and come to the same conclusion) for it to be accepted.

Another notable parameter, which depends on the deployment, is whether parts of the results should be checked by the system. This is done by executing the same program on the same data (called *authoritative check*). As this is a rather resource intensive task it can’t be done on the whole *Task*. Otherwise there would be no need for clients to calculate the answer, the system would authoritatively calculate every answer.

In the system there can be 3 states of a task:

- New – meaning it has been newly created and not yet been sent out to clients.
- Active – these tasks are currently under execution by clients.
- Finished – when the number of valid results reach the confidence level for each *Task Data Part*, a *Task* is considered finished, therefore not assigned to clients anymore.

3.1.2 Task Description

A *Task Description* is a minimal description of a task, it contains its priority, and the number of data parts the task has. Information contained in the task descriptions may be used by the system or the client depending on the working mode. Priority is used when choosing which *Task* to execute in case of multiple concurrent active tasks.

3.1.3 Task Program

A *Task Program* is the code executed by the clients on the *Task Data* parts they receive. The program gets two input arguments, the *Task Data* itself, and its index in the task it belongs to. The *Task Data* can be of any kind mandated by the current application.

Each *Task Program* runs separately, there is no way for them to communicate or synchronize by any system provided means.

3.1.4 Task Data Part

The *Task Data Part* is the data portion of the *Task* on which the program operates on. Each client receives at least one *Task Data Part*, runs the program on it and then returns the *Task Result*.

3.1.5 Task Result

The *Task Result* is what the clients generate for each *Task Data Part* by executing the *Task Program*. It contains the original *Task Data Part's* index and the computed result data. This data can be also of any kind and size required by the actual application of the system, it only has practical limitations.

3.2 Participants and responsibilities

In order to create an easy-to-use, easy-to-join participatory system, we separated the current client-server relations to three different actors and evaluated the requirements for each of them. This section identifies this three main and lists their responsibilities.

3.2.1 Clients

The client is an actor who has computational resources which can be added to the system. We do not presume any special knowledge or intention from the client.

3.2.2 System owner

The system owner is an entity who owns and maintains the infrastructure required to facilitate the distributed computing between clients. The system owner themselves doesn't necessarily uses the system.

3.2.3 Application owner

The application owner is an entity who has a usage scenario for the system. This could be a research institution, a commercial company or even a private person.

3.3 Scope and typical deployment

Our system has two usage concepts, from a technical standpoint they are very similar, but the tasks and challenges associated with them are different.

3.3.1 Standalone deployment

The system can be deployed as a standalone deployment, where the owner hosts the system as a primary service. Users are visiting web sites that are specifically designed to let them join the computational network. This is very much akin to the traditional usage model of volunteer systems, with all its drawbacks and benefits.

This kind of deployment is preferable in case the system owner also owns, or at least have control over the client hardware. In this case the client accessibility helps with minimizing administrational costs when setting up the system.

3.3.2 Secondary deployment

The other usage concept is called *Secondary Deployment*, in this case the system owner has a primary service which attracts the users (e.g. a popular website), and the system is deployed parallel to it. In this mode clients may not even know about participating in the system, if the primary service's terms of service permit it.

Secondary Deployment may raise a few concerns about user privacy, security and morality. Is it ethical to run code on a user's computer without their explicit knowledge of what it does? On one hand it is already happening, websites download all kinds of analytics and tracking scripts to their user's computers, mainly intended for more efficient advertising. In [9] the authors show this as the process of privacy diffusing on the web.

Contrary to the initial belief, the proliferation of secondary deployment participatory systems, could actually help with privacy concerns on the web. In the modern web, a free to visit website has only one good to sell, their users. It may sell space to advertisers or exchange their user data for services like analytics, but ultimately it targets the person sitting behind the computer.

By enabling web content creators to harness and lease their visitor's computational resources, we could provide an alternative source of income. This could very easily be beneficial for both parties, as in the current scheme not only the user is taxed with distracting advertisements, but these usually have high performance and network impact on the client's computer.

3.4 High level architecture

One of the most important considerations when designing the system's architecture was the scalability aspect. We wanted to minimize administration costs associated with maintaining the system, while retaining the maximum number of possible clients.

Usability and extendibility was also a main concern, we wanted the system to be easily available for both the owner and system and its users. From the application creator's perspective, the ability to easily use the system while being able to extend it is an important factor.

Considering these points, we opted to use a loosely coupled, micro-services based architecture as the high level pattern for our system.

3.4.1 Backbone technologies

Usually when designing a new system, the basic architecture comes first and technologies are selected later to suit the requirements stated by the previous step. In our current case however the technologies we use, has a significant effect on the structure of our system. That is why we have to mention a few before further detailing the architecture.

To satisfy the user accessibility requirements, we chose to build the system on web technologies, enabling users to join from their preferred browsers. In our choice of language, the main driving points were penetration, cross-platform support and availability. Considering these points, we choose JavaScript, which is supported by all major and minor web browsers, used in everyday web and requires no user interaction to run.

With the advent of modern JavaScript engines, like Google's V8 or Mozilla's SpiderMonkey [11] performance is no longer considered a major drawback of the language. In case of naïve implementation, modern JavaScript has near native code levels of performance.

It also worth noting that many features required by participatory systems like networking, storage and execution are readily available in a modern web browser, with standardized APIs. Networking can be done over HTTP, Web Sockets or even WebRTC [12]. Thanks to HTML5 [13], for storage we have local storage, IndexedDB or cookies and execution in the background is supported by Web Workers.

System-wide technological uniformity is something what was really hard to achieve until now. Thanks to the performant JavaScript engines and Node.js, it became possible to write high-performance server side applications in JavaScript [14]. This factor was also a major point when we considered the technological choices for our system.

3.4.2 System outline

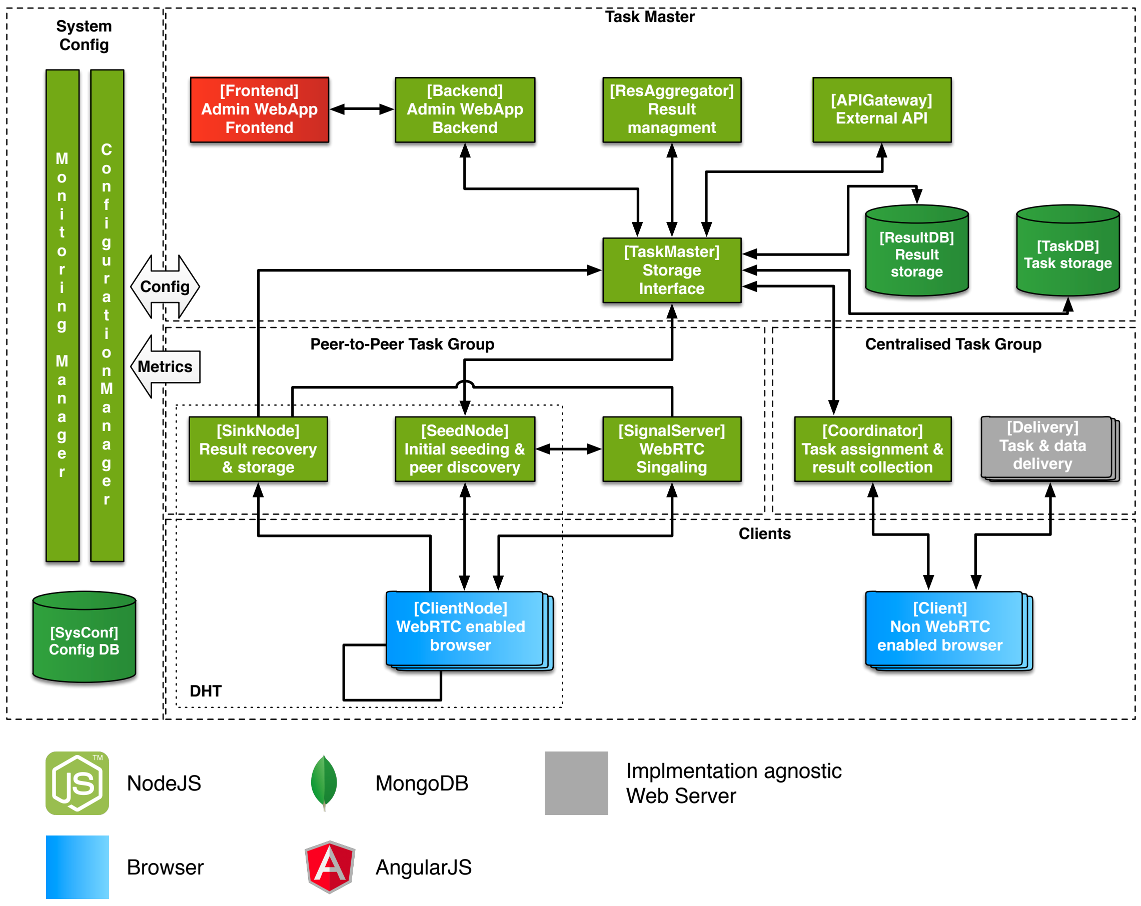

System Config
Monitoring Manager
ConfigurationManager
[SysConf] Config DB
Config
Metrics
Task Master
[Frontend] Admin WebApp Frontend
[Backend] Admin WebApp Backend
[ResAggregator] Result managment
[APIGateway] External API
[TaskMaster] Storage Interface
[ResultDB] Result storage
[TaskDB] Task storage
Peer-to-Peer Task Group
Centralised Task Group
[SinkNode] Result recovery & storage
[SeedNode] Initial seeding & peer discovery
[SignalServer] WebRTC Singaling
[Coordinator] Task assignment & result collection
[Delivery] Task & data delivery
Clients
[ClientNode] WebRTC enabled browser
[Client] Non WebRTC enabled browser
DHT
NodeJS
MongoDB
Implmentation agnostic Web Server
Browser
AngularJS

Figure 1 System architecture

The components in the architecture can be categorized to three main groups, *Task Master*, *Task Group* and *System Configuration*.

3.4.2.1 System configuration

The *System Configuration* module is responsible for storing configuration values for each component in the system and maintaining information about the current system state.

These two functions are served by two different applications; *Configuration Manager* provides configuration data by a pull model for other components. The *Monitoring Service* listens for incoming metrics and information form components declared in the configuration manager.

3.4.2.2 Task Master

The group of components responsible for task management, it includes an administration component, made of an Angular2 web application frontend and the backend, a Node.js based server, using the express framework.

The *TaskMaster* also have a *TaskDB*, which stores *Tasks* and a *ResultDB* which stores completed results. The implementation of these databases can vary, but the contents are accessed using the *StorageInterface*, this enables a greater flexibility for the system. For the current proof-of-concept (PoC) deployment, we used MongoDB [15].

Also for processing completed data we use the *Result Aggregator* component, which works on the data stored in the *ResultDB*.

3.4.2.3 Task Group

The group of components responsible for *Task* execution and client management. The *Task Group* usually concurs with a deployment instance of the system. For example, the clients of a website with a *Secondary Deployment* system can make up a *Task Group*. It has two distinct working modes which can run parallel, this behaviour is detailed in the next section.

3.4.3 Working modes

Depending on client types and technology support a *Task Group* could either work in a distributed (peer-to-peer) or centralised way. These two working modes are not mutually exclusive, when both are enabled, we call it a hybrid working mode.

3.4.3.1 Distributed mode

The *Distributed Task Group* shifts the responsibility of storing and assigning jobs, resources and results to the clients. The common channel for clients to access this data is based on the JavaScript implementation [16] of the Kademlia Distributed Hash Table (DHT) [17]. The connection between clients (nodes in the DHT) are facilitated over WebRTC Peer connections [18]. In order to build these peer-to-peer connections a *Signal Server* is required. The signalling is done over Web Socket protocol.

When joining the DHT, the entry point for the clients are the *Seed Nodes*. These nodes also serve as the injection point for *Tasks* and *Task Data Parts* into the DHT. Results are constantly collected and stored trough the *Storage Interface* by *Sink Nodes*.

Both *Seed* and *Sink Nodes* are implemented by Node.js server applications, using WebRTC support library for Node [19].

Unfortunately, at the time of writing this paper (2016 October) WebRTC is not universally supported by all major browsers. Even in supported cases many of the features are missing or incomplete, some of the implementation details are different from browser to browser. Nevertheless, this working mode of the *Task Group* has much larger scaling potential with minimal infrastructure costs compared to the centralized version.

3.4.3.2 Centralised mode

The *Centralised Task Group* on the other hand uses a traditional client-server architecture, when a client joins the system, it communicates with the *Coordinator*. The *Coordinator* assigns the *Task* and *Task Data Parts* to the client, which receives them either from the *Coordinator* or retrieves them through the *Delivery* component (depends on the configuration and size).

The *Delivery* component handles the serving of larger static resources. Because of the stateless nature of these resources, this component can be scaled horizontally using traditional techniques employed in web environments, like using a content delivery network (CDN).

When finished with the *Task Data Parts*, the clients upload their results through the *Coordinator*, which stores them in the *Result DB* (through the *Storage Interface*).

3.4.3.3 Hybrid mode

The two working modes are not mutually exclusive, it’s entirely feasible to have a scenario where both modes could be used in parallel. Clients supporting the *Peer-to-Peer* mode could join the *Distributed Task Group*, while older clients without the necessary support could fall-back to the *Centralised Task Group*. This is the behaviour implemented by the hybrid mode.

3.5 Components and technologies

The system is made up of several loosely coupled components, each encompassing a specific functionality. This micro services architecture allows for much better scaling and the fine tuning of different system components based on load.

From the technological side accessibility from both client and developer perspectives was our main concern. This mandated the usage of common, widely supported technologies. In the client therefore we had to use JavaScript as the basis of our system. For technological uniformity, we chose to create the server side components also in JavaScript with the support of JavaScript backed databases, namely MongoDB.

Cross component communication is mostly done over HTTP or Web Sockets, while peer-to-peer client connections are facilitated over WebRTC.

3.5.1 Task Master

The components in this group are responsible for administrational tasks and task management.

3.5.1.1 Task DB

The Task DB is a database which is intended for *Task* storage. Because of the nature of the system, this data is frequently read but sparsely written. Read operations originate from clients who access their issued data parts and programs, while the database is only written when the task states are changed or a new task is assigned to the system.

3.5.1.2 Result DB

The *Result DB* stores the execution results for *Tasks* computed at the clients. These results are continuously written in the database as more and more clients execute the *Task*. For *Task Results*, where enough clients have completed the calculations, the *Result Aggregator* creates a final result. *Task Results* associated with completed *Tasks* can be removed from this database when the final aggregation is over.

3.5.1.3 Storage Interface

The storage interface is a thin layer over the *Task DB* and *Result DB,* meant to provide an implementation agnostic way for other components to access these resources. It provides RESTful API [20] over HTTP for basic entities in the system, like *Task, Task Data Part, Task Program, Task Result.* As part of the basic services, these entities are validated then saved for future retrieval.

For our current system we used MongoDB for the databases but thanks to the *Storage Interface* we can change the backing infrastructure to adapt to the specific requirements for the actual deployment.

The *Storage Interface* application is written in JavaScript for a Node.js execution environment, using the Express framework **Error! Reference source not found.** for structuring and serving requests and Mongoose [22] for database access.

3.5.1.4 Admin Backend

The *Admin Backend* is the server-side part of the *Admin Web App*. This application is intended for providing tools for system and task management. Here the user can see information about the *Tasks* currently in the system, view their results and see their progress.

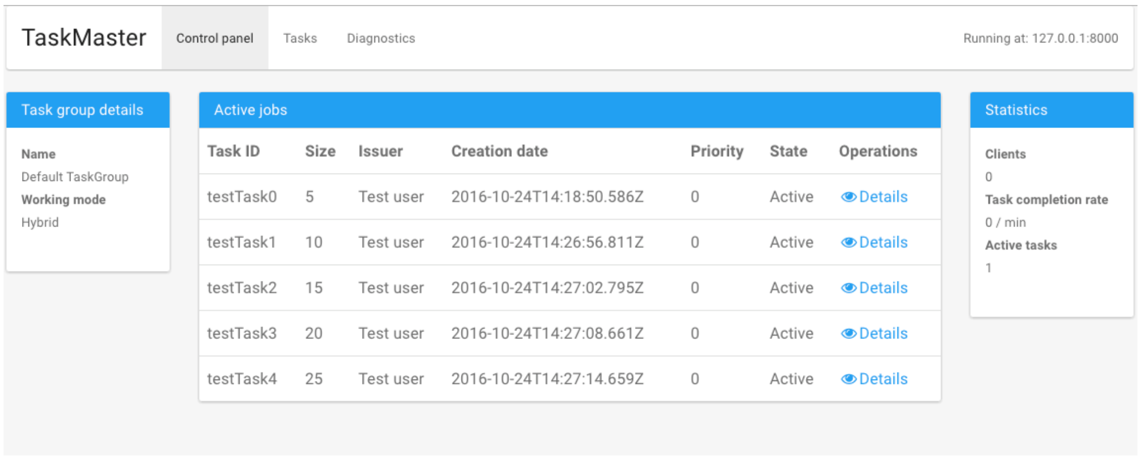
It also allows for viewing and editing system wide configurations managed through the *Configuration Manager* component. Statistics and current status collected by the monitoring service is also displayed in the application.

The *Admin Backend* is written in JavaScript for Node.js and uses the Express framework.

3.5.1.5 Admin Frontend

The *Admin Frontend* is the user facing side of the *Admin Web App*. It features a modern, responsive UI and is based on Angular 2 and Bootstrap.

This component diverges from the common technological stack, instead of JavaScript it's based on TypeScript [23], which is a superset of JavaScript and also compiled to JS. Angular 2 supports both languages, the decision to use TypeScript instead of JavaScript was based on the availability of documentation for the framework.



Running at: 127.0.0.1:8000

Task group details

Name
Default TaskGroup
Working mode
Hybrid

Active jobs

Task ID	Size	Issuer	Creation date	Priority	State	Operations
testTask0	5	Test user	2016-10-24T14:18:50.586Z	0	Active	Details
testTask1	10	Test user	2016-10-24T14:26:56.811Z	0	Active	Details
testTask2	15	Test user	2016-10-24T14:27:02.795Z	0	Active	Details
testTask3	20	Test user	2016-10-24T14:27:08.661Z	0	Active	Details
testTask4	25	Test user	2016-10-24T14:27:14.659Z	0	Active	Details

Statistics

Clients
0
Task completion rate
0 / min
Active tasks
1

Figure 2 Admin Web Application Frontend

3.5.1.6 Result Aggregator

The result aggregator periodically checks the *Result DB* for changes in task results. When the criteria of a *Task* being marked as completed met, it aggregates the results, and set the *Tasks* state to finished.

The requirement for a *Result* to be accepted as the correct result is that the number of matching answers is at least the value of the Task's configured *Confidence Level* parameter.

The basic principle of defending against false results is to collect every result from every client without giving the client any feedback about whether their response was accepted. When enough results are accumulated for each *Task Part,* the aggregator checks them for consistency, if there are conflicting results for a part it tries to resolve the conflict by a simple majority decision.

If it's not possible to resolve the conflict the *Task* will stay in an Active state, with the number *Data Parts* reduced to those with conflicting results. If the results are not conclusive after a specified number of tries the *Task Aggregator* may finish the task.

After pruning the incorrect results from a *Task Data Part,* the offending clients get a negative score, in case of future conflicts the results of this client is weighted by this score. If this score reaches the pre-set limit all results by this client is discarded.

If authoritative checks are enabled, the *Result Aggregator* randomly chooses *Task Data Parts* and executes the *Task Program* on them. The results are then compared to the ones given by clients, changing client scores and accepted results accordingly. As these checks can be quite resource intensive in large numbers the usage of this feature is optional.

The result aggregator is also a Node.js application, which can be deployed in parallel for better performance.

3.5.1.7 API Gateway

The *API Gateway* is responsible for providing a programmable interface for the system. Through this gateway it's possible to create new tasks, retrieve results and manage the system state. It's responsible for authentication and authorization, providing access control to system resources.

From a technical standpoint, the API gateway is a Node.js application with Express framework and Passport library for authentication, serving requests over HTTP.

3.5.2 System Configuration

The *System Configuration* module contains applications responsible for system management and maintenance. The responsibility of this module is significant because the micro services style architecture makes it difficult to maintain a consistent system state.

3.5.2.1 Configuration DB

The *Configuration DB* stores the configuration settings for the system, it's mostly read and rarely written, making scalability easier. As it's a central component, a redundant deployment is preferable. Our choice in database was MongoDB, which provides this redundancy with the usage of replica sets.

3.5.2.2 Configuration Manager

The configuration manager is responsible for storing and distributing centralized configuration for each application component. While the configuration manager describes the static structure of the system, actual component states are not stored in it, therefore it's mostly stateless.

When a component starts, the first thing it does is downloading the relevant configurations from the *Configuration Manager*. This makes it a Single Point of Failure (SPOF), so in order for higher availability it's recommended to redundantly deploy it. Luckily it's mostly statelessness nature helps in great deal achieving this, multiple instances can be deployed over the same replicated *Config DB*.

The *Configuration Manager* is a Node.js application with Express framework and Mongoose for data access.

3.5.2.3 Monitoring Manager

The *Monitoring Manager* is responsible for maintaining information about the system's dynamic state. It collects information and metrics from each system component by providing endpoints for their periodic updates.

The components receive the address of the *Monitoring Manager* in their configuration at start up, and periodically send data to it. The *Monitoring Manager* stores and aggregates this data, in case of missing reports the component’s state is noted as unavailable and corrective action can be taken by the system administrator.

The Monitoring Manager is a Node.js application using the Express framework.

3.5.3 Centralized Task Group

The *Centralized Task Group* is a collection of components responsible for facilitating client joining and computing using the traditional client-server architecture. In this working mode the *Task Data Parts* are assigned to the clients by the system.

3.5.3.1 Coordinator

The *Coordinator* is an application that handles client connections and task assignment. When a client joins the system they communicate with the *Coordinator* which in turn assigns *Task Data Parts* to them. Each client tries to execute every *Task Data Part* in the *Task*, so this assignment is based on which parts are at the least completion level.

3.5.3.2 Client

Each client is assigned a unique *Client ID* upon joining the system, which is persisted on the client side, but checked with the also unique *Connection ID* on the server side. These identifiers are used for statistical purposes and for identifying the work of the same client in multiple results.

When a client in distributed mode joins the system, they get assigned a *Task* and a number of *Task Data Parts*. Depending on system configuration and the size of the *Task Data Parts,* they may either be retrieved directly from the *Coordinator*, or from the *Delivery* component. They fetch the *Task Program* and start executing it on the *Task Data Parts*, creating *Task Results*. These results are then uploaded to the server trough the *Coordinator* and a new set of *Task Data Parts* are requested.

The connection to the *Coordinator* is done over Web Socket, while the connection to the *Delivery* component is over HTTP.

3.5.3.3 Delivery

The *Delivery* component is responsible for serving static content for *Clients*. This content includes *Task Data Parts*, and *Task Programs* as these resources are stateless and read-only from the client’s perspective.

The role of this component can be fulfilled by several well proven web servers, like Nginx [24] or Apache [25]. The only requirement for it is to be able to serve JSON content from the file system.

For our system we used a simple Node.js application with Express framework configured for serving static files.

3.5.4 Peer-to-Peer Task Group

In the *Peer-to-Peer Task Group* the *Tasks*, *Task Data Parts* and *Task Results* are stored in a Kademlia distributed hash table. This shifts the responsibility of serving these resource to the client themselves.

Because there is no centralized state management of task completion levels, each client must choose which *Task Data Part* to execute themselves. Currently this is done by randomizing the starting index and working sequentially from there. In case of a sufficiently large number of clients this will ensure homogenous *Task Data Part* completion numbers.

Communication between clients are over Web RTC peer-connections, each client has a nick name assigned which can be used to reach the client in the overlay network. The connections are created using a Web Socket based *Signal Server*, through the usage of these nick names.

3.5.4.1 Client

Clients connect to the DHT by first joining to a *Seed Node*. They reach these seed nodes by using their pre-determined nick names trough the *Signal Server*. After connecting the DHT trough the *Seed Nodes* the *Clients* are added as nodes to the DHT. The unique ID for each client is stored when joining for later, result retrieval purposes.

Clients first fetch the list of *Tasks* available from the DHT, then chooses from them randomly, weighted by their priority. After choosing the *Task* a starting point is randomly set and *Task Data Parts* are retrieved from the DHT.

When finished with the calculation, the client stores their results with the following key format:

```
[Task ID]_result_[Task Data Part Index]_[Client ID]
```

This ensures that the results can be retrieved by using the previously stored client IDs and knowing the *Task* details such as ID and the number of *Task Data Parts*.

3.5.4.2 Seed Node

The *Seed Node* is a special node in the DHT which is always present contrary to a client which may join or leave freely at any time. Each *Seed Node* have a predefined nickname by which they can be reached by newly joined *Client Nodes*.

For clients a *Seed Node* is the entry point to the DHT, when a client joins the DHT it's ID is stored by the *Seed Node* for later usage.

Another function of *Seed Nodes* is populating the DHT with the necessary data for task execution. Each of them listens for changes in the *Task DB* trough the *Storage Interface* by polling and timestamp based comparison.

The *Seed Node* is a Node.js application using the node-webrtc Web RTC stack for Node and the Kadtools library.

3.5.4.3 Sink Node

A Sink Node is a special node that's sole purpose is watching results accumulate in the DHT and writing it to the *Result DB* trough *the Storage Interface*. They participate in the DHT but do not contribute to any computations.

The *Sink Node* is a Node.js application using the node-webrtc Web RTC stack for Node and the Kadtools library.

3.5.4.4 Signal Server

The *Signal Server* is a Node.js based application using Web Sockets to facilitate signalling for opening Web RTC Peer connections. Signalling is required for building the connections [27] but it's implementation is not specified by the WebRTC standard.

For this purpose, our signalling server works by connecting clients through Web Sockets. When joining the server, each client registers with its nick name and opens a socket. Subsequent messages sent to this nick will be delivered to this socket.

4 Using the system

The system can be considered a platform from the application owner's perspective, as a service from the system owner's perspective and invisible from the client's perspective.

4.1 Use cases

Depending on the actor there can be several use cases of the system. For the system's users the main goal is executing a distributed computing task, while from the system's owner the main goal is forging some kind of advantage form the maintaining of the system.

4.1.1 Scientific uses

Scientific usage would most likely entail a *Primary Deployment* system, as organizations that are interested in solving scientific tasks are rarely in the possession of popular websites that could serve as the host for a *Secondary Deployment*.

Nevertheless, it's not entirely impossible that a scientific organization could profit from a *Secondary Deployment*, for example a university could embed this system in its websites, therefore harvesting the students' computational power.

Use cases would likely closely resemble the current volunteer computing projects as the largest advantage of the system in this model would be ease of access. A well-received press release or a successful, widely covered article could garner the attention of the public, greatly increasing the number of clients, thus computational power, even for short amounts of time.

Some fields where volunteer computing is currently used and could profit from this solution are the following:

- Astrophysics
- Mathematics
- Medicine
- Chemistry
- Engineering

4.1.2 Commercial uses

Commercial usage would likely look very different form scientific uses. In this case *Primary Deployment* is not feasible as most users wouldn't visit a blank page for the sake of participating and generating revenue for a 3rd party. The motivation of contributing to a public interest research is not present in this case, so clients have to be gathered by some other means.

In case of a *Secondary Deployment* the type and visitor characteristics of the host website has significant impact on what kind of *Tasks* the system can handle. Usually the minimum requirement for an operable system is that each client stays in the system for at least the time it takes to complete a single *Task Data Part* and upload it. Otherwise no progress can be made and the administrational costs of unfinished tasks are wasted.

Given a proper host website and configuration a commercial deployment could provide an alternative revenue stream for websites. Compared to advertisements, the user experience doesn't suffer as much when using the visitors as clients. In both scenario (advertisements and participatory computing) the browser downloads data and executable code from the website, while the key difference being purpose.

Advertisements usually contain JavaScript to improve their hit rate by adding visual enchantments, the participatory client uses it to compute the given tasks. Participatory computing has its resource cost, but advertisements can also contain large images or audio, taking up network bandwidth and processing power. The most important difference is that advertisements target the user, distracting them, sometimes even violating user privacy while the participatory client is user agnostic.

There is no need for any kind of interaction from the user and their browsing experience is not disturbed. If the leasing of visitor computational power could become an alternative revenue source, it could have significant effects on the internet. User privacy could be restored to certain level on sites using this revenue stream instead relying on advertisements and user browsing experience would improve, it could be a win-win situation for both users and site owners.

The other question would be how can a website lease this gathered capacity. For this purpose, a larger service could be built that collects several deployments and based on their characteristics assign them tasks. This service could be compared to large cloud computing providers in a way, both of them would provide capacity on demand over the internet.

Potential users of this service would be commercial entities requiring occasional large computing capacity. Computer graphics is one of the fields which could benefit from this service, but other usages are easily imaginable. Business analysis on larger data sets could be conducted using the system's resources and there are several machine learning tasks which could be run in this fashion.

5 Measurements

To prove the feasibility of the concept of a participatory system like this we concluded several measurements, each focused at different aspects of the system. One of the first question is whether the system can have a positive net result of computational power when comparing administrational computing costs to actual task related costs.

Another interesting aspect to examine is the different characteristics of the two working modes. For the *Centralised Working Mode*, we expect close to linear scaling in terms of administrational costs as the number of clients increase. For the *Peer-To-Peer Working Mode* we expect decreasing administrational costs on the server side, but increasing overhead on the client's side.

5.1 Goals

We had two main goals when preparing and conducting these measurements. The first one is proving the feasibility of the system, the second is examining its behaviour while it's working.

5.1.1 Proving feasibility

When studying the feasibility of a participatory system like this, one of the most telling number is the ratio of administrational tasks to actual, computational tasks. In order for a system like this to be profitable in terms of computational gains, the administration overhead of clients requesting and completing tasks must be lower than the cost of executing the tasks themselves. Otherwise it would be faster to use the administrational infrastructure for the calculations.

In other words, the question can be phrased as the following: What is that point in complexity, when the computational gains become lesser than the administrational workload imposed by its execution?

In 5.3.1 we show our results concerning this goal.

5.1.2 Examining behaviour

Our other interest is measuring the typical system behaviour in different working modes. We are interested in client loads and system administration costs, their ratios and how they change when switching working modes.

By measuring these values, we hope to provide some insight when to use which kind of working mode and how to improve them in the future.

In 5.3.2 we show our results for this test.

5.2 Measurement configuration

In our measurement configurations for different test we sought to minimize the distracting effects like network latency, or multi-tasking interference on the participating machines.

However, because of the systems distributed nature, using network connections is unavoidable. To minimize the effect of these, where it was possible we used wired local area network connections, directly between the participants with no other sources of traffic on the network.

In some of the measurements we used a single machine to host all components, in others we used multiple computers, up to 20 physical machines running in parallel. We avoided the usage of virtual machines in order to keep our results consistent and repeatable.

In the following section we list some of the machines we used, in each measurement we'll detail which of the following were used for what purposes.

5.2.1 Hardware configuration

For our measurements we used standard, off-the shelf PCs, just like what we would most likely find in case of a real-world deployment of the system.

5.2.1.1 Workstation A

- Intel Core i7-6700 3.40 GHz
- 16 GB DDR4 2133MHz RAM
- 256 GB Samsung 850 EVO SSD (SATA 6,0 Gbit/s)

5.2.1.2 Workstation B

- Intel Core i5 – 3450 3.6 GHz
- 8 GB DDR3 1300 MHz
- 7200 RPM HDD

5.2.1.3 Laptop A

- Intel Core i7 4720HQ 3.60 GHz
- 16 GB DDR3L RAM 1600 MHz
- 500 GB Samsung 850 EVO M.2 (SATA 6,0 Gbit/s)

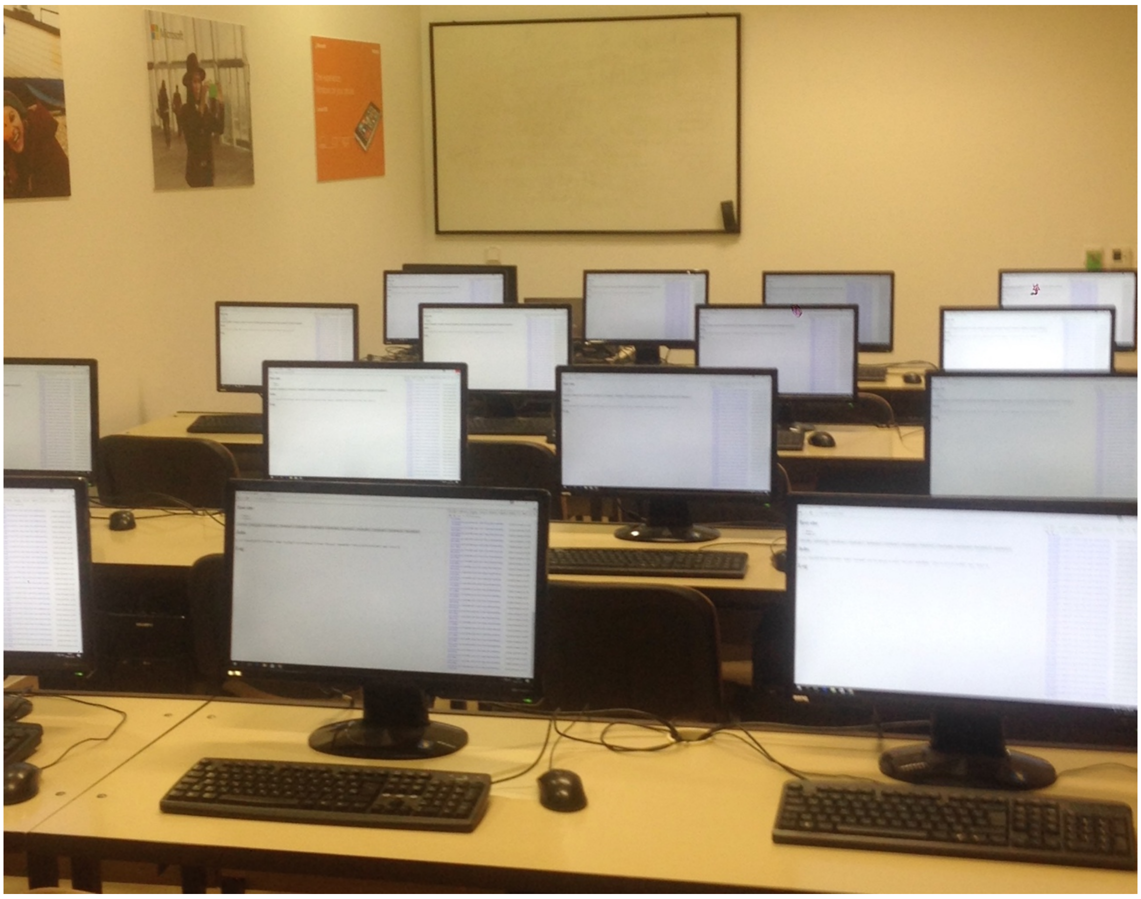


Figure 3 Workstations used in our measurements, running the web client

5.2.2 Software configuration

For our tests we used Windows 10 64-bit edition as operating system and Google Chrome 54.0.2840.71 as the browser. We choose Google Chrome because of its support for WebRTC and extensive developer tooling, used during our measurements.

For our Node.js applications we used Node version 6.7.0. as this was the latest available version at the time of developing the components. Our MongoDB instances were versioned at 3.2.9.

During the tests we closed every non-essential application to avoid influencing system performance.

5.2.3 Executed tests and measured data

In this section we detail the test setups for each measurement.

5.2.3.1 Feasibility test

In our testing setup we used a synthetic task to simulate client load, namely matrix multiplication. Our clients were instructed to create two n * n sized (where n is a given parameter) matrices, fill them with random numbers and multiply them, posting the results to the system. The n parameter takes the powers of 2 from 1 to 1024, each value is run 10 times and results are aggregated.

In this test we used the *Centralized Task Group*, in a *Secondary deployment.* We created a small demo website and embedded the script in it, then visited the site with our client computer (this was a single-host deployment on workstation A - 5.2.1.1).

We chose the *Centralized Deployment* for this test because we wanted to study the impact of client task complexity and result size on system infrastructure. In case of a *Distributed Task Group* these costs would be spread over many clients, not telling much about the feasibility of the system from infrastructure cost vs client performance gain perspective.

We measured the System's processing time, which was calculated as the sum of *Coordinator* and *StorageInterface* processing times. The Client processing time was the time the client was actually doing the job related calculations, excluding network requests. For both measurements we used the JavaScript engine-s built in instrumentation tools for function execution time. As in our case both the client and server applications were using the same V8 engine, we can be assured that the measurements are correct in relation to each other.

5.2.3.2 Working mode comparison

The *Task* executed in the working mode comparison test was much more complex than our previous feasibility test, more closely resembling a real-world application. We chose ray tracing for this test, as it can benefit greatly from parallelization and it's entirely conceivable that it will be a significant use-case for the final system.

We modified Salvatore Sanfilippo's (author of Redis) open-source JavaScript ray tracer [28] to run in our system and used Robert Eisele's similary open-source JavaScript PNG encoder [29] to encode the resulting images and upload them.

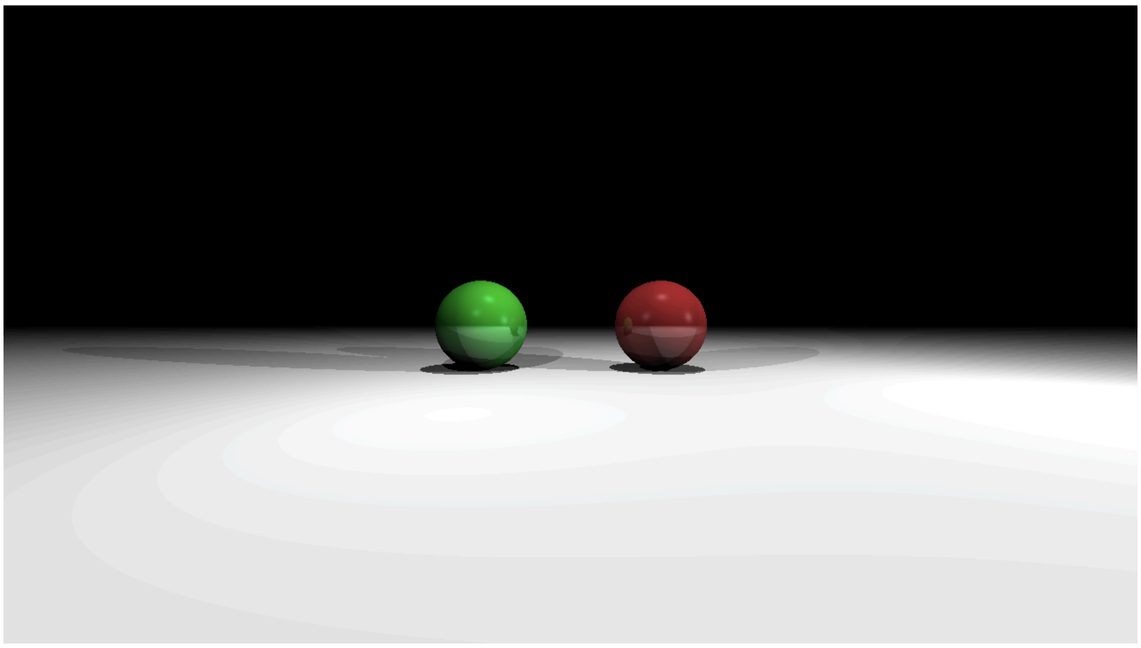


Figure 4 The Scene to be Rendered by the Clients

The overall dimension of the image is 3840 * 2160 pixels (UHD resolution). We partitioned the image to 64 tiles, 8 by 8 in each dimension, meaning each tile is 480 * 270 pixels. In this test the *Task Data Parts* were structures describing which tile the client should render, and the scene was embedded in the *Task Program*.

When a client finished rendering a tile it encoded it in PNG format, then uploaded the result in Base64 encoding, thus each *Task Result* contained an index, and the Base64 string of the image.

We were interested in client and server loads so we set up the *Task* parameters to be possibly the most demanding. We set *Confidence* level to 19 (the number of clients in the system), meaning each client have to render each tile at least once.

We executed this task in both *Centralized Working Mode* and *Peer-to-Peer Working* mode.

5.3 Results and evaluation

On one hand our results supported our expectations about the feasibility of the system. On the other hand, we observed some interesting behaviour in case of *Peer-to-Peer Working Mode*, mandating further research.

5.3.1 Feasibility test

The following table show processing times for clients and the system in case of different matrix sizes.

Table 1 Feasibility measurement results

Matrix size (n*n)	System processing time (ms)	Client processing time (ms)	Ratio % (system / client)
1	18,957	3,77	502,8
2	20,384	3,466	588,1
4	19,884	3,708	536,2
8	21,619	4,161	512,1
16	21,204	5,184	409
32	22,626	14,88	152,1
64	32,011	27,25	117,5
128	70,015	76,73	91,2
256	172,73	521	33,2
512	706,122	4103	17,2
1024	3211,275	55750	5,8

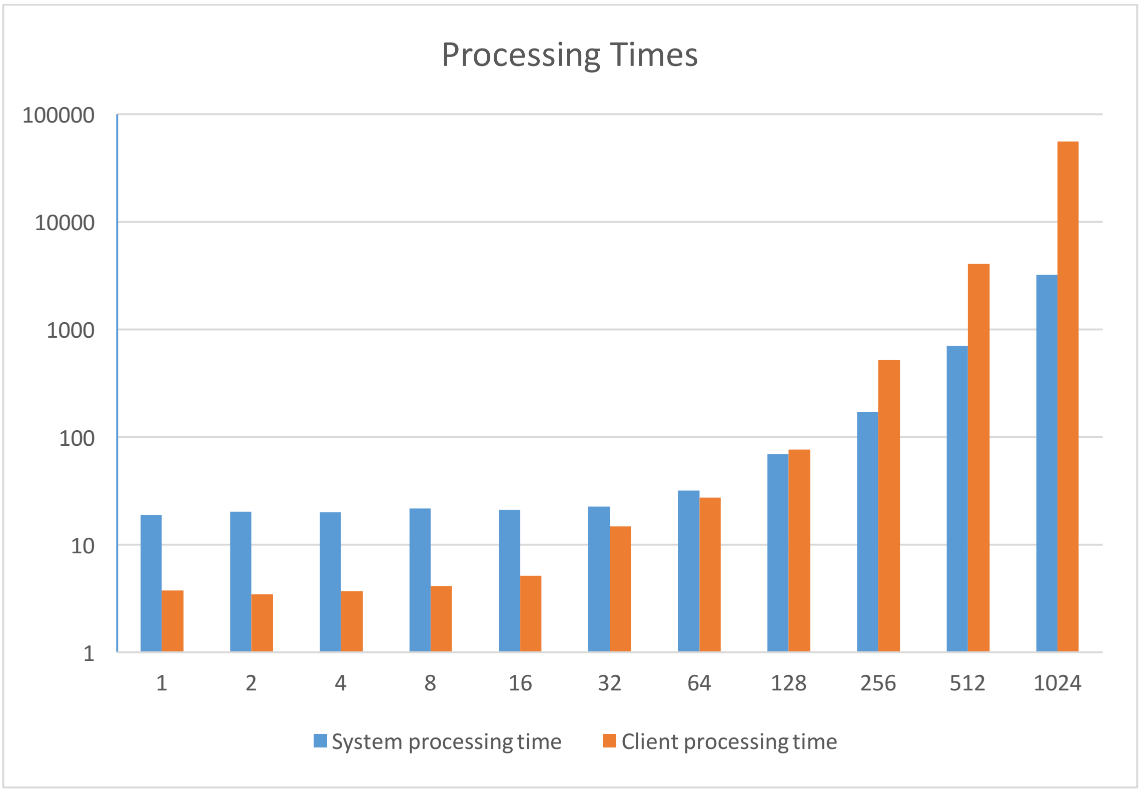


Figure 5 Processing times compared

According to our measurements, in our test configuration the system become computationally profitable at 128 * 128 matrix size. At this point the client spent more time computing the results than the system spent with assigning the *Task* and retrieving the result.

From the data we measured, we can see that after an initial turning point at around 75 ms of client calculations, the gap between client and system processing time starts to widen exponentially. Notice the expanding gap between the two processing times on the visualization of the measured data (note; the Y axis is logarithmic). We predict this behaviour continues for more expensive calculations until it becomes impractical to run on the clients.

From this observation we can conclude that even at relatively low individual task costs (~75 ms) it's profitable to use the system in *Centralised Mode*.

5.3.2 Working mode comparison

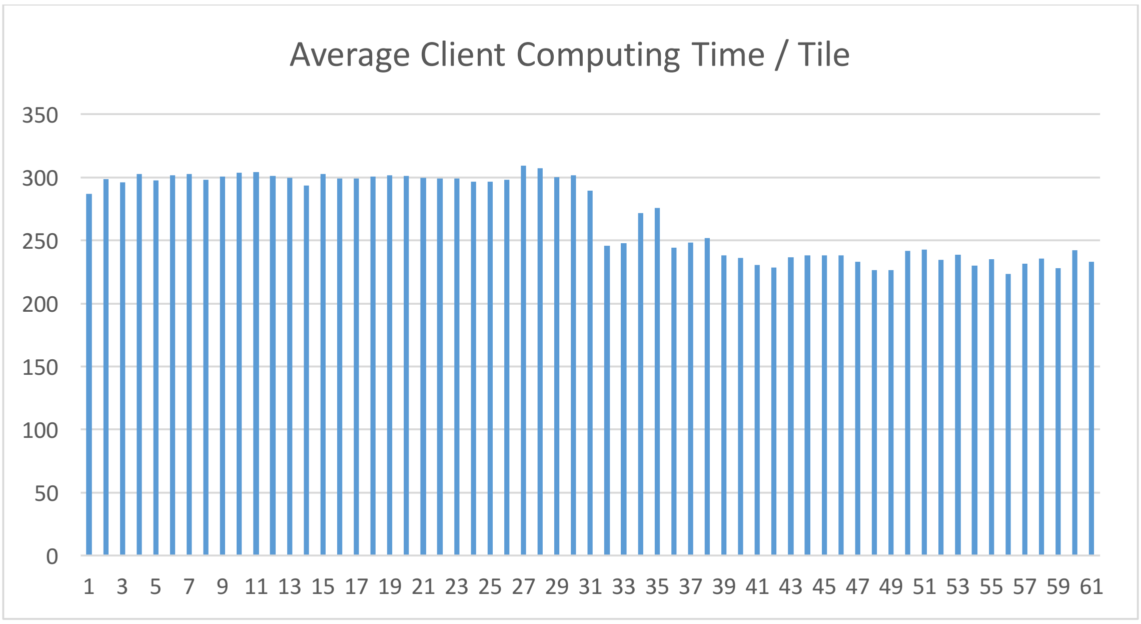


Figure 6 Average client computation time of each tile

The client processing time or *Task Data Part* execution time is the same in both Working Modes, as it's simply the rendering of the given tile. Tiles are numbered from 1 to 64 from the top left corner, from the chart we can see which parts of the scene are computationally more expensive. For example, the bump at 33-35 is caused by one of the specular spheres.

5.3.2.1 Centralised Working Mode

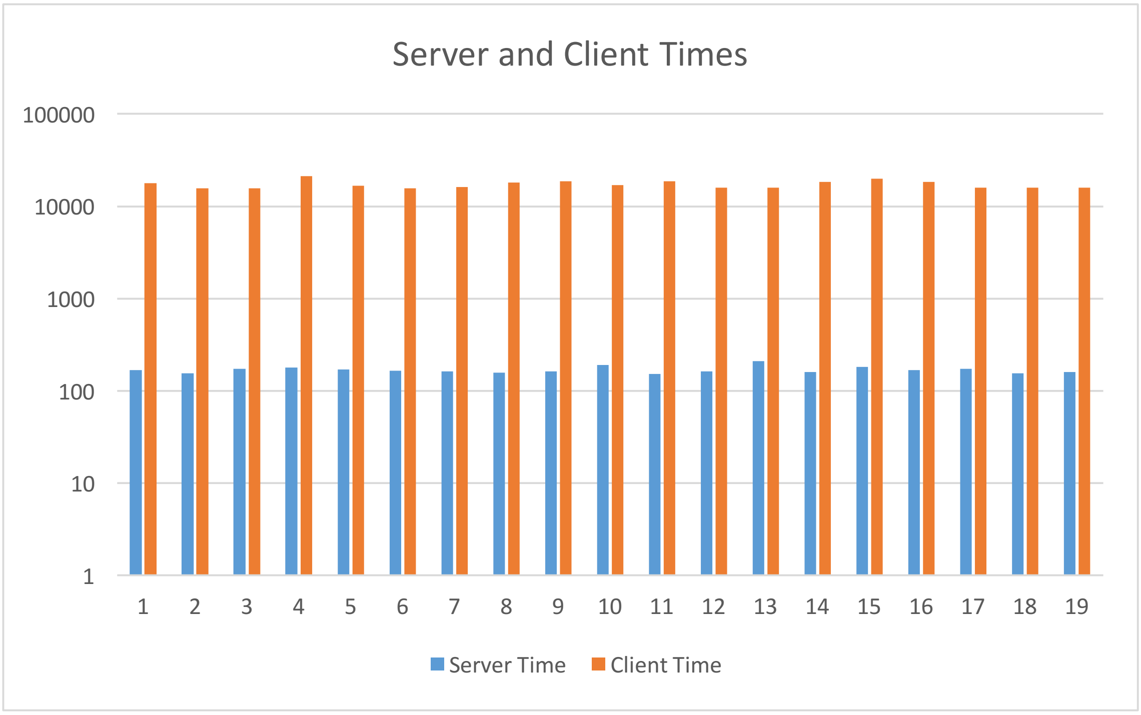


Figure 7 Server and Client Times in Centralised Mode

According to our measurements the total server processing time spent was 3040,19 ms while the total client processing time was 327 453,44 ms. This means a ratio of ~0,9% which is considered very good compared to our previous measurement.

This improvement can be attributed to the type of task executed and the way the results are stored. In the first measurement the resulting matrices are stored as 2 dimensional arrays of integers in the database. This requires considerably more manipulation of the data than simply storing the Base64 string of the current test's result.

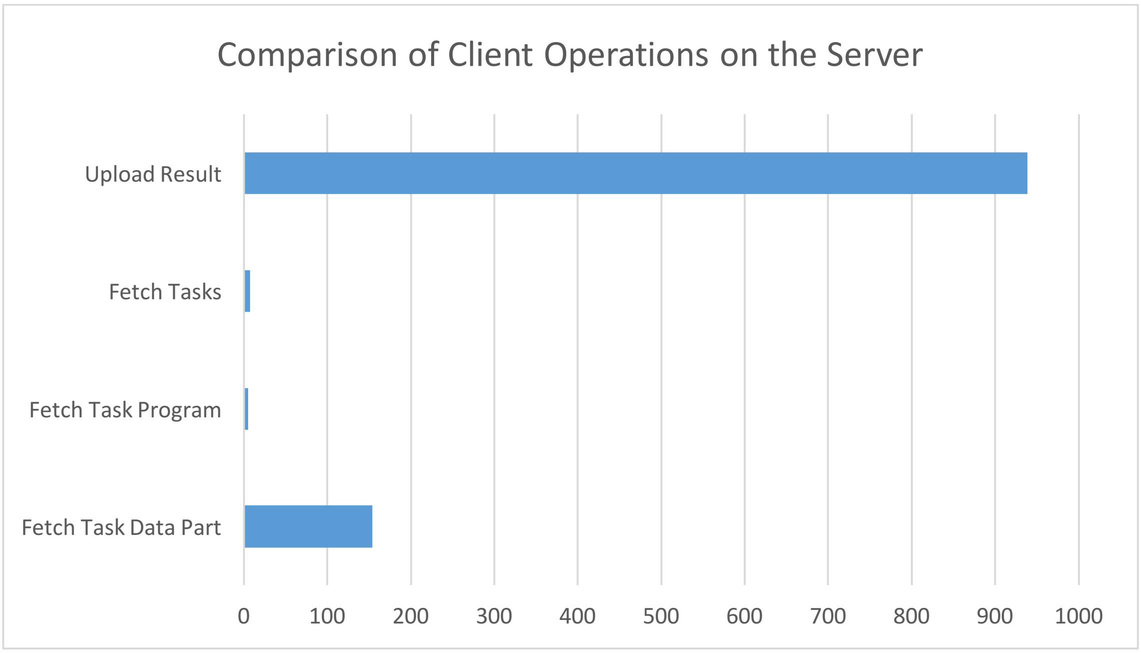


Figure 8 Comparison of Client Operation Costs on the Server

From the server's perspective the computationally most expensive client operation was the storage of the results, followed by the retrieval of the *Task Data Parts*.

5.3.2.2 Peer-to-Peer Working Mode

Peer-to-Peer Working Mode to our surprise proved to be much slower than *Centralised Working Mode* in this measurement. During our tests 2 of the 19 nodes failed to completely execute all of the calculations, this is not completely unexpected as our underlying Kademlia implementation of the DHT has at most 'bet effort' guarantees. Nevertheless, we did not expect this kind of reliability issues in a closed, controlled environment like ours was for the measurements.

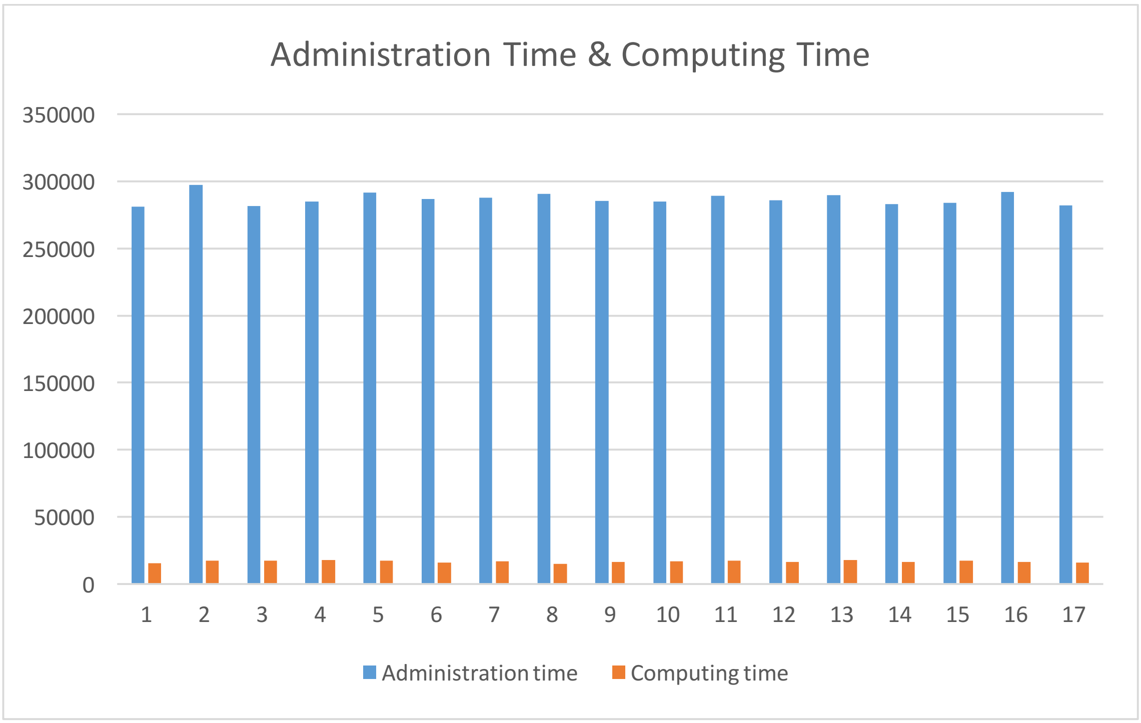


Figure 9 Administration and computing times in clients

For the remaining nodes the computing times were very similar to the ones we measured in the centralised test. This is expected as the *Task* remains the same, despite the delivery and upload method.

What is more interesting to notice is that accessing the DHT provides a rather significant overhead in the clients. On average it took ~1730% more time to retrieve the *Data Parts* and upload the *Results* then actually calculating them.

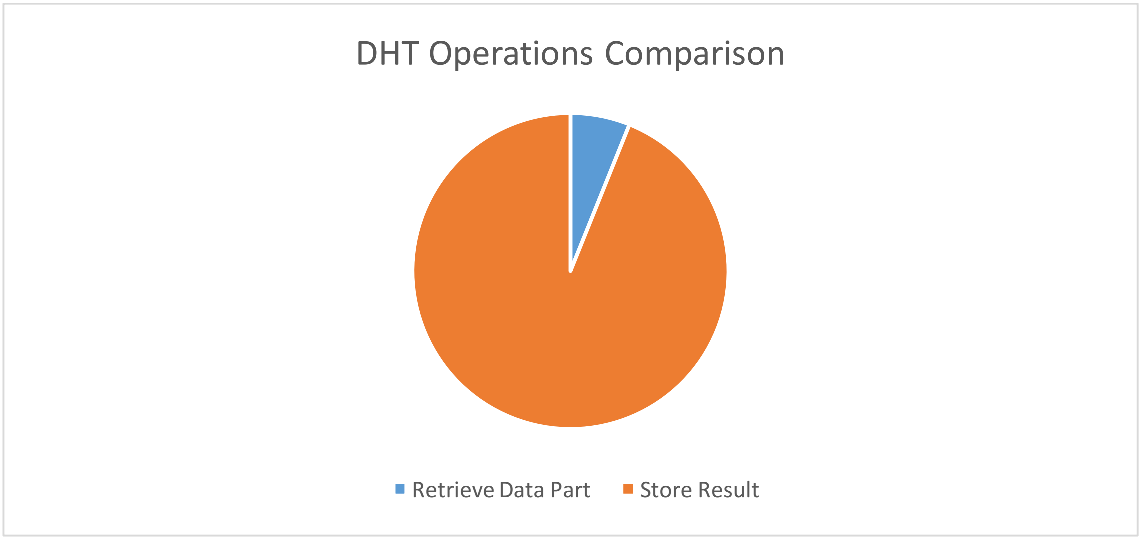


Figure 10 Client DHT Operations Comparison

The bulk of this administrational overhead came from storing the results. The ratio between data retrieval and result storage was around 1 to 15, or ~6,49%. In overall we concluded that, while the *Peer-to-Peer Working Mode* does indeed moves administrational responsibility to the clients, it also does it rather inefficiently.

Ultimately the performance of *Peer-to-Peer Working Mode* proved to be significantly under the *Centralised Mode's* in this measurement. The reasons of this performance drop can be attributed mainly to the overhead introduced with the usage of DHT.

Further study of Kademlia's behaviour in different configurations could lead to optimizations and improvements in efficiency. We could also improve performance by adapting our client specifically for the Kademlia library we use, for example by using node id and locality for Task allocation and management.

6 Overview and future work

At the first part of this paper we presented a problem, examined it from different perspectives and showed a possible solution. In later parts we discussed this solution and its uses and ultimately measured different aspects of it. In this final section we intend to provide and overall summary of the work done and show some future research directions.

6.1 Summary

In this paper we showed some potential barriers in the proliferation of participatory and volunteer computing systems. We identified one of this barriers, accessibility as a major factor in the phenomenon and showed some previous works trying to overcome it.

We listed the key features and requirements necessary from a system to achieve the set of goals. Then we introduced our solution, a participatory system built on web technologies, requiring no active participation from the client. We evaluated the required technologies to build a system like this and selected JavaScript and web browsers for clients, and JavaScript and Node.js for server applications. We detailed the implementation of our system, showed benefits of the proposed architecture and used technologies.

Finally, we measured the feasibility of this system by checking whether the final balance between computational gains by adding clients and its administrational overhead are balanced out in favour of the former.

We also investigated how does the system behaves in the two working modes presented earlier. We've observed the expected behaviour in *Centralised Working Mode,* further proving the operability of the system and found some interesting directions to continue refining *Peer-to-Peer Working Mode* with.

As a result of these measurements we concluded that a participatory system using JavaScript in browsers for clients is feasible, we've also showed trough a real-world example that the system we implemented is capable of conducting these computations in both working modes.

6.2 Future research directions

One of the conclusions we've drawn from the measurements, was that the large administrational overhead of the *Peer-to-Peer Working Mode* affects performance negatively. In the future we intend to explore possibilities for further reducing these administration costs when utilizing this working mode in the system.

Another interesting research direction is associated with the *Secondary Deployment Mode*, when the system is deployed alongside an existing website. In this case, the effects of different visitor behaviours could have very different impacts on the system. For example, client joining and leaving frequency greatly depends on the type of the host website. E.g. in case of a news portal, with long articles we may find users staying on the same page for longer times, therefore it would be beneficial to assign them longer tasks. Measuring how this client behaviour affects different types of *Tasks* could help us in tailoring the configurations more precisely to the given deployment's needs.

We would also like to improve our client software by providing persistence options for Tasks, execution halting and restart, and general performance tweaks and optimizations.

Another field worth investigating is improving the application creators experience. The basic structure of the *Task* creation and execution could be improved. Instead of pre-determined *Task Parts* and sizes, we could enable dynamic *Task Part* creation on the server side.

Ultimately answering these questions could lead to a much larger adaptation of participatory systems, enabling computationally expensive scientific and commercial projects to become viable. By providing an alternative income for websites, namely leasing their visitors computational power, participatory systems could also change the way how the economy of the web works

Bibliography

[1] *Directive 2002/58/EC of the European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector* (Directive on privacy and electronic communications)

[2] David P. Anderson, *"BOINC: A System for Public-Resource Computing and Storage"*, 5th IEEE/ACM International Workshop on Grid Computing (4-10)

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, Dan Werthimer, *"SETI@home An Experiment in Public-Resource Computing"*, Commun. ACM 2002 Vol. 45 (56-61)

[4] Stefan M. Larson, Christopher D. Snow, Michael Shirts, Vijay S. Pande, *"Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology"*, Computational Genomics, Richard Grant, editor, Horizon Press, 2002.

[5] Luis F. G. Sarmenta and Satoshi Hirano, *"Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java",* Future Generation Computer Systems 15(5-6), New York, 1999.

[6] O. Regev, and N. Nisan, *"The POPCORN market. Online markets for computational resources"*, Decision Support Systems, Vol. 28. (177-189) , 2000.

[7] John P. Morrison, James J. Kennedy, David A. Power, "WebCom: A Web Based Volunteer Computer", The Journal of Supercomputing vol. 18 (47-61), 2001.

[8] David P. Anderson, Gilles Fedak, "The Computational and Storage Potential of Volunteer Computing", Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (73-80), 2006.

[9] Balachander Krishnamurthy, Craig E. Wills, "Privacy Diffusion on the Web: A Longitudinal Perspective", Proceedings of the 18th International Conference on World Wide Web (541-550), 2009.

[10] "Chrome V8 | Google Developers", Google Developers. [Online]. Available: https://developers.google.com/v8/. [Accessed: 14- Oct- 2016].

[11] "SpiderMonkey", Mozilla Developer Network, 2016. [Online]. Available: https://developer.mozilla.org/en- US/docs/Mozilla/Projects/SpiderMonkey. [Accessed: 14- Oct- 2016].

[12] "WebRTC Home | WebRTC", Webrtc.org, 2016. [Online]. Available: https://webrtc.org. [Accessed: 14- Oct- 2016].

[13] "HTML5", W3.org, 2016. [Online]. Available: http://www.w3.org/TR/html5/. [Accessed: 14- Oct- 2016].

[14] Stefan Tilkov, Steve Vinoski, “Node.js: Using JavaScript to build high-performance network programs”, IEEE Internet Computing (2010 November) Vol. 14 (80-83), 2010.

[15] "MongoDB", MongoDB, 2016. [Online]. Available: https://www.mongodb.com. [Accessed: 14- Oct- 2016].

[16] "KadTools Index", Kadtools.github.io, 2016. [Online]. Available: http://kadtools.github.io. [Accessed: 14- Oct- 2016].

[17] Petar Maymounkow, David Mazieres, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”, Revised Papers from the First International Workshop on Peer-to-Peer Systems (53-65), 2002.

[18] "kadtools/kad-webrtc", GitHub, 2016. [Online]. Available: https://github.com/kadtools/kad-webrtc. [Accessed: 14- Oct- 2016].

[19] "js-platform/node-webrtc", GitHub, 2016. [Online]. Available: https://github.com/js-platform/node-webrtc. [Accessed: 14- Oct- 2016].

[20] Roy T. Fielding and Richard N. Taylor. *Principled design of the modern web architecture.* ACM Trans. Inter. Tech., 2(2):115–150, 2002.

[21] "*Express - Node.js web application framework*", Expressjs.com, 2016. [Online]. Available: http://expressjs.com. [Accessed: 20- Oct- 2016].

[22] "*Mongoose ODM v4.6.4*", Mongoosejs.com, 2016. [Online]. Available: http://mongoosejs.com. [Accessed: 20- Oct- 2016].

[23] "*TypeScript - JavaScript that scales.*", Typescriptlang.org, 2016. [Online]. Available: https://www.typescriptlang.org. [Accessed: 21- Oct- 2016].

[24] "*nginx news*", Nginx.org, 2016. [Online]. Available: https://nginx.org. [Accessed: 21- Oct- 2016].

[25] *"Welcome! - The Apache HTTP Server Project*", Httpd.apache.org, 2016. [Online]. Available: https://httpd.apache.org. [Accessed: 22- Oct- 2016].

[26] "*js-platform/node-webrtc*", GitHub, 2016. [Online]. Available: https://github.com/js-platform/node-webrtc. [Accessed: 23- Oct- 2016].

[27] Sam Dutton *“Getting Started with WebRTC”,* HTML5Rocks, 2016 [Online]. Available: https://www.html5rocks.com/en/tutorials/webrtc/basics/ [Accessed: 23- Oct- 2016].

[28] "*antirez/jsrt*", GitHub, 2016. [Online]. Available: https://github.com/antirez/jsrt. [Accessed: 24- Oct- 2016].

[29] Xarg.org, 2016. [Online]. Available: http://www.xarg.org/download/pnglib.js. [Accessed: 24- Oct- 2016].

Appendix

Table of Figures

Figure 1 System architecture 17

Figure 2 Admin Web Application Frontend 21

Figure 3 Workstations used in our measurements, running the web client 32

Figure 4 The Scene to be Rendered by the Clients 34

Figure 5 Processing times compared 36

Figure 6 Average client computation time of each tile 37

Figure 7 Server and Client Times in Centralised Mode 38

Figure 8 Comparison of Client Operation Costs on the Server 39

Figure 9 Administration and computing times in clients 40

Figure 10 Client DHT Operations Comparison 40

Table 1 Feasibility measurement results 35