CHALMERS | UNIVERSITY OF GOTHENBURG

A general peer-to-peer based distributed computation network

Bachelor of Science Thesis in Computer Science and Engineering

JACK PETTERSSON
LEIF SCHELIN
NIKLAS WÄRVIK
JOAKIM ÖHMAN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

A general peer-to-peer based distributed computation network

J. Pettersson,
L. Schelin,
N. Wärvik,
J. Öhman

Examiner: J. Skansholm

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000
Department of Computer Science and Engineering
Göteborg, Sweden June 2014

A general peer-to-peer based distributed computation network

Bachelor of Science Thesis in Computer Science and Engineering

JACK PETTERSSON
LEIF SCHELIN
NIKLAS WÄRVIK
JOAKIM ÖHMAN

Abstract

We consider how a decentralised computation system would work when each participant could create computation code as well as executing other participants' code. A protocol is proposed that allows such collaboration to take place, assuming that no participant can be trusted. We consider briefly how the computation code itself can be executed in a safe manner.

The primary focus of this thesis is to investigate how to increase the reliability of the computation results, as some participants can be assumed to return incorrect results. Furthermore, a prototype that demonstrates the key principles of the theoretical results is also developed. The methods developed are found to be correct and working, but unfortunately does not contribute very much in terms of functionality or advantages for the end user.

Sammanfattning

Vi beaktar hur ett decentraliserat beräkningsnätverk skulle fungera om varje nod både kunde utföra och skapa beräkningar. Vidare föreslås ett protokoll som tillhandahåller detta, även under förutsättningen att ingen deltagare är pålitlig. Vi överväger också hur beräkningarna kan genomföras på ett säkert sätt.

Det primära syftet med rapporten är att undersöka hur tillförlitligheten hos beräkningsresultaten kan ökas, då deltagare kan antagas returnera falska resultat. Dessutom utvecklas en prototyp som åskådliggör de viktigaste principerna och teoretiska slutsatserna. De utvecklade metoderna fungerar men tillför dessvärre inga betydande förbättringar för slutanvändaren.

Acknowledgements

This thesis has benefited greatly from the continuous support and enthusiasm of our supervisor Dr. Andreas Abel, to whom we are very grateful. We would also like to thank Prof. Mattias Wahde for suggesting several complex functions we could use as examples. Finally, Jack would like to thank Johan Gustafsson for the interesting discussion which led to the initial idea for the project.

Contents

1 Introduction

Distributed systems have been gaining more and more attention in recent years. One reason for this is that a lot of computation power goes unused for large amounts of time, which instead could be used in a collaborative manner by a distributed system. Systems such as BOINC and Cosm help research projects (for example Folding@Home and Einstein@Home) in harnessing unused resources from thousands of volunteers to find solutions to complex problems (BOINC 2014a; Cosm 2014). These networks work as an alternative to supercomputers which are expensive to buy and maintain.

Such centralised systems generally suffer from two major drawbacks. Firstly, they can be vulnerable to attacks due to the single point of failure, for example denial of service attacks. Secondly, they often require a lot of resources and knowledge in order to set up a server to delegate tasks as well as to collect and validate results.

Decentralised systems such as Freenet, Bitcoin and Namecoin, to name a few, have proven that it is sometimes possible to decentralise tasks that have generally been thought to require central coordination (Freenet 2014; Nakamoto 2008a; Namecoin 2014). Hence, we aim to introduce a peer-to-peer based computation system that lets any peer upload tasks to be performed and choose freely which other computations to work on.

Ideally, using this system to perform a resource intensive computation would only require a bit of programming knowledge and that others find its purpose meaningful. Consequently, one does not need access to supercomputers as long as there are participants willing to provide computation power. This system could be useful for universities which often have extensive resources in standby. People interested in research could also collaborate with the universities by providing computation power. The system would also be useful for the same research as the existing centralised systems for distributed computations, without the need for the resources and knowledge required to set up a central server.

Preliminaries The reader is assumed to know fundamental computer science, in particular data structures, complexity theory and basic cryptography, as well as discrete mathematics and mathematical statistics. Terms and concepts specific for this thesis are explained in text and can also be found in the glossary in the appendix.

1.1 Purpose

The aim of this thesis is to investigate how peer-to-peer based computation systems with free participation would work. Furthermore, we intend to determine methods for such a system to produce reliable computation results. A software prototype is developed and implemented to demonstrate the key principles of these methods.

Essential properties of the system The essential properties of the system can be summarised in the following four points:

- Decentralisation using peer-to-peer technologies
- Free participation
- Reliability of results
- Secure participation

The paramount requirements of the system are decentralisation and free participation as these form the foundation of this thesis. Decentralisation will make the network as a whole hard to disrupt as it does not depend on any single node to function.

Free participation refers to letting any computer join the network to participate in working for others and in requesting computations to be executed. Nodes that execute computation code will hence be referred to as workers. Nodes that create new computations are referred to as job owners.

The computed results in the network should be reliable solutions to the given problems. This is vital for the system to be useful.

Users should be confident that executing computation code is purely computational. BOINC[1] establishes this by only letting a small number of trusted people produce computation code (BOINC 2014a). However, when anyone can write computation code, safety from harm should be guaranteed in a more strict sense. If the users do not feel safe, they will probably be more reluctant to participate in computations.

Main challenges to solve Following the properties are a set of challenges that must be solved. The most central challenges are:

- Byzantine nodes

[1]The centralised computation system used by Einstein@Home and many others.

- Malicious code

A Byzantine node is a computer in the network that behaves in an unexpected and potentially harmful way. This is important to consider, as any computer can join the network and any node can participate in computations. Most importantly, Byzantine nodes can return invalid results for a computation, whether this is intentional by the user or not.

Malicious code must be prevented from being executed on a worker. If this is disregarded, serious consequences like virus injection can happen to a worker, or a worker can unknowingly become part of a botnet[2] that in turn hurts third parties. This would be a huge security threat for a user of the network. If the system can be misused, people may also be more reluctant to volunteer.

1.2 Requirements

For the system to work and function well, certain requirements must be realised. Here we present the essential properties the system needs.

Functional requirements The following functionality should be facilitated by the system.

- Each worker should be free to choose which computations it wants to contribute to.
- Nodes should be able to get computations to work on and upload the respective results to the network. The work itself should be possible to do offline.
- Computation code and results should be stored in a distributed storage.
- All files in the distributed storage should be safe from modification by Byzantine nodes.
- Workers and job owners should be able to go offline without disrupting the network. However, it is reasonable to require that, at all times, a minimum number of nodes should be connected to the network to ensure stability and no data loss, but it should not be required of any specific node.

[2]A group of computers that cooperate over a network, often for a malicious intent against a third party.

- It should be safe for workers to execute computations. If they are not protected, harmful code may destroy a user's data or make it part of a botnet without the user knowing it.

Non-functional requirements Security should be good enough for the prototype to be open source. Even if a hacker knows in detail how it is implemented it should still withstand attacks. This is consistent with good practice in modern cryptography, where security is achieved through the secrecy of the key rather than through obscurity of the cryptographic method (Basin et al. 2011; Furnell et al. 2008; Kerckhoffs 1883).

1.3 Delimitations

It is beyond the scope of this project to search the computation code for bugs. Compilation failures and runtime errors that crash the program will however be reported.

Truthful executions of a task are assumed to always return the same result, even when run on different computers. This is relevant since the results of floating point computations can vary between different architectures (Muller et al. 2010). Code using floating point operations has been shown to behave differently for the same hardware using the same compiler when run on different operating systems. This will have to be taken into consideration by the job owner when implementing a task that extensively uses floating point operations.

The prototype will not facilitate the option for job owners to write computations in an arbitrary programming language. One specific language will be chosen for computations. Only code that can be guaranteed to not perform any unsafe operations, such as IO, during execution will be accepted. This means that all data necessary to perform a computation must be present before it is executed. Note that for stochastic algorithms, a seed used to generate chance can be sent as an input parameter to the computation rather than be created during runtime. If the randomness is only used to generate a few numbers, those can be computed by the job owner and sent as parameters instead of the seed.

Performance can be considered important in a computation system. In this thesis however, security of participation and reliability of results take precedence over performance. It does not matter how fast a system is if the results cannot be trusted.

2 Architecture

The basic architecture of the system consists of six parts, which is visualised in Figure 1. Each part will be discussed later in this thesis.

The connectivity between peers is solved in two steps: the message passing interface allows for direct communication and the distributed storage facilitates efficient file transfer. The secure execution handles executing computation code in a secure manner. The task management takes care of the handling and assignment of tasks, which is very important for the job owner. The controller part binds the system together and the user interface (UI) interacts with the user.

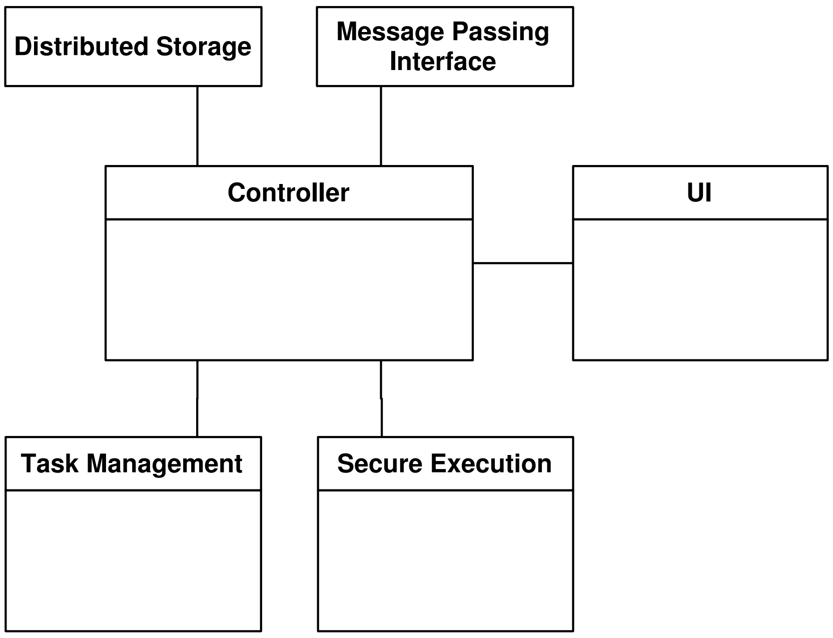


Figure 1: *Illustration of the high-level architecture of the system.*

Simple scenario When the program starts, it connects to the distributed storage and starts listening for messages. At this time, a job owner can check in the distributed storage for results that have been computed while he has been offline. The results can then be validated to see if they are correct. The job owner can also create new tasks with the task management that are uploaded to the distributed storage.

A worker can request a task from a job owner by sending a message. The job owner then lets the task management decide which task the worker should work on and sends a message containing the location for the necessary files in the distributed storage. The worker can download the task and execute it in the secure execution environment. The result of the computation can then be uploaded to the distributed storage and the worker can notify the job owner with a message that the task is complete. If the job owner is currently offline, he will notice the result in the distributed storage when he comes online.

3 Example computations

Two mathematical problems were chosen to demonstrate the capabilities of the prototype. They were both meant to be simple and easily understood while being very different in nature. The first relies on deterministic algorithms and produces results that can be checked for correctness. The second relies on stochastic algorithms and produces results that are hard to check if they are correct. They will be referred to throughout the thesis to exemplify different principles.

3.1 Prime calculations

The first example is the familiar problem of calculating prime numbers. The job owner decides what intervals of primes should be calculated. Each interval becomes a task for a worker to solve. The first task must be to calculate all primes between 2 and n. Using the results from the first task, new tasks can be created to calculate all prime numbers up to n^2. Though more efficient algorithms exist, a very simple algorithm was implemented for the prime search as the searching itself is not the focus of the thesis.

3.2 Stochastic optimisation

In order to demonstrate that the developed prototype can handle very different kinds of tasks, a stochastic problem was chosen as the second example. Stochastic optimisations can be faster than exact analytical solutions to many complex optimisation problems (Schneider and Kirkpatrick 2006). There are many different algorithms or heuristics that use randomness to calculate optimal parameters to a system. One kind of stochastic algorithms are sometimes referred to as Monte Carlo (MC) algorithms. They do not always return the optimal result, but are instead bounded in time. The optimisation problem we chose as our example is described below, along with a brief discussion of the MC algorithm used to solve it.

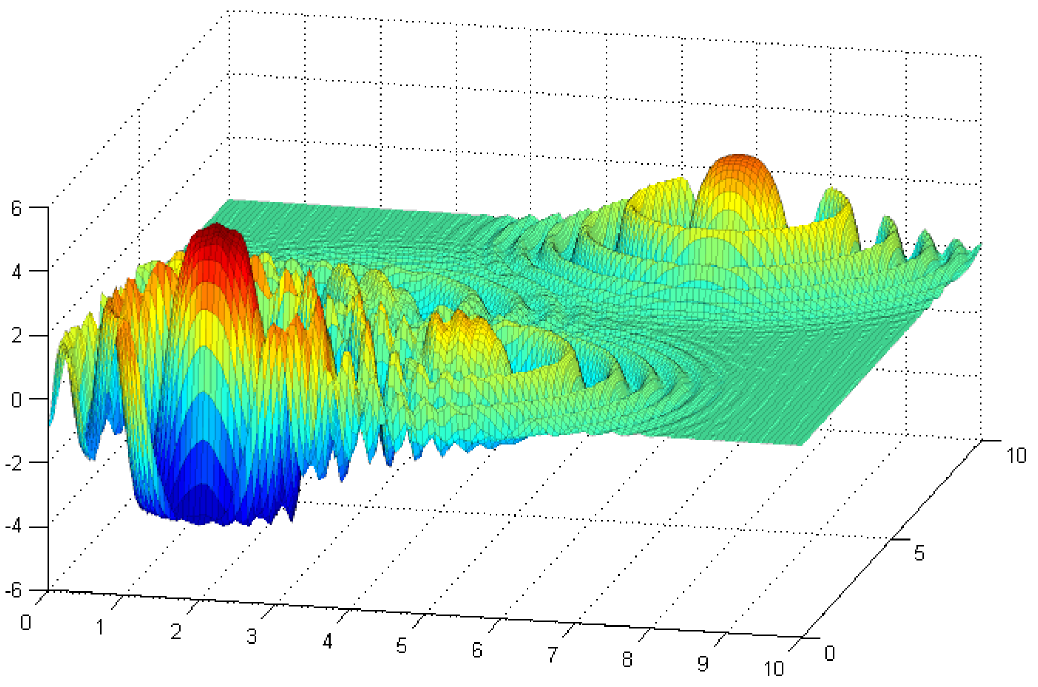


Figure 2: *The Langermann function in two dimensions.*

Langermann The problem example in question is to optimise the two-dimensional Langermann function over a finite interval. The appearance of the function is illustrated in Figure 2. Note especially that there are many stationary points separated by barriers. The multitude of potential optimum points make traditional, analytical methods relying on gradients infeasible. Traditional, numeric methods such as the Gradient descent are instead hindered by the barriers. This makes the Langermann function very useful for benchmarking of optimisation algorithms, for which it is also recommended (Molga and Smutnicki 2005).

This specific application of stochastic optimisation problems was chosen as it is very simple to understand and visualise. An even more complex but realistic scenario is to optimise a simulation with many variables. If the system to be optimised has many parameters, one can divide the parameter space into several tasks and apply either approach to the task of optimising the system in the respective limited parameter space. With the Langermann example, one task could solve the optimal point in the interval $1 \leq x \leq 2$ and $0 \leq y \leq 1$.

Implementation The implemented stochastic algorithm is a very simple MC algorithm that only randomises points and evaluates them; the tested point with the highest function value is returned to the job owner. The implemented Langermann function was positively verified against published Matlab code written by Surjanovic and Bingham (2013).

4 Literature studies on distributed algorithms

As stated in Section 1.1, one of the main challenges to solve is that of invalid results and an essential property is decentralisation. With the correct algorithm(s), it might be possible to combine these two concerns in a way that enables the validation of results to be done in a purely decentralised manner. Thus, the area of distributed algorithms was researched and the results are summarised in this section.

4.1 Failing nodes

When discussing distributed algorithms in general, an important concept is that of *failures* (Lynch 1996). There are two kinds of failures a process can exhibit: *stopping failures* — processes that simply stop communicating — and *Byzantine failures* — processes that fail in arbitrary ways, such as processing requests incorrectly or producing inconsistent output. Naturally, we will have to take both kinds into consideration. Stopping failures are handled with timeouts, which are described further in Section 5.4. This section only discusses Byzantine failures.

4.2 Asynchronous systems

An asynchronous system is a multi-process system in which the actions of the processes are not synchronised by a global clock (ibid.). Instead, each process acts independently of all the others, synchronising with them exclusively through communication channels. Since this is an apt description of the system we aim to develop, the field of distributed algorithms for asynchronous systems is of high interest.

4.3 Consensus about results

The consensus problem is a classical one in distributed computing, with the goal being for a number of processes to propose at most one value each and then agreeing on exactly one of them (Pease et al. 1980). This is relevant because each decision about whether to accept or reject a certain result can be seen as a consensus problem: whenever a worker publishes a result, all the other nodes perform a validation and propose whatever it outputs. For this to function well, the validation has to be relatively cheap. This is certainly not always the case, but could be seen as a delimitation even though it would lead to a less useful system.

Unfortunately, to deterministically achieve consensus in bounded time while allowing processes to fail has been proved to be impossible in an asynchronous system without any constraints imposed on it (Fischer et al. 1985). For most practical applications, this was solved probabilistically in 2008 with the introduction of the *block chain* data structure (Miller and LaViola 2014; Nakamoto 2008a,b). Assuming that the system has certain properties, the problem has also been solved deterministically by Alchieri et al., who referred to the problem as *BFT-CUP*. Both of these solutions are presented and examined below.

4.3.1 Block chain

A block chain is a chain of *blocks*, each containing *transactions*, the data objects that are agreed upon (Nakamoto 2008a). When a participant creates a transaction, it is broadcast to all other participants, which will include it in their next block if they find it to be valid. A new block is created by the first participant to find a certain proof-of-work, which it includes in the new block, together with all valid transactions that have not yet been included in any previous block.

The new block is then broadcast to the network and upon receipt of a valid block, each participant adds it to the end of its copy of the block chain and starts working on the next block. The hash of the latest block is then seen as the challenge for the next proof-of-work[3]. Figure 3 shows the basic structure of a block chain. The process in which nodes continuously compete to find the next proof-of-work is called *mining*.

Honest participants always consider the longest chain to be the correct one. This means that in order to have them accept an invalid transaction, an attacker would have to find proof-of-works consistently faster than the rest of the network together. Provided that the honest nodes control a majority of the network's computational power, the probability of this is negligible, as shown by Nakamoto (2008a).

With all this in mind, it would be very tempting to use a block chain in our system, using results as transactions. Unfortunately, this is based on the false assumption that nodes can use 100% of their computation power to

[3]The point to take from this is that each block depends on the previous one, meaning that blocks cannot be created in advance or changed afterwards. For a detailed explanation of proof-of-work and challenges, see Section 6.3.2.

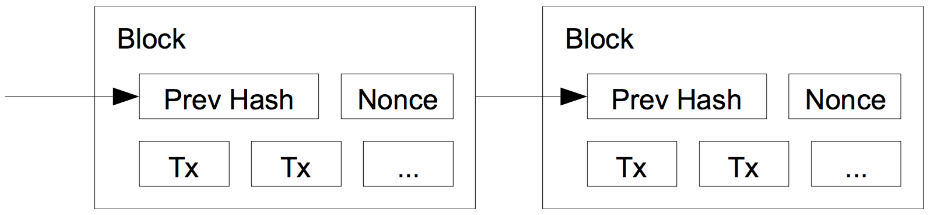


Figure 3: *The basic structure of a block chain. In order for a block to be considered valid, its nonce has to be a proof-of-work for when the hash of the previous block is used as a challenge. The transactions of the block are shown beneath. (Nakamoto 2008a)*

secure the network, while at the same time producing results. In other words, the method for producing results stands in stark contrast to the method for reaching consensus about them.

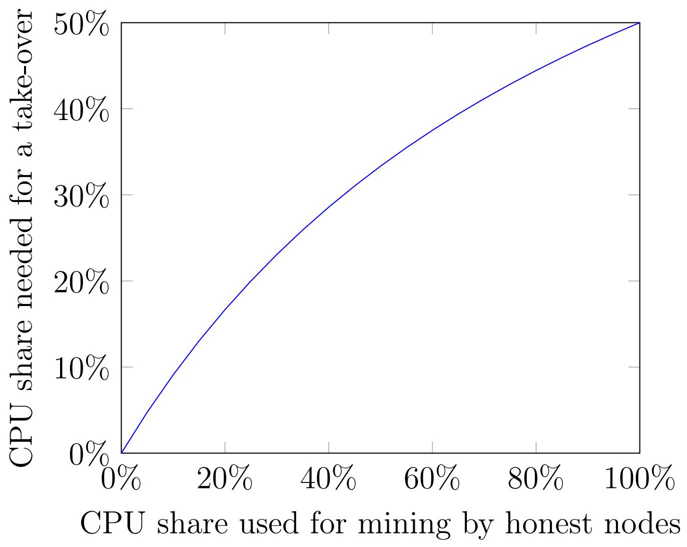


Figure 4: *Plot showing the relationship between the amount of power honest nodes "waste" at mining and the power an attacker needs to be able to take over the system.*

One could of course have honest nodes use only a fraction m of their power for mining and the rest for computing results. This would lead to the relationship $t = \frac{m}{1+m}$, with t being the fraction of the network's power an attacker needs to control it. As seen in the plot in Figure 4, this would either lead to a high "waste" of power or low security. For example, even if the honest nodes "wasted" as much as 30% of their power on mining, an attacker would only need to control 23% of the network's power to be able to control it. This makes a block chain highly unsuitable for our purposes.

4.3.2 BFT-CUP

The block chain solves the Byzantine consensus problem in the quite unusual way of making assumptions on the computational power in the network. Contrastingly, most traditional solutions make assumptions on the maximum number of simultaneous failures instead. Most of these also assume a known set of processes, which we cannot assume, but Alchieri et al. have defined — and solved — the highly relevant BFT-CUP[4] problem (2008).

In order to handle up to f simultaneous failures, their solution requires that the processes have certain knowledge of each other. The weakest possible constraints that this knowledge needs to meet are described below in terms of k which is related to f by $k \geq 2f + 1$. For instance, if the system is to withstand $f = 10$ failures, $k \geq 21$ is necessary.

In the constraints, G refers to the directed graph representing which nodes have knowledge of which.

- G is weakly connected.
- If G is reduced to its k-strongly connected components, the result is a directed acyclic graph with exactly one sink.
- For any pair of k-strongly connected components (G_i, G_j) with a path from G_i to G_j, there are at least k node-disjoint paths from G_i to G_j.

As mentioned, this is only the weakest constraints possible; G could as well be k-strongly connected, or be a complete graph. The problem is that even though these are the absolutely weakest constraints needed to be able to reach consensus, they are still very limiting. At the point in the project where these matters were researched, it had already been decided that a distributed hash-table (DHT) should be used as the foundation for the overlay network (see Section 6.3.1 for details), but none of the common DHT algorithms satisfy these constraints (Tanner 2005).

A possible course of action would of course be to design a new overlay network that enforces these constraints. This is well outside the scope of this project, but an interesting future direction that should be researched further. For the purpose of this project though, the idea of performing validations in a decentralised manner is nonviable.

[4]Byzantine Fault-Tolerant Consensus with Unknown Participants

5 Theoretical results

The disheartening results of our research into distributed algorithms led us to construct our own model for ensuring a high probability of the system producing reliable results.

When designing a peer-to-peer based computation system with free participation, there are several problems to consider. In this section we will explore some properties of such a system and present our solutions to these problems. Below, we will go through how to ensure validity of computation results and different models that further ascertain correctness in results. Most of these concepts describe the part of the architecture called task management, mentioned in Section 2.

5.1 Validity of results

With all nodes being mutually unknown and untrusted, a job owner cannot trust that the workers' results are indeed the correct solutions to their corresponding tasks. The results will have to be validated in some way before they can be accepted, either as a part of the solution to the larger problem or as input to dependent tasks. The result that is considered most correct after validation is called canonical.

Tasks in the prime number example will always return a list of numbers. As a whole, a result can be considered to be either *correct* or incorrect. However, it can be difficult to know if all prime numbers in the given interval are returned, that is if the result is *complete*. As an example, when computing all primes in the range $[1, 10]$, the result $\{2, 3, 4, \dots, 10\}$ is clearly incorrect as it contains non-prime numbers. For the same task, the result $\{2, 3, 7\}$ is incomplete as the number 5 is missing.

Any element in the list can be tested to be true or false by using absolute or probabilistic methods but it is more difficult to test that no numbers are missing. Incorrect results are thus often rather easy to discover with a simple validation function that checks some property of a result. However, running a similar function on each element in an incomplete result will fail to discover any missing solutions.

5.1.1 Replication and the active job owner

It is enough that one worker node calculates a task if it is possible to create a validation function with low time complexity that checks if the result is both correct and complete. Otherwise, the task must be replicated to multiple workers in order to recognise deceitful nodes, which return faulty results.

Thus, from one task, many replicas are created and each is given to a different worker.

Since replication as a method for validity relies on that at least one truthful node works on a task, allowing untrusted nodes to manage the distribution of replicas would open up for attacks. For example, a group of deceitful nodes could vote for all replicas of a task to be sent to themselves. Additionally, in order to avoid assigning the same task to very many workers while another task goes unassigned, some record needs to be kept over which workers are working on which tasks. Should this information be distributed over the network (and by extension to untrusted nodes), there can be no guarantees that it would be correct.

Hence, we introduce an active job owner, who manages the replication of tasks and comparing the replica results. Note that this weakens the notion in which this project can be considered peer-to-peer. It was considered motivated as it makes the set of problems which the system is useful for substantially larger. Note further that for tasks that are in no need for replication, the process of computing and validating their tasks can be done completely in a peer-to-peer manner. The prototype developed for this project always works with an active job owner for the sake of generality and simplicity.

5.1.2 Quality function

Since the job owner should be able to compare the results of each task replica in a way that distinguishes which result is the correct one, or at least which is the better one, the need arises for a function that enables comparison of different results for the same task. This function will be referred to as the quality function and should be implemented by the job owner. For some optimisation problems, it can be difficult to validate that the result was the expected one given the input, but the quality of the result can be checked easily by applying the target function to the solution.

The example task of optimising the Langermann function, which is described in Section 3.2 , will always return one coordinate rather than a list of elements. Therefore, the result will always be complete. Any result that is syntactically correct cannot be said to be false, it can only be said to have a certain quality. That quality can be evaluated in constant time and thus be quickly compared to other results. For this example, the quality function in the validation evaluates the Langermann function on the proposed coordinate. This kind of validation would also be useful for other optimisation problems such as the Travelling Salesman Problem.

For prime calculation on an interval, the quality function can be implemented in several ways. For example, the computation algorithm can return all non-primes in the interval, with a factor for each non-prime as proof. The quality function would then have to check that the proof is correct and also check a number of samples of proposed prime numbers in the interval with a test for primality such as the Fermat primality test or the Polynomial test (Batten 2013).

As the quality function runs on the job owner's computer, it should be considerably less expensive to compute than the task itself. If this was not the case, distributing the computation would not increase the job owner's performance. Possibly, validation could be distributed to be performed by workers. However, it would be difficult to ensure that the validation had been done correctly as the validation itself would have to be validated.

In the prototype, all validation is done by the job owner. This has the advantage that even though the devious programmer that constructs deceitful nodes can read the computation code, the quality function is unknown to him. Faulty results can be constructed intelligently in general but can never be designed to satisfy an unknown quality function.

The quality function can also return an error code when a result is considered invalid, for example if it does not meet the conditions in the task. This means that tasks that can be validated fast with certainty, such as recreating hashes, only need to use one replica per task. If the result is correct, a quality value will be returned, otherwise an error code is returned. If the quality method exits abruptly, a different error code is returned which is interpreted as a bug in the quality method code.

5.2 Probabilistic model

In this section we calculate the probability that a set of cooperating deceitful nodes, so called Sybil nodes, would be assigned all the replicas of some task. If they were, they could return the same faulty result and thus fool the job owner that their result is correct. Otherwise, if at least one truthful result was returned, the false results would be discarded. This model motivates the use of the concept *proof-of-work*.

Assume a job owner will give a task replica to any node that asks for one. Assume further that the job owner has R replicas for each task to hand out and that no worker may get two replicas from the same task. Let k denote the average time it takes to work on any task and let d denote the time it takes to be assigned a new task. Let N denote the total number of nodes that are working for this job owner whereof B are cooperating deceitful nodes. Then on average λ_r requests will be sent from truthful nodes to the job owner each time interval:

$$\lambda_r = \frac{N - B}{d + k}$$

Let t_b denote the time it takes for R deceitful requests to finish. This depends on the ceiling function since B requests can be sent in parallel:

$$t_b = \left\lceil \frac{R}{B} \right\rceil d$$

Let λ_r' denote the average amount of requests from truthful nodes during a time interval of length t_b:

$$\lambda_r' = \lambda_r t_b$$

Let p denote the ratio of the time it takes to be assigned a new task with the time it takes to compute a task:

$$p \equiv \frac{d}{k}$$

Let X be a stochastic variable that denotes the number of truthful requests within time t_b. The probability of no truthful request within time t_b can then be modelled using a Poisson distribution:

$$\begin{aligned} P\left(X = x; \lambda\right) &= \frac{\lambda^x e^{-\lambda}}{x!} \\ P\left(X = 0; \lambda_r'\right) &= e^{-\lambda_r'} \\ \lambda_r' &= \frac{p}{1 + p} \left(N - B\right) \left\lceil \frac{R}{B} \right\rceil \end{aligned}$$

If p was zero, the probability of Sybil success equals $e^0 = 1$. A deceitful node in this model could send infinitely many requests right at the start and thus precede any truthful nodes. In reality, p is not absolutely zero but indeed very small since sending a request takes much shorter time than solving a complex, perhaps NP-complete, problem.

Given this model and assuming a fixed number of nodes N and B, one must increase p which is the relative time it takes to be handed a task compared to the time it takes to compute a task. This uses the concept of *proof-of-work* that has been proposed in earlier work to minimise the effect of false nodes (Douceur 2002). It should however be noted that increasing the challenge difficulty wastes computation power in the system. The impact of p on λ_r' is shown in Figure 5. Note that the gain of increasing the difficulty drops after about $p > 0.3$. In other words, the gained reliability by improving

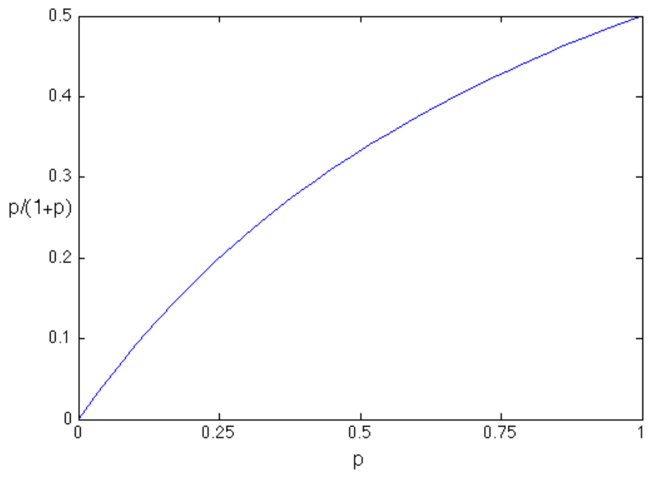


Figure 5: *Plot of $\frac{p}{1+p}$: how higher relative difficulty in proof-of-work p affects λ'_r. Higher λ'_r leads to lower success rate for a Sybil attack.*

challenges is not worth the cost of the wasted computation power, after a certain point.

Proof-of-work Requiring connecting nodes to solve a small computational puzzle (i.e. presenting a proof-of-work, which should be easily verifiable by the job owner) is an old way to counter denial of service (DoS) attacks and reduce spam in a network. If the deceitful nodes try to corrupt a result, they will have to request tasks as often as possible in order to maximise their probability of acquiring all the replicas of a certain task. Thus, imposing a cost on nodes seeking to acquire a new task will be more expensive for the deceitful nodes than for the truthful nodes.

Increase number of replicas A partial countermeasure would be to increase the number of replicas, R, per task which is quite intuitive. Adding more replicas without introducing the proof-of-work concept would however not be sufficient.

Impact This mathematical model gives some insight for probabilistic defence against deceitful nodes. Most importantly it shows that if any node is trusted to cooperate in the computations, some proof-of-work must be added in order to safeguard the validity of the result. It also demonstrates how adding replicas compared to increasing the proof-of-work difficulty scales for many nodes.

5.3 Reputation model

The previous model demonstrated how proof-of-work was necessary given that any node was trusted to cooperate. One way to continue is to identify which nodes are truthful and which are not. This can be done by storing a value representing the trust for a node called reputation. Since the network is entirely asynchronous, the reputation of a node cannot be global lest it would risk corruption. If nodes indeed could vote on reputation, many Sybil nodes could vote on each other and then dominate the network. Instead, trust must be handled separately for each job owner.

Digital signatures Since the network is asynchronous and does not require any registration in the conventional sense, it is impossible to identify deceitful nodes because they can create a new identity simply by rejoining the network or spawning a new Sybil node. Truthful nodes can however keep their identity, even after being offline, by saving a private identifier. This identifier must be able to be authenticated by the job owner without risking interception of the message.

Using asymmetric cryptographic key-pairs as identities is both quite practical and secure. Results can be signed with the node's private key before being uploaded and then be verified using the corresponding public key (Kerry and Gallagher 2013). This ensures that the result was not uploaded by someone else, so that if it is found to be unsatisfactory in some way, action will not be taken against an innocent node.

Reputation Since the job owner can with certainty recognise nodes that keep their identities, the job owner can choose to trust previously truthful nodes more than unknown nodes. Nodes can gain reputation by working on the job owners tasks. When the results for the same task differ, only the workers that produced the best solution, measured by quality, are rewarded with increased reputation. The others are considered deceitful and thus lose reputation. This comparison of results is done at the time of validation.

Before validation is allowed to occur, enough replicas must have been returned and the sum of the respective workers reputation must be high enough. As a result, new worker nodes can only gain reputation by returning the same result as nodes who already have some reputation. Hence, it will sometimes be necessary for the job owner to work himself since at the starting point, only himself is trusted. To work oneself is a decision taken dynamically whenever the following conditions hold true:

1. there is currently no task that has its reputation demand fulfilled,

2. the sum of the reputation of all *active workers* is less than what the job owner expects for one task.

When the first condition is false, any worker, including a newcomer, could finish that task and get increased reputation which may satisfy the current task. For the second condition, active workers must be defined. A worker is considered active when it is given a replica to solve. The worker becomes passive by not asking for work again for a certain amount of time. A passive worker that is given a new replica is considered active once again.

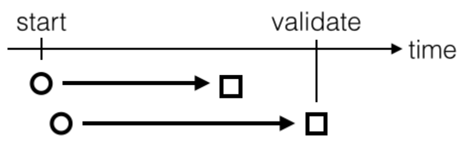
5.4 Replica timeout

When a worker is given a replica, the job owner assumes that the worker will return a result for it. However, at some point a replica must be considered lost as it is assumed that workers can drop out at any time. This is referred to as a *timeout* which is a concept also used by BOINC 2014. Two example scenarios are shown in Figures 6a and 6b.

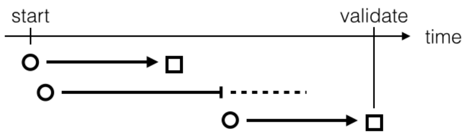
Note that the timeout of a replica differs from the timeout of a worker. A replica timeout makes the job owner assume that the worker will not return a result. A worker timeout makes the job owner assume that the worker will not come around to ask for a new task any time soon. The time interval for both types of timeouts should however be related. In the prototype, the timeout interval for workers equals twice the timeout interval for replicas.

When validation can be made but there are pending replicas that have not yet timed out, either the validation could be made directly or postponed until they have either returned or timed out. These alternatives are depicted in Figure 6c where a validation would occur at time a and a postponed validation at time b. In the prototype in order to achieve maximum certainty in the validation, postponing was chosen.

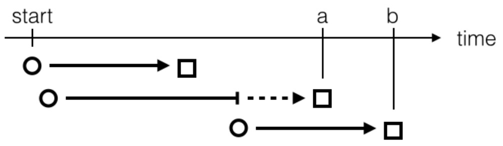
For direct validation without waiting, a potential Sybil attack could be made by two workers who know that they are working on the same task through deduction on the input files and subsequently return identical but false results. One might think that such a behaviour would be a threat since one does not wait for the other replicas before validation. However, the reputation model requires a certain amount of trust before validation is made which would either require the Sybil nodes to gain reputation before doing the attack or require there to be another well trusted worker with a returned result. In the first case, the Sybil attack is counter productive as it helps the job owner more than it can hope to destroy. In the second case, the trusted worker is very likely to give a truthful result which would negate the attack.



(a) *Two workers are given replicas of the same task. Validation occurs when both return their respective results.*



(b) *The second replica times out. That replica is assumed to be lost so a new replica from the same task is given to a third worker.*



(c) *The second replica times out so a new replica is given to a third worker. Validation could occur at point a when the second worker returns his result. The job owner must choose if it should validate directly or wait for the third replica to return.*

Figure 6: *These are three possible scenarios of timeouts depicting when validation can occur. The giving of replicas is depicted with circles while the publishing of results is depicted with squares. Timeout is depicted with a dashed line.*

For postponing the validation, a new attack strategy may be produced. Every new Sybil node could acquire replicas at regular time intervals and thus postpone the validation indefinitely. Two trivial solutions exist: either the job owner could deny giving a new replica from a task that can be validated or the job owner can remember which workers arrived after validation was possible. In the latter case, the validation will only be postponed until all the original workers have returned or timed out. The excess workers can still gain reputation and or improve the result after validation as *latecomers*.

Latecomers After a task has been validated, additional results from replicas may return from excess workers or timed out workers, such as in Figure 6c. These results can be assimilated with the canonical result by comparing if they are equal and also by comparing their quality. If the added result equals the canonical, the worker will gain reputation. Otherwise, the two results are compared in further detail by calculating the quality of the late result. If the quality is higher than that of the canonical result the worker's reputation increases, the stored result is replaced and the previous reputation of workers advocating the false result decreases. If the new result is deemed worse than the previously proposed, the worker loses reputation and the stored result is unaffected.

Removal of a task A task must not be given to new workers after it has been validated. Latecomers that have already started on it may finish and assimilate their result as was described above but no new workers should be able to start working on it. If they were allowed, one worker node could solve the task truthfully and store the result locally. When the job owner would be out of tasks, new Sybil nodes could be assigned old tasks and simply return the stored result. This would allow an intelligent attacker to give his Sybil nodes free reputation and thus nullify the protection of the reputation system.

5.5 Smart assignment of tasks

When a job owner has several tasks and there are several workers with different reputation, the job owner should assign tasks to workers as intelligently as possible in order to speed on the process of computing the tasks. Optimally, tasks that require much reputation should be given to workers with high reputation and vice versa. If the workers' reputation is not taken into account, work may be performed in vain as extra replicas may need to be produced in order to fulfil the reputation requirement. When a worker then asks for work, the job owner should find a task that suits the worker's reputation:

$$\begin{aligned} w &= \text{reputation of worker} \\ p &= \text{additional reputation needed for a task} \\ r &= \text{additional replicas needed for a task} \\ \bar{p} &= \text{average reputation per replica} \\ \bar{p} &= \frac{p}{r} \end{aligned}$$

Optimally, a task should be finished without extra replicas and minimal excess reputation. If it was possible, a task should be found such that the worker reputation equals the average need of the task: $\bar{p} = w$. If that could always be the case, no extra replicas would ever have to be produced and there would be no excess reputation. This scenario is depicted by the middle arrow in Figure 7. As shown in the picture, the average does not change value in this case since $\frac{3}{3} = \frac{2}{2}$.

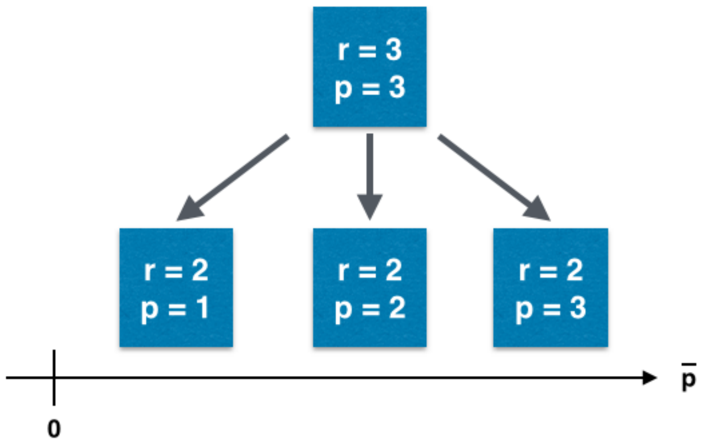


Figure 7: *Depiction of how a task changes $\bar{p}$ value after being given to a new worker depending on the workers reputation.*

Normal case Often in a running system, there is no task with exactly the same expected reputation as a given worker. In order to meet the reputation need without extra replicas, a task should be assigned such that the worker reputation exceeds the anticipated need: $\bar{p} \leq w$. Eventually this can lead to excess reputation but that can be minimised by choosing a task with the average $\bar{p}$ as close to the worker reputation w as possible. Any excess reputation will decrease the average value $\bar{p}$ for the next worker which subsequently can have a lower reputation to work on this task than otherwise. This is illustrated by the left arrow in Figure 7. Thus, the higher reputation a worker has, the lower reputation his co-workers on the task can have. This is an essential characteristic as this regulates trust and tests untrusted workers, who could be Sybil nodes, against the most trusted workers who have proven themselves during a long time.

If there are no $\bar{p} \leq w$, the job owner should do the second best thing: to give a task with $\bar{p} > w$ as close to w as possible. This reputation deficiency will increase the average need $\bar{p}$ for this task so next time it will be given to workers with higher reputation. This scenario is depicted by the right arrow in Figure 7. Also for this case, the self-regulating characteristic described above is proven to be true.

The concept of the average need $\bar{p}$ is extended to be more abstract than a simple average. To deal with some special cases, it must be considered an arbitrary number used to represent the order in a collection of tasks so that tasks can be assigned workers in an intelligent way.

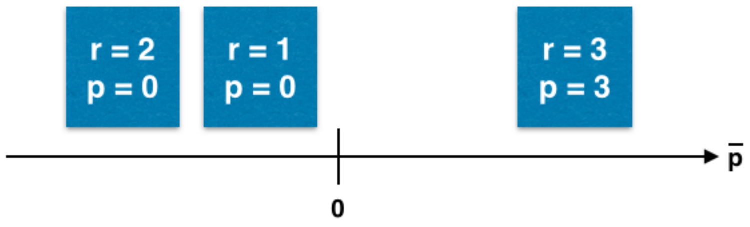


Figure 8: *Example of a scenario when the reputation requirement of a task is met but not the requirement for replication. The two tasks on the left do not need more reputation to be validated compared to the task of initial state on the right.*

Special case 1 The first special case occurs when the reputation requirement is met but not the minimal number of replicas. Any node could move such a task closer to validation by working on it, disregarding the node's reputation. Newcomers who have reputation zero will primarily work on these tasks that already have enough reputation since $\bar{p} = \frac{p}{r} = 0$.

However, tasks closer to completion should be prioritised, so instead $\bar{p}$ will be considered negative with the distance to zero reflecting how many replicas there are left. One successful model for this is $\bar{p} = -r$. This model is illustrated in Figure 8. In the figure, the task with only one replica left will always be chosen before the task with two replicas left. If the worker has a reputation greater than or equal to $\frac{3}{3}$, he will be given the task to the right instead.

Special case 2 Another special case is the inverse of the first special case: when the requirement for minimal number of replicas is met but not the requirement for reputation. This can happen when the majority of workers cannot find a suitable task such that $\bar{p} \leq w$. When this condition occurs, the task would only need one extra replica if the worker had high enough reputation. Therefore, let $\bar{p} = p$. Even if there is no such worker with that high reputation, eventually, the workers with highest reputation will get to work on this. If the sum of the workers are insufficient, the job owner himself will work on it as described above so that the workers can gain reputation for the next task.

Special case 3 The very last case is when all requirements for a task are met. It is possible for such a task to still not be validated because it might be waiting on a replica that has not yet returned. This kind of task should be chosen as the very last alternative if any, thus $\bar{p}$ is proposed to take the value of positive infinity.

Summary The four cases are summarised below:

$$\bar{p}(p,r) = \begin{cases} \frac{p}{r} & p > 0, r > 0 \\ -r & p \leq 0, r > 0 \\ p & p > 0, r \leq 0 \\ \infty & p \leq 0, r \leq 0 \end{cases}$$

The parameters p and r are not trivial to optimise as several goals need to be achieved. First of all, security of validation must be maximised while minimising replication and job owner effort. To produce and analyse a mathematical model for this is considered future work.

Reputation in the Prime number example For the prime number example, reputation can be used quite straightforwardly to increase the reliability in the results. For prime numbers, it is critical that no errors are missed as the results are used as input for future tasks.

Reputation in the Langermann example The Langermann example, described in Section 3.2 is a stochastic computation and hence depends on a seed for generating pseudo-random numbers. Because of this, the problem of optimising can be structured in tasks in two different ways: either as a single task that is replicated many times or as many tasks.

The most natural would be to follow the one-task approach where different nodes get the same input data but generate the seed dynamically. If this is done, only the worker that produces the very best result can be given reputation to not risk rewarding deceitful nodes. The other workers should not lose reputation since they might have been working truthfully but have had worse luck with the seed. The main advantage of this approach is that all results are compared automatically by the validation which creates less work for the job owner.

For the second approach with many tasks, the respective seed for each task would have to be generated in advance by the job owner. As the same seed is used for all replicas of a task, reputation can be given fairly as they can be compared against each other. However, this would waste computation power. Similarly to the other case, one could also set the number of minimal replicas to one but requiring a certain amount of reputation. Then truthful nodes could get the reputation they deserve while Sybil nodes still would have to be checked against nodes with high reputation. Compared to the previous approach of only one task, it creates a small overhead for the job owner but makes the reputation system more fair. In either case, a useful solution is very likely to be produced while cheating nodes do not constitute a problem as long as they do not gain reputation without deserving it.

6 The prototype

We implemented a prototype in Java (Source code 2014) to demonstrate our theoretical results (see Section 5). As a result, it is platform independent. This section describes the prototype-specific solutions; other implementations could be used to solve the same problems as the ones described here. The process and technologies used when developing the prototype are discussed in appendices A and B.

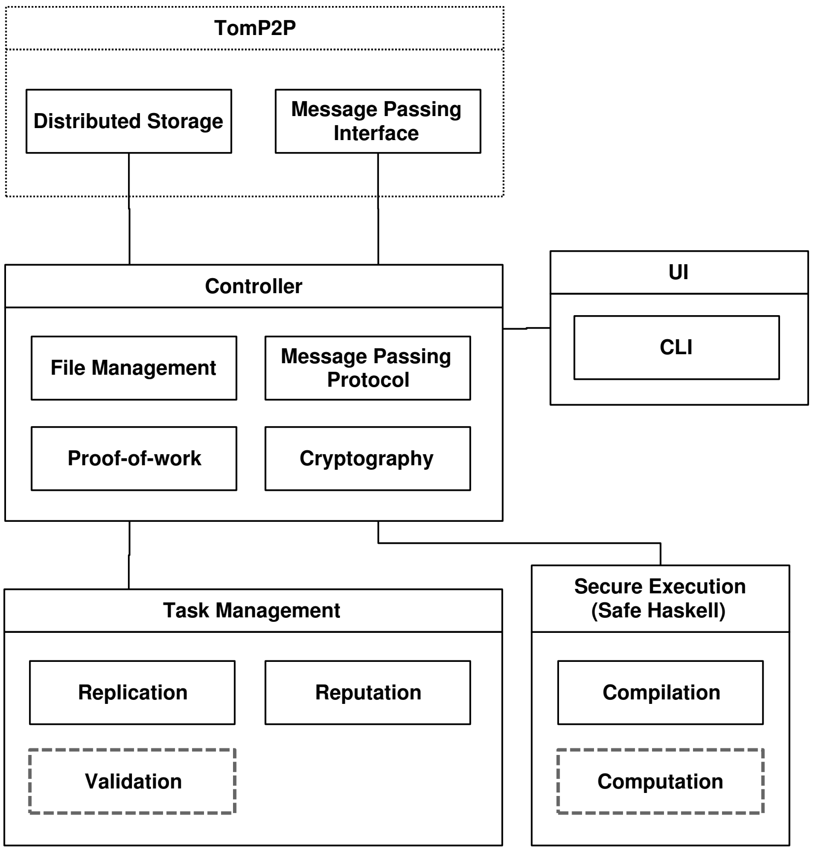


Figure 9: *Illustration of the architecture of the prototype. The dashed subparts, validation and computation, are written in Haskell by the job owner.*

6.1 Prototype architecture

The realised prototype implements the general architecture that was introduced in Section 2. The specific architecture used for this prototype is briefly summarised in this section and illustrated in Figure 9.

The connectivity parts: distributed storage and message passing interface are realised with an external library, which will be discussed in Section 6.3.1. The details of the task management have been discussed previously in Section 5. Our methods presented there, such as reputation and validation, are implemented in a module called Client in the prototype (Source code 2014). The secure execution part will be discussed in detail in Section 6.2. This part is implemented in the prototype as the module called TaskBuilder.

The controller part is also implemented in the module called Client. The respective subparts will be discussed in different sections. The proof-of-work was motivated in the theory in Section 5.2. The specific implementation of proof-of-work in this prototype will be presented in Section 6.3.2. The cryptography subpart facilitates encryption of messages and signing of results. This will be discussed in Section 6.3.3. The message passing protocol we have designed for exchanging information between workers and job owners is described in Section 6.3.4. The file management will be discussed in Section 6.3.5.

The user interface (UI) consists of a command line interface (CLI). In the source code, this can be found in the module called UI. The user can only interact with the prototype through the UI module, which consequently defines what a user can do.

6.2 Secure execution

Because all nodes have the right to post jobs on the network and participants are mutually unknown and untrusted, great care must be taken in order to ensure that the code in a job cannot do anything else than perform computation on the given input. The potential consequences could be devastating, both for the individual users if their hard drives were erased or injected with malware, but more importantly a lot of completely unrelated entities could suffer if our network was used to create a botnet. This is a key ethical issue to solve.

In order to provide safety for the worker nodes, the computations must be guaranteed to be secure or run in a sealed environment, referred to as a sandbox. Secure languages have the advantage of giving a guarantee that the code cannot do unauthorised I/O-operations. On the other hand, there have been cases where code running inside a sealed environment have broken out and taken control of the machine (Ray and Schultz 2009). Safe environments also seemed to be platform dependent which would contradict the requirements of portability.

Safe Haskell The language that job owners can implement computation code in is Safe Haskell. Before it can be described, some features of the ordinary Haskell language need to be explained. Haskell is a programming language composed of pure functions, in the sense that the functions behave like mathematical functions: the same input of a function call always result in the same output and does not cause any side-effects (Milewski 2013). Input and output operations however, causes side-effects and may return different results, therefore they are considered to be actions and they have the type *IO*. This makes it easy to see if a piece of code can access a system and its files or if it only may perform a computation in memory. However, other features of Haskell lets input and output operations be performed unsafely in pure computational typed functions, preventing safe behaviour to be ensured.

First introduced in GHC 7.2, Safe Haskell is an extension to the Haskell language that disables all features of Haskell that are deemed unsafe and prevents unsafe code from being compiled (Haskell 2014; Terei et al. 2013). This ensures that computational functions remain pure. Using this, an external program can call pure computational functions safely, only allowing computation in memory. One disadvantage however is that, as of yet, Safe Haskell does not guarantee safety during compilation. Terei et al. state that disabling certain functionality, including the C preprocessor, should make compilation safe. To be perfectly safe however, the compilation must run in a sandbox.

For purposes of practicality, some modules need to be allowed to use unsafe operations. In order to allow a program to be compiled that uses a module with unsafe code, that module most be considered *trustworthy*. This imposes a risk: if a module exporting an unsafe function is trusted, that function may be used in an untrusted program. Therefore the choice of code to trust must be handled with care.

Alternatives to Safe Haskell Other safe languages were considered such as Joe-E (Joe-E 2014) and E (ERights 2014). Joe-E was discarded since it required the installation of Eclipse and a plug-in to compile the Joe-E code. This was not acceptable as a minimal installation of Eclipse itself use about 150 MB (Eclipse 2014). E on the other hand was an interpreted language but proved to be quite unsuitable for computations which is demonstrated by our performance test below.

Testing language performance The performance of C, E and Safe Haskell for usage as computational languages were evaluated and compared. While C cannot be used in our implementation since it is not a safe language, it is usable as a comparison against E and Safe Haskell. A test was created that was intended to be as fair as possible. As the test was very simple to program without using any library functions and mostly required CPU power, it should be a reasonably fair comparison between C, E and Haskell. The algorithm checked for every integer up to a certain point if it was divisible with any other number, that is if it was a prime number.

The test was run multiple times on different computers. Safe Haskell performed well in test, only taking 4 times longer than C, making it a strong candidate for demanding computations. E however did not perform well, being 1,100 times slower than C, making even simple computations unfeasible.

6.3 Functionality

In the prototype the reputation model and smart assignment are implemented as described in Section 5. Tasks are grouped together as a job to be easier to organise and see which tasks belong together. A job is deemed completed once every task is finished. For workers to be able to work on a task a job owner has to post a job to make it available for the network. The job owner replicates each task in the job and assigns the replicas to workers. Additionally, the implementation of distributed storage, proof-of-work and cryptography is presented below.

6.3.1 Distributed storage and network: TomP2P

TomP2P is a library used for distributed data storage (TomP2P 2014b). As can be seen in Figure 9, TomP2P implements both the distributed storage and the message passing interface. The data storage in the system has a few requirements that TomP2P fulfills. Firstly, nodes need to know each other over a fault tolerant network that expects nodes to drop out sooner or later as it is not expected of the users of the system to be online at all time. Secondly, the nodes also need to be able to send files and messages to each other, otherwise there is no means to communicate. Furthermore nodes must be able to send files to each other without being online at the same time to make the work of the prototype more fluid. However, it is not possible to send messages to nodes that are offline.

TomP2P implements a DHT (ibid.). It is a fully decentralised Key-Value storage model that often is categorised as a type of NoSQL database (NoSQL 2014). Being master-less is an important requirement as single-point failures need to be avoided. Generally, using cryptographic methods, files in a DHT can be ensured not to be modified but they can still be denied access to, or be deleted, by Sybil nodes (Balakrishnan et al. 2003). TomP2P however offers functionality for protecting values against modification and deletion(TomP2P 2014a). Other libraries such as Voldemort (Project Voldemort 2014) were also studied but they did not offer the same key protection as TomP2P did. Furthermore, both the prototype and TomP2P are open source and therefore anyone who is interested in the source can look up TomP2P's source code as well.

Since TomP2P met the requirements, other distributed storage solutions were disregarded such as the Column store model implemented by the Cassandra library (Apache 2014c; NoSQL 2014). The greatest difference was the possibility to do search queries in the database, as Cassandra facilitates advanced queries in its own query language, ”CQL”, while a DHT required the peer to always know exactly what key data is stored at. One advantage of the strict Key-Value model was that it was more difficult for Sybil nodes to find and delete objects in the database. Cassandra was designed for running on a server cluster where each node is to be trusted whereas TomP2P was designed for a peer-to-peer network where keys could be protected both implicitly by unknown key and explicitly by key domain (TomP2P 2014a). The security features of Cassandra took place between the querying clients and the server cluster (DataStax 2014).

6.3.2 Proof-of-work: Hashcash-cookies

As mentioned previously, a proof-of-work system is needed in order to lessen the risk of deceitful nodes succeeding to report incorrect results. This section presents the proof-of-work system used in the prototype and aims to motivate the choices made when developing it.

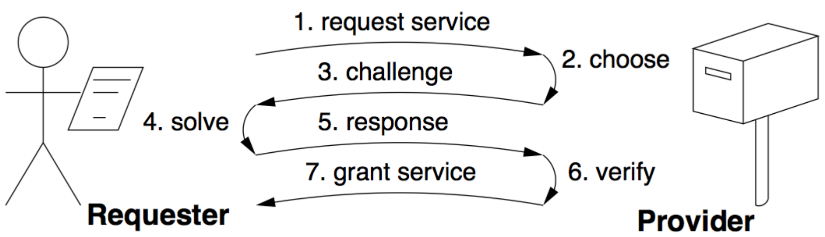


Figure 10: *The basic structure of a challenge-response protocol. (Coelho 2008)*

Challenge-response The proof-of-work systems we deem as most appropriate to use are those based on the so-called *challenge-response* protocol. In these, a challenge is issued to each node requesting access to a resource, as seen in Figure 10. The challenge is then solved by the requester and only upon successful verification of the solution is the requester granted access to the resource. The fact that the challenge is picked by the resource provider allows the difficulty of the challenges to be tweaked according to the current state of the system in general and the requester in particular.

However, if such a protocol only included what is outlined above, the provider would have to remember which challenges it has issued in order to prevent nodes to create their own challenges. This would be $\mathcal{O}(n_{pending})$ in space complexity, with $n_{pending}$ being the number of issued challenges that have not yet been solved. Since any node could request any number of challenges, this would make the proof-of-work system itself vulnerable to DoS attacks, the very problem it is trying to counter! Fortunately, this problem has a relatively simple solution: the provider sends a message authentication code (MAC) of the challenge together with the actual challenge and requires both of them to be returned together with the solution. This allows the provider to verify in constant time that the challenge was indeed issued by himself, without having to store any information about it while waiting for a response (Back 2002).

Reusable solutions The ability to tweak difficulties and the lack of space overhead makes the challenge-response protocol seem very attractive, but the advantage of not having to save either challenges or solutions introduces a new problem. Without any information on which solutions it has accepted, how does the provider know if a particular solution is being reused? In the setting where proof-of-work systems are traditionally used (and are indeed designed for), a unique service name is included in each challenge, specifying which resource its solution grants access to. Relying on this method would require our system to reserve a specific task for a specific requester *before* a challenge has been solved, which would be very inefficient and vulnerable to DoS attacks.

Instead, the job owner in our prototype relies on a ”score” associated with each registered worker. The score is included in the challenges sent to the worker and changes in an unpredictable way upon successful verification of a solution. This makes old solutions unusable, while requiring $\mathcal{O}(n_{workers})$ space complexity and constant time for lookups, using a hash table. As the score is not saved until the worker has registered, Byzantine node(s) seeking to maximise $n_{workers}$ would have to solve one registration challenge per increment. This makes $\mathcal{O}(n_{workers})$ a *huge* improvement over $\mathcal{O}(n_{pending})$, which would increment on each request made.

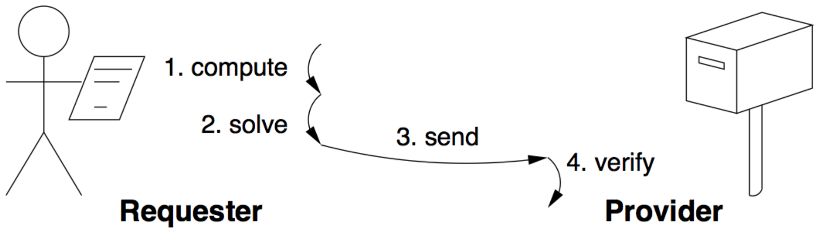


Figure 11: *The basic structure of a solution-verification protocol (Coelho 2008).*

Solution-verification In contrast to the challenge-response protocol, the *solution-verification* protocol works by allowing a proof-of-work to be computed without the need for the resource provider to issue the challenge, as seen in Figure 11 (Coelho 2008). This is primarily applicable whenever the requester can be assumed to know exactly which resource it is requesting access to, or when timestamps are a good way to differentiate between different requests from the same node. The traditional setting is when the requester wants to send an e-mail through the resource provider's e-mail server — the requester chooses a challenge consisting, for example, of a timestamp and the sender's and receiver's e-mail addresses. The challenge and its solution is then sent along with the e-mail, which proves that the requester has performed some work in order to send the e-mail, with higher difficulties presenting stronger proofs that the requester is not a spammer.

Upon receipt of this proof-of-work, the provider has to check *1)* that the choice of problem is acceptable, *2)* that the solution indeed solves the problem and *3)* that the same problem has not been used by a requester before. The first two conditions can definitely be checked efficiently in our setting, but the last one requires checking new solutions against all the previous ones. This can be a cheap process with a good data structure ($\mathcal{O}(1)$ with a hash table), but it will of course be of $\mathcal{O}(n_{solutions})$ space complexity. As each worker will be required to present a proof-of-work for each task it works on, it is reasonable to assume that $n_{workers} < n_{solutions}$, even though the worst case would be $n_{workers} = n_{solutions}$. This, together with the need for a unique identifier for each request and the inability to tweak difficulties as needed, led to the discarding of the solution-verification protocol.

Hashcash The Hashcash system was found to be satisfactory to the needs of this project. It is well-tested and most notably used in Bitcoin mining and Microsoft's "Coordinated Spam Reduction Initiative", albeit with a slightly modified and incompatible format in the latter case (Microsoft 2004; Nakamoto 2008a). Solutions are ensured to actually be proofs of work through the *pre-image resistance* property of cryptographic hash functions, stating that, given a hash h, it should be hard to find a message m such that $hash(m) = h$ (Back 2002; Rogaway and Shrimpton 2004).

Specifically, a challenge in Hashcash consists of a bitstring s and an integer difficulty d. The requester's task is to find another bitstring t, such that $hash(s||t)$ has a prefix of at least d zero-bits, with $||$ denoting concatenation of strings. The fastest known algorithm to solve this problem is brute force, causing the expected time to find a solution to rise exponentially with d (Back 2002).

The prototype employs a modified version of Hashcash with MAC (or "Hashcash-cookies" (ibid.)), with $s = hash(id_{jobowner}||id_{worker}||score_{worker})$. Due to time limitations, d does not depend on either the worker's reputation or the general state of the system, even though such a feature was seen as very attractive. It is instead fixed, with registration requiring more work than authentication.

6.3.3 Cryptography: Apache Shiro and Bouncy Castle

Recalling the probabilistic model once more, digital signatures will be needed in order to avoid punishing innocent nodes. The following section discusses this and other cryptography-related features in the prototype as well as how they relate to each other.

Digital signatures In order to provide integrity and authenticity of results, we use the digital signature algorithm (DSA) (Kerry and Gallagher 2013). Workers sign hashes of their results using their respective private DSA key, uploading both the result and the signature to the DHT before notifying the job owner. Being an asymmetric algorithm, a registering worker has to provide the job owner with its public DSA key in some sort of handshake before the job owner can verify the worker's results.

Secure message passing Integrity and authenticity of results are not enough however: numerous messages are sent back and forth between job owners and workers, some of them containing information that only the intended recipient should be able to access. The standard way to provide confidentiality of messages as well as integrity and authenticity, is to use an authenticated encryption algorithm (Bellare and Namprempre 2000). AES-GCM was deemed to be the most appropriate such algorithm for the prototype, because of AES's strong security together with GCM's relative efficiency for short messages and unpatented status (Bogdanov et al. 2011; Švenda 2004).

Key agreement Since AES is a symmetric encryption algorithm, the participants are required to establish a shared secret key before any secure communication can take place. To do this, while at the same time preventing any adversary to retrieve the key, is a classic problem in modern cryptography, solved by the key agreement algorithm Diffie-Hellman. In it, each of the two participants have one private key — which is kept secret — and one public key — which is sent to the other participant, possibly over an insecure channel.

The algorithm ensures that the two participants obtain the same secret key through a property of modular arithmetic, stating that $(g^a)^b \equiv_p (g^b)^a$. Here g and p are publicly known, while a and b are the private Diffie-Hellman keys of the participants and g^a and g^b are their respective public keys (Diffie and Hellman 1976). It is hard to retrieve a from g^a due to the discrete logarithm problem which is thought to be NP-intermediate, that is, being in NP but neither in P nor NP-complete (Atallah 1998). It is worth noting that is not proven that NP-intermediate exists as it is still debated whether P equals NP or not.

Handshake Subsequently, in order for the nodes to be able to agree on a secret key, the aforementioned handshake will have to consist of a public Diffie-Hellman key as well as the public DSA key. The former will have to be accompanied by a signed hash in order to discover man-in-the-middle attacks where an active adversary replaces the Diffie-Hellman key in a handshake. This will guarantee that the secret key is negotiated with the correct node, even though we do not know anything about the node in question, only that it is in possession of *both* keys.

A summary of the protocol for exchanging public keys and establishing a shared secret key follows below (verification of signatures omitted for clarity):

$$\begin{aligned}
A \to B: &\quad \{DSA_{A_{pub}}, DH_{A_{pub}}, sign_A(hash(DH_{A_{pub}}))\} \\
B \text{ computes}: &\quad S_{AB} = agree(DH_{A_{pub}}, DH_{B_{priv}}) \\
B \to A: &\quad \{DSA_{B_{pub}}, DH_{B_{pub}}, sign_B(hash(DH_{B_{pub}}))\} \\
A \text{ computes}: &\quad S_{AB} = agree(DH_{B_{pub}}, DH_{A_{priv}})
\end{aligned}$$

A and B now share the same secret key S_{AB} and both have verified that the node that owns the private DSA key is the same as the one they agreed on the secret key with. Thus, in further communications both can be sure that they are communicating with the correct node. Additionally, when either one of them uploads a result for the other to see, it can verify that the result was uploaded by the same node with whom it is communicating.

Implementation Since all versions of JDK 7 does not provide support for block ciphers in GCM mode (Wetmore 2011), our cryptographic functions rely on the Bouncy Castle API (Bouncy Castle 2013). It has a footprint of only 2.6 MB, a cheap price to pay to avoid either risking broken code on certain machines or building our own authenticated encryption algorithm through composing an encryption function with a MAC-function. The latter option is certainly non-trivial and opens up to programming errors (Krawczyk 2001). Apache Shiro is also used, in order to provide a simpler and more intuitive wrapper interface to Java's standard cryptographic libraries (Apache 2014b).

6.3.4 Message passing

Message passing is the protocol used by the job owner and the workers for communication pertaining to tasks. It requires both the job owner and the worker to be online simultaneously when requesting and assigning tasks. When a worker has finished executing a task, the job owner will be notified immediately with a message if he is online. Otherwise, the job owner will obtain the result as soon as he comes online by querying the distributed data storage. If the task could not be compiled or if it crashed unexpectedly while the job owner was offline, that information could be uploaded to the same location as the result would have been uploaded to.

As illustrated in Figure 12, a worker initiates the communication with the job owner by sending a challenge request, receiving a challenge as reply. The worker solves the challenge and sends a task request containing the solution to the job owner, who then returns a task. Finally, the worker executes the task, uploads the result and notifies the job owner.

In this process, two things can halt the communication. Firstly, if the job owner has no tasks available, for example if no job has been posted or if all jobs are completed, then the worker will be notified and will not receive a task. Secondly, if the task is implemented incorrectly so that the worker cannot execute it, the job owner will be notified and no result uploaded.

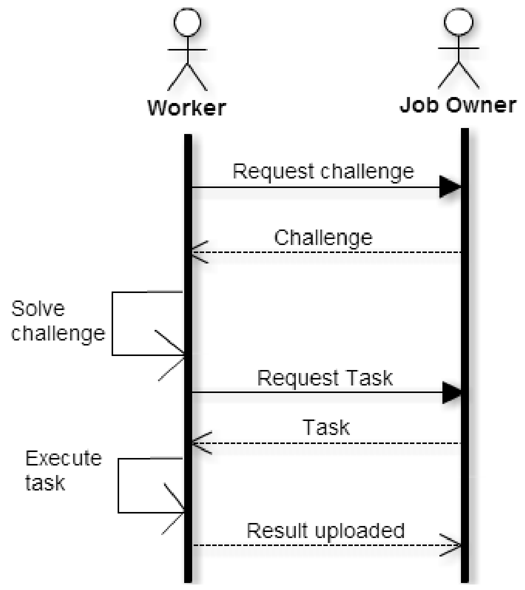


Figure 12: *The message passing protocol when successfully working on a task. The normal line represents a message that needs an answer and the dotted line represents a message that do not require a reply.*

6.3.5 File management

During installation, the directoy `.gdcn` is created in the user's home directory where local files are saved. Files are created and updated during runtime to maintain the state of the program when it is restarted. Most important are the cryptographic keys, as they are necessary for the peers to keep their identities in the network. The task files are located in subdirectories, one for each job. The information about a task is stored in a JSON-file so that the client knows which files are to be used when executing the Haskell code.

6.4 Connecting to the peer-to-peer network

A peer-to-peer network was chosen as the underlying structure and consequently there is no central server for peers to connect to. Two peers in the same local network can find each other by broadcasting, but this is not possible if they are on different networks. However, peers on different networks still need to be able to find each other, which is commonly accomplished through bootstrapping (Buford and Yu 2010).

Bootstrapping is when a peer initially connects to a peer-to-peer network. As there are no servers to connect to, a joining peer instead connects to an already existing peer in the network. The existing peer then provides the connecting peer with information about the state of the network and protocols used, allowing it to communicate with the rest of the peers in network. The peer that provides this information is called a bootstrapping peer.

For new peers to find the network, a number of stable peers should be used as a core network. New peers can bootstrap to this core network and find the rest of the network through them. The core network is not acting as a server but is a normal peer and consequently does not have any more control over the network than any other peer in the network. The bootstrapping peer currently used by the prototype is predefined, so that new users can bootstrap to it easily.

Peers that have previously connected to the network initially attempt to connect to peers they have connected to earlier. This means that even if the core network should go offline, the only result would be that new peers cannot find the network, as old peers would be able to reconnect without any trouble. Naturally, if no earlier known peer is online, old peers need to bootstrap to the core network as well.

7 System properties

In this section the results of this thesis are recapitulated. Specifically, we summarise which problems have been solved and to what extent.

7.1 Security against malicious code

A major concern was how to ensure the safety of the workers when the author of the computation code is unknown and untrusted. As any malicious code would require IO operations, malicious code produces compiler errors in Safe Haskell (Terei et al. 2013). Safe Haskell can guarantee that no unsafe code is compiled by users. Anything that Safe Haskell does not catch is beyond the scope of this project. Notably, Safe Haskell cannot guarantee that the compilation itself is completely safe.

The input files for computation code however poses a potential security threat. Currently, any binary files are accepted as input. This means that the input files may contain malware that go undetected. Possibly they could be started by preprocessor commands at compilation time of the computation code. How to counter this kind of attack is considered future work.

7.2 Reliability of results

The proposed system and implemented prototype cannot guarantee that every result is correct. Sybil nodes could be assigned all replicas of a task and return identical but false results. It would however require them to work truthfully for some time, which would help the job owner. As the Sybil nodes are unaware of what settings the job owner uses, they can never know when to strike with certainty. Furthermore, if a fraud is successful and the same nodes are caught later on, their previously accepted results may be re-evaluated.

The probability of deceitful success also depends on the kind of computation. If it is a Monte Carlo algorithm optimising a system, one false result has little impact as it will be compared against other solutions from other tasks. On the other hand, for an isolated task, a deceitful result is impossible to discover as it cannot be incorrect per definition, only of very low quality.

7.3 Comparison with BOINC

As mentioned earlier, BOINC is a common platform for making distributed volunteer computations (BOINC 2014a). Here, we point out the key differences between BOINC and our methods and prototype. This is interesting as our system, during development, has converged towards the BOINC design, motivated by security reasons.

The asymmetric relation between job owner and worker exist in both systems. Workers are considered untrusted, which motivates the replication of tasks in both systems. In BOINC the workers always trust the job owner, while in our system the job owner need not to be trusted as the execution is considered safe. BOINC computations are written in C which generally run faster than Haskell, but needs to be compiled in advance by the job owner for different platforms. This also means that the computation code is unknown by the workers in BOINC. In contrast, our system is highly platform-independent and the computation code is available for workers to read.

An alternative implementation of BOINC is V-BOINC that runs BOINC in a sealed off virtual environment (McGilvary 2013). This keeps the advantages of BOINC while reducing platform-dependence and the required trust of the job owner.

The greatest achievement of our system is the protection against Sybil nodes by proof-of-work and reputation. BOINC allows workers to gain ”credit” but it does not affect the system; it is merely a token of appreciation and recognition to the user (BOINC 2014a). BOINC however facilitate a feature called "adaptive replication" that use probabilities to decide when a task should be replicated based on a worker's failure rate. The concept is similar to our reputation system but is implemented differently.

Furthermore, BOINC requires the job owner to run a server continuously. In our system, it is enough for the job owner to come online once in a while. On the other hand, if the reputation method is used, the job owner will be required to work on tasks if the available workers have too low reputation.

8 Discussion of theoretical results

In this section, we discuss our solutions presented in Section 5. As long as deceitful nodes can be assumed to exist and results cannot be ascertained to be correct but need to be reliable, some replication of tasks must be made. However, replication alone cannot guarantee success as Sybil nodes can cooperate and deduce which replica belongs to which task.

Two methods are proposed to counter this kind of attack: proof-of-work and reputation. Neither of them can guarantee absoluteness but they can increase the expected reliability of results as well as making it more difficult for an attacker.

Proof-of-work The proof-of-work method increases the probability for the Sybil nodes to fail in their attack but more importantly it increases the cost for an attack. The mathematical model shows that reliability increases with more replicas, harder challenges and more truthful workers. More replicas and harder challenges however lead to more wasted computation power.

Increasing challenge difficulty overall however leads to a much higher cost for performing a Sybil attack as each Sybil node must solve an individual challenge before getting a task to work on. This increased cost may demotivate potential attackers, but will also waste more of the truthful worker's computation power.

Challenge difficulties could however be changed dynamically for specific workers. As an analogy, when a book is not returned in time to a library, a fee should be paid. Similarly, when a worker has been given several tasks that have timed out without returning any result for them, a more difficult challenge can be given as a "fee" when asking for a new task. This would not affect truthful nodes, even when working on several tasks in parallel, as the difficulty only increases after timeout and truthful nodes solve tasks before asking for more. Deceitful nodes that attempt to prolong the computation process by asking for many tasks without solving any one of them, would however be countered.

Reputation The reputation method considers any unknown worker as untrusted. In order for a Sybil node to do any damage, it must first receive trust by working truthfully, since its results will be compared primarily against well-trusted workers. Only after becoming trusted, it can hope to return false results without the job owner noticing. After spending considerable time to earn reputation, the attacker can never know for sure that the false result will be accepted as the number of replicas of a task is dynamic. The trusted Sybil node will sooner or later be compared against a truthful node and lose all of its accumulated reputation.

The reputation model thus increases the reliability of results. A successful Sybil node would have to be more complex in deducing when it should return a truthful result and when it should give a false result in order to dupe the job owner as efficiently as possible. The cost of making an attack would be very high as a Sybil node would have to solve multiple tasks before getting trust. The chance of a successful Sybil attack would also be quite low, assuming there are any truthful nodes at all.

The greatest disadvantage of the reputation model is that in the beginning, the job owner client will have to work itself, as it is the only trusted worker.

Combination The two methods can be combined by using both challenges and reputation. Workers with higher reputation can be given easier challenges to solve, so the proof-of-work will only significantly affect untrusted workers. This would maintain a high cost for creating Sybil nodes while reducing wasted computation power for truthful workers.

Smart assignment The smart assignment feature makes the system more efficient by reducing both average wait time for tasks and wasted task replication. A positive side-effect is that low-reputation workers most often will be paired with high-reputation workers rather than allowing a group of low-reputation workers to finish a task.

As the procedure is primarily designed for newcomer scenarios, it may have to be adjusted for steady state scenarios. For example, if the job owner has very many tasks and only a few but truthful workers, the workers will gain a very high reputation and hence will be assigned to work on different tasks. This means that many tasks are started but that few are finished. This depends on what parameter values have been chosen for the reputation model. If only one replica is required but large reputation, this scenario would work quite well. However, that setting is disadvantageous for a scenario with many newcomers. Throughput in this case would typically benefit from requiring many replicas but low reputation.

Hence, parameter values for the reputation model can be tailored to fit one scenario very well but it is difficult to find a set of values that work well for any possible scenario. In the future, the procedure could be made more intelligent to dynamically adapt to different situations.

9 Social impact of the system

We have previously discussed what effects our theoretical solutions have on an arbitrary decentralised computation system with free participation. We will here discuss how such a system and the prototype can affect society at large.

Future uses Our research about peer-to-peer technologies and our solutions can be used in many situations where computation power can be increased through the help of untrusted participants. It might be more resource heavy for each node participating compared to systems designed for environments with trusted participants, however the gain in resources by allowing untrusted participants is huge considering most users on the Internet are untrusted.

Research potential This network was designed to keep down the resource use of the job owner while still producing reliable results in an environment with untrusted participants. As such, professional and hobby researchers without much computer power can gain more resources from volunteers. If our network is widely used for this purpose, more research can be done in fields were much computation power is needed.

Malicious jobs It is impossible in our system to force a worker to execute malicious code. However, jobs with results that can be used for harmful purposes cannot be prevented. As an example, a job specified to crack cryptographic keys, can harm third parties that use such keys. A system that prevents this requires a method to judge if a computation is morally right to perform. This is not in scope of this project.

Haskell Since our prototype requires computation code to be written in Haskell, awareness about Haskell and its advantages would increase. Following this and the fact that job code must be distributed as readable source code, gives workers the opportunity to study a job owner's code. With this, users can learn from each other.

However, if a job owner obfuscates his code by changing function and variable names, the code will be much harder to study and the job owner can keep his code a little more secret (Sun and Huang 2014).

10 Future work

To the best of our knowledge, this thesis is the first attempt to design a peer-to-peer based distributed computation system with free participation. Hence, many things can be improved upon and explored further. The proposed methods on reputation and proof-of-work can be improved upon by optimising parameters and extending them. One proposed extension is to dynamically change the difficulty of challenges.

Improving security is always a concern. Especially, methods for checking input files are essential. Furthermore, compilation of computation code can be made more secure by introducing a sandbox environment.

A desired feature, that can be difficult to achieve, is the ability to validate results and assign workers to tasks in a decentralised manner that is safe against Byzantine nodes and does not require the job owner to be online. The results regarding BFT-CUP presented in Section 4.3.2 are most likely highly relevant when researching this further. One approach is to create a new overlay network that meet the requirements for using the the BFT-CUP solution presented by Alchieri et al. (2008).

Recently, a DHT implementation was proposed for higher resistance against Byzantine nodes (Young et al. 2013). Even though that solution is not directly applicable to this system, new insights may be drawn from related work that are yet to be published.

The prototype itself can be improved upon by increasing ease of use and options to limit use of computer resource such as storage and CPU consumption. Furthermore, enabling use of GPU for computations would potentially improve system performance.

11 Conclusion

The produced network offers free participation without a formal registration process. The network is autonomous without any central point of control but the relationship between worker and job owner is, of necessity, asymmetric. However, any peer can be a job owner; thus there is opportunity for efficient cooperation between users.

No trust between nodes is necessary in the network: workers do not need to trust the job owner and vice versa. In a different system, where all nodes are known to be truthful, any node could post a job to a global queue. However, if this design was used in the presence of Byzantine nodes, they could corrupt the queue. Hence, the job owner must be online when a worker is assigned a task.

Potential attackers can create Sybil nodes that eventually can dupe a job owner in accepting a false result. Such an attack would have a very low success rate and high cost in terms of computer resources. Concerning malicious intents, our system can never be used as a botnet. However, without further measures, the input files of jobs may contain malicious content. Nevertheless, that content cannot be activated within the computation code, neither during compile time nor runtime.

Decentralised computation networks with free participation may not be motivated as they do not improve usability significantly over existing systems such as BOINC. Most of the results are however applicable to any distributed computation system. For example, BOINC could benefit from using Safe Haskell, the reputation model and the proof-of-work system. For decentralised computation systems to be useful, distributed assignment of tasks and distributed validation of results should be investigated further.

References

Alchieri, Eduardo A. et al. (2008). "Byzantine Consensus with Unknown Participants". In: *Proceedings of the 12th International Conference on Principles of Distributed Systems*. OPODIS '08. Luxor, Egypt: Springer-Verlag, pp. 22–40. ISBN: 978-3-540-92220-9. DOI: 10.1007/978-3-540-92221-6_4. http://dx.doi.org/10.1007/978-3-540-92221-6_4.

Apache (2014a). *Apache Ant*. http://ant.apache.org (2014-05-19).

— (2014b). *Apache Shiro Features Overview*. http://shiro.apache.org/features.html (2014-05-13).

— (2014c). *Cassandra*. http://cassandra.apache.org/ (2014-03-26).

— (2014d). *Maven*. http://maven.apache.org (2014-03-21).

Atallah, Mikhail J. (1998). "Reducibility and Completeness". In: *Algorithms and Theory of Computation Handbook*. Chap. 28, p. 14.

Back, Adam (2002). *Hashcash - A Denial of Service Counter-Measure*. http://www.hashcash.org/papers/hashcash.pdf (2014-04-15).

Balakrishnan, Hari et al. (2003). "Looking up data in P2P systems". In: *Communications of the ACM* 46.2, pp. 43–48. DOI: 10.1145/606272.606299.

Basin, David, Patrick Schaller, and Michael Schläpfer (2011). *Applied Information Security*. English. Springer Berlin Heidelberg. Chap. 1, p. 3. ISBN: 978-3-642-24473-5. DOI: 10.1007/978-3-642-24474-2_1. http://dx.doi.org/10.1007/978-3-642-24474-2_1.

Batten, Lynn (2013). *Public Key Cryptography:Applications and Attacks*. Wiley-IEEE Press. Chap. 6, pp. 133–150. ISBN: 9781118482261. DOI: 10.1002/9781118482261.ch7.

Belapurkar, Abhijit (2009). *Distributed systems security : issues, processes, and solutions*. Chichester, UK: John Wiley & Sons, pp. 21–22. ISBN: 978-0-470-51988-2.

Bellare, Mihir and Chanathip Namprempre (2000). "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". English. In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 531–545. ISBN: 978-3-540-41404-9. DOI: 10.1007/3-540-44448-3_41. http://dx.doi.org/10.1007/3-540-44448-3_41.

Bogdanov, Andrey, Dmitry Khovratovich, and Christian Rechberger (2011). "Biclique Cryptanalysis of the Full AES". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by DongHoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 344–371. ISBN: 978-3-642-25384-3. DOI: 10.1007/978-3-642-25385-0_19. http://dx.doi.org/10.1007/978-3-642-25385-0_19.

BOINC (2014a). *Computing with BOINC*. http://boinc.berkeley.edu/trac/wiki/ProjectMain (2014-03-20).

BOINC (2014b). *Redundancy and errors*. http://boinc.berkeley.edu/trac/wiki/JobReplication (2014-04-30).

Buford, John and Heather Yu (2010). "Peer-to-Peer Networking and Applications: Synopsis and Research Directions". In: *Handbook of peer-to-peer networking*. Ed. by Xuemin Shen et al. Heidelberg: Springer. ISBN: 978-0-387-09750-3.

Coelho, Fabien (2008). "An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol based on Merkle Trees". In: *Africa Crypt 2008*. Ed. by Serge Vaudenay. LNCS 5023. Cryptology eprint Archive 2007/433. Springer Verlag, pp. 80–93. http://eprint.iacr.org/2007/433.pdf.

Cosm (2014). *The Cosm Project*. Mithral Inc. http://www.mithral.com/cosm/ (2014-05-14).

DataStax (2014). *Securing Cassandra*. http://www.datastax.com/documentation/cassandra/1.2/cassandra/security/secure_intro.html (2014-03-26).

Diffie, Whitfield and Martin E. Hellman (1976). "New directions in cryptography". In: *Information Theory, IEEE Transactions on* 22.6, pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

Douceur, John R. (2002). "The Sybil Attack". In: *Peer-to-Peer Systems*. Cambridge, MA, USA: Springer Berlin Heidelberg, pp. 251–260. ISBN: 978-3-540-45748-0. DOI: 10.1007/3-540-45748-8_24.

Eclipse (2014). *Eclipse*. https://www.eclipse.org/downloads/ (2014-03-28).

ERights (2014). *ERights*. http://www.erights.org (2014-03-28).

Fischer, Michael, Nancy Lynch, and Michael Paterson (1985). "Impossibility of distributed consensus with one faulty process". English. In: *Journal of the ACM (JACM)* 32.2, pp. 374–382. www.summon.com.

Freenet (2014). *What is Freenet?* https://freenetproject.org/whatis.html (2014-05-14).

Furnell, Steven, Sokratis Katsikas, and Javier Lopez (2008). *Securing Information and Communications Systems : Principles, Technologies, and Applications*. Norwood, MA, USA: Artech House. Chap. 7.2, p. 106.

Git (2014). *git*. http://git-scm.com (2014-04-30).

Google (2014). *Google-Gson*. https://code.google.com/p/google-gson/ (2014-05-19).

Haskell (2014). *Safe Haskell*. https://www.haskell.org/ghc/docs/7.4.1/html/users_guide/safe-haskell.html (2014-03-28).

Joe-E (2014). *Joe-E*. https://code.google.com/p/joe-e/ (2014-03-28).

JUnit (2014). *JUnit*. http://junit.org/ (2014-05-19).

Kerckhoffs, Auguste (1883). "La cryptographie militaire". In: *Journal des sciences militaires* IX, pp. 161–191.

Kerry, Cameron F. and Patrick D. Gallagher (2013). *Digital Signature Standard (DSS)*. National Institute of Standards and Technology.

Krawczyk, Hugo (2001). "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)" English. In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in

Computer Science. Springer Berlin Heidelberg, pp. 310–331. ISBN: 978-3-540-42456-7. DOI: 10.1007/3-540-44647-8_19. http://dx.doi.org/10.1007/3-540-44647-8_19.

Legion of the Bouncy Castle Inc. (2013). *The Legion of the Bouncy Castle*. https://www.bouncycastle.org/java.html (2014-05-15).

Lynch, Nancy A. (1996). *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Chap. 1. ISBN: 1558603484.

McGilvary, Gary (2013). *V-BOINC Manual*. http://homepages.inf.ed.ac.uk/s0678915/vboinc/VBOINC_Manual.pdf (2014-05-16).

Microsoft Corporation (2004). *The Coordinated Spam Reduction Initiative: A Technology and Policy Proposal*. http://download.microsoft.com/download/7/6/b/76b1a9e6-e240-4678-bcc7-fa2d4c1142ea/csri.pdf (2014-04-15).

Milewski, Bartosz (2013). *Basics of Haskell*. https://www.fpcomplete.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io (2014-05-05).

Miller, Andrew and Joseph LaViola (2014). "Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin". https://socrates1024.s3.amazonaws.com/consensus.pdf (2014-05-16).

Molga, Marcin and Czesław Smutnicki (2005). "Test functions for optimization needs". http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf (2014-05-03).

Muller, Jean-Michel et al. (2010). *Handbook of Floating-Point Arithmetic*. Boston: Birkhäuser. Chap. 7.1, pp. 205–209. ISBN: 978-0-8176-4705-6. DOI: 10.1007/978-0-8176-4705-6.

Nakamoto, Satoshi (2008a). "Bitcoin: A Peer-to-Peer Electronic Cash System". https://bitcoin.org/bitcoin.pdf (2014-04-15).

— (2008b). *Re: Bitcoin P2P e-cash paper*. http://www.mail-archive.com/cryptography@metzdowd.com/msg09997.html (2014-05-16).

Namecoin (2014). *Namecoin*. http://namecoin.info/ (2014-05-14).

NoSQL (2014). *List of NoSQL databases*. http://nosql-database.org/ (2014-03-26).

O'Brien, Timothy and Sonatype (2008). *Maven: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, pp. 8–10. ISBN: 9780596517335.

Pease, M., R. Shostak, and L. Lamport (1980). "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2, pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. http://doi.acm.org/10.1145/322186.322188.

Project Voldemort (2014). *Project Voldemort*. http://www.project-voldemort.com/voldemort/ (2014-03-26).

Ray, Edward and Eugene Schultz (2009). "Virtualization Security". In: *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. CSIIRW '09. Oak Ridge, Tennessee: ACM, 42:2–42:3. ISBN: 978-1-60558-518-5. DOI: 10.1145/1558607.1558655. http://doi.acm.org/10.1145/1558607.1558655.

Rogaway, Phillip and Thomas Shrimpton (2004). “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In: *Fast Software Encryption*. Ed. by Bimal Roy and Willi Meier. Vol. 3017. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 371–388. ISBN: 978-3-540-22171-5. DOI: 10.1007/978-3-540-25937-4_24. http://dx.doi.org/10.1007/978-3-540-25937-4_24.

Schneider, Johannes J. and Scott Kirkpatrick (2006). *Stochastic Optimization*. Germany: Springer-Verlag Berlin Heidelberg. ISBN: 978-3-540-34559-6.

Schwaber, Ken and Jeff Sutherland (2013). *The Scrum Guide*. https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf (2014-04-10).

Source code (2014). *GDCN*. GitHub. https://github.com/GDCN.

Sun, Yongyong and Guangqiu Huang (2014). “Code Obfuscation Technology Based on Renaming Identifier”. English. In: *Proceedings of the 9th International Symposium on Linear Drives for Industry Applications, Volume 1*. Ed. by Xiaozhu Liu and Yunyue Ye. Vol. 270. Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, pp. 625–631. ISBN: 978-3-642-40617-1. DOI: 10.1007/978-3-642-40618-8_81. http://dx.doi.org/10.1007/978-3-642-40618-8_81.

Surjanovic, Sonja and Derek Bingham (2013). *Langermann function*. http://www.sfu.ca/~ssurjano/Code/langerm.html (2014-05-03).

Švenda, Petr (2004). *Basic comparison of Modes for Authenticated-Encryption (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS)*. report. Masaryk University in Brno, Faculty of Informatics. http://www.fi.muni.cz/~xsvenda/docs/AE_comparison_ipics04.pdf (2014-05-09).

Tanner, Timo (2005). *Distributed Hash Tables in P2P Systems - A literary survey*. Seminar on Internetworking.

Terei, David et al. (2013). “Safe Haskell”. In: *Proceedings of the 2012 Haskell Symposium*. Ed. by Andy Gill. New York, NY, USA: ACM, pp. 137–148. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364524.

TestNG (2013). *TestNG*. http://testng.org/doc/index.html (2014-04-10).

TomP2P (2014a). *Domain and Entry Protection Mechanisms*. http://tomp2p.net/doc/advanced/ (2014-03-28).

— (2014b). *TomP2P*. http://tomp2p.net (2014-03-21).

Wetmore, Brad (2011). *Provide API changes to support future GCM AEAD ciphers*. Oracle. http://mail.openjdk.java.net/pipermail/security-dev/2011-April/003097.html (2014-05-12).

Young, Maxwell et al. (2013). “Towards Practical Communication in Byzantine-Resistant DHTs”. In: *IEEE/ACM Transactions on Networking* 21.1, pp. 190–203. ISSN: 1063-6692. DOI: 10.1109/TNET.2012.2195729.

A Work process

To be able to ensure progression and quality of this project, the group needed a strategy. The work process and individual responsibilities are described in this section.

A.1 Responsibilities

In interest of quality and productivity, five roles or sets of responsibilities were elicited and assigned to different members of the development team, namely Product owner, Project leader, Security manager, Editor and Calendar master. Some responsibilities were shared by all members such as documenting.

Product owner The Product owner was given the responsibility of defending the interests of the end user. The key effort was to prioritise and update the backlog of features that were going to be implemented. The Product owner was also specifically consulted whenever an implementation decision would affect the end user. Jack Pettersson was assigned this role.

Project leader The Project leader was responsible for managing the development process. This consisted primarily of preparing and leading each group meeting as well as organising the development iterations. It was important for the Project leader that the right decisions were taken at the right time with the necessary information. Another important responsibility of the Project leader was to ensure that each member participated in the discussions. Leif Schelin was assigned this role.

Security manager Since security is difficult or impossible to append to an existing project (Belapurkar 2009), Joakim Öhman was assigned the responsibility of always advocating the product security, especially the security of the worker nodes.

Editor Jack Pettersson was given this responsibility of advocating the quality of the final report so that it would not be forgotten in the midst of development. He also facilitated technical support with LaTeX throughout the writing process.

Calendar master The Calendar master was responsible for updating the group calendar and ensuring that each group meeting would have a room booked. It was also his responsibility to remind each team member of upcoming deadlines and events. This role was given to Niklas Wärvik.

A.2 Development process

The development process was based on *Scrum* (Schwaber and Sutherland 2013) but with a few minor modifications. A *feature backlog* was created and updated but no *story points* were assigned. The time interval for each *Sprint* was set to two weeks. Meetings were held each Sprint for *Sprint planning* and *Sprint retrospective* but not for *Sprint review*. During development, each member took initiative to work rather than being assigned work.

A.3 Version control system

Distributed development with the version control system Git was used throughout the project (Git 2014). The master branch contained the different releases of the prototype to make certain that there always were a working version without any dysfunctional parts. To minimise merging problems, a development branch was used for the latest functioning version and this branch was merged with the master branch after each sprint. When a new feature was to be implemented, a new branch was created from development branch and was merged back into development when it was finished and bug-free. Consequently it was certain that every feature in the master branch was finished and could be used in the final revision and when a new branch was created from the development branch the code was functioning.

B Technologies

In the development of the prototype, several technologies were used that are not directly related to its functionality. This section briefly introduces and motivates the use of them.

B.1 Java

The prototype is implemented in the programming language Java. Java was chosen for multiple reasons. Firstly, all group members had previous experience with Java and knew the syntax. Secondly, Java is cross-platform, making the prototype executable on all computers which has Java installed as it does not contain any operating system dependent binaries. Finally, Java is a widely used programming language and owing to this there are numerous libraries. A couple of those libraries are used in the project such as TomP2P and the ones mentioned later in this section.

B.2 TestNG

TestNG is a testing framework used to test the Java code in the project (TestNG 2013). It can be used for unit testing, in addition to integration and system testing. Being inspired by JUnit (JUnit 2014), it has a similar syntax and moreover it can perform the same tasks as JUnit. TestNG is customisable in multiple ways, for instance it is possible to specify which group a test should belong to, enabling to run faster tests more often and slower tests less often. Customisation is done in an XML-file, but it is not a requirement for the tests to be executed.

B.3 Maven

Maven is a management tool for software projects that facilitates automated building and testing (Apache 2014d). With Maven, a project can be built effortlessly from command line no matter what IDE was used to develop it. Unit tests run automatically when building. Compared to Ant (Apache 2014a), it is much simpler to use Maven if you follow the standard conventions (O'Brien and Sonatype 2008). These conventions are easy to follow, especially when using an IDE with Maven support.

Dependencies, such as libraries, are downloaded by Maven, ensuring that all developers use the same versions of the libraries. When developing, another advantage of using Maven is that dependencies between modules are always explicit. Thus it is very easy to make sure that the modules follow a good structure, for example that no module depends on the user interface.

B.4 Gson

Information concerning each task, such as which files should be used as input to it, must be specified by the job owner. For each task, a json file is created by the job owner and put in the proper directory. This information is read dynamically by the program, using the Gson parsing library (Google 2014).

Glossary

Byzantine A process in a system that exhibits unexpected behaviour, such as processing requests incorrectly or producing inconsistent output. (One example is a deceitful node)

canonical Result that is considered correct by validation.

deceitful Node who attempts to disrupt the network, for example by giving false results or spamming.

DHT (distributed hash-table) A decentralized distributed storage with a Key-Value interface.

DoS (denial of service) An attack in which the attacker aims to make the target system unusable, often by flooding it with traffic in order to deplete its resources.

job Problem that has been expressed in code for nodes.

job owner The node who has put a certain job on the network.

MAC (message authentication code) A short piece of information used to ascertain the integrity and authenticity of a message.

man-in-the-middle Type of attack in which an eavesdropper communicates with the victims separately and relays messages between them, while pretending to be the other party to both.

MC (Monte Carlo) A stochastic algorithm that produces a result in known time without guarantee of optimum.

node A participator in the network. A single computer can run several nodes.

post Job owner posts a job to be computed.

problem An actual problem, not yet expressed to be understandable by the system.

proof-of-work Either an algorithm which aims to produce a proof that a certain amount of computation power has been used, or the result of such an algorithm (i.e. the proof).

quality Property of a result that describes how good it is.

quality function Haskell function created by job owner to compute the quality of a result.

replica Replication instance of a task.

replication Sending the same task to multiple nodes.

reputation A value representing how truthful a node has been in the past towards a certain job owner.

result The output from a finished task.

sandbox A sealed environment that can only interact with the rest of the computer through a secure interface.

Sybil A Byzantine node that cooperate with other Sybil nodes for a deceitful or otherwise malicious intent.

task A small self-contained part of a job that is computed by a single node.

truthful A node that behave expectedly, that is, not Byzantine.

validation The process of determining if a result is correct or not.

worker A node that works on a task.