

Grid Computing Framework

A BOINC Client for Breaking the ECC Challenge
With Distributed Checking

Master Thesis

Christoph Schwank

<cschwan@student.ethz.ch>

Advisors:

Prof. Dr. Roger Wattenhofer

Michael Kuhn

Stefan Schmid

Distributed Computing Group
Computer Engineering and Networks Laboratory (TIK)
Department of Computer Science

7th June 2007

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Distributed
Computing Group** 

Abstract

The evolution of the Web and the growth of computational power of the clients, have led to new possibilities to solve computational puzzles such as the factorization of large numbers. Instead of using a mainframe to do this kind of work, clusters of distributed machines, called grids, are used. These grids are built with the help of lots of participants devoting their unused CPU cycles to the grid computing projects.

This thesis' goal was to evaluate the different existing frameworks for grid computing and to use one of these frameworks to attack the ECC Challenge, avoiding selfish behavior. We decided to extend the BOINC framework for our purposes. In particular, we added distributed checking mechanisms to fend off cheaters while the overhead at the server is low.

Contents

1	Introduction	1
1.1	Task Description	1
1.2	Results	1
1.3	Thesis Overview	2
2	Evaluation Of Different Existing Framework And Projects	3
2.1	Frameworks	3
2.1.1	BOINC	3
2.1.2	XtremWeb	4
2.1.3	Alchemi	5
2.2	Projects	6
2.2.1	SETI@home	6
2.2.2	Distributed.net	7
2.2.3	GIMPS	7
2.2.4	M4	7
2.2.5	HashClash	7
2.2.6	Assault on 13th Labour	7
2.2.7	BBC Climate Change Experiment	8
2.3	Evaluation Results	8
2.3.1	BOINC	8
2.3.2	XtremWeb	8
2.3.3	Alchemi	9
3	The Discrete Logarithm Problem on Elliptic Curves	11
3.1	The Discrete Logarithm Problem	11
3.2	The Algorithm	12
3.2.1	The Pollard Rho Algorithm For Logarithms	12
3.2.2	The Distributed Version of Pollard's Rho Algorithm for Logarithms	13
4	Distributed Checking	15
4.1	Related Work	16
4.2	The Distributed Checking Algorithm	17
4.3	Analysis of the Algorithm	19
4.4	Simulation Results	21
5	Architecture	25
5.1	The Whole System	25

5.2	The Server	26
5.2.1	The Data Part	26
5.2.2	The Web Site Part	30
5.2.3	Security	30
5.3	The Client	30
5.4	An Example	31
6	Implementation	35
6.1	The Server	35
6.1.1	General	35
6.1.2	The Distributed Checking	37
6.2	The Client	40
7	Conclusions	41
7.1	Achievements	41
7.2	Future Work	41
A	Installing The BOINC Server	43
A.1	Requirements	43
A.1.1	Install MySQL, apache2 and PHP	44
A.1.2	Setting up MySQL	45
A.2	Getting The Source Code	45
A.3	Creating a Test for the BOINC Server	46
A.3.1	Creating the Hello Project	46
A.3.2	Modifying the Preferences	47
A.3.3	Adding the Application	47
A.3.4	Creating the Work Units	48
A.4	Final Steps	50

1

Introduction

1.1 Task Description

Grid computing allows to aggregate the CPU cycles of thousands of machines, which makes this paradigm interesting for solving computational puzzles such as the factorization of large numbers. In his thesis, Christoph will develop a grid computing framework that allows to tackle such problems. First, he will investigate existing systems and decide whether one of them can be used as a starting point, or whether our framework has to be implemented from scratch. Second, Christoph will search for interesting computational puzzles whose solution is financially interesting. The framework should be designed generically, such that it can be used to solve different problems. As an example, we could attack RSA challenges by implementing the Number Field Sieve. Christoph will have to opt for one interesting problem that he wants to tackle in the course of the project. In this phase, he will also have to consider the parallelizability of a good algorithm for the chosen problem. Third, Christoph will implement the framework. Thereby, it is important to come up with mechanisms which inhibit free-riding and other selfish behavior. In this more theoretical part, Christoph has the opportunity to conduct scientific work which, dependent on the results, may lead to a research paper.

1.2 Results

The two main results of this thesis are the implementation of a distributed computation infrastructure to compete on the ECC Challenge [12] and a distributed checking algorithm which can be used to improve the distributed computation.

To distribute the load of checking the clients' results we aimed at improving on the straightforward approach of just computing the same unit several times. For this reason we decided to use distributed checking, where the clients will do most of the checking work. Also the checking in our case is much less time-consuming than recomputing the results. Generally,

this might not be the case, because the checking highly depends on the problem to compute; the checking is not the same for every computation problem. So the checking algorithm must be able to deal with different kinds of computation problems and therefore no optimizations for special kinds of data sets can be made.

In this thesis we present a checking algorithm which should be usable for every kind of computation and also, very important, both for reliable and unreliable clients. The support for unreliable clients is necessary because if the computation is not done on a cluster the clients can join and leave anytime.

For the implementation of the distributed computation infrastructure first an applicable framework had to be found. We then implemented a server part and a client part to attack the open ECC Challenges. To do this an algorithm for the given problem has to be found and implemented on the framework. We also wanted to use our distributed checking and for this a couple of changes were needed. Now we are able to present an implementation which uses distributed checking and can be used to compete on the open challenges.

1.3 Thesis Overview

We begin by evaluating existing projects and frameworks in Chapter 2. This contains an overview of the frameworks and the projects as well as a detailed overview of the evaluation results. We have chose to tackle the Discrete Logarithm Problem, which is explained in Chapter 3. An overview of the distributed checking will be given in Chapter 4. We also give a short analysis of the algorithm and some simulation results. Chapter 5 will then give an overview of the architecture of our system, containing an explanation of the different parts as well as a short example. The implementation details of the different parts are then given in Chapter 6. Finally we give summarizing conclusions and future work in Chapter 7. In Appendix A we give short installation instructions to use our system.

2

Evaluation Of Different Existing Framework And Projects

The first phase of the thesis consisted of the evaluation of existing frameworks, and projects that base on them. The list of projects is far from complete, and we only give a selection of the analyzed frameworks.

2.1 Frameworks

First we will give an overview of a few existing frameworks which seem interesting to use as a basis for the implementation. Then we have a look at some projects using distributed computing. At the end of this chapter we present the evaluation results.

2.1.1 BOINC

BOINC [7] stands for Berkeley Open Infrastructure for Network Computing and is a software framework which allows to build open-source software for volunteer computing and desktop grid computing. It has been developed at U.C. Berkeley Space Sciences Laboratory by the group operating SETI@home for their SETI@home project.

The goal of BOINC is to get a large number of computers spending their free computation cycles on one or more projects. As presented in [3], the goals include:

Reduce the barriers of entry to public-resource computing. BOINC provides a scalable framework meaning that a server for a BOINC-based project can either consist of one single machine running common open-source software (Linux, Apache, PHP, MySQL, Python) handling all the work, or of a couple of machines handling only parts of it. Also the administrative effort will not be very high.

Share resources among autonomous projects. Every BOINC-based project is running independently. There is no centralized control over the various projects, which means

that every project has to operate its own server. On the client side this is a bit different. Every user running the BOINC client can participate in multiple projects. To do this, a user only has to download the client application once and then assign it to each project he wants to participate in. He can then also determine how his resources (CPU, disk space etc.) should be distributed among the projects. If a project is temporarily not available, the client will recognize this and use the free capacities for the other projects. It is also possible that the CPU on a computer is working for one project and in the background, the network is transferring files for another project.

Support diverse applications. BOINC supports many applications. It provides all mechanisms necessary to distribute data in a flexible and scalable way as well as an intelligent scheduling algorithm to match requirements with resources. It also allows to run existing programs in the languages C, C++ and FORTRAN with no or only little modification. A program can consist of only one file as well as of multiple files (e.g. multiple programs and a coordinating script). New versions can be easily deployed and will be replaced on the client side automatically.

Reward participants. To make public-resource computing projects interesting, the projects must provide “incentives”. Without them the participants would not be attracted and will not stay with the project. Because the projects normally can not pay the participants with real money, another incentive is needed. The primary incentive for many participants is **credit**: a numeric measure of how much computation they have contributed. BOINC implements a credit-accounting system reflecting the usage of multiple resource types (CPU, network, disk). It is common across multiple projects and resistant to “cheating” (attempts to gain undeserved credit). BOINC also comes with the possibility to add visualization to the applications which allows them to be used as screensaver.

Implementation

Most parts of the BOINC framework are written in C or C++, the Web site parts on the server side are written in PHP and a few Python scripts are also needed. For the information storage like user accounts, credits, work handling etc. a MySQL database is used.

BOINC is distributed under the Lesser GNU Public License [16].

2.1.2 XtremWeb

XtremWeb [30] was designed at the Computer Science Laboratory on the Paris XI University of France. It is a software platform designed to be a middleware for Global Computing (also called Large Scale Distributed System) experiments. Like other distributed systems, it uses remote resources (workstations, servers etc.) connected by the Internet or a LAN. XtremWeb belongs to the so called Cycle Stealing Environment family, which means that every participant provides his CPU idle time to the XtremWeb project. It allows to run multiple applications distributed to the various participants and implements some security features to prevent applications from being able to corrupt data or resources.

The architecture looks as follows: XtremWeb consists of three parts. First there is the server. It is the centralized part of the system. On the server, everything is organized. Then there is the worker. The worker is one of the distributed parts of the system. Every participant needs to run a worker to get jobs from the server and execute them on the

free cycles of the CPU. The worker computes in a non-intrusive way so that the owner of the machine is not disturbed, according to an activation policy. The third and last part is the client. The client is also distributed. Its usage is more on the administrative side, because it is used to insert and delete applications, users and jobs as well as to retrieve results.

Implementation

XtremWeb is implemented in Java. This has the advantage that you have to compile every part only once. Nevertheless the client and the applications will run on every client computer regardless of the operating system if Java is installed. This gives enormous flexibility and the testing is much easier.

An interesting feature is XtremWeb's own message passing system MPICH which it uses to optimize the communication.

XtremWeb is developed as Free Software (GPL) [15]. It is open source and non-profit software.

2.1.3 Alchemi

Alchemi [2] is a framework that allows to aggregate the computing power of networked machines into a virtual supercomputer and to develop applications to run on it. The design idea was to build an easy-to-use and flexible framework. The system is also scalable, reliable and extensible.

The authors of [21] claim that the key features supported by Alchemi are:

- Internet-based clustering of desktop computers within a shared file system
- federation of clusters to create hierarchical, cooperative grids
- dedicated or non-dedicated (voluntary) execution by clusters and individual nodes
- object-oriented grid thread programming model (fine-grained abstraction)
- Web services interface supporting a grid job model (coarse-grained abstraction) for cross-platform interoperability, e.g. for creating a global and cross-platform grid environment via a custom resource broker component.

Alchemi has a central component that dispatches and manages independent units of parallel execution to workers. The entire system consists of the following components: the Manager, the Executor, the Owner and the Cross-Platform Manager.

The Manager is equivalent to the central component. Every Executor registers himself at the Manager which then keeps track of the availability of the Executor. The Manager schedules and distributes the threads according the given priority to be executed by the Executors and collects the the sent-back results.

The Executor executes the threads he gets from the Manager on the contributor's computer. The Executor supports two different modes: dedicated and non-dedicated. Dedicated means that the resource is centrally managed by the Manager, and non-dedicated means that the volunteer can decide how the resource is used. For the dedicated mode, two-way communication is needed whereas for the non-dedicated mode, one-way communication is sufficient. Due to needing two-way communication for dedicated execution

this is the preferred mode for Local Area Networks with the advantage that the Manager can control all the participants. The non-dedicated execution is more appropriate when the Manager and the Executor are connected over the Internet or if they are in different LANs.

The Owner “owns” the application. He submits the work to the Manager and receives the completed work. He is the connection between the grid and the application developer.

The Cross-Platform Manager is an optional sub-component of the Manager. It is a generic web service interface which allows to execute platform independent grid jobs on the Alchemi framework. It transforms grid jobs into a form that is executable on the Executor and then schedules them to be executed.

Alchemi allows the use of different grid configurations: desktop cluster grid, multi-cluster grid, and cross-platform grid (global grid).

A cluster is the basic form. It consists of a single Manager and multiple Executors getting their work from the Manager. One or more Owners can execute their applications on the cluster by sending the work to the Manager.

Multi-Clusters arise when Managers are connected in a hierarchical way. In addition to the Executors connected to the Manager, also Managers can connect to other Managers. This results in a tree architecture. For a Manager there is no difference if the child is a Manager or an Executor, because the Manager behaves like an Executor to another Manager.

A grid is called Cross-Platform Grid when the Cross-Platform Manager is used to connect an Alchemi grid to a classical global grid like Globus [27] or an other similar network.

Implementation

Because most of the computers within enterprises are running Microsoft Windows operating systems, the developers of Alchemi implemented their framework on the Microsoft .NET Platform which only supports Microsoft Windows. With the .NET Framework they have a powerful tool set to deal with security, heterogeneity, reliability, application composition, scheduling and resource management as well as support for remote execution (via .NET Remoting and Web Services).

Alchemi is published under the GNU General Public License (GPL) and GNU Library or Lesser General Public License (LGPL).

2.2 Projects

Now we list a selection of distributed computing projects running around the globe. Some of them use a framework we introduced before, while others using their own framework.

2.2.1 SETI@home



Probably the best known project in this list is SETI@home [25] which is a scientific experiment using distributed computing for the Search for Extraterrestrial Intelligence (SETI). The project collects data with the telescope in Arecibo, Puerto Rico, edits them and lets the distributed clients search for well-known patterns in it. This is ideal work for distributed computing, since the continuous collection of data makes it easy to divide the data that have not been processed yet among different users.

With over 5.2 million participants worldwide, the project is currently the biggest distributed computing project. It offers a ranking of the contributing users, the computers of the different users, as well as of teams of users. This will probably motivate the users to do more work than they would otherwise do.

SETI@home uses the BOINC Framework for running its client. With the current version it is possible to let SETI@home be run as screensaver or continuously using the unused cycles of the processor on the clients.

2.2.2 Distributed.net



Distributed.net [9] is a distributed computing project, which tries to create a distributed computing platform for solving large-scale problems and to provide an accessible pool of computational power. At the moment, it tries to find the solution for the RC5 challenge from RSA Labs as well as to solve the OGR problem. Currently it has over 260,000 participants running its client.

The client is written in C++, using its own framework, not BOINC or a comparable framework. It also has a ranking on the users to motivate them to do more work.

2.2.3 GIMPS

The Great Internet Mersenne Prime Search [14] tries to find Mersenne [23] primes with the help of distributed computing. At the moment it has over 50,000 participants which is not that much compared to SETI@home. Therefore users are offered money if they find a solution, but only if they find it with the provided software.

The client provided is written in C and assembly and is not using another framework.

2.2.4 M4

The M4 Project [22] tries to break three original Enigma messages by using distributed computing. The messages were intercepted in the North Atlantic 1942 and were never broken. This project is different from other projects, because it tries to simulate an encryption system which uses the four rotor Enigma M4, which was used in the Second World War. Its client is written in C.

2.2.5 HashClash



The idea of the already completed project Hashclash [19] was to find MD5 collisions using the BOINC framework.

2.2.6 Assault on 13th Labour

Assault on 13th Labour [1] is a project which tries to decrypt an RC5 code used in the on-line/offline game "Perplex City"¹ by brute force. The language used for the client is not documented, but the fact that it needs the Microsoft .NET framework suggests that it is C# or another .NET language.

¹<http://www.perplexcity.com/>

2.2.7 BBC Climate Change Experiment



The BBC Climate Change Experiment [5] studies global climate warming. By processing different scenarios it tries to find better models to predict the climate change in the future. It uses the Transient Coupled Model from the climateprediction.net citeclimateprediction experiments.

This is another very nice example which uses the BOINC framework.

2.3 Evaluation Results

In the following part the strengths and weaknesses of the three frameworks will be compared.

2.3.1 BOINC

Strengths: BOINC was developed by the developers of SETI@home for their SETI@home project, which means that they have a lot of experience in developing distributed computing frameworks. This leads to a strong framework where all the teething problems are gone. They realized that it does not make sense to write a dedicated client for each project, so their client can be used for different projects. This means a contributor must only run one client and can participate in various projects. The fact that the project is open source also helps to improve it.

Another advantage is the fact that a lot of projects use BOINC. On the homepage of BOINC over 20 projects are listed. This means that there is a big community running and improving the framework.

Weaknesses: BOINC is written in C/C++ which implies that the application computing solutions that must be compiled for every operating system which should be supported. This means if your application should support Windows, Linux and Mac OS X, you have to compile the application three times. To make the whole thing even worse, you have to use a lot of precompiler statements in the code to make it work on the different operating systems. This also means that in reality every time the application is updated three applications must be updated.

2.3.2 XtremWeb

Strengths: XtremWeb's biggest advantage is the fact that it is implemented in Java. This reduces the complexity of developing the applications. Every operating system can use the same client running the same application, and the server is not bound to a specific operating system.

We also like the idea of having a component to control the applications which will run on top of the framework. This greatly simplifies the management.

Weaknesses: XtremWeb was only used at a few French universities. We found no project running on XtremWeb and also the entries in the mailing list were not up-to-date. Also the fact that we were not able to compile the framework without a lot of errors does not speak for XtremWeb. The whole website gives the impression that the project is currently put aside.

2.3.3 Alchemi

Strengths: Alchemi has two major features which are quite interesting. First there is the possibility to “connect” the framework to classical grids. It makes the framework interesting for scientists, because they can use the power of a large number of computers for their computations, without having to buy huge clusters. And they do not need to change the applications already running on the grid.

The second feature is the possibility to connect a manager as a client. This allows to build a tree topology. We think this is very good to distribute the load of the server to the network. With this topology the managers do not have as much clients as a single server would have and therefore they must not handle a big load.

The fact that this framework is implemented in C# has the advantage that the development is easier, because it has to run on Windows only. The testing is also simpler, because everything has only to be tested on Windows. But this also has disadvantages which will be described next.

Weaknesses: Alchemi is limited to clients running Windows because it is implemented on the .NET framework. The developers say that they chose this approach because most of the computers running on workplaces are running Windows. By covering these computers you have most of the unused computing power for your work. Nevertheless, for distributed computing, it would be preferable if everyone could join, independent of their operation system, which is not possible with this framework.

This leads us to the decision to use the BOINC framework. The advantage of having a large number of other projects using this framework and with this having a lot of documentation and current mailing lists will weight more heavily than the disadvantage of the use of C/C++. The fact that a lot of projects already use this framework will hopefully acquire many contributors faster because everyone having installed the BOINC client only needs to join our project without having to download another client. Probably a user already having installed the BOINC client will rather join our project if he can reuse his client than if he would have to download another one.

3

The Discrete Logarithm Problem on Elliptic Curves

This chapter gives an overview of the elliptic curve discrete logarithm problem and of the algorithm we use to solve it.

3.1 The Discrete Logarithm Problem

The discrete logarithm problem is the problem of “inverting” the process of exponentiation. In more detail, discrete logarithms are group-theoretic analogues of ordinary logarithms. There are two versions of the problem which are often studied:

1. The discrete logarithm problem in a finite field: Let F_p be a finite field, and given $p, q \in F_p$, find an integer l such that $p = q^l$ in F_p , given that such an l exists.
2. The elliptic curve discrete logarithm problem: Let E be an elliptic curve over a finite field F_p , and given $P, Q \in E(F_p)$, find an integer l such that $P = l * Q$ in E , given such an integer exists.

Even though these two problems do not look the same, they share a number of properties. a^l means *multiplying* q by itself l times. On an elliptic curve, $l * Q$ means exactly the same, namely *adding* Q to itself l times. The only differences are the notation and the operations, which are defined differently on the elliptic curve and on the finite field. Both build a group with the following characteristics: group G , elements $a, b \in G$, group operation \oplus with $b = a \oplus a \oplus \dots \oplus a = a^x$. There is yet another small difference; the operations on elliptic curves are more difficult to implement than the operations on finite fields. For further details see [13].

3.2 The Algorithm

For solving the elliptic curve discrete logarithm problem different algorithms exist. We have decided to use the distributed version of Pollard's Rho algorithm for logarithms.

3.2.1 The Pollard Rho Algorithm For Logarithms

In [24], the original algorithm is presented for the first time. It is a randomized version of the "baby-step giant-step" algorithm [4].

As said above, the goal is to compute l such that $r = q^l$, given only r and q . It computes integers a, b, A and B such that $r^a * q^b = r^A * q^B$. Given that the underlying group is of cyclic order n , we can calculate l as a solution of the equation $(B - b) * l = (a - A) \pmod{n}$.

To find the integers a, b, A and B , the algorithm uses Floyd's cycle-finding algorithm to find a cycle in the sequence $x_i = p^{a_i} q^{b_i}$. The x_i s will be created by a function $f : x_i \mapsto x_{i+1}$ which creates a sequence of values in a "random" manner. The solution l can be found if $x_i = x_{2i}$, given that $x_i = p^{a_i} q^{b_i}$ and $x_{2i} = p^{A_{2i}} q^{B_{2i}}$.

The algorithm starts with initial values for a, b, A and B and then goes stepwise until $x_i = x_{i+1}$ (see Figure 3.1). In every step we do "one" step for a and b and "two" steps for A

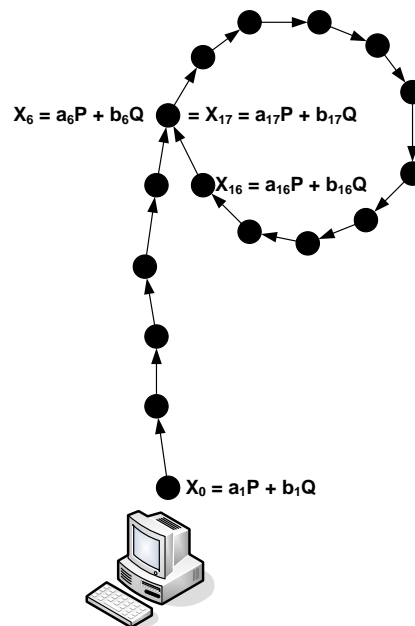


Figure 3.1: Pollard Rho

and B .

Since the algorithm has been introduced, a lot of improvements have been added to find a collision faster, but the idea is still the same. The algorithm has an expected runtime of \sqrt{n} in a set of n elements. This follows from the birthday paradox, which is quite similar, namely finding the same value in a set twice. A short listing of the algorithm is given below:

Let G be a cyclic group of prime order p . Given $a, b \in G$, and a partition $G = G_0 \cup G_1 \cup G_2$, let $f : G \rightarrow G$ be a map

$$f(x) = \begin{cases} b \cdot x & x \in G_0 \\ x^2 & x \in G_1 \\ a \cdot x & x \in G_2 \end{cases}$$

and define maps $g : G \times \mathbb{Z} \rightarrow \mathbb{Z}$ and $h : G \times \mathbb{Z} \rightarrow \mathbb{Z}$ by

$$g(x, a) = \begin{cases} a & x \in G_0 \\ 2a \bmod p & x \in G_1 \\ a + 1 \bmod p & x \in G_2 \end{cases}$$

$$h(x, b) = \begin{cases} b + 1 \bmod p & x \in G_0 \\ 2b \bmod p & x \in G_1 \\ b & x \in G_2 \end{cases}$$

Inputs: p a generator of G , q an element of G

Output: An integer x such that $px = b$, or failure

1. Initialize $a_0 = 0, b_0 = 0, x_0 = 1 \in G, i = 1$
2. $x_i = f(x_{i-1})$,
 $a_i = g(x_{i-1}, a_{i-1})$,
 $b_i = h(x_{i-1}, b_{i-1})$
3. $x_{2i} = f(f(x_{2i-2}))$,
 $A_{2i} = g(f(x_{2i-2}), g(x_{2i-2}, A_{2i-2}))$,
 $B_{2i} = h(f(x_{2i-2}), h(x_{2i-2}, B_{2i-2}))$
4. If $x_i = x_{2i}$ then
 - (a) $r = b_i - B_{2i}$
 - (b) If $r = 0$ return failure
 - (c) $x = r^{-1}(A_{2i} - a_i) \bmod p$
 - (d) return x
5. If $x_i \neq x_{2i}$ then $i = i + 1$, and go to step 2.

3.2.2 The Distributed Version of Pollard's Rho Algorithm for Logarithms

The Pollard Rho algorithm for logarithms cannot be distributed in a straightforward manner. This is not so important if the set of possible values is small. But if the number range grows larger, e.g. a field with 100 bit integers, the non-distributed Pollard Rho algorithm will run for a very long time.

This was the motivation to use the distributed version of the algorithm. Finding the solution with the distributed version is more difficult than with the centralized algorithm, even when the complexity stays the same: In the distributed case every client creates a trail of points on the set with the function $x_i = f(x_{i-1})$, $i = 0, 1, \dots$. It then reports these points to the server, which searches for the collisions to compute the solution. Because every client only creates one trail and does not look for collisions, it will never note when the solution is found or if he is looping. The second difficulty is to ensure that the clients do not flood the server with results. To avoid this, the clients do not report every point they pass; they only report

distinguished points. “Distinguished points” are a subset of all points, which have special characteristics. For example, a distinguished point has all the leading x bits set to zero. If every client uses the same function f and the same selection criterion for the distinguished points, the server will find the collision if one occurs (see Figure 3.2).

The collision will also be detected if the collision point is not a distinguished point. In

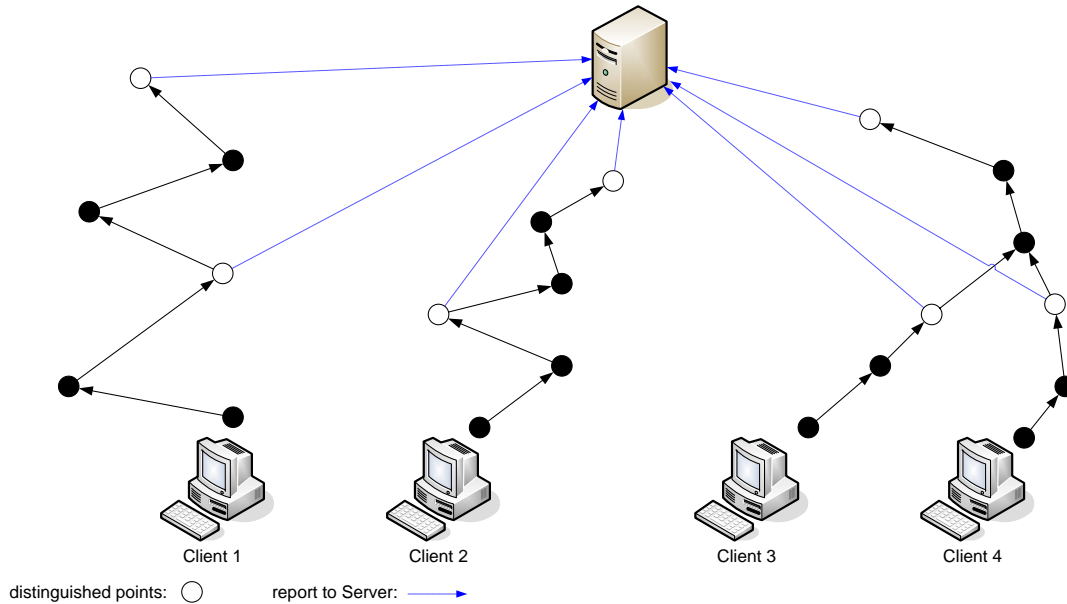


Figure 3.2: Distributed Pollard Rho

that situation, the collision will be detected slightly later, but both clients will come to the same distinguished point in exactly the same number of steps starting at the real collision point. And also the next distinguished point reached by the clients will be the point where the collision is detected. This relies on the fact that all the clients use the same function which does its pseudo-random steps depending on the last point computed. So every client will perform the same step if they are on the same point before that step. This is also the case if two clients are on the same point but with different parameters a and b . In the case where the expected running time until a collision occurs is very long, e.g. a year it is not a problem if the collision is detected one hour later than it could be.

It is possible that the starting point of a walk collides with a trail, and then we do not have a solution, because we do not have two autonomous trails which collide but the same trail twice. This means that only duplicates (a duplicate is the same point with the same parameters reported from different hosts) will be generated. It is also possible for a walk to fall into a loop where no distinguished point is contained. This problem can be handled by setting a maximum trail length and abort the walk if it goes over this boundary.

4

Distributed Checking

This chapter gives an overview of distributed checking.

First of all we will try to answer the question why distributed checking is necessary. Every time a problem should be solved with a distributed approach, the question of validating the reported results of the clients arises. Depending on what has to be computed it is rather important to be sure that the result is correct. If you want to analyze a huge amount of data which is growing continuously it is often enough to let every part be analyzed by a few clients and take the result that is returned the most. If at a later date a solution is found to be wrong it is not a big deal to recompute this part. The impact of this wrong decision is not very big. However, there are also problems where one wrong decision could fail the whole project. For example, if you are searching for a private key given the corresponding public key, it is important to be sure that the reported results are correct. If the server lets a client check a part of the set of all possible solutions, he must be completely sure that the client did not find the key if he tells the server that the key was not in his part. Otherwise the whole work has to be done a second time. This means that a majority decision may not be the best solution. Checking the results instead of computing everything at least a second time will lead to a better performance, because normally checking is faster than recomputing. A second reason for using checking instead a majority decision is that it is more robust against collaborations of cheaters. This means every result has to be validated. This makes a function necessary, which validates the results reported to the server. This could be done on the server. The disadvantage of this approach is that the server has to spend a lot of computation power on result validation which is not so good, because the server has to control the computation in the first place. The validation problem could also be solved with distributed checking. Instead of validating every result on the server, we can let the clients check themselves, always appointing the result of one client to another one for checking. With this approach, some load of the server will be distributed to the clients. But with the advantage of disburdening the server some new problems are created. The distributed checking must be tamper-proof. It should not be possible for a group of cheaters to cover their cheating during the checking. The reasons for cheating are various. The reward for participating is mostly some sort of

credit. The reason for participating is then to earn as much credit as possible to be on top of the participant list. To reach the top as fast as possible a very small amount of participants will try to cheat, and this is probably the main reason for cheating. Cheating clients try to trick the project by claiming work they did not do. This can be done for example by sending random data back instead of the true results of the work that was appointed to them. With this they can produce a lot more data in the same time, for which they will earn credit.

Another example of cheating is, if we search for example for a key, that a participant can try to hide the fact that he has found the key and earn the prize for himself, if a prize is given for finding the key. So the project thinks the key has not been found and everybody is still spending computing power even though no computation power would be needed anymore. In the same case, another participant can try to slow down the project by sending wrong results or not checking his part of the possible keys he has to check, hoping that he misses the key and the search will continue.

There is also a very small group of cheaters who will try to cheat for the reason of finding a way to cheat. If the project is successful they might give their knowledge to the project owners helping them to improve the project.

4.1 Related Work

As mentioned above there are some reasons for uncheatable grid computations. Most of the projects have to defend themselves against participants claiming credits for work they have not done.

Golle and Mironov proposed an interesting *ringer* scheme [17] to overcome such “lazy” cheaters. Their scheme is restricted to problems involving the inversion of a one-way function (IOWF). In their basic ringer scheme, several inputs x_i from a participant’s domain D are randomly selected by the manager in the initialization stage. These inputs, called *ringer*, are kept secret with the manager. It then creates *ringer images* for each x_i , that is, It evaluates the one-way function $f(x_i)$ on these inputs.

The manager then assigns the grid computation tasks in D to the participants, it also assigns the ringer images. The participant needs to compute f for every $x \in D$ it gets from the manager as well as find the ringer corresponding to the given ringer images. On the returned results the manager can check if for every ringer image the participant has computed the ringer and also if it has found all the assigned ringers. If the manager recognizes that a participant has found all the assigned ringers, it is assured with high probability that the participant has indeed done all computations.

Golle and Mironov extended their basic ringer scheme in several ways, to prevent the participant from knowing the number of ringers he has to find. These improvements added a new level of security, but still the schemes are limited to IOWF computations.

Szajda, Lawson, and Owen augmented the ringer scheme to deal with general classes of computations. These also include optimization and Monte Carlo simulations. In [26] effective ways to choose ringers for these computations are explained. But it is still unknown if the schemes proposed in [26] can be extended for generic computations, because their constructions are not generic.

Another technique to protect computations against cheaters, who try to get more credit than they deserve or who try to hide some results, is the injection of chaff as described in [11] by Wu and Goodrich.

Closer to our approach, Du et al. [10] propose checking algorithms for cheater detection in grid computations. Their approach is that the supervisor randomly selects and verifies some samples from the task domain D . To prevent a participant from cheating on these samples, the samples will not be checked until the participant has committed the corresponding results. This works with a commitment-based sampling scheme based on the Merkle tree. The advantage of this approach is that it is generic. The disadvantage is that the complete checking has to be done on the server side, which creates a heavy load on the server for checking, by redoing computations the participants have already done.

Finally, another approach for uncheatable grid computing is the method used by SETI@home [20]. The idea is very simple. They double-check every result by letting at least two participants do the same work and then selecting the hopefully right result by a majority decision. Golle and Stubblebine [18] proposed a more efficient scheme based on using probabilistic redundant execution to implement this approach. They assume that all computations occur in a single round.

There are also some alternative ways to defeat cheating participants, not discussed in this thesis, e.g. the use of tamper-resistant software. Nevertheless, also these approaches have their weaknesses.

4.2 The Distributed Checking Algorithm

In [6] a round-based distributed checking algorithm was introduced. This algorithm is designed for a set of processors that have to check each other. It assumes that all processors can reach each other at all times, an assumption that does hold for grid computing. It further uses Hamiltonian paths to build a strongly connected component containing only good clients. Finding this strongly connected component is complex and time-consuming.

The algorithm performs a preprocessing step in which nodes are paired up and test each other. After discarding all pairs with at least one negative test, only pairs containing two good or two bad clients remain. Inspired by this preprocessing step we developed an algorithm where this Hamiltonian paths and the strongly connected component are not necessary, as we could show that it is not required in grids of realistic size. We also make sure that our algorithm can be used with non-reliable clients as opposed to clients which are always online as in [6]. Our algorithm requires a checking function for the results, which runs much faster than finding the solutions with the solution finding algorithm. To make the algorithm robust we use groups of two in the first step, groups of three in the second step and groups of six in the third step. This means after step three we have super groups of 36 clients.

The algorithm for group sizes 2-3-6 works as follows:

1. Pair up participants and let them test each other. Discard pairs with at least one identified bad test (for at least one of them the test must be bad). Now only pairs of good or pairs of bad participants will go to the next round. Pairs of bad participants can not be detected if both participants say that the other one is not cheating. This will normally happen because a cheating participant will always say that the checking was a success otherwise he will eliminate himself in this round.
2. Group pairs of participants from the round before into sets of size three and let them test each other. Due to the fact that every participant of a pair from round one is behaving the same it does not matter which client of another pair a client checks. Also discard

groups that have an identified bad test (at least one client of a pair from round one must fail the test).

3. Group groups of participants from the previous round into sets of size six (i.e. 36 clients) and let them test each other. Again, discard groups with an identified bad test. All participants reaching the end of this round will now be used as identified good nodes.
4. With the identifies good nodes from the step before we can now check every discarded participant from the three rounds before. This ensures that every client is checked and every cheating client will be detected.

This algorithm is also useful if we have non-reliable clients as said above, because we do not need to wait until round one is complete to start with round two. And every proven good client from round three can also check unchecked clients from round one to three, so the checking gets finished faster. The only important thing is that enough participants get through the first three rounds, to have at least more than half of all clients in the group of proven good clients to check all the unchecked clients in the fourth round.

Figure 4.1 gives an overview of the steps one to three. Note that for this algorithm it is

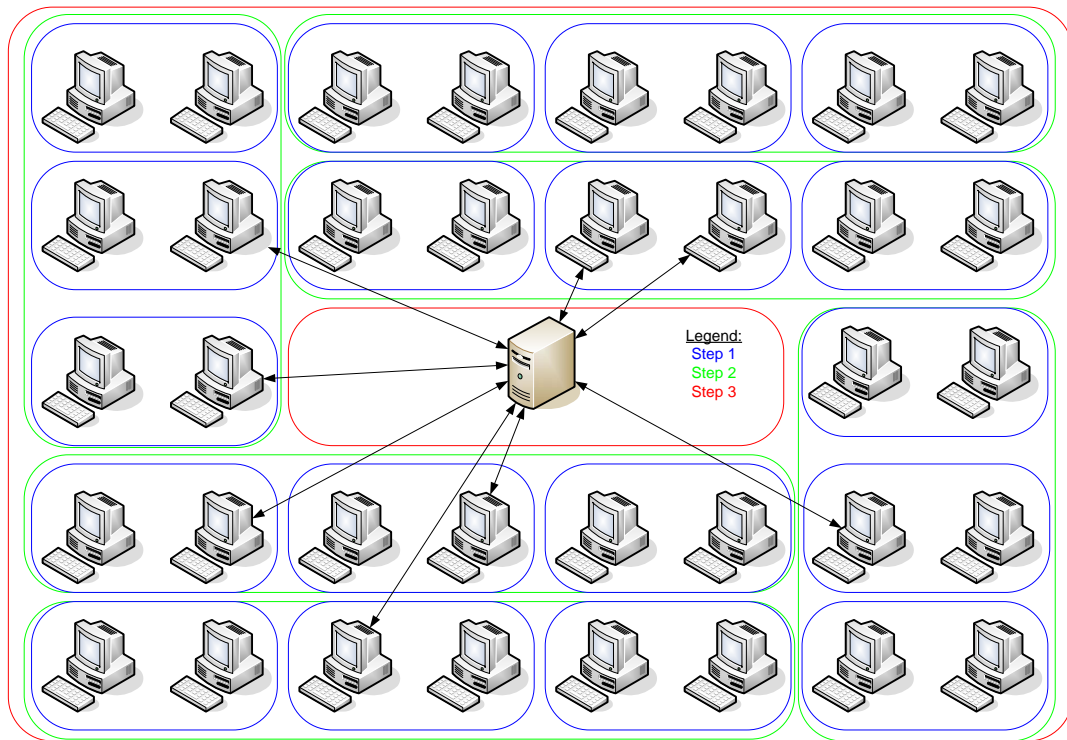


Figure 4.1: Distributed Checking: Step 1 - 3

highly important that one execution of the checking algorithm is considerably faster than recomputing the result. Because otherwise the checking will take longer than the computation itself. This means the checking function has to be fast, because it is executed several times during one checking round.

4.3 Analysis of the Algorithm

Let n be the number of clients, and m the number of groups consisting only of cheating clients. In the following, we will analyze three rounds 2-3-6, where in the first round, the nodes are paired up, in the second round the resulting groups from the previous round are grouped into sets of size three, and in the last round, groups of size six are formed.

Round 1

Let W denote the total number of groups consisting only of good nodes, B the total number of groups consisting of only bad nodes, and M the remaining groups. Moreover, let $n_1 = n$ be the total number of nodes and $m_1 = m$ be the number of bad nodes before round 1.

The idea for the analysis is as follows: By building pairs of clients three different groups are possible. A group of two good clients (γ), one of two bad (or cheating) ones (β) and one with one good and one bad one (δ). Given that we have n clients the number of groups built is $n/2$. If we build this groups one after each other then in every grouping step we have either γ , β or δ with a given probability. This is in the first grouping $\frac{m(m-1)}{n(n-1)}$ for β , $\frac{(n-m)(n-m-1)}{n(n-1)}$ and $\frac{2m(n-m)}{n(n-1)}$ for δ . Summing this up for all the pairs will give

$$\sum_{i=0}^{m_1/2} 2^{m_1-2i} \binom{n/2}{i} \binom{n/2-i}{m_1-2i}$$

In Figure 4.2 we show this graphically.

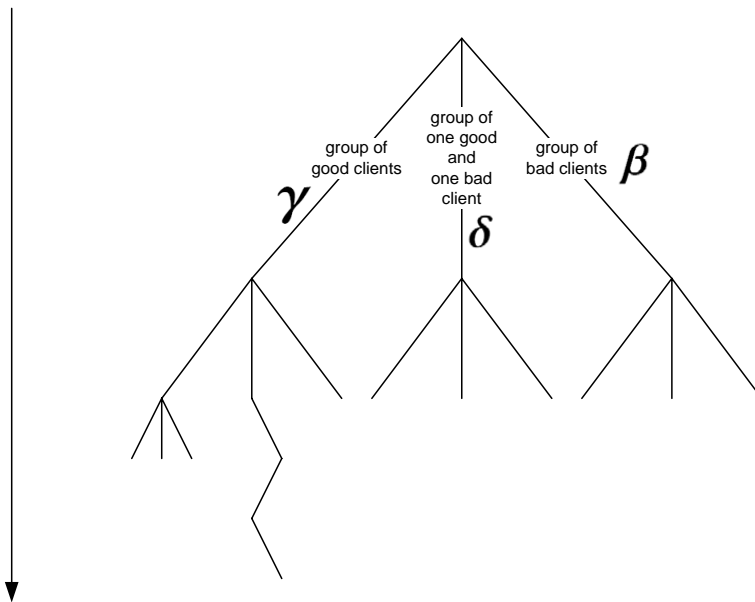


Figure 4.2: Distributed Checking Analysis Idea: Pairing Sequence

Summing over all possibilities, we have

$$Pr(B \geq k) = \frac{\sum_{i=k}^{m_1/2} 2^{m_1-2i} \binom{n/2}{i} \binom{n/2-i}{m_1-2i}}{\sum_{i=0}^{m_1/2} 2^{m_1-2i} \binom{n/2}{i} \binom{n/2-i}{m_1-2i}}$$

Concretely, for $n = 1,000$ and $m = 100$, the probability that there are more than 10 groups consisting of only bad nodes is 0.0172. Clearly, for larger n and the same fraction of bad nodes, the probability can only become smaller.

Observe that there are at most m mixed pairs, thus the remaining number of groups is at least $n_1/2 - m_1$. This means, if every cheater gets detected in the first round we have at least $n_1/2 - m_1$ groups for the remaining rounds. As seen above in the case where not every bad node is detected we have a bigger number of good nodes building good groups. The fraction of bad groups is even smaller and in round two and round three we get another chance to catch them.

Round 2

The groups of the preceding round now form groups of three. Let W now denote the total number of groups consisting only of good groups, B denote the number of groups consisting only of bad groups, and M denote the remaining triples. Let n_2 and m_2 be the total number and the number of bad groups of nodes respectively before round 2.

We have

$$Pr(B \geq k) = \frac{\sum_{i_1=k}^{m_2/3} \binom{n_2/3}{i_1} \sum_{i_2=0}^{n_2/3-i_1} 3^{i_1} 3^{n_2-3i_1-2i_2} \binom{n_2/3-i_1}{i_2} \binom{n_2/3-i_1-i_2}{m_2-3i_1-2i_2}}{\sum_{i_1=0}^{m_2/3} \binom{n_2/3}{i_1} \sum_{i_2=0}^{n_2/3-i_1} 3^{i_1} 3^{n_2-3i_1-2i_2} \binom{n_2/3-i_1}{i_2} \binom{n_2/3-i_1-i_2}{m_2-3i_1-2i_2}}$$

For example, for $n = 400^1$ and $m = 10$, the values we get by tacking the results from the round before, the probability that there are more than 6 groups consisting of only bad groups of nodes is practically zero. Clearly, for larger n and the same fraction of bad nodes, the probability can not become larger.

Observe that there are at most m_2 mixed groups, thus the remaining number of groups is at least $n_2/3 - m_2$. This means, that most bad nodes are recognized after this round. The remaining number of bad groups therefore will be very small and recognized in the next round.

Round 3

Similarly, for this round, using again W , B and M to denote the number of good, bad, and mixed groups at the end of the round n_3 the number of groups, and m_3 the number of bad groups round 3, we have

$$S(n, m, k) = \sum_{i_1=k}^{m_3/6} \binom{n_3/6}{i_1} \sum_{i_2=0}^{n_3/6-i_1} \binom{n_3/6-i_1}{i_2} \sum_{i_3=0}^{n_3/6-i_2} \binom{n_3/6-i_2}{i_3} \sum_{i_4=0}^{n_3/6-i_3} \binom{n_3/6-i_3}{i_4}$$

¹ $n = 400$ is a worst case value, given that no bad group was built in the step before. With the results of the previous round n would be 460.

$$\sum_{i_5=0}^{n_3/6-s_4} \binom{n_3/6-s_4}{i_5} \binom{n/6-s_5}{m-6i_1-5i_2-4i_3-3i_4-2i_5} 6^{i_5} 15^{i_4} 20^{i_3} 15^{i_2} 6^{i_1}$$

where s_j denotes $i_1 + \dots + i_j$.

4.4 Simulation Results

The simulations are done for the static case only (due to time constraints). This means no client will leave during the checking. Also different checking times for the clients are not taken into consideration.

For the analysis of the checking algorithm we have assumed that ten percent of our clients cheats, which is high compared to some real life projects. SETI@home for example only has one percent cheaters. Our distributed checking algorithm has even worked for this extremely high percentage, proving the validity of our approach.

The first simulation arrangement was as follows: The number of clients n was 1,000,000, the number of cheating clients m was 100,000. We simulated 50,000 checking rounds. The group sizes of the three steps were 2-3-3, which would be a bit worse than using group sizes 2-3-6 like we used in Chapter 4.3. The results are presented in Table 4.1. As we can see the number of groups only containing bad clients is zero and the number of proven good clients is greater than half of all clients (43908.51634 groups a 18 clients). This means we have enough proven good clients to check all the discarded clients in one step.

A second simulation scenario was to compare the algorithm with different numbers of clients.

Table 4.1: Distributed checking simulation results ($n = 1,000,000$, $m = 100,000$ and group sizes: 2-3-6)

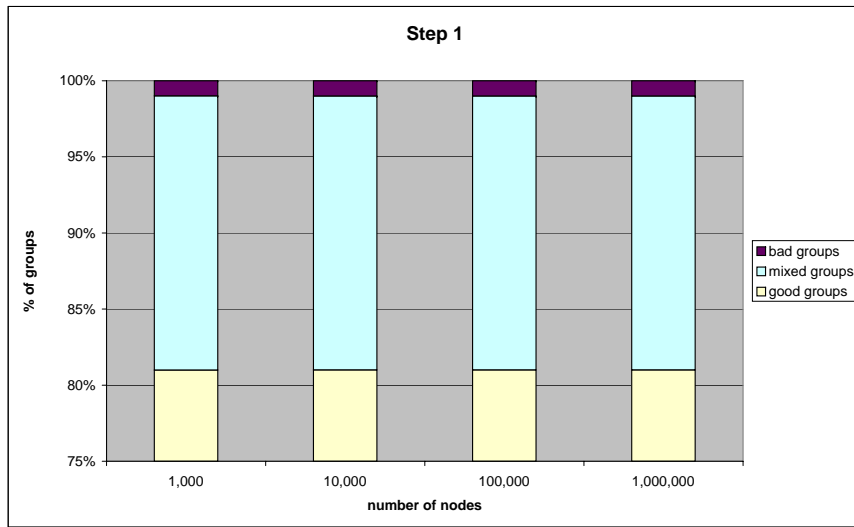
	○ nr. of good groups	○ nr. of mixed groups	○ nr. of bad groups
Step 1	404999.996864	90000.06272	4999.96864
Step 2	131727.0356	4939.69784	0.24494
Step 3	43908.51634	0.90996	0.0

The group sizes in this case were 2-3-6. The number of cheating clients was as before ten percent of the total number of clients.

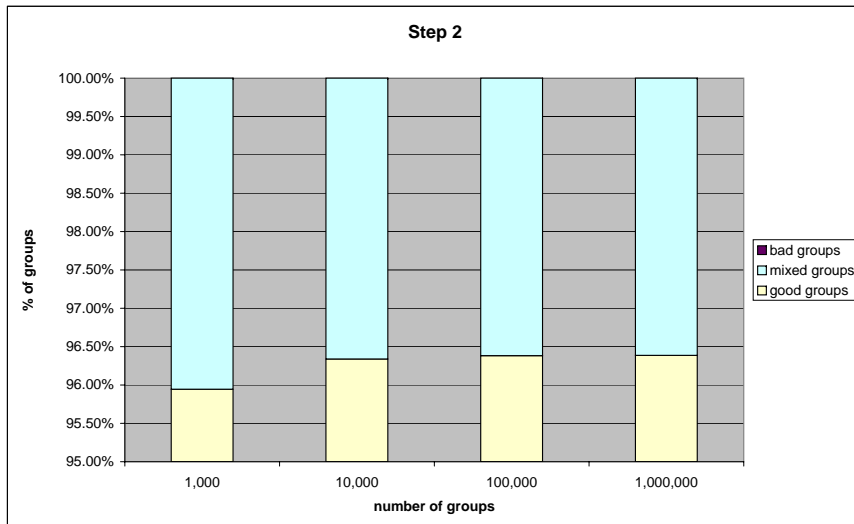
As we can see in Figure 4.3(a) most of the bad clients will be in a mixed group (one good and one bad client) and therefore discarded for the second step of the algorithm. The number of bad groups (only containing cheating nodes) is very small for every number of clients.

In the second step the ratio of good groups versus bad groups is even better. Also the percentage of mixed groups is shrunk. A ratio of over ninety five percent of good groups makes the initial position for step three very good. In Figure 4.3(b) we can also see that the values are getting better for a bigger number of clients which is very good for our application.

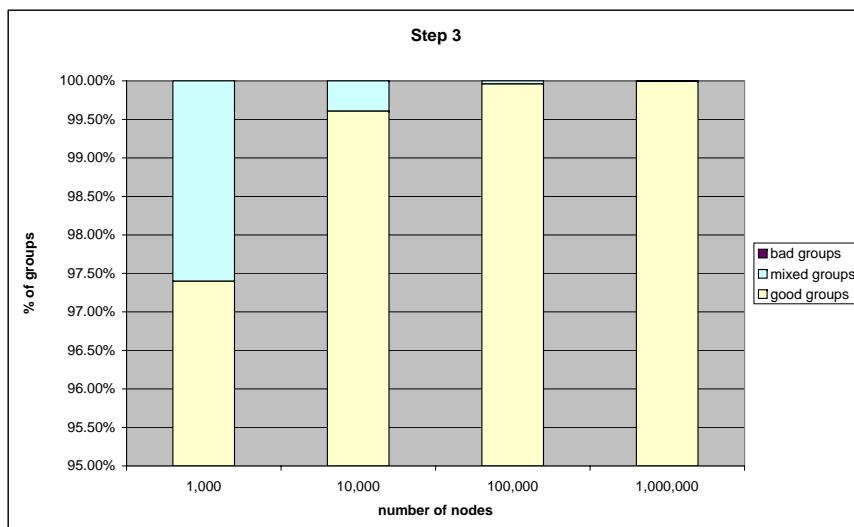
For the analysis of the checking algorithm is it very important that the number of bad groups after step three is zero. This means every group finishing this round is a group of only good nodes. In Figure 4.3(c) we can see, that the percentage of good groups is growing the more clients participate and more important that the number of bad groups be zero. Also very good is that for more clients the percentage of mixed groups is shrinking. This means that the more clients participate the more cheaters will be detected in the first two steps of the algorithm.



(a) Step 1



(b) Step 2



(c) Step 3

Figure 4.3: Simulation Results: Percentage Of Good, Mixed And Bad Groups (Observe Percentage Bias!)

For the dynamic simulation the main problem is the simulation of the dynamics of the clients. It is not clear which pattern for the leaving and reappearing of the clients should be used. We think that the more cheating clients can be tolerated in the static case, the more disappearing clients can be tolerated in the dynamic case. In Figure 4.4 we can see, that even for 25 per cent cheaters no group of only cheaters will be built in the static case. Therefore if the number of cheaters is small also disappearing clients should be tolerable.

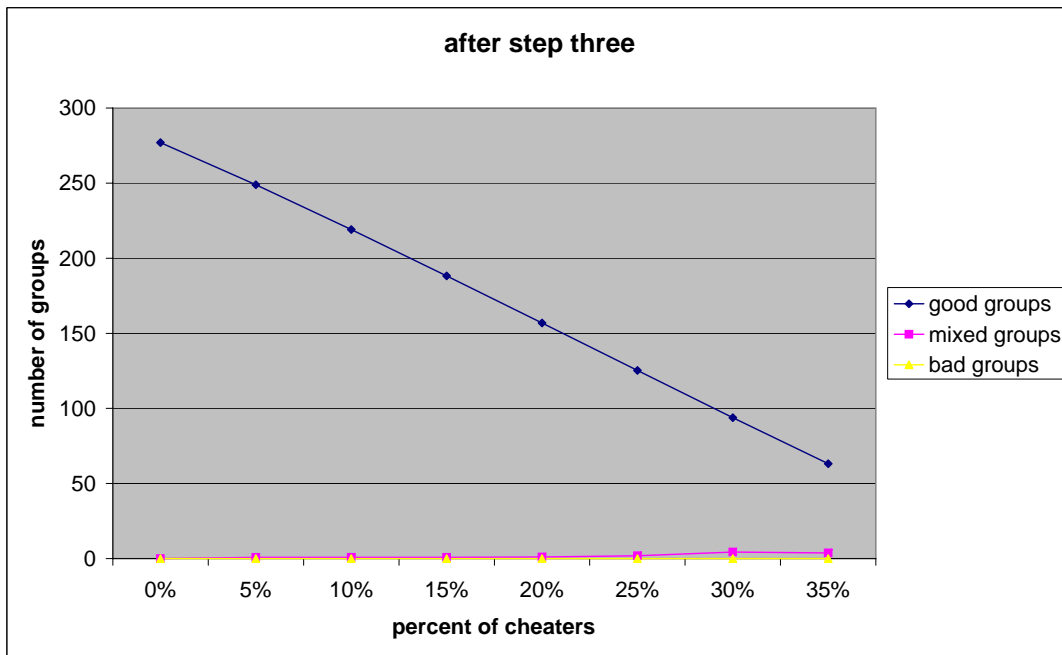


Figure 4.4: Simulation Results For Different Percentages Of Cheaters (Number Of Clients: 10,000)

5

Architecture

In this chapter we will give an overview of the architecture. First we start with a short description of the whole system. After that we will look at the different parts, mainly the server side and the client side, and we finish with a short example of a complete computation cycle.

5.1 The Whole System

As said in Chapter 2 we use the BOINC framework. This framework provides a client part and a server part. Figure 5.1 gives an overview of how the client and the project server(s) interact.

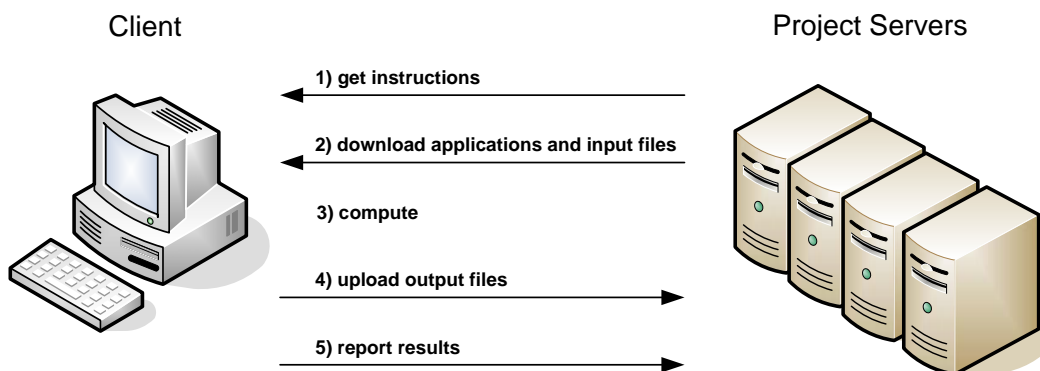


Figure 5.1: Interaction Between Server And Client

1. The client gets a set of instructions from the project's scheduling server. The instructions depend on the client's PC: for example, the server will not give it work that requires more RAM than the client has. The instructions may include multiple pieces

of work. Projects can support several applications, and the server may send the client work from any of them.

2. The client downloads the executable and input files from the project's data server. If the project releases new versions of its applications, the executable files are downloaded automatically to the client.
3. The client PC runs the application programs, producing output files.
4. The client PC uploads the output files to the data server.
5. Later (up to several days, depending on the client's preferences) the client PC reports the completed results to the scheduling server, and gets instructions for more work.

This cycle is repeated infinitely. The BOINC framework does all this automatically.

An overview of the BOINC system architecture is given in Figure 5.2. All the objects that have a blue color are part of the framework, whereas the red objects have to be built by the project that is being hosted on the BOINC System. The participant is a person donating his computer's idle processing capabilities to a project. The participant has two interfaces used to interact with the system. The first is the Project's Web Server that is hosted by the project and the second is the BOINC Manager used to manage the applications running on the participants host.

The communication between the clients and the server is done through HTTP requests and the client always starts the communication. With this mechanism the problems with firewalls are mostly solved.

5.2 The Server

The server is the biggest component of the BOINC framework. It can be divided roughly in two parts. On the one side we have the data part, where all the data handling is done, and on the other side we have the Web site part, where all the information about the users, data etc. is produced for participants and administrators in a human-readable form.

5.2.1 The Data Part

This is the main part of the server. Here everything concerning data is handled. The data part consist of a few daemons, tasks and a database.

We used the database provided by the framework and extended it at some places if necessary. We also had to add additional tables to store other information, mainly for the distributed checking.

The basic setup provided by the framework looks as follows:

First of all, work has to be generated. This has to be done by a work generating daemon which generates workunits to be computed by the clients. Every workunit then gets distributed to one or more participants to be computed. For this, the server creates one or more results for every workunit. Every result corresponds to one client computing the workunit. These results, which are not assigned to any participant at the beginning, will be placed in an array of unspent work in a shared memory segment by a daemon called "feeder". Every time a client requests more work, the scheduler called "cgi" takes one or more unassigned results

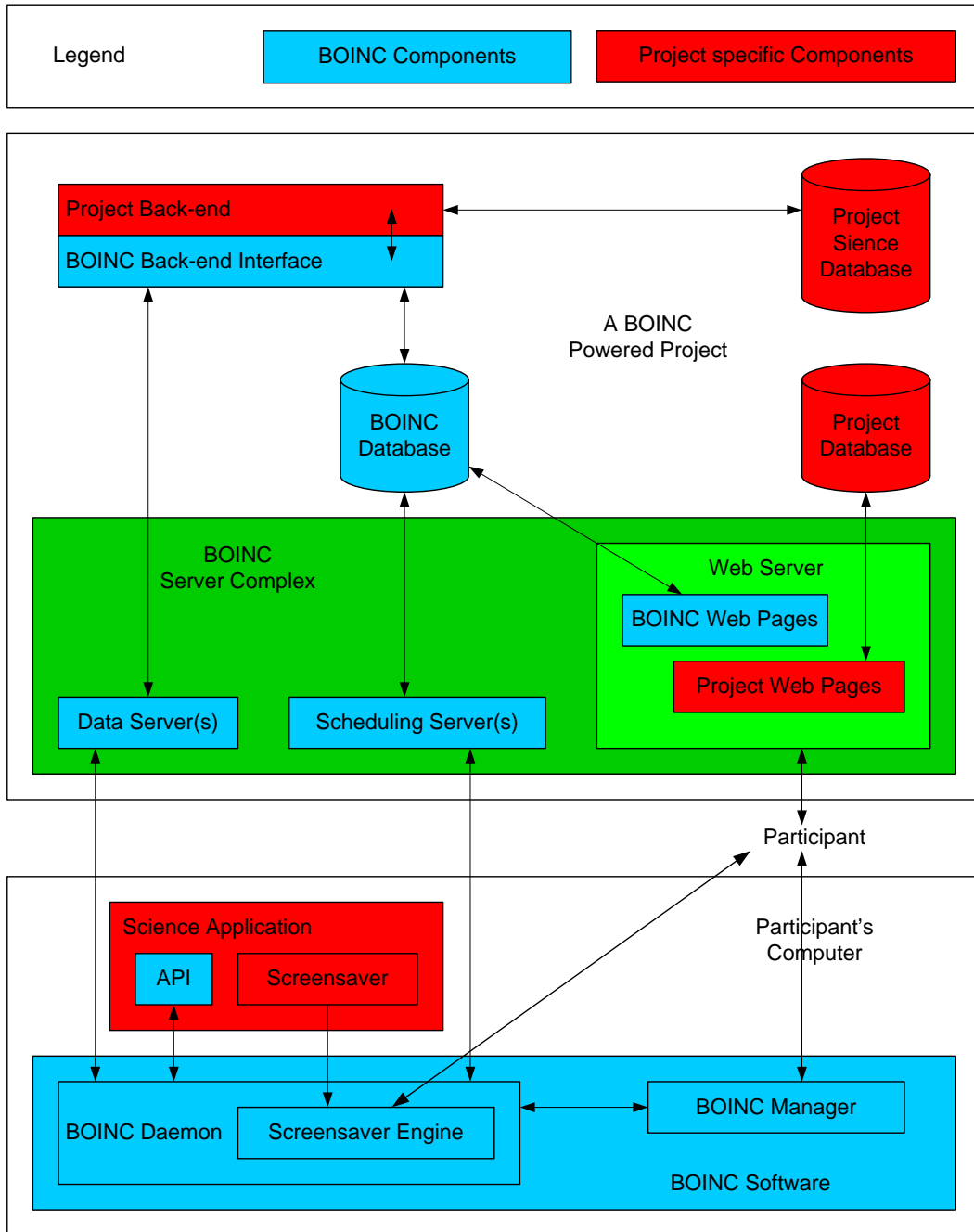


Figure 5.2: BOINC System Architecture

from this array and assigns them to this client. The client can then download these workunits and process them. After the computation is done by the client, he has to upload the results. This process will be handled by the `file_upload_handler`. After a result is reported to the server it gets validated by a validator. The standard distributed validator does a majority decision on the reported results. If the result passes the validation, it will be assimilated, meaning that the results will be stored in a useful manner. Finally, the file deleter daemon deletes all the files not used anymore. The transitions of the different results is controlled by a transitioner daemon. It takes care of the fact that a result has to be reported as computed and after this, the result gets validated, assimilated and the files deleted. Figure 5.3 gives an

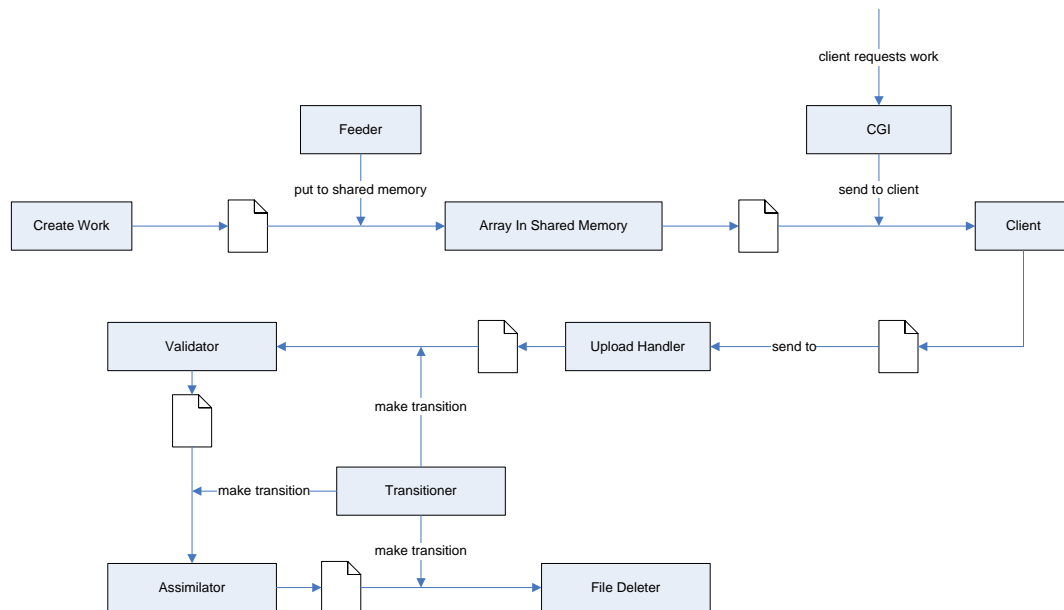


Figure 5.3: BOINC Standard Data Flow

overview of this process.

Because we wanted to do distributed checking we had to change some parts. The validation will now only be done on the server if the number of clients is very small (< 1000 clients). Otherwise the validator will only check whether the reported files have the right format. This means we now can have results which are invalid while the workunit to which the results belong is valid. This is not a problem, because every reported result is stored as invalid until it gets validated by either a checking round or the server. If a collision occurred it will not be checked until the results are validated. The idea of the distributed checking is also to know who is cheating on the results and to be able to lock this participant out.

To make the distributed checking possible we had to create two sorts of workunits. A normal computing workunit which is the same as the workunits from the framework, and a so called checking workunit, which is created on demand and assigned to a specific client to check the computed results of the different clients. For the client, there is no difference in the file handling and the way he gets his work or how he has to report his results. Both sorts of workunits are stored in the normal workunit table labeled with a newly added flag, to allow an administrator to see on the workunit list which is a normal and which is a checking workunit. To be able to assign a workunit to a specific client, we had to change the CGI program. Now

it does not only pick an unsent entry from the shared memory and sends this to the client but also checks if this specific client has to do some checking work. If this is the case it creates a checking workunit and assigns this workunit to this client.

The computation on the client is the same as with the standard framework. Also after the output file of the client is sent to the server the steps are the same. We have our own validator which validates the files and our own assimilator. The assimilator differentiates between output files from normal computing and output files from checking. With the information in the output files from a checking workunit it updates the checking controlling status.

To control the whole distributed checking part we had to add some tables to store the different possible states and we created a daemon which does the state transitions with help of the information given by the output files of the checking workunits. One important piece of information for checking is in which round a participant is at the moment. Thus, for every host the checking state is saved in the database. If a client fails the checking we add one to his fail count. This allows to control when a client has to be excluded from participating in the future. But not every client delivering a wrong result, for example generated by a computation error, will lead to a ban of this client.

We also have a process which checks every valid result created by a participant on collisions with other valid results to find the solution as fast as possible.

This process is shown in Figure 5.4.

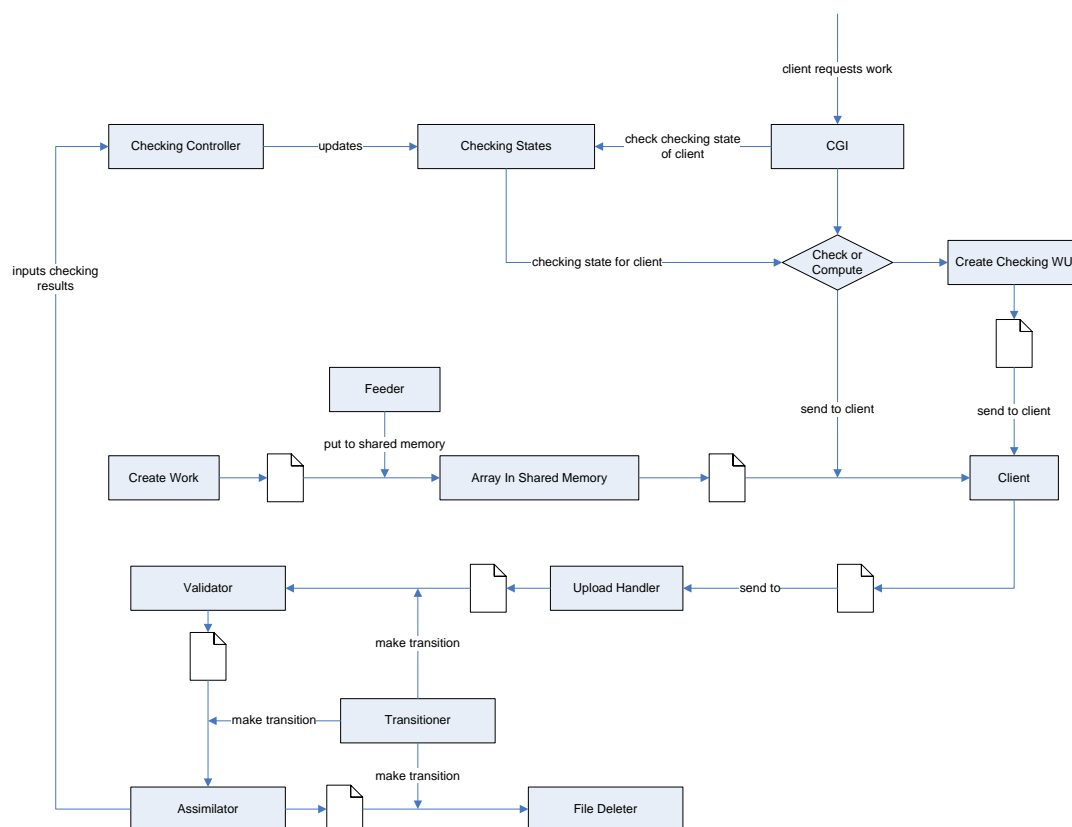


Figure 5.4: New BOINC Data Flow

5.2.2 The Web Site Part

On the Web site part we had to change a couple of things. On the publicly viewable website all the dummy entries for the project have to be replaced by project specific entries. E. g., we had to insert some information about the project and remove the dummy news. This part is deliberately held very simple by the framework, because every project has to adapt this part. Therefore the pages are held very modular.

In the second part, the administration Web site, we had to make more important changes. The most important changes all concern the changes we have made in the framework. We wanted to be able to see for every host if he is checking results or just computing results. Also on the list of workunits we wanted to be able to see if a workunit is a normal computation workunit or a checking workunit. We had to adjust all the outputs for database tables we changed for our project.

On this part, a lot of work could be done in the future to make the project more attractive for participants.

5.2.3 Security

A reported point can easily be checked. For every reported point, the client also reports the parameters corresponding to this point. The checking can be done by computing the point corresponding to the given parameters and then comparing this newly computed point to the submitted one.

To be sure that a client really computes a trail and sends back distinguished points from this trail, the application reports a pair of points every time: the distinguished point and the point one step before. Along with the points, it sends the values to compute this point. With this we can avoid that a cheating client just chooses random values and then computes the corresponding point. If a cheating client only computes one point and sends this point back several times, it will be recognized in the checking round, because the step from one to the next point will not be valid. Also if he does the step one time and reports this step several times, this will be recognized, because the cheating client then only has one result pair in the result set, due to the fact that double results will be discarded.

It is clear that a cheating client still can cheat by choosing two random values, computing the point, do a step on the trail and report these two points. But if a client does these steps he could also do a whole trail, since this is not harder to do. The problem of just sending random data back should be solved with this. It is not to be expected that a cheating client will invest a lot of computing power to create data that is accepted by the checking algorithm.

A yet unsolved problem is the dealing of replay attacks. If a client computes one workunit and then reports for every future workunit he has to compute the result of the first computed workunit without doing any work for it our system will not recognize it. This happens because no checks are done if the given results correspond to the given input of the workunit. Some ideas to solve this problem will be given in Chapter 7.2.

5.3 The Client

As client software we wanted to use the original client provided by BOINC. The client of the BOINC framework allows to join multiple projects with only one client. If we change the client, this would not be possible and the number of possible contributors would be a lot

smaller.

As said before, the framework is designed to let the client run an application for every project. This means that the client gets an application with which he then computes the work. The client software is actually not only one program running on the participant's PC. It consists of a core client, a GUI, a screensaver and one or more applications (see "Participant's Computer" part in Figure 5.2).

The decision to use the provided client does not mean that distributed checking is not possible. Our application switches between computing and checking given the mode defined in the input file. If the client gets a normal workunit, defined by the line # `MODE: COMPUTE` in the input file, it computes the solutions normally. If it gets a checking workunit, defined by the line # `MODE: CHECK` in the input file, it checks the included results.

Our application in the computation mode does the following:

It initializes the computation with the values from the input file. Then it does a random walk until a defined number of distinguished points has been reached. Every distinguished point will be written to the output file, in which the results to the server are reported. If enough distinguished points have been found the client terminates the workunit and sends the output file back to the server.

In the checking mode the application behaves as described below:

Every result in the input file is checked whether it is valid. The application writes every result to the output file if the result is valid. If all results are valid, the application also writes the status "valid" for the checked client to the output file. If at least one result is invalid the status of the checked client will be set to false in the output file. The reason for also giving a status flag is that the interpretation of the output file on the server side gets faster.

5.4 An Example

Here we try to give an overview of the way data has to go from the creation until the storage at the end and, hopefully, the finding of the solution.

First of all, a workunit is created. Let us call it workunit number one or *WU1* for short. *WU1* then gets scheduled in the array of ready results. At some time the CGI program assigns *WU1* to, say, host "Oahu". The host will download the input file from the BOINC server and compute a trail with the given input. After this is finished the host "Oahu" uploads the output file to the BOINC server and reports *WU1* as finished. The output file contains a number of possible collision points with their corresponding values. After the workunit is reported to be finished, the validator will validate the file and if the format of the file is OK, set the workunit to valid. If this has happened, the assimilator will write all the points to the database as unvalidated results. After this is done the file deleter will delete the uploaded output file of client "Oahu" to make room for new results.

Some time later the client "Oahu" will join in a checking round. Now his produced results will be checked by another client. Let's call this client "Molokai". The checking controller has paired the client "Oahu" and the client "Molokai" for the first checking round. The next time the client "Molokai" requests new work, he will receive a checking workunit with the results of client "Oahu". He validates these results and sends the output file back to the BOINC server. Again, the validator validates if the file has the right format and the assimilator updates the checking informations for the client "Oahu" and his results. Here also if all this is done successfully, the file deleter will delete the files corresponding to this workunit.

The results now have to endure the next two checking rounds, where also other clients than “Molokai” will check the result, and finally get set to valid if the client “Oahu” does not fail in a checking step.

After the results are marked as valid, a daemon will grab them and check if a point collides with another already validated point stored in the database. If this is the case we have found the solution and rejoice. Otherwise the results will be stored in a table to be checked with for future results.

A schematic overview is given in Figure 5.5. In Figure 5.6 we give a schematic overview of the whole system during the checking round.

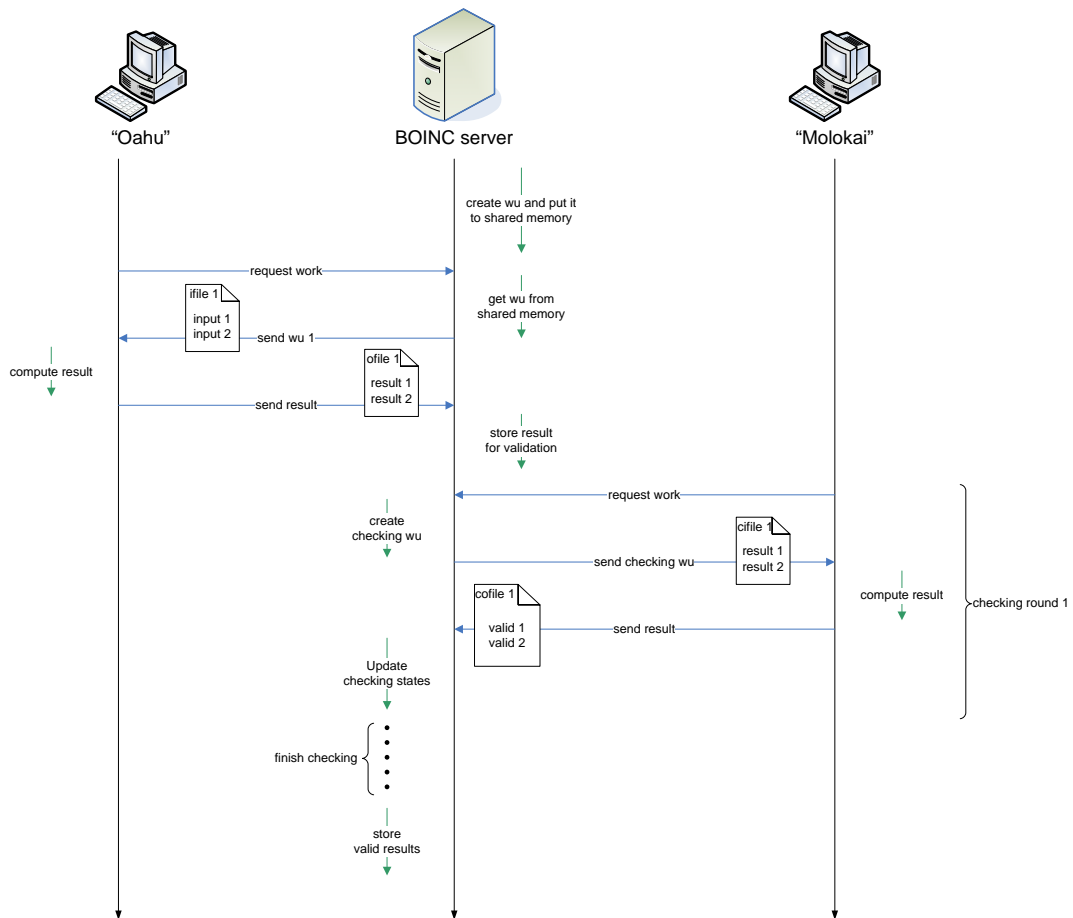


Figure 5.5: BOINC Time Line

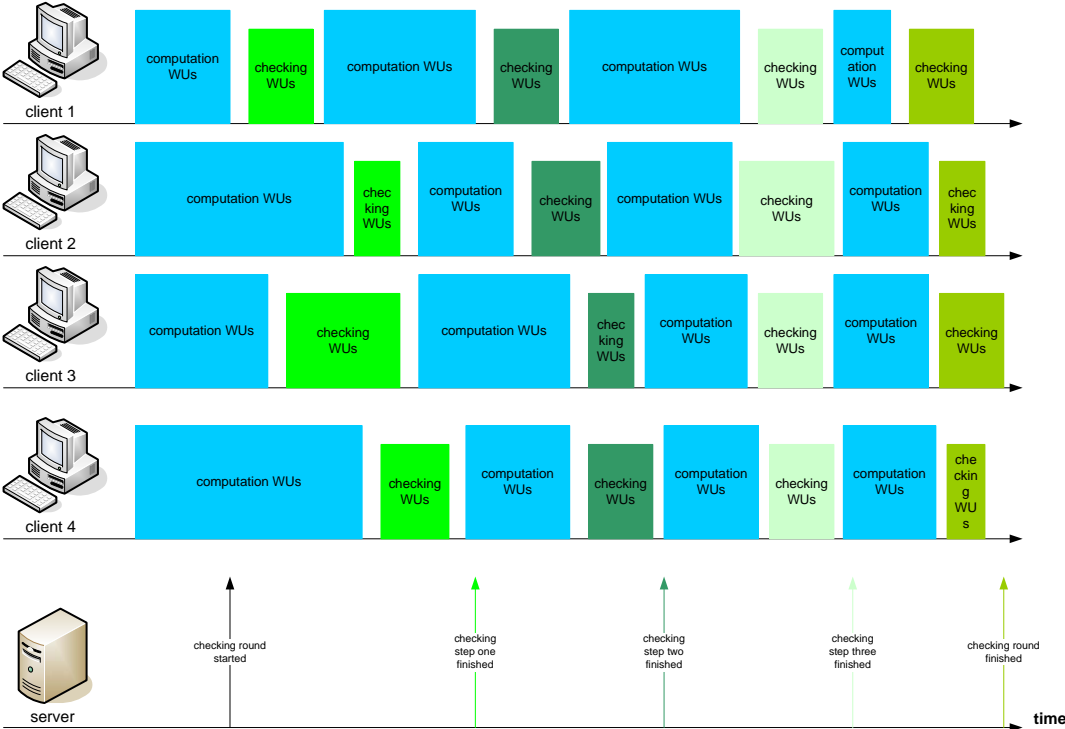


Figure 5.6: Schematic Overview Of A Checking Round

6

Implementation

Since the BOINC Framework is implemented in C++, we had to implement all the changes to the framework and the application running on the clients in C++. We separated the description of the implementation into two parts: the server side and the client side. Instead of giving a detailed overview of the implementation, this chapter will describe the problems we encountered during the implementation and their solutions

6.1 The Server

6.1.1 General

The server consists of some daemons and tasks running. Daemons are programs running continuously while tasks are programs running periodically. The tasks are executed by a program which should be run by cron. The first part we had to implement was a daemon for creating work. This daemon has to create the input files for the workunit and call a function from the core of the BOINC framework which creates a workunit for the given file. The file will hold the start values for the Pollard Rho algorithm and some configuration options for the client. Since these start values are chosen randomly from the complete space of the field on which the elliptic curve relies and can be very big (over 100 bits), it was a bit tricky to generate this values.

The `feeder`, running as a daemon, which puts the ready-to-compute results of the workunits in the shared memory segment, had to be trimmed so that he only takes ready-to-compute computation workunit results and no checking workunit results. This was very simple; all we had to do was setting the flag for computation workunits on the selection criteria. No further changes were needed.

The trickiest part on the implementation of the server was to change the CGI program “`cgi`” which handles all the client requests and assigns the workunits to the clients. The assignment of a client to a workunit works as follows:

On the creation of a workunit 1 to n , the exact number is defined in the workunit template,

so-called results are generated. ‘The value n in the template is equal to the quorum size for the majority decision normally made by the BOINC framework. Since we only send every workunit once, only one result is created every time. A result corresponds to a client computing the workunit. Every result is scheduled separately and will be assigned to a client by the CGI program.

When using the original CGI program from the BOINC framework, a client gets a random ready-to-compute result from the shared memory segment, filled by the `feeder`.

For our distributed checking we have to be able to assign a specific workunit to a specific client to give a client a checking workunit result if needed, or a normal computing workunit result otherwise. To make this possible we slightly changed the way a client gets its work. Instead of simply grabbing a ready-to-compute workunit result we now check first if the client has to do some checking work. If this is not the case the client gets a random ready workunit result to compute as with the standard CGI program. If the client still has outstanding checking work the CGI program will create a checking workunit and assign this workunit result immediately to the requesting client.

What sounds very simple here was in fact quite complex. First we had to find where the assignment of the workunit results to the clients is done. After this we had to extend it such that before the assignment is done the program checks for the checking work. This was the simple part. The checking for the checking work, the creation of the checking workunit and the assignment of this specific workunit by assigning the assigned workunit result was the complicated part. We had a lot of trouble with the creation of the checking workunit. For the checking workunit, an input file containing all the computation results to check must be generated. But at the time of the file creation we do not know the name of the file, since it has to be unique exactly as the name of the checking workunit. The uniqueness of the file name is needed because otherwise it can happen that a client downloads the wrong input file. So we had to create a temporary file and copy the content into the right file after the creation, creating the name of the right file according to the IDs of the results which should be checked. Here we had some problems with access rights as well as problems with uninitialized pointers for the checking workunit creation. Mainly the uninitialized pointer problem was very hard to find.

The second big problem was that after the creation of the checking workunit the program does not know the ID of the workunit result assigned to the newly created checking workunit. This is the case because normally the workunit results will be created and the `feeder` puts the ready workunit results into shared memory, from where they get randomly assigned to clients. Because we needed to assign the workunit result directly after the creation to a given client, we had to get the ID of this workunit result. We wanted to read this ID from the database, but if we query the database right after the creation of the workunit the ID can not be read. The result set was empty, even if the transaction is set to `READ_UNCOMMITTED` mode. We solved this problem with a `sleep(5)` statement after the checking workunit creation and before the selection of the ID from the database. With this the problem vanished and the ID can be selected. This `sleep(5)` statement seems to be very bad, because a huge amount of request to the server may block each other. The fact that every request starts a new instance of the program solves this problem.

For implementing the `validator`, also running as a daemon, we simply extended the `sample_bitwise_validator` from the BOINC framework. The `sample_bitwise_validator` does a bit wise comparison of the result files if there are

more than one and an additional validation of the result files. Our `validator` will now only check if the file contains data readable by our `assimilator`.

Our `assimilator` is relatively simple. Like the original `assimilator` our `assimilator` is running as a daemon. It checks whether the file is a result of a computation workunit or of a checking workunit. For the computation workunits it simply goes through the results and stores them in the database, setting the valid flag to false, independent of whether we are doing “server-side” checking or distributed checking. If the file belongs to a checking workunit it updates the statuses of the clients in the different checking rounds according to the information stored in the file. It also sets the results to valid if the result file is that of the final checking round.

The `transitioner` and the `file_deleter` will be the same as provided by the BOINC framework and run as daemons as they do on the standard framework.

To handle the unvalidated results of the Pollard Rho algorithm we store the results in two tables. One for the unvalidated or validated results, which have not been checked for collisions, the so-called “reported results”, and one table for the validated, which have been checked for collisions, the so-called “stored results”. To find the solution we have a periodically running task which selects all valid results from the reported results, checks them for collisions with results already stored and then inserts the selected results into the group of stored results. If a collision occurs, a file with the two colliding points is generated by this task.

The first implementation of this task was in Python. We noticed that this Python script is running very slowly and after a while the number of validated but not for collision checked results was very big. On a refactoring we tried to solve this problem by implementing the new task in PHP, which seems to be faster. Some tests with more than 10 clients showed that the PHP script is still slow, thus we decided to implement a daemon to do this work. After solving some SQL problems the daemon was a lot faster than the PHP script.

6.1.2 The Distributed Checking

For the distributed checking we had to change some parts of the BOINC framework and add some additional parts. The changes in existing parts are explained above. New is a checking controller daemon. The `checking_controller` controls the checking as its name tells. It has two modes: “server-side checking” and “distributed checking”. The mode is set according to the number of hosts participating. If the number of hosts is small (only a few hundreds), the `checking_controller` will check the unvalidated results himself. Here all the computations on the elliptic curve will be done with the Crypto++ library [8]. To not block the `checking_controller` for a long time it only checks a configurable number of results each time. The process is as follows: check if the mode has to be updated, then check the mode, and check the configured number of results. After this is done the `checking_controller` will restart at the beginning.

If we are in the “distributed checking” mode it behaves as follows. At the beginning of a checking round it puts all the clients who interacted with the server in the last 48 hours into checking step one. The maximal allowed time between the last interaction of a client with the server can be manipulated with an input parameter of the `checking_controller`. Also the group size of the step two and the group size of the step three can be manipulated like this.

During the checking round the clients get paired up. After this is done the clients will get

checking work when they arrive at the server the next time. The `checking_controller` then does the transitions from checking step one to checking step two, after this, from checking step two to checking step three and at the end of checking step three to checking step four. After this is done it will restart at the beginning.

Here the problem was how to do the switch from “server-side checking” to “distributed checking”. Checking the number of hosts registered on the database every time we restart at the beginning of the while loop, would mean a database lookup every time the `checking_controller` restarts. This not is necessary, because switching the mode is not time critical. If the switch is not done at the moment when the boundary of switching to distributed checking is reached, will not produce invalid results. So we decided to lookup the number of hosts once every hour and save the state on the `checking_controller` and not lift the load on the database more than needed.

A state diagram of the `checking_controller` is given in Figure 6.1.

In principle the checking algorithm is round based, but because we have non-reliable clients where it could take a very long time until a client reports again, waiting until a round is completely finished will not work or take ages. So we decided to do a kind of interleaved round-based approach. This means that we check the transitions between the steps after each other and cycle through this process for ever. For every transition we check the number of clients ready to go to the next step. If we find enough ready clients to build a complete group in the next round we put them together into the next round. This will lead to a finished complete checking round much faster. We let also every client which is in checking round four check both the discarded pairs of the rounds one to three and, the unchecked clients of the rounds one to three. With this we can “replace” clients which started at the beginning of the checking round but did not report their checking results for a long time. This prevents that the checking gets stuck when a client should check some results but does no longer report any checking results and with this block all other clients in his group.

This also has some disadvantages. The fact that every time when enough clients of one step are ready they will be grouped for the next step, will open some space for attacks. This pseudo-random group building is resistant against single cheaters, because one cheater can not remain undetected in his group. But if a group of cheaters works together this approach is not so robust anymore. If a group of cooperating cheaters delivers their checking results together and manages to get all their groups ready to go to the next step and also bluff all the non-cheating clients on the groups if there are some, it may happen that all these cheaters will come as a group to the next step or in the worst case to be accepted as good clients if they manage to pass every step of the algorithm. Preventing this possible attack is one reason for a round-based `checking_controller` running as standalone daemon. In principle the transition of the clients could also be done event-based, every time a client reports its results. With the round-based approach we work against the attack by the fact that at the time the cheaters report their checking results they do not now in which state the `checking_controller` is. They also do not know when the state transition effectively will happen. In the time between the moment the collaborating cheaters report their checking results and the moment the transition is made other clients can also report their checking results and with this reduce the chance that the collaborating cheaters will be grouped in the next step. Nevertheless it is possible. To eliminate this, the `checking_controller` must do a check. For example for every newly build group the `checking_controller` has to check one random results of every client in the group, by comparing the delivered result with

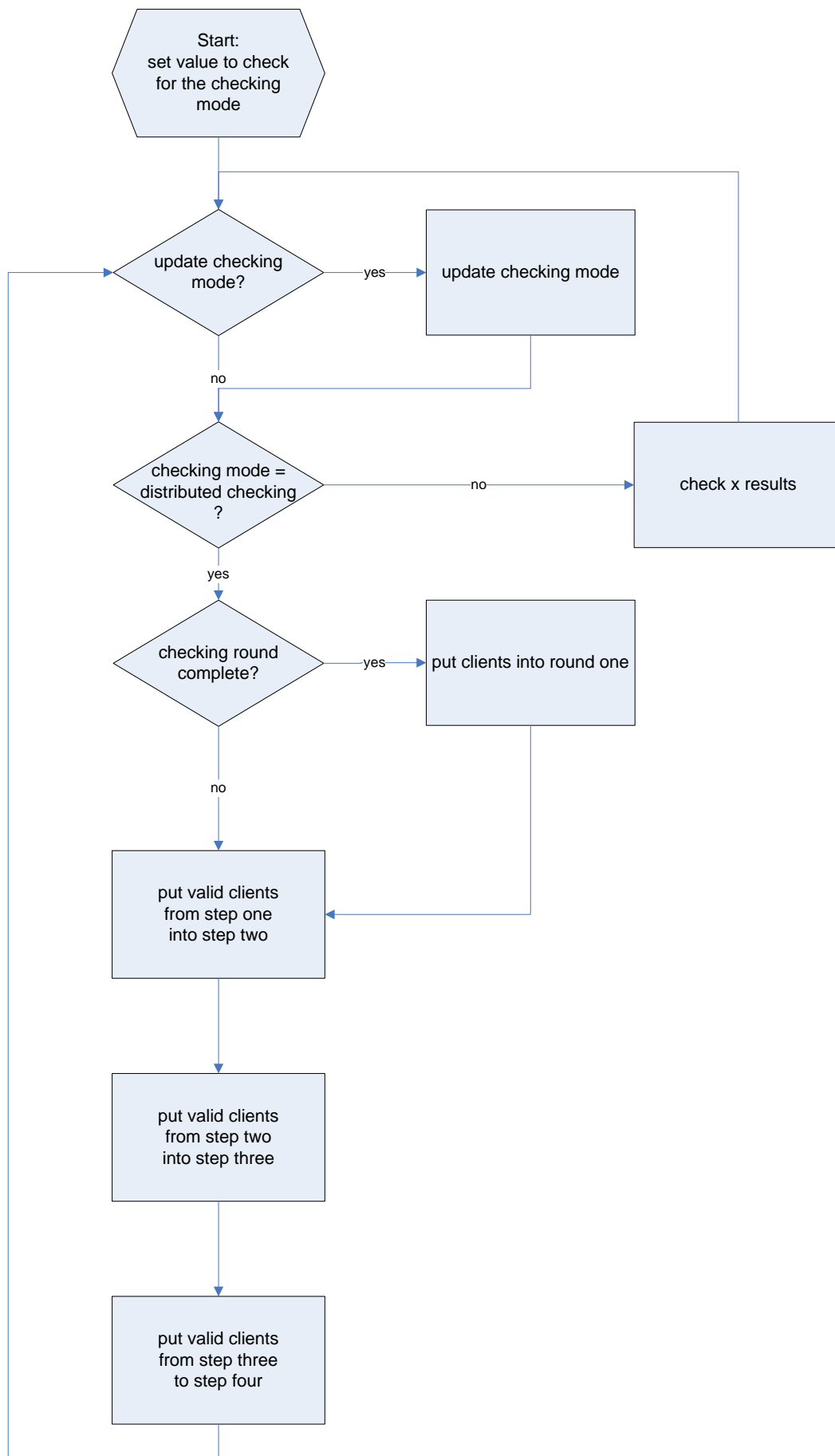


Figure 6.1: State Machine Of The Checking Controller

the self computed one. Certainly there are also other approaches to solve this problem.

6.2 The Client

As client we use the normal client of the BOINC framework mentioned in Chapter 5.3. The application running on this client is a simple C++ program taking an input file of the workunit which generates an output file. Given the input file the application decides if it has to compute solutions for the Pollard Rho algorithm or check the results in the input file.

As on the server side, the client uses the Crypto++ library for computations on the elliptic curve regardless of whether it is computing points or checking results.

The decision criterion for the distinguished points we use is based on the computation of $Hash(x, y) \pmod{m}$. We then select every point for which the results lies in a given range of the possible results.

A problem on the client side was to find the moment when the computation of a trail has to stop. We implemented two stopping criteria in parallel.

First, the number of distinguished points is bounded. If a certain number of distinguished points is reached, the client will finish the computation and send back the results.

The second criterion is the number of steps made since the last distinguished point. If this number crosses a given barrier, the computation will also be terminated. This barrier is chosen according to the elliptic curve and the criterion for choosing the distinguished points.

This abort criterion is a pseudo loop detection for the trail. As explained on Chapter 3.2.2, it is possible to loop on points not containing a distinguished point. To detect this loop normally some loop detection algorithm has to be used. But these algorithms are often complex and we wanted to leave the computation as simple as possible.

Given the criterion for a distinguished point, we can define a maximum number of steps until the next distinguished point is reached on average. We now define that a loop is happening if we do not reach a distinguished point after x times this maximal number of steps, even if we are not in a loop. This is much simpler than using real loop detection. Unfortunately this can be used for attacks, by just sending an empty result and claiming that a lot of computer power was needed to create it.

We have not developed any graphical parts which can be displayed while the screensaver is running.

7

Conclusions

7.1 Achievements

The goal of this master thesis was to develop a grid computing framework that allows to tackle computational puzzles such as the factorization of large numbers. The beginning of this master thesis was an evaluation of the different existing frameworks and projects. The next step was to decide which framework should be used and which computational challenge should be solved. After that the implementation had to be done, as well as finding and implementing some strategies to avoid cheating. At the end all the different parts had to be put together to one system and this system had to be tested.

7.2 Future Work

As in every project there is still room for improvements. We will give an overview of some improvements that can be done in future.

- **Improve the website**

One of the most important parts of a BOINC project is the website. The website represents the project to the world. At the moment we mostly have the standard website, improved with some customization. For the future it would be very nice if the website will have some more information about the project and a unique look instead of the standard look from the dummy website. There are a lot of possibilities to improve the website which are not named on this report, but would also be nice to be made.

- **Implement a graphical part for the screensaver**

At the moment when using the BOINC screensaver only the standard dummy screensaver will appear. If some graphics, for example the status of the search, would be shown on a later time would be lovely. For this it will be necessary to make oneself familiar with the graphical library of BOINC.

- **Improve the computation of the Pollard Rho algorithm**

To speed up the finding of a solution it is possible to use some optimizations of the Pollard Rho algorithm. At the moment various optimization to speed up exist and it will be good to evaluate these and if possible use one or more of them.

- **Try to find stakeholders helping to distribute the client**

To get a lot of users helping to find a solution faster having some stakeholders helping to distribute the client would bring the project faster to a lot of participants.

- **Protection against replay attacks**

As said before in the thesis, replay attacks on the system will not be recognized. This has different reasons: No double checking of the results is done and the results have no relationship to the input values, given in the input file of the workunit. Also no checking is done to catch cheaters who report distinguished points not lain on the same trace.

So for the future a solution for this problem has to be found. Some solution ideas will be given below.

To solve the problem of unrelated distinguished point the client has also to report the number of the iteration step in which the distinguished point was created. With help of this information the checking not only has to check the validity of the distinguished point but also check for one random interval between two distinguished point if the number of steps between them is the number the client says.

The problem of the results which are not related to the input can be solved by storing for every workunit the start values. With this help on the checking the checker can check if with the given start values the first reported distinguished point is reachable in the reported number of steps.

The two described solutions can surely also be combined. The problem of this approach is that the checking will have to recompute around one per mille of the work. A possible solution to this problem can be to also report some “checking points” between the distinguished points which have less steps between them. Then the checking if a client has done the whole trail can be done with these “checking points” which reduces the number of steps computed by a factor two. Still the relation to the input values has to be kept in mind.



Installing The BOINC Server

A.1 Requirements

First of all, you need a recent Linux release. We used a Ubuntu 6.10 [29] installation. For the Server to run correctly you need the additional packages listed in Table A.1:

Table A.1: Additional packages

Packages
autoconf 2.58+
automake 1.8+
make 3.79+
m4 1.4+
GNU tools
libtool 1.4+
pkg-config 0.15+
GCC 3.0.4+
g++
libcurl 7.15.5+ (we used libcurl3-dev)
php4.0+
apache2
mysql 4.1+ server and client
openSSL 0.9.8+ (also libssl-dev)
python 2.2+

We installed the packages with apt-get. Pay attention that you install the developer parts as well to prevent a lot of errors during the installation. We will explain the installation of some other required components, e.g. apache2, later in this chapter.

A.1.1 Install MySQL, apache2 and PHP

First, install the MySQL server and client as well as the python-mysqldb package. We have done this by executing the following command:

```
apt-get install mysql-server mysql-client libmysqlclient15-dev
apt-get install python-mysqldb
```

After this is done you have to set a password for the root account on MySQL. This can be done with the following command:

```
/usr/bin/mysqladmin -u root password 'myPassword'
```

To suppress the password checking for the shell user root, you can create a file `.my.cnf` in the home directory as described below.

```
joe .my.cnf

[mysql]
user=root
password=myPassword

[mysqladmin]
user=root
password=myPassword
```

Make sure that the root user is the only one who can access it:

```
chmod 0600 .my.cnf
```

To make the changes happen, restart the MySQL server:

```
/etc/init.d/mysql restart
```

To get the web server working correctly you have to install apache2 and PHP 4+. We used PHP 5 instead of PHP 4 for our server. To allow the BOINC framework to run PHP scripts we also needed to install a PHP command line interpreter. The commands we used to install apache2 and PHP 5 are:

```
apt-get install apache2
apt-get install php5
apt-get install libapache2-mod-php5 php5-mysql php5-gd php5-cli
```

Before we start the installation of the BOINC server, we will create a user and a group under which the server tasks are running.

```
adduser boincadm
addgroup boinc
```

Now add this newly created user to the new group and because the clients access the server via the Web server, also add the user under which the Web server is running to this group, allowing the Web server to access the shared memory block.

```
adduser boincadm boinc
adduser www-data boinc
```

Set the default group for this user to boinc:

```
adduser -g boinc boincadm
```

We recommend to do all the following steps with the newly created user boincadm.

A.1.2 Setting up MySQL

To check that the MySQL server is running you only have to type

```
mysql -u root -p
```

in the command line. Enter the password you defined above and if you get a

```
MySQL ?
```

prompt you can proceed, otherwise you have to check your MySQL installation on this computer.

Now we have to add the Boinc and Apache user and define passwords for them. When you have the MySQL prompt type the following:

```
mysql> grant all on *.* to boincadm@localhost identified by 'myPassword';
mysql> grant all on *.* to boincadm identified by 'myPassword';
mysql> grant all on *.* to 'www-data'@localhost identified by
' myPassword';
mysql> grant all on *.* to 'www-data' identified by 'myPassword';
mysql> exit
```

A.2 Getting The Source Code

Normally you will have the source code in `/usr/local/src`. To get the source code execute the following command from that directory:

```
svn co http://boinc.berkeley.edu/svn/trunk/boinc
```

If you also want the samples from BOINC also execute the following command:

```
svn co http://boinc.berkeley.edu/svn/trunk/boinc_samples
```

To get the project running we need to modify some files before we compile the server part. First of all we need to put the source for our programs from the CD into the right location. We give a list of the location where each file should be copied to (the paths are relative to `/usr/local/src/boinc/`):

```
sched/hello_assimilator.C
sched/hello_validator.C
sched/hello_make_work.C
sched/ecc_curve.h
sched/checking_controller.C
sched/collision_finder.C
sched/db_purge.C
sched/create_checking_wu.C
sched/create_checking_wu.h
sched/distributed_checking_feeder.C
sched/file_constants.h
sched/handle_request.C
sched/result_checker.C
sched/result_checker.h
sched/sched_check.C
sched/sched_check.h
sched/sched_send.C
sched/sched_send.h
sched/Makefile.am
```

```

db/boinc_db.C
db/boinc_db.h
db/schema.sql
db/constraints.sql
html/inc/collision.inc
html/inc/db_ops.inc
html/ops/find_collision.php
lib/error_numbers.h
lib/str_util.C
py/BOINC/database.py
py/BOINC/setup_project.py
tools/make_project
tools/create_work

```

After that we have to modify the template for the Makefile. To do this replace the original `Makefile.am` by our modified version. Before replacing existing files with our modified versions check for differences between them. If parts in the original file are not found in our files, copy them to our modified file before replacing the original file. This can happen if parts on the framework are changed since the time we changed the files. Changes not colliding with our changes should be adopted. Also check the `Makefile.am` for changes. Another option is to copy our complete source to `/usr/local/src/boinc/` and then do an update. After the update you have to solve all the SVN conflicts. Now we are ready to compile and install the server.

To install the BOINC server you only have to execute the next 4 commands in `/usr/local/src/boinc/`:

```

./_autosetup
./configure --prefix=/usr/local/boinc --disable-client
make
make install

```

After this, the first steps are done.

A.3 Creating a Test for the BOINC Server

To test if the BOINC server is working as expected, you need to create a project. You can do this by using the “uppercase” application from the BOINC samples repository. The sources can be downloaded as explained above. After downloading it, you have to compile it for your platform and test it locally. When the application works as expected we go forward to the project creation. To create the sample application you have to use “uppercase” instead of “hello” in the commands of the following sections.

A.3.1 Creating the Hello Project

Normally the projects will be created in the folder `$home/projects/`. If you want them to be in another location you have to create a directory to use. We will use the default location. To create a new project execute these commands:

```
cd /user/local/src/boinc/tools/
./make_project -v --delete_prev_inst --drop_db_first
--db_user boincadm --db_passwd myPassword
--url_base http://URL/ hello Hello@Home
```

If you do not want to install to the default location you can use the

```
--project\_root path\_to\_install\_directory
```

option on the `make_project` command.

Answer Yes to all the questions and if everything worked you will find a new folder called `hello` in `$home/projects/`. It will be a good idea to define a `HELLO` variable to be the directory of the project. Using bash this is done like this:

```
export HELLO=/home/boincadm/projects/hello
```

A.3.2 Modifying the Preferences

Now its time to adjust the preferences for our project. This means changing the following things:

- **config.xml**

In this file you will have to modify a few things. Change the option `disable_account_creation` from 1 to 0. Also insert the XML element `<profile_screening/>` into the `<config>` element. The remaining changes will be explained later.

- **project.xml**

In this file you will have to update the `<app>` element, so it looks like the following:

```
<app>
  <name>hello</name>
  <user_friendly_name>Hello Project</user_friendly_name>
</app>
```

Without this element the application will not be distributed to the clients.

- **html/project/project.inc**

In this file you have to check that the URL of your server is correct. Normally it will be `http://server/hello`. `Server` is either the name or the IP of our server.

A.3.3 Adding the Application

The next step is to add the application that the BOINC client is going to use. For this, go to the project folder and execute the following statements:

```
mkdir -p hello/hello_1.00_windows_intelx86.exe
cp /path_to_application/hello_1.00_windows_intelx86.exe
$HELLO/apps/hello/hello_1.00_windows_intelx86.exe/
```

Then we have to sign every file in the folder `hello_1.000_windows_intelx86.exe`. It is important that the signature file has the same name as the original file plus the extension `.sig`.

Do this by executing the next two commands.

```
cd $HELLO
./crypt_prog -sign apps/hello/hello_1.00_windows_intelx86.exe/
hello_1.00_windows_intelx86.exe keys/code_sign_private > apps/
hello_1.00_windows_intelx86.exe/hello_1.00_windows_intelx86.exe.sig
```

Now we can add the project to the BOINC server:

```
cd $HELLO
bin/xadd
bin/update_versions
```

A.3.4 Creating the Work Units

For the creation of work units we need templates. Copy the templates from the CD into the following directory `$HELLO/templates`

If you want to create a single workunit by hand and not with the workunit creation program you can do this by executing the `create_work` command:

```
cd $U
bin/create_work -appname hello -wu_name helloWU
-wu-template templates/world_wu_mod.xml
-result_template templates/hello_re_mod.xml
```

Next we have to add a few daemons for creating work, checking the results of the BOINC clients and searching for collisions. This is done by adding the following lines at the end of the `<daemons>` element to the file `config.xml`.

```
<daemon>
  <cmd>
    hello_validator_with_checking -d 3 -app hello
  </cmd>
</daemon>
<daemon>
  <cmd>
    hello_assimilator_with_checking -d 3 -app hello
  </cmd>
</daemon>
<daemon>
  <cmd>
    hello_make_work -appname hello -d 3 -sleep_time x
    -nr_of_bits_p x -wu_name wu_name_template -wu_template
    templates/world_wu_mod.xml -file_counter_offset x
    -result_template templates/hello_re_mod.xml
    wu_filename_template
  </cmd>
</daemon>
<daemon>
  <cmd>
    collision_finder -d 2 -number_of_results_to_check x
  </cmd>
</daemon>
```

Remember to replace the placeholders “x” by some real numbers. The workunits will get a name of the form “wu_name.templateXX” where “XX” is a continuous counting number. The input files for the workunits will get a name of the same form:

“wu_filename_templateXX.txt”.

An overview of the different options possible in the `config.xml` file is given on the BOINC wiki [28]. The options for our programs are explained below:

distributed_checking_feeder The `distributed_checking_feeder` behaves the same as the original feeder. The options are explained on the BOINC wiki.

checking_controller The `checking_controller` command looks as follows:

```
checking_controller [arguments]
```

The mandatory arguments are listed in Table A.2, the optional ones in Table A.3.

Table A.2: Mandatory Arguments of `checking_controller`

Argument	Description
-appname name	Application name
-sleep_time time x	# of seconds to wait before restart
-selection_hours x	The last RPC call has to be at least x hours in the past
-groupsize_round_two x	The number of clients in a group of checking round two
-groupsize_round_three x	The number of clients in a group of checking round three

Table A.3: Optional Arguments of `checking_controller`

Argument	Description
-checking_bound	If # of clients > checking_bound then do distributed checking else check on server
-number_of_results_to_check	The number of results to check at once if the server is checking
-config_dir path	Path to <code>config.xml</code> , if it is not at the default location

collision_finder The `collision_finder` command looks as follows:

```
collision_finder [arguments]
```

The optional arguments are listed in Table A.4.

Table A.4: Optional Arguments of `collision_finder`

Argument	Description
-sleep_time time	# seconds to wait before restart
-collision_filename	The filename of the file where the collision will be written to. If no filename is given the default name “possible_solutions.txt” will be used.
-number_of_results_to_check	The number of results to check at once if the server is checking
-config_dir path	Path to <code>config.xml</code> , if it is not at the default location

hello_validator_with_checking The `hello_validator_with_checking` takes the same optional arguments as the `sample_bitwise_validator`, which are also explained on the BOINC wiki.

hello_assimilator_with_checking The `hello_assimilator_with_checking` takes the same optional arguments as the `sample_assimilator`. The details are explained on the BOINC wiki.

hello_make_work The `Hello_make_work` command looks as follows:

```
hello.make.work [arguments] wu.filename.template
```

Mandatory arguments are listed in Table A.5, optional ones in Table A.6. The job parameters, explained in Table A.7, may be passed in the input template, as arguments, or not passed at all (defaults will be used).

Table A.5: Mandatory Arguments of `hello_make_work`

Argument	Description
<code>-appname name</code>	Application name
<code>-wu_name name</code>	Workunit name
<code>-wu_template filename</code>	WU template filename relative to project root; usually in <code>templates/</code>
<code>-result_template filename</code>	Result template filename, relative to project root; usually in <code>templates/</code>
<code>-sleep_time time</code>	# of seconds to wait until the next WU is generated
<code>-nr_of_bits_p</code>	# of bits of p
<code>wu_filename_template</code>	The input files for the WU will get a name of the form: <code>"wu_filename_templateXX.txt"</code>

Table A.6: Optional Arguments of `hello_make_work`

Argument	Description
<code>-file_counter_offset</code>	Offset for filename and WU name generation, the default value is 1
<code>-batch n</code>	Limits size of database query
<code>-config_dir path</code>	Path to <code>config.xml</code> , if it is not at the default location

A.4 Final Steps

The last steps are now to start the server and test it. Starting is done by the following command:

```
cd $HELLO
bin/start
```

The commands for getting the status of the server and stopping it are:

```
cd $HELLO
bin/status
```

```
cd $HELLO
bin/stop
```

Testing is done by downloading a BOINC client, running it and attaching it to the project. You can get some information about the participants in http://server/hello_ops/. We recommend to restrict the access for ops to authenticated users with `htaccess`. First create the directory `$HELLO/usernames` and change the rights.

```
cd $HELLO
mkdir usernames
chmod 711 usernames
```

Go in the directory and give the boinc user access to the ops web page:

```
htpasswd -c .htpasswd boincadm
```

This will create a file `.htpasswd`. You can add or remove entries with `htpasswd .htpasswd <name>`. Nobody from outside the server should have access to this directory so create a file named `.htaccess`. This file should contain the following lines:

```
AuthGroupFile /dev/null
AuthName "DO NOT think to access this file"
AuthType Basic

deny from all
```

No let only users who are in the `.htpasswd` access the ops web page:

```
cd $HELLO/html/ops
joe .htaccess

AuthName "administration access"
AuthUserFile $HELLO/usernames/.htpasswd
AuthGroupFile /dev/null
AuthType Basic

<Limit GET POST>
require valid-user
</Limit>
```

Now you can remove the protection in `$HELLO/html/ops/cancel_wu_action.php` by inverting the following condition.

```
if (1) {
echo "
WARNING! Make sure the html/ops directory is password-protected,
then edit html/ops/cancel_wu_action.php by hand to remove
this message.
";
exit();
}
```

Now, you are ready to go!

Table A.7: job parameters of `hello_make_work`

Parameter	Description
<code>-command_line "-flags foo"</code>	The command-line arguments to be passed to the main program.
<code>-rsc_fpop_est x</code>	An estimate of the average number of floating-point operations required to complete the computation. This is used to estimate how long the computation will take on a given host.
<code>-rsc_fpop_bound x</code>	A bound on the number of floating-point operations required to complete the computation. If this bound is exceeded, the application will be aborted.
<code>-rsc_memory_bound x</code>	An estimate of application's largest working set size. The workunit will only be sent to hosts with at least this much available RAM.
<code>-rsc_disk_bound x</code>	A bound on the maximum disk space used by the application, including all input, temporary, and output files. The workunit will only be sent to hosts with at least this much available disk space. If this bound is exceeded, the application will be aborted.
<code>-delay_bound x</code>	An upper bound on the time (in seconds) between sending a result to a client and receiving a reply. The scheduler won't issue a result if the estimated completion time exceeds this. If the client doesn't respond within this interval, the server 'gives up' on the result and generates a new result, to be assigned to another client. Set this to several times the average execution time of a workunit on a typical PC. If you set it too low, BOINC may not be able to send some results, and the corresponding workunit will be flagged with an error. If you set it too high, there may be a corresponding delay in getting results back.
<code>-min_quorum x</code>	The minimum size of a 'quorum'. The validator is run when there are this many successful results. If a strict majority agrees, they are considered correct. Set this to two or more if you want redundant computing.
<code>-target_nresults x</code>	How many results to create initially. This must be at least min_quorum . It may be more, to reflect the ratio of result loss, or to get a quorum more quickly.
<code>-max_error_results x</code>	If the number of client error results exceeds this, the workunit is declared to have an error; no further results are issued, and the assimilator is triggered. This safeguards against workunits that cause the application to crash.
<code>-max_total_results x</code>	If the total number of results for this workunit exceeds this number, the workunit is declared to be in error. This safeguards against workunits that are never reported (e.g. because they crash the core client).
<code>-max_success_results x</code>	If the number of success results for this workunit exceeds this, and a consensus has not been reached, the workunit is declared to be in error. This safeguards against workunits that produce nondeterministic results.
<code>-additional_xml 'x'</code>	This can be used to supply, for example, <code><credit>12.4</credit></code>

Acknowledgements

First of all, I would like to thank Prof. Dr. Wattenhofer for allowing me to write this interesting thesis in his group. Furthermore, I want to thank my supervisors, Stefan Schmid and Michael Kuhn for helping and supporting me on this thesis. Their valuable feedback helped me whenever I was unsure of what was the best way to solve the upcoming problems.

List of Figures

3.1	Pollard Rho	12
3.2	Distributed Pollard Rho	14
4.1	Distributed Checking: Step 1 - 3	18
4.2	Distributed Checking Analysis Idea: Pairing Sequence	19
4.3	Simulation Results: Percentage Of Good, Mixed And Bad Groups (Observe Percentage Bias!)	22
4.4	Simulation Results For Different Percentages Of Cheaters (Number Of Clients: 10,000)	23
5.1	Interaction Between Server And Client	25
5.2	BOINC System Architecture	27
5.3	BOINC Standard Data Flow	28
5.4	New BOINC Data Flow	29
5.5	BOINC Time Line	32
5.6	Schematic Overview Of A Checking Round	33
6.1	State Machine Of The Checking Controller	39

List of Tables

4.1	Distributed checking simulation results ($n = 1,000,000$, $m = 100,000$ and group sizes: 2-3-6)	21
A.1	Additional packages	43
A.2	Mandatory Arguments of <code>checking_controller</code>	49
A.3	Optional Arguments of <code>checking_controller</code>	49
A.4	Optional Arguments of <code>collision_finder</code>	49
A.5	Mandatory Arguments of <code>hello_make_work</code>	50
A.6	Optional Arguments of <code>hello_make_work</code>	50
A.7	job parameters of <code>hello_make_work</code>	52

Bibliography

- [1] Assault on 13th Labour. <http://www.13thlabour.tk/>.
- [2] Alchemi - .NET Grid Computing Framework. <http://www.alchemi.net/>.
- [3] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing.*, pages 4–10. IEEE Computer Society, 2004.
- [4] Baby-step giant-step - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Baby-step_giant-step.
- [5] BBC Climate Change Experiment. <http://bbc.cpdn.org/>.
- [6] Richard Beigel, Grigori Margulis, and Daniel A. Spielman. Fault diagnosis in a small constant number of parallel testing rounds. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 21–29, New York, NY, USA, 1993. ACM Press.
- [7] BOINC - Berkeley Open Infrastructure for Network Computing. <http://boinc.berkeley.edu/>.
- [8] Crypto++ Library - a Free C++ Class Library of Cryptographic Schemes. <http://www.cryptopp.com/>.
- [9] distributed.net. <http://www.distributed.net/>.
- [10] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *The 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 4–11, Tokyo, Japan, March 23–26 2004.
- [11] Wenliang Du and Michael T. Goodrich. Searching for high-value rare events with uncheatable grid computing. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 122–137, 2005.
- [12] ECC Challenge. http://www.certicom.com/index.php?action=ecc,ecc_challenge.
- [13] ECC Challenge details. http://www.certicom.com/download/aid-111/cert_ecc_challenge.pdf.
- [14] GIMPS. The Great Internet Mersenne Prime Search. <http://www.mersenne.org/prime.htm>.

- [15] GNU General Public License.
- [16] GNU Lesser General Public License. <http://www.gnu.org/licenses/lgpl.html>.
- [17] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. *Lecture Notes in Computer Science*, 2020:425–440, 2001.
- [18] Philippe Golle and Stuart G. Stubblebine. Secure distributed computing in a commercial environment. In Paul F. Syverson, editor, *Financial Cryptography*, volume 2339 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2001.
- [19] Hash Clash. <http://boinc.banaan.org/hashclash/>.
- [20] Leander Kahney. Cheaters bow to peer pressure. *Wired Magazine*, 2 2001.
- [21] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .net-based enterprise grid computing system. In Hamid R. Arabnia and Rose Joshua, editors, *International Conference on Internet Computing*, pages 269–278. CSREA Press, 2005.
- [22] M4 Message Breaking Project. http://www.bytereef.org/m4_project.html.
- [23] Mersenne prime - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mersenne_prime.
- [24] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math.Comp*, 32(143):918–924, July 1978.
- [25] SETI@home. <http://setiathome.berkeley.edu/>.
- [26] Doug Szajda, Barry G. Lawson, and Jason Owen. Hardening functions for large scale distributed computations. In *IEEE Symposium on Security and Privacy*, pages 216–224. IEEE Computer Society, 2003.
- [27] The Globus Alliance. <http://www.globus.org/>.
- [28] The project configuration file. <http://boinc.berkeley.edu/trac/wiki/ProjectConfigFile>.
- [29] Ubuntu 6.10. <http://www.ubuntu.com/>.
- [30] XtremWeb - A Global Computing Experimental Platform. www.xtremweb.net.