Master Thesis

Secure and convenient distributed computing

Victor Budilivschi

March 10, 2011

Advisors: Prof. Dr. Srdjan Čapkun, Ghassan O. Karame, Dr. Aurélien Francillon
Department of Computer Science, ETH Zürich
Abstract

The recent years have witnessed the increased development of distributed computing platforms that use the idle computing power of the volunteer computers to perform computationally expensive tasks on behalf of academic research groups, industry labs or even individual clients. The distributed computing projects try to solve problems in various areas like biology, medicine, earth sciences, mathematics, physics, astronomy etc. Some existing distributed computing platforms, by hosting several projects, offer the possibility for people to join them. The problems of increasing the number of participating volunteers and ensuring that the gathered results are correct are gaining increasing attention nowadays. For instance, for inexperienced users it is not always easy to install or uninstall software needed to participate in a distributed computing project.

We propose a solution that offers the possibility to make computations using web-based clients and we study its feasibility. We analyse and test the existing technologies that would allow making the computations in browsers efficiently. We analyse the possibility of interconnecting our platform with existing distributed computing frameworks. In order to make our framework scalable, we design it to have an efficient way of extending it. We use a secure scheme that offers a mechanism for validation of the results with small overhead. We show that making computations in web-based clients is not only possible, but also efficient, without disturbing the browsing experience of users.
Contents

Table of Contents

List of Figures

1. Introduction 1

2. Background and Related Work 3
   2.1. Background: Distributed Computing Projects ................. 3
   2.2. Related work .............................................. 4

3. System Architecture 6
   3.1. Overview .................................................... 6
   3.2. Security Considerations ..................................... 9
   3.3. Server Architecture (Nest) ................................ 11
   3.4. Client Architecture (Pafi) ................................ 12
   3.5. Content Provider Library ................................... 14

4. Implementation Setup 15
   4.1. Evaluation of different technologies for client-side .......... 15
       4.1.1. JavaScript .............................................. 15
       4.1.2. Google Web Toolkit ..................................... 16
       4.1.3. Java Applet ............................................. 17
       4.1.4. Silverlight ............................................. 17
       4.1.5. Adobe Flash ............................................ 17
       4.1.6. Google Native Client ................................... 18
   4.2. Performance comparison of several technologies ............. 19
   4.3. Client-side Implementation .................................. 21
       4.3.1. Structure .............................................. 21
4.3.2. Initialization of Pafi .................................. 22
4.3.3. Communication between Pafi and Pafi Workers .............. 25
4.3.4. Requesting work units ................................ 26
4.3.5. Starting workers .................................. 26
4.3.6. CPU usage throttling ................................ 27
4.3.7. Receiving results from workers and sending them to the server .. 28
4.3.8. Handling lost connections ............................ 28
4.4. Server-side Implementation ................................ 30
  4.4.1. Nest ............................................ 30
  4.4.2. Rooms ......................................... 31
4.5. Integration of Other Projects ................................ 31
  4.5.1. Server-Side ...................................... 32
  4.5.2. Client-Side ...................................... 32

5. System Evaluation ....................................... 34
  5.1. Prototype Evaluation .................................. 36
    5.1.1. Prototype version 1.0 ............................ 36
    5.1.2. Prototype version 2.0 ............................ 37
  5.2. Limitations of the Prototype .......................... 38
  5.3. Future work ....................................... 39
  5.4. Applications ....................................... 40

6. Concluding Remarks ..................................... 41

Bibliography ............................................... 43

A. Source code ............................................. 47
  A.1. RC4 implementation (JavaScript) ......................... 47
  A.2. RoomController class (Java) .......................... 48
  A.3. ClientWork class (Java) .............................. 49
  A.4. PafiThread class (Java) ............................. 50
## List of Figures

3.1. General architecture of the framework .................................. 7  
3.2. Overview of the Server Side (Nest) structure .......................... 11  
3.3. Overview of the Client Side (Pafi) structure .......................... 13  
4.1. Class diagram of \textit{Pafi} ........................................... 21  
4.2. Class diagram of \textit{Pafi Applet} .................................. 22  
4.3. Class diagram of \textit{Pafi Silverlight} ................................ 23  
4.4. Package diagram of \textit{PafiApplet} with \textit{RC4Worker} .......... 24  
4.5. Package diagram of \textit{PafiSilverlight} with \textit{RC4Worker} ....... 25  
4.6. Package diagram of \textit{Nest} ....................................... 30  
4.7. Class diagram of \textit{Nest} .......................................... 33  
5.1. Prototype screen shot: First visit on the page ......................... 35  
5.2. Prototype screen shot: Viewing and downloading an article ........ 35  
5.3. Prototype screen shot: Insufficient credit ............................ 36
Chapter 1

Introduction

The recent years have witnessed the increased development of distributed computing platforms that use the idle computing power of the volunteer computers to perform computationally expensive tasks on behalf of academic research groups, industry labs or even individual clients [54]. Some of the platforms like BOINC [30] are open source, and anybody can use them to start their own distributed computing project.

The distributed computing projects try to solve problems in various areas like biology, medicine, earth sciences, mathematics, physics, astronomy etc. Some existing distributed computing platforms, by hosting several projects, offer the possibility for people to join them. One example, World Community Grid is the largest non-profit computing grid that offers to solve problems that benefit the humanity [43].

The distributed computing platforms provide software for the users who want to participate. These are applications compiled for different platforms that need to be installed on their machines. When joining a project, a user has to first download the core client and install it on his system. Another option would be to download the source code of the client software and compile it.

One of the challenges that the distributed computing projects are facing is the limited exposure they acquire online. More specifically, the number of volunteers who participate is still modest. Although the results are considerable [44, 5], there is still room for improvement. Most of the projects try to provide clients for as many platforms as possible in order to give the possibility of joining the project to any volunteer. Also, by releasing the source code of the client software, there is a possibility for the community to modify and build the software for a variety of operating systems and hardware. Even when the client software is available, volunteers must install and configure it on their systems. In many cases this is not a straightforward task for an unsophisticated user. Also, if a participant is not willing to contribute anymore, he has to uninstall software
from his system. On several platforms a user (depending on his access level) might not even have credentials for this kind of operations.

Another challenge of the distributed projects is the evaluation of the correctness of the returned results from clients. Given the nature of the distributed computing projects, the participants could return incorrect results intentionally or unintentionally. The possible incorrect results returned by some participants might compromise the results of the whole project.

In this thesis, we address these challenges and we explore the possibility of performing remote computations using existing software that is already installed on the machines of the volunteers. This removes the problem of software installation on client systems, thus making easier to increase the number of participants. More precisely, we study the feasibility of executing computations in web browsers on client machines. We analyze several technologies to identify the ones suitable for making computations. We build a framework that allows making remote computations in web browsers on client machines, using several different technologies like JavaScript, Silverlight or Java applets. The design of our framework allows interconnection with content providers and existing distributed computing projects. The content providers can use the framework to “charge” users for viewing content – by making them execute computations – instead of displaying advertising on web pages, as described in [53]. For validation of the computation results, our projects is using precomputed checks or ringers as described in [54] and [51]. We also conduct some user studies to observe the reception of our scheme.

The remainder of this thesis is organized as follows. In Chapter 2 we overview the related work on distributed computing projects. The general architecture of our framework and the security considerations are contained in Chapter 3. In Chapter 4 we describe in detail the implementation of the server-side, client-side and the interconnection of the framework with current distributed computing projects and content providers. The evaluation of our platform and its limitations are presented in Chapter 5. Also in Chapter 5 we discuss about possible applications.
2.1. Background: Distributed Computing Projects

There are millions of computers connected to the Internet that participate in various distributed computing projects [5, 44]. These distributed computing projects use the idle time of volunteer computers for executing computationally expensive tasks. These projects try to solve problems in various areas like biology, medicine, earth sciences, mathematics, physics, astronomy etc. Some existing distributed computing platforms, by hosting several projects, offer the possibility for people to join them. One example, World Community Grid is the largest non-profit computing grid that offers to solve problems that benefit the humanity [43]. As of March 2011, it has more than 550 thousands of registered users, and uses the computing power of more than 1.7 million devices. The aggregated running time of the whole grid is more than 450 thousand years [44]. The most widely used framework for volunteer distributed computing is Berkley Open Infrastructure for Network Computing (BOINC) [6]. The total computing power of all projects that use BOINC is around 5.6 PFLOPS (March 2011) [5].

The distributed computing projects offer client software for the users who want to participate. These clients represent applications compiled for different platforms that need to be installed on the machines. For some client applications, the source code is also available. When joining a project, a user has to first download the core client and install it on his system. Another option would be to download the source code and compile it. In the BOINC framework for example, after the core client is installed and started, the user is able to choose the project he wants to join. When the project is chosen, the core client will request the binary files needed for the specific project. In case the binaries for that specific platform are available, they will be provided. These binary files are later executed on the machine of the volunteer by the core client. During its work on the client machine, these programs will constantly request work units form
the server, make computations and report back the results of the computations to the
server. The binary files have access to the local file system where these can store files
that contain different data needed for computations.

The project-specific clients very often make use of specific CPU or GPU architectures.
Because of this, the server must keep binary files for as many platforms as possible.
Using this approach is very effective because the applications can be very efficient in
making computations. The applications also have access to the operating system APIs.
For example, this is helpful for CPU usage throttling, when the application needs to
know the current CPU load to avoid excessive use of resources.

But it also has some disadvantages. For unsophisticated computer users is not straight-
forward how to download, install/compile applications on their systems. In some situ-
ations the users are not able to install any program on their machines because they do
not have the credentials for making such operations. It also poses a security threat when
people, instead of choosing the correct programs, might download and install malware.

A volunteer computing projects can have issues with the validation of the returned re-
sults. Participants could return bogus results intentionally – to get more credit, or
unintentionally – because of some software or hardware errors [3]. The BOINC platform
has some mechanisms for validating the results by relying on job replication [4, 2], that
is sending the same work units to more than one participant and upon the receipt, it
compares the returned results. If a consensus is reached, the result is validated.

2.2. Related work

In the literature there are several contributions that address the security of outsourced
computations in distributed systems. One of these, Gore et al. [51] offers a solution to
secure a class of parallel problems that are executed on remote machines: inverse one-
way functions, where the participants are required to compute the pre-images of some
one-way functions. A solution for the problem of securing remote sequential computa-
tions is given by Karame et al. [54] and Szajda et al. [57].

Distributed computing implementation that cracks DES using web browsers was pro-
posed in [7]. The author built the system as a proof of concept, to show that distribute
computing in browsers is possible using JavaScript. The DES cracker tries to find the
key that was used to encrypt a block of plaintext into corresponding cyphertext. The
security of the results was not addressed in the implementation. The client is fully
supported on Firefox and having different issues with other browsers. The CPU usage
throttling, was implemented with an user definable amount of time to wait (through a
drop-down list) after a block of 128 keys are checked. This basically only allows to make
sure that the browser does not freeze during the computations.

Horton et al. [52] show that Java applets in web-browsers can be used to perform
covet distributed computations without the knowledge of users. The authors claim
that computations can be executed on computers such that the users are not able to
notice that this is done. This can be accomplished by periodically putting the working thread to sleep for an amount of time, thus avoiding excessive use of resources. The idea of users paying for accessing services or content on the web through the use of their idle computer time was also introduced by Horton et al. [52]. The authors claim that the content or service providers could sell the CPU resources in the same way as advertising is sold. Barabasi et al. [46] show that the TCP protocol can be exploited and hosts in the Internet can be transformed into a distributed computer that can make computations on behalf of remote nodes. They analyse the possibility of a machine forcing other nodes to solve complex computation problems, that is “parasitic computing”.

In the work of Provos et al. [56] threats originating from scripts that run in the browsers are analyzed. They show that using such a powerful scripting language like JavaScript is possible to determine the exact vulnerabilities (e.g. by checking the Java version) a system has and to exploit them from a central server.

Remote computations as a way of combating spam are proposed by Goel et al. [50] and Hashcash [20]. The former work introduces a concept of certified micro-donations as a way of combating spam. Hashcash is a tool for counter-measure denial-of-service attacks. The main use is to avoid losing emails due to content based and blacklist based anti-spam systems. It uses a concept of a hascash stamp which is a proof that the sender made some computations. The recipient can easily verify whether the sender made the computations it claims.
Chapter 3
System Architecture

3.1. Overview

Our goal is to construct a framework that is not only efficient when making computations, but also easy to integrate with current distributed projects and content providers. One way of implementing the web page viewing is to split the content needed by the user into parts, and “charge” the user one unit of credit for viewing each piece of content. For example, if a user wants to read an article, then he would need one unit of credit for one page he wants to read. The credit is obtained by making computations. The client obtains credit by completing computation bundles on his machine. While viewing the web page, the browser will make the computations thus increasing the balance of the user. The amount of computations needed to get one unit of credit can be adjusted depending on the capability of the user machine (ex: smaller work units for mobile phones).

In figure 3.1 is presented an overview of the framework structure. In what follows, we explain all the components and steps in detail.

The Computation Server manages many distributed computing projects. It could be any existing platforms like BOINC [30]. Any of these servers has an interface for clients to connect to them, request work units, make computations and return results.

The Intermediary computation server acts as a client for the Computation Server, manages the outsourcing of computations to the clients. It splits the work units into smaller parts, inserts ringers, distributes them to connected web browsers and aggregates the results.

The Content providers are websites that provide content to clients. It could be for example an on-line library that allows users to read books, articles, news, or watch movies, pictures etc.
3.1. OVERVIEW

The Clients. The clients represent browsers that request content from the content providers. These are the clients that would run computations in exchange for the content offered by the Content provider.

The scheme description (Figure 3.1):

1 **Client requests content.** The Client (C) contacts CP requesting some content from it. For example, this could be a page of an article. In the request, the browser will embed the Session ID (SID). If C does not have an SID, it will pass null as session identifier, meaning this is its first request.

2 **Request client credit.** The CP will ask the Intermediary Computation Server (ICS) the amount of credit C has based on the SID provided by the browser.

3 **Reply with client credit.** The ICS will return the amount of credit C has. Based on this amount, CP will decide what to return back to C, the content, or a message to inform that C has to first make some computations to be able to receive content.

![Diagram](image)

Figure 3.1.: General architecture of the framework
CHAPTER 3. SYSTEM ARCHITECTURE

4 Reply to the client. When $C$ makes the first request for content to $CP$, the later will generate an $SID$, build a `<script>` tag with an address pointing to a script on $ICS$ and return it. The browser will load the script as specified in the reply. If this is not the first request from $C$, then it will receive the content based on the credit it has with $ICS$.

5 Request code from $ICS$. Based on the reply from $CP$, if this is the first request, $C$ will have to load a script from $ICS$, and to run it.

6 Reply with script from $ICS$. $ICS$ will return the script that is needed to be executed by the browser. This script will load the core client and prepare it for running computations.

7 Request work unit $i$. After the core client is loaded in the browser and is ready, it will request work units from $ICS$ based on the client browser capabilities.

8 Return work unit $i$. Based on the client browser capabilities, $ICS$ will return a work unit to the client.

9 Client makes computations on work unit $i$. After receiving a work unit, the client will start the necessary thread in the browser to make computations according to the information in the work unit metadata.

10 Return computation results for work unit $i$. When the client finishes the computations, it will return the results to $ICS$. If the result is correct, the server will increment the amount of credit the client has.

The communication between $ICS$, $CS$ and their functions:

A Request work unit from Computations Server. While the $ICS$ can also host its own projects, it can as well to request work units from other Computations Servers($CS$) that host many projects, thus contributing to existing projects.

B Receive work unit from $CS$. Usually, the work units given by $CS$ are intended to be completed in relatively big time intervals (ex: 24 hours [8]). Because we want the client to complete the computations faster, the big work unit received from $CS$, is split into smaller work units suitable for web browser-based clients.

C Process incoming requests. The $ICS$ is handling the requests between $CS$, $CP$ and clients. It keeps track of the active sessions and the amount of computations completed.

D Return results to $CS$. When all the small work units are completed, the $ICS$ assimilates the results and return them to $CS$. 

8
3.2. Security Considerations

A distributed computing framework may have issues with the validation of the results that come from the clients. In order to ensure that the results of the computations are correct, some security mechanisms are implemented at different levels.

Some clients might want to gain more credit by making the computation once and resending the result of the same work unit to the server several times. In a group of clients, these might share the results of their own computations, such that all of them resubmit the same results and gain undeserved credit. Malicious users may also want to get credit by generating fake results for fake work units and send them to ICS.

In order to protect the framework from the above-mentioned security issues, the server keeps track of all the information regarding work units sent to the clients. When a result is received, all the fields are checked to identify if the respective work unit was indeed sent by the server to the client who replied and if the work unit is not expired. If no problem is detected, then the computation results are passed for validation to the next layer (RoomController). Otherwise, if the client is trying to cheat, subsequent collaboration with him is stopped.

**Verifying the remote execution of parallel computations**

Parallel computations consist of the evaluation of a function or algorithm \( f : D \to R \) for every input value \( x \in D \). The work units are created by splitting \( D \) into subsets \( D_i \). For evaluating the results of remote computations, the most efficient solution is to rely on selective redundancy or to selectively embed indistinguishable precomputed checks (ringers [51]) within the work units for the clients [53].

The server chooses \( n \) uniformly distributed random values \( r_1, r_2, \ldots, r_n \) from \( D_i \). It computes the set \( S = \{ f(r_1), f(r_2), \ldots, f(r_n) \} \) (if these calculations require too much computation power, it will use as ringers results reported by other clients). After receiving the replies from clients, the computation results will be considered valid if and only if \( \forall r_i \in D_i, f(r_i) \) is correctly computed. Since the clients cannot distinguish the ringers from other data values in \( D_i \), and do not know how many ringers are present, they will have to make all the computations for their results to be accepted by the server [53]. This mechanism allows the validation of the results with a small overhead.

The probability of detecting malicious behaviour (Table 3.1) is given by the formula \( P = 1 - (1 - P_{C})^n \) where:

- \( P_{C} \) - fraction of incorrect inputs returned by participants
- \( n \) - number of ringers per work unit

The generation of ringers, and insertion into the Work Unit is performed by the Room Controller as this task is project specific.
Verifying the remote execution of sequential computations

For validating results of sequential computations ringers can also be used. However, ringers can only be efficiently used when several sequential tasks are permuted together and outsourced to users [54].

In what follows, we describe the scheme in [54], that allows securing the remote execution of $N$ distinct and independent tasks on the remote machines of $Q$ ($Q \geq N$) different users.

The ICS divides each task into smaller tasks. The sever then sends the $N$ tasks to $Q$ different users in $M$ consecutive rounds. In round $i$, ICS according to some probability, will try to verify the credibility of the user. ICS inserts “security checks” within the computations; in other words, it assigns a task whose result is already known to it. The clients do not know what kind of tasks they are running. Round $i$ ends when all $N$ users are assigned a job. In this way, the ICS can check several clients in each round [53].

At the beginning of round $i+1$, ICS collects the results and checks the correctness. If the results are right, ICS re-permutes the next logical sub-tasks among the users while using the corresponding output of the last round as inputs to the sub-tasks for the current round [53].

The detection probability of participant misbehavior is given by the formula $P = 1 - (1 - PC(P_R + PP(1 - PM)))^M$, where:

- $PC$ - probability of malicious participant cheating
- $P_R$ - probability that the server inserts an indistinguishable check per subtask
- $PP$ - probability of assigning the same subtask to another participant
- $PM$ - fraction of malicious participants

The probability of detecting malicious participants for several different $PC$, $P_R$, $PM$ and $PP$ are presented in Table 3.2.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$PC$</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>0.9375</td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
<td>0.99609</td>
</tr>
<tr>
<td>16</td>
<td>0.5</td>
<td>0.999985</td>
</tr>
</tbody>
</table>

Table 3.1.: Probability of detecting malicious participants in parallel tasks with respect to different input parameters. $n$ - number of ringers per computation bundle, $PC$ - probability of cheating, $P$ - detection probability. (Adopted from [51, 53])
3.3. Server Architecture (Nest)

An overview of the server side (Nest) structure of the framework is presented in Figure 3.2.

Table 3.2.: Probability of detecting malicious participants in sequential tasks with respect to different input parameters. $M$ - number of subtasks per task (adopted from [54]).

<table>
<thead>
<tr>
<th>$M$</th>
<th>$P_C$</th>
<th>$P_R$</th>
<th>$P_P$</th>
<th>$P_M$</th>
<th>Overhead</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.5</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>30%</td>
<td>0.77869</td>
</tr>
<tr>
<td>10</td>
<td>0.7</td>
<td>0.4</td>
<td>0.2</td>
<td>0.2</td>
<td>60%</td>
<td>0.993097</td>
</tr>
<tr>
<td>20</td>
<td>0.5</td>
<td>0.15</td>
<td>0.15</td>
<td>0.2</td>
<td>30%</td>
<td>0.945005</td>
</tr>
<tr>
<td>20</td>
<td>0.7</td>
<td>0.15</td>
<td>0.15</td>
<td>0.2</td>
<td>30%</td>
<td>0.98485</td>
</tr>
<tr>
<td>40</td>
<td>0.5</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>40%</td>
<td>0.999643</td>
</tr>
<tr>
<td>40</td>
<td>0.7</td>
<td>0.25</td>
<td>0.25</td>
<td>0.4</td>
<td>50%</td>
<td>0.999998</td>
</tr>
<tr>
<td>60</td>
<td>0.5</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>40%</td>
<td>0.999993</td>
</tr>
<tr>
<td>60</td>
<td>0.7</td>
<td>0.1</td>
<td>0.3</td>
<td>0.4</td>
<td>40%</td>
<td>0.999998</td>
</tr>
</tbody>
</table>

Figure 3.2.: Overview of the Server Side (Nest) structure
Because in this framework there are multiple levels of the server-side controller and applications, we named them with the analogy of an ant nest as described below.

**Main server controller (Nest Controller).** The Nest Controller is the main controller of the server side framework. It is the main point where the Clients and Content Providers interact with the server side. It keeps track of the sessions, the credit associated with these sessions, generates and assimilates work units. It provides security mechanisms to counteract intentional or unintentional cheating from clients (the security methods are discussed in 3.2).

**Project Controller (Room Controller).** Every Room controller is a controller for a computation project (Room) like [32] or [17]. The Room controller is responsible for work unit generation, assimilation and validation. It also secures the computation as described in 3.2 and provides the applications for different platforms. Every Room Controller has a working folder that can be used at its discretion during the execution.

**Application (Client Worker).** An application is a set of .class, .dll, .js, etc., files that represent the software that will be executed on the client depending on the technology supported by the browser (Java applets, Silverlight, etc.). The prototype currently supports JavaScript, Java applets and Silverlight applications. But it is easily extensible, and other technologies could be added, like ActionScript, Google Native Client, etc.

**Work unit.** A work unit sent to the client encapsulates metadata and data. The metadata contains information needed by the core client to know how to process the work unit. The data is project specific and consists of concrete data needed for computations by the Client Worker. Structure of a Work Unit:

- **Metadata**
  - ID - the identification number of the work unit.
  - AppInfo = Application information needed by the core client to correctly process this work unit.

- **Data**
  - Project specific data.

### 3.4. Client Architecture (Pafi)

A general overview of the Core Client (Pafi) is depicted in Figure 3.3. This pafi_loader.js script provides functionality to load the Pafi client into a web browser. It is embedded into the Content Providers’s web page.
Core Client (Pafi)

*Pafi* is the core client that is started in the web browser. It is loaded inside an `<iframe>` tag constructed by the `pafi_loader.js` script. Its main functions are:

- Handling communications between client and server. All the communication with the server is handled by *Pafi*.
- Unpacking work units and packing results to be sent to server
- Handling lost connections. When the connection to the server is lost, it will collect the results from the threads and will wait until the connection is reestablished, to send them to the *Nest*.
- Check browser capability. It will check the browser’s capability of making computations with different platforms (Applet, Silverlight, etc.) and report them back to the Server.
- Prepare frameworks (Applet, Silverlight, JavaScript etc.) for thread execution
- Create threads
- Start threads
- Stop threads
- Pause/Resume threads
- Destroy threads

Figure 3.3.: Overview of the Client Side (Pafi) structure
• CPU usage throttle. It will make sure that all the threads do not use too much CPU, and will balance the CPU usage during the execution of the threads.

Pafi is able to run multiple frameworks (Java Applets, Silverlight, etc.). For each of these frameworks is able to start multiple Room Client Workers for different Rooms.

3.5. Content Provider Library

A CP must be able to communicate with the ICS through an interface. For easing this communication a library for the content providers was developed. It provides interfaces for connection to the Nest and functions for Session ID generation. We assume a secure channel between the content provider and the intermediary computations server. Before sending content to the clients, CP can check their balance in order to provide the content or inform them that they should make some more computations.

The main function of the provided library are:

• Check connection with ICS
• Get client credit from ICS based on the Session ID
• Request from ICS to decrement credit for a specific client in return for content
• Generate Session ID
• Register the Session ID with ICS
Chapter 4
Implementation Setup

4.1. Evaluation of different technologies for client-side

One important challenge for executing remote computation in browsers is choosing the right technology. It should not only be supported by most of the browsers, but it also has to be efficient when making computations. Browsers support different frameworks that are able to execute programs on the client machine. Usually, these frameworks are used to create Rich Internet Applications (RIA) and to provide a more interactive interface except of static HTML. There is no universally adopted framework that is guaranteed it is available and enabled in the browsers. JavaScript is one that is supported by most major browsers. Other frameworks like Java Applets or Silverlight can be made available to the browsers by installing the respective plug-ins. Several technologies are tested for this project.

4.1.1. JavaScript

JavaScript, also known as ECMAScript [9] is a prototype-based object-oriented [11] scripting language that is dynamic, weakly typed and has first-class functions. It is also considered a functional programming language like Scheme and OCaml because it has closures and supports higher-order functions.

JavaScript is primarily used in the form of client-side JavaScript engine which is implemented as part of a web browser. It is used to provide enhanced user interface. It allows programmatic access to computational objects within the browser.

The JavaScript syntax is similar to that of C. Some names and naming conventions are the same as in Java. The key design principles are taken from the Self and Scheme programming languages [10].

JavaScript supports all the structured programming syntax (for example if statements, while loops, switch statements, etc.). There is a distinction between expressions and statements in JavaScript.
CHAPTER 4. IMPLEMENTATION SETUP

The same as most scripting languages, types are associated with values and not with variables. For example, a variable `s` can hold an integer value at some point in time, and later a string can be assigned to it.

The objects in JavaScript are associative arrays. The properties and their values can be added, changed or deleted at run-time. It is possible to execute statements as strings with the function `eval`. Functions are objects themselves, thus can be assigned to variables, passed as arguments, returned by other functions. The nested functions defined within another function are created whenever the outer function is called.

As JavaScript is using prototypes instead of classes for inheritance, many class-based features can be simulated with this.

The support for JavaScript in browsers is given by their JavaScript interpreter, that interprets the code and executes it.

The way the JavaScript engine is implemented in browsers can vary from one vendor to another and some browsers do not even have support for it, or the implementation is considerably different than standards. JavaScript support might not be enabled for security reasons as well, users can disable it in the configuration of the browser. For making JavaScript code compatible cross-browser, one can use available libraries like jQuery when accessing the Document Object Model tree or making Ajax requests.

The statistics show that 93%-95% of the browsers have the JavaScript engine enabled ([35], [40]). This makes JavaScript a very good choice for the client side computations. The main drawbacks are that this is an interpreted language (thus slower than compiled), and no possibility for multi-threading. There is also no possibility of using the local file storage on the client machine.

4.1.2. Google Web Toolkit

One very convenient technology is Google Web Toolkit (GWT) [15]. GWT is an open source set of tools that allows developers to create and maintain complex JavaScript front-end applications in Java [16]. With GWT it is possible to automatically compile Java code to highly optimized JavaScript code that runs across all browsers, including mobile browsers for Android or iPhone [16].

GWT supports most of the core Java language syntax and semantics, but there are some differences. Primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`), `Object`, `String`, arrays, user-defined classes are supported. In JavaScript, the only available numeric type is 64-bit floating point value. All Java primitive types are implemented (when compiled to JavaScript) on doubles. Because there is no 64-bit integral type in JavaScript it is implemented as a pair of two 32-bit integers. This might result in performance degradation when this data type is used a lot. Even though in Java multi-threading and synchronization is possible, the JavaScript interpreters are single-threaded, thus no multi-threading support [18].

Because of the limitations of JavaScript, GWT can only support a subset of the classes available in the Java 2 Standard and Enterprise Edition libraries and GWT has also no access to the local file system [19].

The benefits of using GWT are that a lot of distributed computing code can be reused
and Java developers do not have to adapt to writing cross-browser compatible JavaScript code.

### 4.1.3. Java Applet

Java Applet is a class of Java program that browsers enabled with Java technology can download and run using the Java Virtual Machine. An applet must be a subclass of the `java.applet.Applet` class. The interface between the Java applet and the browser is provided by the `Applet` class [23]. Java applets are generally written in Java. It is also possible to produce applets using other programming languages like Jython [25] or JRuby [49].

Java Applets are executed by browsers using a sandbox model, preventing them from accessing local data like clipboard or the file system. The code of the applet (.class or .jar files) is downloaded from the server by the browser and it embeds the applet into the page. In case the applets are unsigned, they cannot communicate with domains other than the one from where they are loaded. However, a signed applet has access to the local file system and can communicate with any domains [21]. To sign an applet a digital signature should be attached to it. This signature should indicate who is the creator of the applet and specify a local security policy that indicates the required access to the local system resources.

The applet can be embedded in a web page by using several different tags. Only the `applet` tag is currently consistent across browsers [39].

Because Java applets are delivered to the web browser as Java bytecode, and run in the same manner as any other Java program, the execution speed should be the same as for any other Java application. Applets also support multi-threading which is also a very good advantage compared to JavaScript.

### 4.1.4. Silverlight

Silverlight is a cross-browser cross-platform framework ([33]) that allows the creation of rich media applications and business applications for the Web desktop and mobile devices [42]. Silverlight applications can be created using several languages like C#, VisualBasic or IronPython. Silverlight applications represent .NET Assemblies that contain code in Common Intermediate Language.

The Silverlight support for browsers (see Table 4.1) running on Windows and Mac OS, represents a free plug-in powered by .NET framework that is compatible across multiple browsers.

Under Linux it is supported with the Moonlight browser plug-in [26]. Silverlight, besides using compiled assemblies, it supports also multi-threading which means it is possible to create high performance applications.

### 4.1.5. Adobe Flash

Adobe Flash is a multimedia platform. It is mainly used to add animation, video (for example YouTube.com) and interactivity to web pages. Flash files are delivered to the
browser in SWF format. For executing SWF files, Adobe is providing plug-ins for browsers such as Adobe Flash Player for many operating systems and mobile devices. The applications are written in ActionScript. To run the compiled bytecode from the SWF files, the plug-in has to first extract it and compile it to native code. That is, Adobe Flash is using just-in-time compilation [55].

ActionScript is a dialect of ECMAScript, thus sharing the same syntax and semantics. A free tool, part of the Adobe Flex can be used to compile ActionScript code [1]. ActionScript is object-oriented. Currently, the latest version of ActionScript is 3.0. ActionScript 3.0 consists of primitive data types which are used to create other data types. ActionScript 3.0 has support for signed (int) and unsigned integers (uint), floating-point 64-bit long (Number) types. The support for strings is given by the String data type that stores sequences of 16-bit Unicode characters using UTF-16 format.

According to Adobe Flash Player Statistics [45], Flash Player is installed in 99% of browsers.

4.1.6. Google Native Client

Native Client is a sandboxing technology for safe execution of platform-independent untrusted native code in a web browser. This technology is designed to build applications that are compute-intensive and/or interactive (e.g., games, media, large data analysis, visualizations) and make use of the client machine resources, in the same time running in a secure environment with restricted access to the host [28].

Currently, Native Client (NaCl) is in an early development stage. It is designed to allow
4.2. PERFORMANCE COMPARISON OF SEVERAL TECHNOLOGIES

<table>
<thead>
<tr>
<th>Framework</th>
<th>RC4</th>
<th>Primes ($10^6$)</th>
<th>Primes ($10^7$)</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javascript (Firefox)</td>
<td>9.97</td>
<td>4.23</td>
<td>4.24</td>
<td>102.73</td>
</tr>
<tr>
<td>Javascript (Chrome)</td>
<td>4.96</td>
<td>3.59</td>
<td>3.57</td>
<td>114.52</td>
</tr>
<tr>
<td>Javascript (Opera)</td>
<td>8.12</td>
<td>5.02</td>
<td>5.29</td>
<td>58.32</td>
</tr>
<tr>
<td>Java Applet</td>
<td>0.81</td>
<td>1.01</td>
<td>1.03</td>
<td>5.00</td>
</tr>
<tr>
<td>Java JVM</td>
<td>0.82</td>
<td>1.01</td>
<td>1.01</td>
<td>0.97</td>
</tr>
<tr>
<td>Java compiled with gcj</td>
<td>4.60</td>
<td>6.49</td>
<td>6.46</td>
<td>2.53</td>
</tr>
<tr>
<td>ActionScript</td>
<td>34.20</td>
<td>6.51</td>
<td>5.94</td>
<td>8.16</td>
</tr>
<tr>
<td>Silverlight</td>
<td>1.70</td>
<td>1.85</td>
<td>1.75</td>
<td>1.11</td>
</tr>
<tr>
<td>Silverlight (converted from Java)</td>
<td>1.75</td>
<td>2.89</td>
<td>3.00</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Table 4.2.: Relative execution speed of several frameworks. Should be interpreted as Framework X is n times slower than C++ implementation of the same code.

Before choosing the frameworks for the client-side, we make some tests to see which one performs better. In Table 4.2 the results of the tests are presented.

The implementation of RC4 encryption for the benchmark is adapted to Java, C#, C++, and ActionScript from this [31] JavaScript library. The Fast Fourier Transform libraries are also adapted from JavaScript code from here [47] based on [48]. The test were performed on a laptop with an Intel(R) Core(TM) Duo CPU @ 2.26 GHz, 3GB RAM and 1066MHz FSB. Operating system used was Windows 7 Professional. The tests with Java compiled with gcj were carried out on the same machine booted with Ubuntu 10.04. For each category, the tests were executed 20 times, and the arithmetic mean of the execution time was recorded. Each test was effectuated as follows:

- RC4 - 100 000 encryption/decryption rounds (total 200 000)
- Primes ($10^6$) - find the number of primes from 3 to $10^6$.
- Primes ($10^7$) - find the number of primes from 3 to $10^7$;

web-based applications to run at near-native speeds [29]. NaCl is open-source and is developed by Google [14]. The Native Client is supported by Google Chrome running on Windows, Mac OS, or Linux on x86 hardware [29]. The Native Client SDK is used to create Native Client executables (abbreviated as nexe) from scratch or from platform-specific web-based native applications [28]. If the code is trusted, it can perform privileged operations. In case the code is untrusted, it is limited by the sandbox. By isolating the code, it is possible to avoid misbehavior by malicious applications.

4.2. Performance comparison of several technologies

Before choosing the frameworks for the client-side, we make some tests to see which one performs better. In Table 4.2 the results of the tests are presented.

The implementation of RC4 encryption for the benchmark is adapted to Java, C#, C++, and ActionScript from this [31] JavaScript library. The Fast Fourier Transform libraries are also adapted from JavaScript code from here [47] based on [48].

The test were performed on a laptop with an Intel(R) Core(TM) Duo CPU @ 2.26 GHz, 3GB RAM and 1066MHz FSB. Operating system used was Windows 7 Professional. The tests with Java compiled with gcj were carried out on the same machine booted with Ubuntu 10.04. For each category, the tests were executed 20 times, and the arithmetic mean of the execution time was recorded. Each test was effectuated as follows:

- RC4 - 100 000 encryption/decryption rounds (total 200 000)
- Primes ($10^6$) - find the number of primes from 3 to $10^6$.
- Primes ($10^7$) - find the number of primes from 3 to $10^7$;
CHAPTER 4. IMPLEMENTATION SETUP

<table>
<thead>
<tr>
<th>Initialization type</th>
<th>loop</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type used</td>
<td>array</td>
<td>string</td>
</tr>
<tr>
<td>Operator used</td>
<td>%256</td>
<td>%256</td>
</tr>
<tr>
<td>%256 &amp; 0xFF</td>
<td>%256</td>
<td>%256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Firefox</th>
<th>Chrome</th>
<th>Opera</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>14.36</td>
<td>17.30</td>
<td>10.76</td>
</tr>
<tr>
<td>static</td>
<td>14.47</td>
<td>9.97</td>
<td>9.14</td>
</tr>
<tr>
<td>operator used</td>
<td>16.89</td>
<td>12.50</td>
<td>8.59</td>
</tr>
<tr>
<td>%256 &amp; 0xFF</td>
<td>12.00</td>
<td>4.32</td>
<td>7.15</td>
</tr>
</tbody>
</table>

Table 4.3.: Javascript RC4 relative (to C++) execution speed with different modifications to the code.

- FFT - 100 Fourier Transforms.

Versions of browsers and plug-ins:
- Firefox 3.6
- Chrome 9.0
- Opera 10.62
- Adobe Flash Player 10.1
- Silverlight 4.0
- Java 1.6

The Google Web Toolkit was the slowest in these tests, and after showing a very low performance compared to the other frameworks (from 50x slower for RC4 to 140x slower for Primes \((10^7)\)), was not considered anymore. Extensive tests with JavaScript in Internet Explorer 8 were not made, because the preliminary tests showed that its JavaScript engine is considerably slower compared to other browsers.

During the tests some interesting details came up, while trying to optimize the JavaScript code. It turned out that some modifications in the code could considerably decrease the execution time in Firefox, and slightly increase it in Chrome. For example, consider the code in A.1. The array \(s\) from line 2, is initialized in lines 3-5. The lines 3-5 could be replaced by a static initialization \((s = \{0,1,2,...,253,254,255\})\). When making this modification, there was no increase in the performance of Firefox. But the execution time for Chrome and Opera went down with 30% and 15% respectively. When the arguments to the function `rc4Encrypt` are passed as string instead of array and the computations in the body of the function are done on strings, Firefox shows a worse performance. In contrast, Chrome and Opera perform better. For all the browsers, the execution time was better when the operation \(\% 256\) was replaced with \& 0xFF. All these findings are summarize in Table 4.3.

By modifying slightly the code, the execution time decreased with 30.6% for Firefox, 32.2% for Chrome and 24.5% for Opera. This example shows how different results can one obtain by making only very small modifications in the code.
4.3. **CLIENT-SIDE IMPLEMENTATION**

The obtained results are encouraging and show that even using JavaScript for the client-side can give acceptable execution speed.

### 4.3. Client-side Implementation

#### 4.3.1. Structure

The core client was implemented in JavaScript for the following main reasons:

- The core client must be able to start at least when only JavaScript is enabled in the browser, because usually browsers come with JavaScript support enabled. It is also possible to easily activate it when it is disabled.

- It must allow easy communication with other frameworks.

While for most of the frameworks, in order to use them (Java applet, Silverlight, Flash) some installation is required on the client-side, for JavaScript, in most browsers, only the activation in the settings is needed. In all major browsers it comes active by default. Also, JavaScript provides easy mechanisms for checking if the other frameworks are available to the browser and simple intercommunication.

In Figure 4.1 an overview of the core client structure is depicted. The `Pafi` object is instantiated using object literals. The `PafiWorker` class is an object that represents a

![Class diagram of Pafi](image-url)

Figure 4.1.: Class diagram of `Pafi`
worker. A worker can be of any type, that is it can use any of the frameworks. For our prototype implementation, workers can be JavaScript, Java applet or Silverlight. 

*Pafi* is organizing the flow at a high level, manages communication between client and server, CPU usage etc. There are also framework-specific clients, their task being creating of an interface between *Pafi* and the framework-specific workers and providing the possibility of managing the workers. Their class diagrams are depicted in Figure 4.2 and Figure 4.3.

![Figure 4.2: Class diagram of Pafi Applet](image)

### 4.3.2. Initialization of Pafi

*Pafi* is initialized when the *iframe* is loaded from the *Intermediary Computations Server (ICS)*. The *iframe* tag is constructed by the *pafi_loader.js* script. This script builds an *iframe* tag with an *src* attribute pointing to the main page on the *ICS*, and having appended the *Session ID* of the client. The server stores the *Session ID* and returns a page that contains two scripts, *pafi.js* that encloses the core client, and *pafi_utils.js* with additional helper classes (like *jQuery* [24]) for *Pafi*.

When *Pafi* is started, it has to identify the frameworks (Java applets, Silverlight, etc.) available to the browser. *Pafi* has to check the availability of all the frameworks supported. When it identifies each platform, it has to set the value of the property *capabilities*, which is an integer. This works as follows - for every detected frame-
work, a the respective bit of capabilities is set to 1. For example, if Java support is
detected, then the LSB is set to 1. In case Silverlight support is detected, the second
bit is set to 1. Thus, if there is only Java support the value of capabilities will be 1,
if there is Silverlight support only, the value is 2, and if both are supported, the value
is 3. This way is possible to add support for other technologies like Flash and use the
next bits as markers for capability of the browser.

Pafi Applet

Before the applet is loaded, Pafi has to check if the browser supports applets. This is
done using the helper library from [38]. If there is Java support, then the Pafi Applet is
loaded with the following Pafi method shown in Listing 1.

Listing 1 Pafi method to embed Pafi Applet

```javascript
embedApplet: function(){
  jQuery(this.container).append('<applet name="pafiapplet" codebase="applets"
  code="com.amaise.pafi.applet.PafiApplet.class" width="0" height="0"
  MAYSCRIPT>'); },
```

The MAYSCRIPT attribute is very important as it allows the communication between
the applet and Pafi. When the loading of the applet is complete, it will report its ready

![Class diagram of Pafi Silverlight](image)

Figure 4.3.: Class diagram of Pafi Silverlight
state to \textit{Pafi}. The core client will update the flag \texttt{isAppletReady} to true.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{package_diagram.png}
\caption{Package diagram of \textit{PafiApplet} with \textit{RC4Worker}}
\end{figure}

\textbf{Pafi Silverlight}

For checking if there is Silverlight support in the browser, the library from [36] is used. If the browser supports Silverlight, the function is loading the Silverlight application (see: Listing 2).

\begin{lstlisting}[language=JavaScript]
embedSilverlight: function(){
    jQuery(this.container).append(''
        <div id="silverlightControlHost">
            <object
                id="pafisilverlight"
                data="data:application/x-silverlight-2,"
                type="application/x-silverlight-2" width="0px" height="0px">
                <param name="source"
                    value="silverlight/PafiSilverlight/PafiSilverlight.xap"/>
                <param name="enablehtmlaccess" value="true" />
                <param name="onError" value="onSilverlightError" />
                <param name="background" value="white" />
                <param name="minRuntimeVersion" value="4.0.51204.0" />
                <param name="autoUpgrade" value="true" />
            </object>
        </div>'');
}
\end{lstlisting}

\textbf{Pafi Javascript}

While \textit{Pafi} is written in JavaScript and it is working on the client machine, it is obvious that JavaScript is enabled and active, so there is no need to check, all we have to do is to start the \textit{Pafi Javascript} client as shown in Listing 3.
4.3. CLIENT-SIDE IMPLEMENTATION

Listing 3 Pafi method to embed Pafi Javascript

```javascript
embedJavascript: function(){
    jQuery(this.container).append('<script type="text/javascript"
        src="javascript/PafiJS.js"></script>');</
},
```

4.3.3. Communication between Pafi and Pafi Workers

Pafi keeps track of all the workers currently active. When it needs to communicate with them, it calls some functions on the PafiWorker object. For general purpose it is using the method jsCommand. The PafiWorker object knows how to handle the communication with different frameworks. All the frameworks disclose the same interface for communication, but it is handled slightly different for each of them.

Pafi Applet

In order to communicate with the Pafi Applet, the core client must first retrieve the pafiapplet object from the Document Object Model (DOM) tree. In Listing 4, an example of requesting the progress of computations is presented. Pafi passes as arguments the WorkerID and the Command. The third parameter is not used in this example.

Listing 4 Request of the progress of current computations example for applet

```javascript
document.pafiapplet.jsCommand(this.ID, "progress", "");
```

Pafi Silverlight

Silverlight framework offers an easy way of communicating with JavaScript. An example of how Pafi can call a function from Pafi Silverlight is shown in Listing 5.

Pafi JavaScript

For calling JavaScript function nothing special is needed because all the scripts run in the same context.

![Diagram](Image)

Figure 4.5.: Package diagram of PafiSilverlight with RC4Worker
4.3.4. Requesting work units

When the incoming queue is empty, or contains some minimum number of elements, the core client makes an XHTTP request to the server, asking for more work units. In this request it will communicate the capabilities of the browser (1 - Java support, 2 - Silverlight support, 3 - Java and Silverlight support), such that the server knows what kind of work unit to return. The server can return multiple work units at once. The possibility to load multiple work units at once is made to decrease the number of requests. After receiving the work units, Pafi will load these into the incoming queue.

4.3.5. Starting workers

The incoming queue will constantly be checked for elements. In case it is not empty, one by one, the elements will be extracted and objects of type PafiWorker created. The work unit contains the data needed for computations plus metadata. Based on this metadata, the PafiWorker object will be created for a specific framework. When the object is ready, it is stored in a hash map by Pafi and the method start (on the PafiWorker object) is called. The method jsStartWorker of the respective framework will be called. Inside this method, based on the information in the header of the work unit and the type of framework the following steps will be made (sample code for java in Listing 6, and for Silverlight in Listing 7):

1. Get the full class name including package (ex: com.amaise.room.rc4.RC4ClientWork), or get the path to the JavaScript file
2. Create object of this class
3. Pass the data needed for computations for deserialization
4. Create a PafiThread object
5. Store the Thread ID - PafiThreadObject in a hash map
6. For the beginning set the CPU usage to 10%, or less (will be adjusted later).
7. Start the thread.

For JavaScript the whole mechanism is basically the same with small differences. Because all the JavaScript files are loaded in the same context we must avoid reloading a file, or otherwise we get an error. For this reason, an additional parameter, which represents a JavaScript function as in Listing 9, that is passed to the jsStartWorker method. This function is evaluated, thus identifying if we need or not to request the file.
again. It works by checking the value of the global unique variable defined in every .js (ex: RC4WorkerLoaded). The same mechanism is used to ensure that the needed class is loaded after requesting it, because the calls are asynchronous.

**Listing 6** Example starting worker for Pafi Applet

```java
try {
    Class aClass = Class.forName(pack+className);
    try {
        AmaiseClientWork obj = (AmaiseClientWork) aClass.newInstance();
        obj.setData(DATA);
        PafiThread thread = new PafiThread(obj, workerId, this, wuId);
        this.threads.put(thread.id(), thread);
        thread.setPerformance(10);
        thread.Start();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

When the thread is started (see Listing A.4), it will begin the execution of the calculate function of the object (passed to it as argument), which is implemented for each project. The calculate function of the ClientWork class is the main point where the calculations are done. This will be executed many times until the computing is completed. It will be called as long as it is returning true. One execution of this function should take milliseconds. More details in 4.3.6 and 4.5.2.

**4.3.6. CPU usage throttling**

The PafiThread object will observe the time of execution of the calculate function at each round (see Listing A.4). Based on this time, and the CPU priority dictated by Pafi, it will adjust the sleep time for the thread such that it does not use more CPU that it was assigned to. It works as follows:

1. Observe calculation time $t_c$.
2. Calculate the sleep time $t_s = t_c(100 - c)/c$ (where $c$ is the allowed CPU usage in percent).
3. Sleep for $t_s$ amount of time before going to the next iteration.
CHAPTER 4. IMPLEMENTATION SETUP

Listing 7 Example starting worker for Pafi Silverlight

```csharp
WebClient client = new WebClient();
MainPage tempThis = this;
client.OpenReadCompleted += new OpenReadCompletedEventHandler((object sender, OpenReadCompletedEventArgs e) => {
    AssemblyPart assmPart = new AssemblyPart();
    Assembly assm = assmPart.Load(e.Result);
    
    Type t = assm.GetType(classname);
    AmaiseClientWork obj = (AmaiseClientWork) Activator.CreateInstance(t);
    PafiThread thread = new PafiThread(obj, workerid, tempThis, wuId);
    thread.setPerformance(10);
    tempThis.threads.Add(workerid, thread);
    thread.Start();
});
client.OpenReadAsync(new Uri(dll, UriKind.Relative));
```

In order not to overload the CPU with calculating $t_s$ at each round, a parameter $s$ is used to adjust $t_s$ once every $s$ iterations (ex: $s = 5$);

4.3.7. Receiving results from workers and sending them to the server

When the calculate function of the Client Work object returns false, it means that it ended the calculations and is able to return the data. The serialize method is called. This method will prepare some XML data. This data is project specific and is implemented differently for each project. The data contained in the XML will represent the results of the computations. The format of this data should be understood by the respective Room Controller. Serialized data is returned to Pafi, where it is packed in a result envelope and sent back to the server. The envelope represents another XML string that encapsulates an element data that contains the XML returned by the worker.

4.3.8. Handling lost connections

Pafi will constantly monitor the connection with the server. At fixed time intervals, the core client is making XHTTP requests to the server. The server replies to these “ping” requests. In case there is no connection to the server, and error will occur. This way, Pafi can spot when the connection is lost. When connection is lost, it will limit the request attempts not to overload the browser with errors.
4.3. CLIENT-SIDE IMPLEMENTATION

Listing 8 Example starting worker for Pafi Javascript

```javascript
jsStartWorker: function(pack, className, params, DATA, workerId, wuId) {
  // check if script file already loaded
  if (eval(params) == false) {
    // load the script
    jQuery.getScript(pack);
  }
  this._jsStartWorker(pack, className, params, DATA, workerId, wuId);
},

_jsStartWorker: function(pack, className, params, DATA, workerId, wuId) {
  // wait until script is loaded by the browser
  if (eval(params) == false) {
    var obj = this;
    setTimeout(function() { obj._jsStartWorker(pack, className, params, DATA, workerId, wuId) }, 100);
  } else {
    // script is loaded, we can start worker
    var obj = eval("new " + className + "();");

    // fix for IE
    if (!window.DOMParser) {
      var xml = new ActiveXObject("Microsoft.XMLODOM");
      xml.async = "false";
      xml.loadXML(DATA);
      DATA = xml;
    }
    obj.setData(DATA);
    var thread = new PafiJSThread(obj, workerId, this, wuId);

    this.threads.put(thread.ID, thread);
    thread.setPerformance(10);
    thread.Start();
  }
},
```

29
Listing 9 Example checking if a JavaScript library is loaded by the browser
function checkRC4WorkerLoaded(){
    if (typeof(RC4WorkerLoaded) != 'undefined' &&
        typeof(PafiJSThreadLoaded) != 'undefined'){
        return true;
    }
    return false;
}
checkRC4WorkerLoaded();

4.4. Server-side Implementation

The server side was implemented as a Java servlet, although it can be replaced with any other technology. This is possible because the core client communicates with the server through simple XHTTP POST requests. This way, for Pafi is not important how the server works, as long as the server can reply with JSON to the POST requests.

![Figure 4.6.: Package diagram of Nest](image)

4.4.1. Nest

The server will receive the Session ID from the client at its first connection. Any communication with the server includes the Session ID in the request such that the server knows with whom it communicates.

The Nest provides and interface for interaction with the clients and the Content Providers. For the clients it provides:

- **Get page** - the server returns the main page that will be loaded into the iframe.
- **Get work unit** - the server returns work units.
4.5. **INTEGRATION OF OTHER PROJECTS**

- *Ping* - process the ping request needed for *Pafi* to know that the connection is alive

- *Receive result* - server receives the results of the computations

and for *Content Providers*:

- *Get credit* - the *Content Provider* requests the amount of credit attached to a specific *Session ID*.

- *Decrement credit* - the *Content Provider* will request the *Nest* to decrement the amount of credit after it sends the content to the client.

All the *Session ID’s* of the clients are kept in a hash map with a corresponding *SID* (Figure 4.7) structure associated to it.

The server will always check first the validity of a received result as described in 3.2. Depending on the return message from the *Room Controller*, it will update the information regarding the current client, by marking it as suspect or trusted. Even if the client is constantly returning incorrect results, it will not be notified about this, such that it does not learn anything about the validation process on the server.

4.4.2. **Rooms**

Every project has an implementation of a *RoomController* class. This object is responsible for work unit generation, validation and assimilation. These operations can to be implemented in code as methods of the *Room controller*. It is also possible use binary files stored on the server. this way it is easier to reuse binary files from existing projects created for BOINC.

Each *room controller* has an object of class *RingerController*. This object helps the room controller with storage and validation of ringers.

The *RingerController* object associated with the *RoomController* class, has a variable through which is possible to specify the probability of returning a ringer to a client.

The *RoomController* generates the ringer when it has to, and asks the *RingerController* class to hold it and sends the work unit to the client. After receiving the result, it will ask the *RingerController* if the received outcome is a ringer and whether it is valid.

4.5. **Integration of Other Projects**

This framework allows easy integration of new projects. For the reason that all the computations are done in browsers the use of the file system of the client machine is problematic. Because the computations should take a relatively small amount of time, a project should have the possibility to split the computationally expensive task into small sub-tasks. Also, this framework would make inefficient a scheme where the computations should be made on very big amounts of data (even if computation time is small) because of the big network load that would be generated.
CHAPTER 4. IMPLEMENTATION SETUP

4.5.1. Server-Side

For adding a new project for the server-side, it is only required to implement a child of the RoomController class (see Figure 4.7, class RC4RoomController). A small configuration file (see Listing 10) is needed to be available in the project folder for the server to know what projects are available. When the server starts, it will automatically extract all the needed information, and the project will be ready. The application can also use external binary files for work unit generation, validation and assimilation. This would make easier the transformation of a project from a platform like BOINC.

In case communication with other computations servers is needed, it can be implemented in the child of the RoomController class (see A.2). For each project a working folder is available for storage. Also Database support can be added if necessary.

4.5.2. Client-Side

For the client-side, for each framework an implementation of a child of type AmaiseClientWork (see A.3) is required (see Figure 4.2, class RC4ClientWork or 4.3, class Calculate. The compiled .class, .dll and plain .js files will be placed in the rooms/ folder on the server. The information about these files are added to the configuration (See Listing 10), and Pafi finds and downloads them when needed.

Listing 10 Example of Room configuration file

```java
name=RC4
capabilities=3
slpackage=./room/RC4/RC4.dll
slclass=RC4.Calculate
jpackage=com.amaise.pafi.room.rc4ex.
jclass=RC4Worker
jspath=javascript/RC4Worker.js
jsclass=RC4Worker
jcheck=function checkRC4WorkerLoaded()
    {if (typeof(RC4WorkerLoaded) != 'undefined' &&
        typeof(PafiJSThreadLoaded) != 'undefined')
        {return true;}
    return false;}
checkRC4WorkerLoaded();
roomControllerClass=com.amaise.nest.room.rc4ex.RC4RoomController
dumpRingersInterval = 60000
```

32
4.5. INTEGRATION OF OTHER PROJECTS

Figure 4.7.: Class diagram of Nest
Chapter 5
System Evaluation

In order to test our prototype, as a project we implemented an RC4 cracker. This cracker has to find the key $k$ for a given plaintext $P$ and a cyphertext $C$ such that $RC4_k(C) = P$.

We also simulated the *micro-computations as micro-payments* concept described in [53], by making the users “pay” with computations for viewing a list of articles.

On the created page, a list of links to articles is presented. In front of every link, there is an indicator which can be green or red, showing whether the corresponding link is accessible or not. For viewing the first article the readers do not need to make computations (as you can see in Figure 5.1). When clicking an accessible link an overlay pop-up window is opened, where the abstract of the article is displayed and a link that allows downloading the PDF file (see Figure 5.2). In case an article is not accessible, the PDF link is also not accessible.

On the right side of the page, in a box, the available credit is displayed, that is the number of successfully completed computations. When a work unit is completed, sent back to the computations server and validated, the credit is incremented. The prototype is trying to perform as many computations as possible and these are not bound to viewing a specific article.

When there is not enough credit for viewing an article, a message is displayed (see Figure 5.3).

For our prototype, there is no need for a user to be logged in. This makes it easier to use the website. Also it preserves the privacy of users, as they do not have to provide any data to the system. Even if a user is navigating away from the page, the credit associated with the current session is not lost for a specified period of time (for example 30 minutes) until the session expires. When returning back to the page, the user will see the amount of credit it had before navigating away. Also, the visited links are kept
available.

Figure 5.1.: Prototype screen shot: First visit on the page

Figure 5.2.: Prototype screen shot: Viewing and downloading an article
5.1. Prototype Evaluation

5.1.1. Prototype version 1.0

The first version of the prototype was tested with a group of students at ETH Zürich. As a project, an RC4 cracker in JavaScript was implemented. This prototype used a slightly different setup than the last version. For the students, a web page was prepared, and they were asked to read a book chapter. The pages of the chapter were delivered to the web page as JPEG images. When the user opened the website, the first page of the chapter was displayed. In the lower part of the web page a “Next” button was positioned. This button was disabled while the core client on the page was making computations. When the computations end, the button is enabled, allowing users to view the next page of the chapter. When pressing the “Next” button, a new work unit would be requested for the server such that computations could be performed. It would continue like this, until the last page of the book chapter was displayed. In the end, the students were asked to answer a list of questions. The students were not informed why the button stays disabled for a while, instead they were told that the disabling of the button ensures that they actually read each page of the chapter. For 25% of the people chosen randomly, computations were not made in their browsers, instead the “Next” button would stay disabled for about 15 seconds.

The statistics of the experiment with the first version of the prototype are presented in Table 5.1 (also see [53]). We could observe that using JavaScript for making computations in browsers is possible, even though the implementation was not the most efficient.
5.1. PROTOTYPE EVALUATION

<table>
<thead>
<tr>
<th>Total number of keys searched</th>
<th>$2^{20}$ keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of work units sent</td>
<td>1548</td>
</tr>
<tr>
<td>Number of individual sessions</td>
<td>92</td>
</tr>
<tr>
<td>Aggregate computation time</td>
<td>188 minutes</td>
</tr>
<tr>
<td>Total network load</td>
<td>8.65 MB</td>
</tr>
<tr>
<td>Average server CPU time</td>
<td>8.34 seconds</td>
</tr>
<tr>
<td>Average user CPU time</td>
<td>15 s per work unit</td>
</tr>
</tbody>
</table>

Table 5.1.: Prototype version 1 deployment statistics (adapted from [53])

The total CPU time of JavaScript execution was 188 minutes, which compared to the total server time, gives a ration of 867, that is 1 day of server computation corresponds to 2.3 years of client computations [53].

The network load was relatively small, of about 80 KB of scripts per session. These scripts are loaded once and cached by the browser [53].

The results of the questionnaire displayed when people finished reading the book chapter showed that the micro-computations did not disturb their browsing experience. None of the students detected unusual CPU load. 97% of the participants said that they would generally agree to support existing research efforts by running computations in their browsers [53].

5.1.2. Prototype version 2.0

The second version supports more frameworks than JavaScript only. As described in previous chapters it also has more functionality (for example CPU usage throttling, multi-threading etc.) and more efficient implementation. For the same amount of time on the client machines, the RC4 crackers can check from 10 to 500 times more keys (than the version 1.0 prototype) depending on the framework used (JavaScript, Silverlight or Java applet). The last version of the prototype was not tested with users. The prototype worked normally on the following platforms:

- Windows 7
  - Firefox 3.6, 4.0
  - Safari 4
  - Chrome 10
  - Opera 9
  - Internet Explorer 9
- Ubuntu 10.04 LTS
  - Firefox 3.6
  - Chrome 9
- Mac OS X
The prototype was tested also on mobile devices. It worked normally on the following devices that were available to us:

- Nokia E71 (OS: Symbian)
  - Opera Mobile 10
- iPhone 4 (OS: iOS 4)
  - native browser
- HTC Desire (OS: Android)
  - native browser
  - Firefox 4
  - Opera Mobile 10
- Nook Color (OS: Android)
  - native browser
  - Firefox 4
  - Opera Mobile 10

The execution speed on the mobile devices was noticeably slower than on the other tested computers, because the work units sizes were optimized for the later. But the responsiveness of the devices was normal. We concluded that computations even in mobile devices is possible, just the work unit size needs to be adjusted according to the computation power of the device.

5.2. Limitations of the Prototype

When making computations using web-based clients instead of binaries deployed on client machines, obviously there are some limitations. There is no access to the operating system APIs, thus there is no possibility to get information about the CPU load of the system. Because of this, the prototype is set to use less than 50% (configurable) of CPU time.

Another important limitation is that there is no low level access to the CPU or GPU instructions. Some current projects make use of the possibilities of the high performance GPUs (for example CUDA).

The lack of possibility for accessing the local file system makes hard to implement the projects that need a lot of storage during the computations process. JavaScript has no possibility at all for storing data on the hard drive (except small amount of data in
5.3. FUTURE WORK

Cookies). If the Java applets are signed, these could have access to the local file storage [21]. A signed applet will contain a signature that the browser could check with a certification authority. The Silverlight applications can store data on the local file system of a user through the Isolated Storage mechanism [34]. The data is stored in a protected folder on the client machine. Several different classes allow to store data from simple scalar values and strings, to complex serialized objects. It is worth mentioning that the network load should be minimal, because otherwise it might disturb the user experience or spend more time downloading work units or uploading results, than making useful computations.

Not all the projects are suitable for our scheme. For some projects, there could be no possibility to create micro work units. Also, to convert an existing project to be part of our framework a considerable effort could be needed, mostly by rewriting the code for the target languages (Java, C# or JavaScript). The different data types used by different languages can make this task even harder.

5.3. Future work

Checking client CPU capabilities. Currently the prototype is not adjusting the work unit size depending on the client CPU performance. In the future development, this feature should be added. It could be possible to send a specific reference (for which we know the execution time on a reference CPU) work unit to the client, and request the execution time for it. Based on the reported time to finish the computations we could make the decision on adjusting the work unit size. This functionality is very helpful especially when executing computations on devices (for example mobile phones, eReaders, etc.) with CPU capabilities that are very different compared to desktop computers. Because our core client can start multiple workers at the same time, it would be possible to find the number of cores a system has and starting a number of workers equal to the number of cores of the CPU. This could be done also with a reference work unit, by measuring the execution time of one worker, two workers, four workers etc. For example, if the execution time is the same for 1 worker, and four workers running in parallel, than it is possible to suppose that the client CPU has at least 4 cores.

Adding support for other frameworks. Currently our platform supports JavaScript, Java applets and Silverlight applications. While Flash is widely available in browsers and Native Code for Chrome is claimed to have very good performance, support for these technologies should also be added.

Extensive testing. More extensive testing is needed with our framework, both with different platforms and with a big number of subjects. The system should cover as many platforms as possible. Different platform can introduce floating-point errors when making computations. These issues should be taken into account. A survey with a big number of participants could give more decisive conclusions on the
acceptability and usability of our scheme. This could be possible to do using Amazon MTurk platform [37]. Also, an existing project should be ported to our framework in order to compare the performance to the existing solutions.

5.4. Applications

**Extending current distributed computing projects.** There are plenty of distributed computing projects and computing grids that try do scientific research that benefit humanity. In order to speed up the computations, these projects need to get as many participants as possible. Some volunteers donate their idle computing power of their computers or game consoles using client binaries installed on these systems. For some people is easy to use these systems, for others is not. By using our framework connected to an existing computation server, it is easy to offer the possibility to more participants to help with the research, because using our solution is a lot easier. Moreover, as discussed in previous chapters, our solution offers comparable functionality and efficiency when compared to current projects. Users can join a project only by opening a web page. It is also very easy to stop making computations just by closing the web browser. As shown in [53], this framework could aggregate up to 0.7 TFLOPS of computing power when used with social networking websites, email providers or on-line news sites.

**Micro-computations as micro-payments.** Another use of our system could be for the replacement of the advertising on web pages with the execution of micro-computations as described in [53]. Studies show that users do not trust on-line advertising [13] and sometimes use tools to block them. Because of this and the fact that the revenues from advertising drops some websites want to give up to the advertising on their pages in favor of charging people for on-line content. People might not want to pay for reading news or using social network websites. In fact, only a small fraction of users - less than 3% - would accept to pay for reading news on-line [27, 12]. In our scheme, when a user opens a web page and requests content, his browser will be asked to make some computations in exchange for the content. After the results are gathered from the users and sent back to the computations server, the content provider will receive payment. Also, this scheme guarantees user privacy because private data about users is not gathered. In fact, they do not even need to open user accounts. This is more convenient that the targeted advertising which needs to make user profiling for the publicity to be efficient.
Chapter 6

Concluding Remarks

Distributed computing projects can be very helpful for academic groups or research labs by offering the possibility to make computationally expensive tasks cheaper by using the idle computing power of volunteer computers instead of building computer grids. But, in order to efficiently complete the project, a big volunteer base is necessary. Also, given the nature of the distributed computing projects, the frameworks should have a good validation mechanism to identify wrong results returned by users intentionally or unintentionally. The wrong results could put at risk the entire project.

Usually, for any distributed computing framework, the participants have to first download the software for their platform, install and run it. This creates problems when users do not have credentials for installing new software on their systems. Also, for unsophisticated users, these tasks are not straightforward. Another issue comes up when users would like to participate, but for a short period of time - they still have to go through the above mentioned process.

We offer a new way of performing the computations. Instead of client-side software that needs to be installed, we study the feasibility of making the computations using the web browsers already present on the client machines. After assessing the current technologies we identify the ones that would allow making computations in browsers efficiently.

By using such widespread programming languages like Java, it is possible to also reuse code for easy adaptation of existing projects to our framework. The later also offers the possibility for interconnection with existing frameworks, thus extending the number of participants and speeding up the computation process.

The implementation of ringers scheme to secure the results is a good option that shows little overhead.

After making a survey with a group of students at our university, who used our prototype the results are very good. None of the participants had a bad experience while browsing and having computations made in the background.

More surveys can be made by using the Amazon Mturk platform [37]. The preliminary
tests made with this platform show that it is possible an easy integration with it, in order to test the entire framework with a big group of people.
Bibliography


Appendix A

Source code

The full source code of the client-side, server-side, content provider library was delivered on a CD, attached to the thesis.

A.1. RC4 implementation (JavaScript)

```javascript
function rc4Encrypt(key, pt) {
    s = new Array();
    for (var i=0; i<256; i++) {
        s[i] = i;
    }
    var j = 0;
    var x;
    for (i=0; i<256; i++) {
        j = (j + s[i] + key.charCodeAt(i % key.length)) % 256;
        x = s[i];
        s[i] = s[j];
        s[j] = x;
    }
    i = 0;
    j = 0;
    var ct = '';
    for (var y=0; y<pt.length; y++) {
        i = (i + 1) % 256;
        j = (j + s[i]) % 256;
        x = s[i];
        s[i] = s[j];
        s[j] = x;
        ct += String.fromCharCode(pt.charCodeAt(y) ^ s[(s[i] + s[j]) % 256]);
    }
    return ct;
}
```
A.2. RoomController class (Java)

```java
package com.amaise.nest.room;
import java.util.Random;
import com.amaise.nest.Log;
import com.amaise.nest.RingerController;
import com.amaise.nest.SID;

public class RoomController {
    public String workingFolder = "";
    public RingerController ringerController;

    //probability to return ringer
    //starting with this request number the probability is this
    private int[][] prob=
    {
        {0, 100},
        {5, 80}
    };

    public void init(String workingFolder, int ringerBufferSize){
        this.workingFolder = workingFolder;
        this.ringerController = new RingerController(ringerBufferSize);
    }

    public boolean validate(String datai, String datar, String sid){
        return false;
    }

    //the function called by the WorkUnit Controller
    public final String[] getWorkUnit(String datai, String datar, String sid,
        boolean small){
        if (this.shouldReturnRinger(sid)){
            return this.returnWorkUnit(datai, datar, sid, small, true);
        }
        return this.returnWorkUnit(datai, datar, sid, small, false);
    }

    //override in child class
    public void receiveExpiredWorkUnit(String datai){
        //handle expired work units
    }

    //override in child class
    public String[] returnWorkUnit(String datai, String datar, String sid, boolean small, boolean returnringer){
        //must return a work unit
        return null;
    }

    //checks if must return ringer
    public final boolean shouldReturnRinger(String sid){
        long requestNumber = SID.requestNumberFor(sid);
        //look into the table
        int p = 100;
        for (int i=0; i<prob.length; i++){
            if (prob[i][0] <= requestNumber){
                if (prob[i][0] <= requestNumber){
                    break;
                }
            }
        }
    }
}
```
A.3. ClientWork class (Java)

```java
package com.amaise.pafi.applet;

public class AmaiseClientWork {

    protected String DATA = "";
    protected long INITIALITERATIONS = 0;
    protected long ITERATIONS = 0;
    protected int BATCHSIZE = 0;
    protected String RETURNDATA = "";

    //override in child class
    public boolean calculate(){
        //make computations, it is executed while it returns true
        return false;
    }

    //override in child class
    public boolean serialize(){
        //prepare the data to be sent to the server
        return true;
    }

    //override in child class
    public boolean deserialize(){
        //unpack the date received from the server
        return true;
    }

    public AmaiseClientWork(){
    }

    public final long getIterations(){
        return this.ITERATIONS;
    }

    //pass data needed for computations
    public final void setData(String data){
        this.DATA = data;
        this.deserialize();
    }

    p = this.prob[i][1];
}
```

A.4. PafiThread class (Java)

```java
package com.amaise.pafi.applet;
import java.util.Date;

public class PafiThread extends Thread{
    private String ID;
    private AmaiseClientWork clientWork;
    private boolean Stopped = true;
    private boolean Suspended = true;
    private int SLEEPTIME = 100;
    private int PERFORMANCE = 10;
    private long TIME = 0;
    private boolean isPerformanceSet = false;
    private PafiApplet master;

    //how often to check for the execution time and adjust
    //the sleep time (cycles)
    private int SETPERFORMANCEINTERVAL = 3;
    private int CSETPERFORMANCE = 1;
    private long timeToCompute = 0;
    private String wuId;

    //constructor
    public PafiThread( AmaiseClientWork clientWork, String id, PafiApplet master,
        String wuId){
        this.clientWork = clientWork;
        this.ID = id;
        this.master = master;
        this.wuId = wuId;
    }

    //return Identification number of the thread
    public final String id(){
        return this.ID;
    }

    public final void run(){
        while (!this.isStopped() && this.isAlive() && !interrupted()){
            if (this.isSuspended()){
                try {
                    sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```
A.4. PAFIThread CLASS (JAVA)

```java
50  }
51  
52  continue;
53  }  
54  Date start = new Date();
55  long startmilis = start.getTime();
56  
57  //////////////////////////////////////////////////////////////////////
58  boolean terminate = !clientWork.calculate();
59  //////////////////////////////////////////////////////////////////////
60  
61  Date end = new Date();
62  long endmilis = end.getTime();
63  this.TIME = endmilis - startmilis;
64  
65  this.timeToCompute += this.TIME;
66  
67  if (terminate){  
68     this.returnResult();
69     this.interrupt();
70  }
71  if (!interrupted()){  
72      if (!this.isPerformanceSet){
73          this._setPerformance(this.PERFORMANCE);
74      }
75      
76      this.CSETPERVORMANCE++;
77      if (this.CSETPERVORMANCE == this.SETPERFORMANCEINTERVAL){
78          this._setPerformance(this.PERFORMANCE);
79          this.CSETPERVORMANCE = 1;
80      }
81      
82      try {
83          sleep(this.SLEEPTIME);
84      } catch (InterruptedException e) {
85          e.printStackTrace();
86      }
87  }
88  else{
89      return;
90  }
91  
92  }  
93  
94  //return the time spent on computations
95  public final long timeToCompute(){
96      return this.timeToCompute;
97  }
98  
99  //returns status if stopped
100  public final boolean isStopped(){
101      return this.Stopped;
102  }
103  
104  //returns status if suspended
105  public final boolean isSuspended(){
106      return this.Suspended;
107  }
108  
109  //stop thread and return results
110  public final void Stop(){
111      this.Stopped = true;
```
this.returnResult();
}

//suspend execution of the thread
public final void Suspend(){
    this.Suspended = true;
}

//resume the execution of the thread
public final void Resume(){
    this.Stopped = false;
    this.Suspended = false;
}

//prepare the Client Worker
public final void Start(){
    this.clientWork.prepare();
    this.Resume();
    this.start();
}

//return the progress of the computations
public final double getProgress(){
    return Math.round((double) (this.clientWork.INITIALITERATIONS-
    this.clientWork.ITERATIONS) / this.clientWork.INITIALITERATIONS * 100 * 100)
    / 100.0; }

//return results back to the applet
public final void returnResult(){
    this.clientWork.serialize();
    this.master.jsSendResults(this.ID, this.wuId, this.clientWork.getReturnData());
}

public final void setPerformance(int performance){
    this.PERFORMANCE = performance;
}

//return CPU usage of the thread
public final int getPerformance(){
    return this.PERFORMANCE;
}

//set the CPU usage of the thread
private final void _setPerformance(int performance){
    this.PERFORMANCE = performance;
    int rem = 100-this.PERFORMANCE;
    this.SLEEPTIME = (int) ((this.timeForBatch() + rem / this.PERFORMANCE));
    this.isPerformanceSet = true;
    if (this.SLEEPTIME>1000){this.SLEEPTIME = 1000;}
}

//return the time spent on computations on a single batch
private final long timeForBatch(){
    if (this.TIME == 0) this.TIME = 1;
    return this.TIME;
A.4. PAFITHREAD CLASS (JAVA)

```java
s
```