Riccardo Volonterio

Web-based Human and Machine-Driven computation

Tesi di laurea specialistica

Relatore: Alessandro Bozzon
Co-relatore: Luca Galli

Politecnico di Milano
Polo regionale di Como
Dipartimento di Elettronica e Informazione
Ottobre 2012

La citazione è un utile sostituto dell'arguzia.

— Oscar Wilde

Dedicato a tutti gli appassionati di LaTeX.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SOMMARIO

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

ABSTRACT

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abbiamo visto che la programmazione è un'arte,
perché richiede conoscenza, applicazione, abilità e ingegno,
ma soprattutto per la bellezza degli oggetti che produce.

— Donald Ervin Knuth

RINGRAZIAMENTI

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Como, Ottobre 2012 L. P.

INTRODUCTION

In the field of distributed computing have been used several methods to create a common layer able to execute code on different systems and platforms. The paradigm of distributed computing is based on the paradigm of grid computing and on that of cloud computing. These paradigms leverage on the core concept of creating an abstraction layer on top of the available resource in order to make them consistent, for example grid computing abstract only part of the available resources, meanwhile cloud computing abstract the whole hardware.

The distribution of the computation can be done at **hardware** or **software** level.

At **hardware** level we have similar distributed resources, or at least can be easily abstracted, so we can distribute and gather the results. This paradigm is used in frameworks like Dean and Ghemawat, 2008 where the computation is spread on large cluster of computers.

The distribution of computation at **software** level uses the concept of distributed systems, where the automatic computation is spread among different machines usually separated by a network. Once the computation is executed by a node, the result is processed by the server and if needed another computation is triggered by ther server, an so on.

Another paradigm has been outlined in this field, **human computation**. The paradigm is the same as above because we need to computation but here the nodes have the ability to perform computation that other standard nodes, like pc and similar, are not able to do.

As one may notice, the idea of human computation is very similar to distributed computation also it leverage on web-based distribution technologies. Usere get engaged using the web, and also the tasks are executed within a web browser. Human computation application or Game With A Purpose (GWAP) usually relies on the web as a common platform like Von Ahn, Liu, and Blum, 2006 or *MTurk*. Another solution is to create a standalone normalized software platform like *FoldIt*.

Given this general overview one can spot that we reached a condition where we have the technical ability to use all the web-users as nodes able to perform arbitrarily complex computation either automatic or human.

As far as we know there are no methods or tools able to stress this opportunities, because they focus on human or automatic computation[1]. The matrix in Table 1 is the representation of the available online

1 Not web-based, but using standalone clients.

tools categorized using as dimensions the will of the user of performing such tasks and the *complexity* of the algorithm. When using the term *complexity* we refer to two main types of computational complexity *workload complexity* and *algorithm complexity*.

Workload complexity indexes all that algorithms that need to perform a huge amount of simple (or not so simple) computation on a lot of data. To address this problem we need use the *Divide et impera* paradigm, like the one used in Dean and Ghemawat, 2008, allowing to split algorithms that operates on huge amount of data into atomc steps that can be executed by any node. When dealing with this type of complexity we need to do **automatic** computation.

Algorithm complexity addesses the other dimension, here we consider the complexity as the computational feasibility of each step of the algorithm. As an example consider the following algoritm:

```
input : a set of tweet about a politician
output: each tweet marked as in favor or against the politician

foreach tweet in tweets do
    opinion ← check(tweet);
    if opinion≠IN_FAVOR then
        contactCIA();
    end
    setTweet(tweet, opinion);
end
```

Algorithm 1: Tweet validation

The algorithm itself is not complex but operation like `opinion ≠ IN_FAVOR` cannot be done by a normal node, like a pc, or they took too long to be computed. These cases belongs to the field of **human** computation.

Table 1.: Task distribution and execution matrix.

	Automatic	**Human**
Voluntary	BOINC	*MTurk*
Involuntary	Parasitic computing	GWAP

A limitation of the available frameworks for automatic computation is the ease of access of the tool for the end-users. Let's take Search for Extra-Terrestrial Intelligence *at* home (SETI@home) as an example, this tool uses the Berkeley Open Infrastructure for Network Computing (BOINC) platform to search for extraterrestral activity using radio telescope and analizing narrow-bandwidth radio signal. A user who want to partecipate to this priject must install the BOINC platform and then enter a specific URL to start contributing. This steps, despite their semplicity, have hidden overhead to the user and to the SETI@home project. The installation of ad-hoc clients can be a problem when a user work an a machine with strong restriction, also the SETI@home project must adapt their data and computation to be executed within the BOINC platform.

ORIGINAL CONTRIBUTION

The aim of this thesis is to present a model for distributing and executing task that covers all the matrix dimension expressed in table 1, and on top of that provide:

- ease of access to the tasks
- usage of standardized protocols/languages
- ease of implementation by the *requester*
- ease of execution by the users

The original contributions are:

1. Definition of a model for automatic, human and hybrid computation
2. Implementation of a reference web-based architecture for human and automatic implementation
3. Implementation of an infrastructure supporting the defined model
4. Validation through 3 use cases (automatic, human, hybrid)

OUTLINE

The thesis is organized in four main parts.

THE FIRST CHAPTER

NEL SECONDO CAPITOLO

NEL TERZO CAPITOLO

NELL'ULTIMO CAPITOLO

1 THE BACKGROUND

Recent years have seen an increasing interest in *Human Computation* and *Crowdsourcing* areas. One of the reason they are becoming so attractive is the growth of the Web. This has allowed to leverage the ability of people over the internet to perform tasks that even modern computers cannot achieve properly.

This chapter, first, focus on the key steps and developments in these fields that lead to the purposes of this thesis. We provide an overview of human computation and parasitic computing, then we introduce the technologies that enables the distributed computation on the web such as HTML5 for the task distribution and execution and WebCL for the task execution.

1.1 CROWD-BASED COMPUTATION DISTRIBUTION

Distributing computation (task computation) in the crowd means splitting the task execution into atomic subtask that can be executed by a host (human or not).

Figure 1.: General structure of a distributed computing system.

Generally speaking *computation distribution* is composed by few steps that can be spotted in all its specialization (i.e. grid computing, cloud computing, parasitic computing, etc.), here is the list:

1. the server has to **split** the workload into smaller parallelizable operations
2. the server **send the code**, among with the needed data, to the clients
3. the client **run** the code
4. the client **send the results** to the server
5. the server **gather** the results from all the clients
6. the server **join** the results

The previous operations are the cornerstone of every distributed computing system, altough they can be done in different ways of can be merged. For instance the client can request the code to the server, the join process of the results can be performed by the clients with subsequent distributed computing task, and so on.

Under the general name of *distributed computing* we can refer to a wide range of distributed application and/or architecture supporting it. Human computation & GWAP are good examples of a specialization of the core concept of *distributed computing*, here we have that the computation is performed by human being used as nodes for high level computation. Other specialization are *Grid computing*, where the nodes form a super virtual computer, or *Jungle computing*[1], where using diverse, distributed and highly non-uniform high performance computer systems to achieve peak performance.

Following the subdivision presented in Table 1 we separate the concept of crowd-based computation distribution in two parts, *Human computation & GWAP* and *Automatic computation*.

1.1.1 Human computation & GWAP

Computers are capable of performing many tasks, they can process large amounts of data and do billions of operation in a few seconds. However, there are still many problems that computers cannot solve or take too much time to solve even for the powerful pc.

Some of these are very simple tasks for humans, for example natual language processing and object regonition are hard to solve problem for a computer but natural for a human being. A great example for this kind of problem is recognizing hand-written text, even after years of research, humans are still faster and more accurate than any computer.

1 Winner of the coolest name 2012

Furthermore, there are problems that are too computationally expensive, such as many NP-complete problems like Traveling Salesman problem, scheduling problems, packing problems, and FPGA routing problems.

The expression *Human Computation* in the context of computer science is already used by Turing, 1950. However is Law and Ahn, 2011 to introduce the modern usage of the term. He defines human computation as *a research area of computer science that aims to build systems allowing massive collaboration between humans and computers to solve problems that could be impossible for either to solve alone*. But, in my opinion simple and direct definitions are better to get the point:

> *Some problems are hard, even for the most*
> *sophisticated AI algorithms.*
> *Let humans solve it. . .*
> — Edith Law

In pratice when speaking about human computation we refer to an algorithm that involves humans interaction during the computational process. As an example infrastructure one can think of *MTurk*, a *centralized* human computation platform that leverage on *Workers* to perform task.

By using the term *centralized* we mean that all the computation is performed in one central place (e.g. *MTurk* website). On the other hand we have *distributed* human computation, here the code is offloaded to the client using some ad-hoc software clients.

Centralized

The *centralized* paradigm is the most common because it do not require the creation of ad-hoc clients to perform computation, that can run directly in the browser. Now we present some of the well known online tools that use this kind of algorithm.

MTurk is an online tool that allow the execution of crowd-based human computation task, called Human Intelligent Task (HIT). Each HIT is created by a *Requester* whom decides also the revenue for the execution of the tasks and some other constraints to the execution (i.e. user qualification). Once the creation is completed the HIT is ready to be executed by a *Worker*. A *Worker* is a human being registered to the *MTurk* platform willing to perform simple task in exchange for money, the revenews can be as low as 0.01$ per user per task. Once the whole HIT is completed the *Requester* check the result and if it is satisfied proceed with the payment. All these operation must be performed within the *MTurk* website so there is no need of any software installed.

This system has been *extended* as presented in Little *et al.*, 2010 to create complete algorithm able to use human computation as functions during the execution. Here is an example algorithm implemented in Turkit:

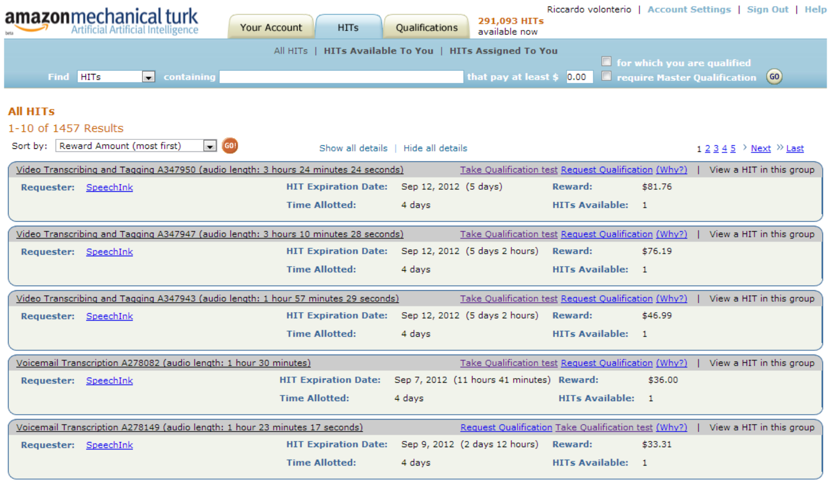

Figure 2.: Web interface of *MTurk*.

```
ideas = []
for (var i = 0; i < 5; i++) {
        idea = mturk.prompt("What's fun to see in New York City?
            Ideas so far: " + ideas.join(", "))
        ideas.push(idea)
}
ideas.sort(function (a, b) {
        v = mturk.vote("Which is better?", [a, b])
        return v == a ? -1 : 1
})
```

Where `mturk.prompt` and `mturk.vote` are human computation task executed on the *MTurk* platform.

ESP is a GWAP developed by Luis von Ahn to perform image their task is to agree on a word that would be an appropriate label for the recognition using the power of humans as described in Von Ahn and Dabbish, 2004. The game rules are simple, once logged in a user will be automatically matched with a random partner. Once matched, they will both be shown the same image, their task is to agree on a word that would be an appropriate label for the image. Similar to the ESP game is Peekaboom another GWAP proposed by von Ahn described in Von Ahn, Liu, and Blum, 2006.

CROWDSEARCHER TODO ???

Distributed

This solution relies on the creation of ad-hoc clients that are able to download and run the code on all the platforms. With this solution the user do not need to go to a website to run the code remotely. The most known implementation of this paradigm is the *FoldIt* game.

Figure 3.: The ESP game.

FoldIt is a puzzle game about protein folding, developed by the University of Washington's Center for Game Science in collaboration with the UW Department of Biochemistry. The objective of the game is to fold the structure of selected proteins to the best of the player's ability. The highest scoring solutions are analysed by researchers, that can determine whether or not there is a native structural configuration that can be applied to the relevant proteins.

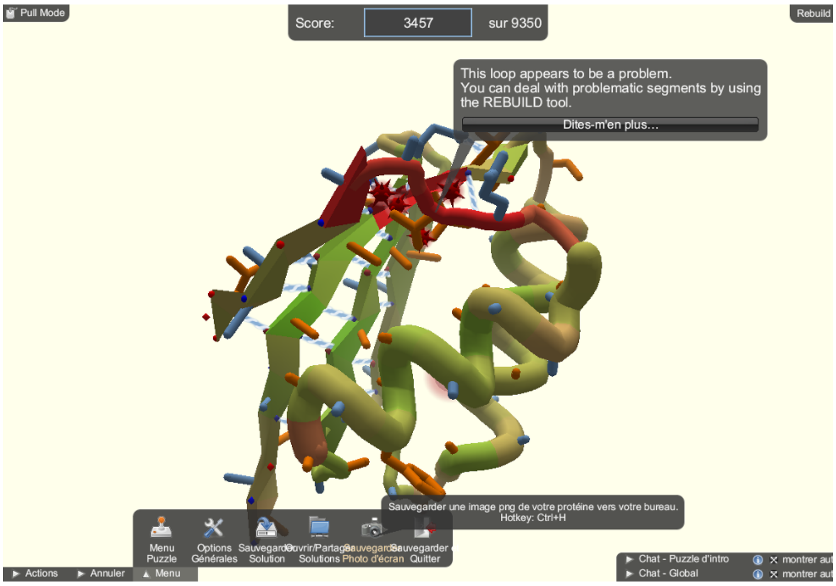

Figure 4.: The FoldIt client.

1.1.2 Automatic computation

Unlike human computation, *automatic computation* aim at executing task, or part of it, in an automatic fashoin, without user interaction. This kind of *distributed computation* leverage on the existence of a *grid* of connected nodes able to perform data intensive calculation.

The platforms that implement these solution use different frameworks for splitting algorithms into atomic operation executable by the nodes. One of these frameworks is MapReduce[2] that, using the core concept of *Divide et impera* can produce highly parallelizable algorithms.

Automatic computation cen be further subdivided accordingly to the will of the user to perform computation on its computer.

Voluntary computing

When a user want to share the computational power of its computer to some project he/she think are worth of it, then might think of using the BOINC system.

Figure 5.: The BOINC logo.

The BOINC system was originally developed to support the SETI@home project, before it became used as a platform for other distributed computing applications. BOINC is an open source middleware system for volunteer and grid computing. It consist on two parts, the backend server, running on linux pletfomrs, and the client, cross-platform, for the end-user.

This piece of software allow a user to connect to the BOINC grid, by doing so a user is allowing the BOINC client to use the idle time of its CPU to perform computation. The client can now download all the necessary data from the chosen project site alonside with the code to run, once all the downloads are completed the BOINC client can run the code and send the results back to the project site.

There are over 40[3] projects that leverage on the BOINC platform to perform computation on different application areas, for example the aforementioned SETI@home project use this framework to search for extraterrestrial intelligence by analyzing the narrow-band radio signal coming from the Arecibo radio telescope.

2 Dean and Ghemawat, 2008.
3 At the time of writing.

Parasitic computing

Parasitic computing[4] is a technique that, using some exploits and ad-hoc code, allow a *malicious* user to use the computational power of the *victim* computer without this being aware. As one can notice parassitic compiting has a strong relationship with *distributed computing*, in fact is a specialization of the general class of *voluntary computing*, where the user is unaware of the execution[5].

This approach was first proposed by Barabási *et al.*, 2001 to solve the NP-complete 3-SAT problem using the existing TCP/IP protocol stack and its error handling routines. The satisfiability problems or "SAT" involves finding a solution to a boolean equation that satisfies a number of logical clauses. For example, $(x_1 \oplus x_2) \wedge (x_2 \wedge x_3)$ in principle has 2^3 potential solutions, but it is satisfied only by the solution: $x_1 = 1, x_2 = 0, x_3 = 1$. Problems problem like the one in the example are known as 2-SAT problem because each clause, shown in parentheses, involves two variables. The more difficult 3-SAT problem is known to be NP-complete, which in practice means that there is no known polynomial-time algorithm which solves it.

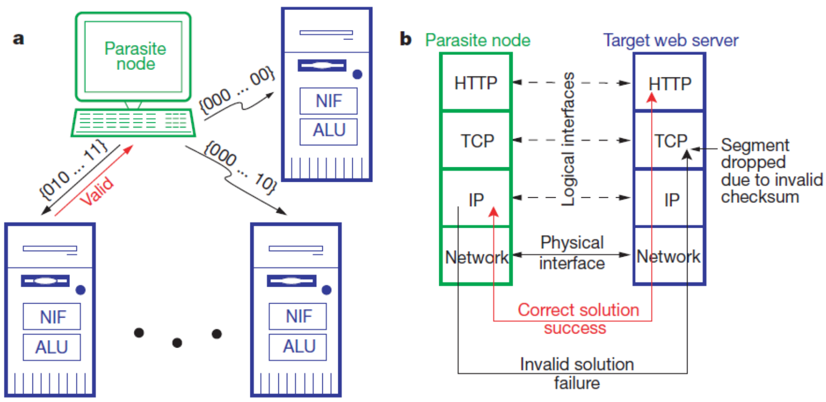

Figure 6.: Schematic diagram of the parasitic computer solving the 3-SAT problem.

The approach proposed in the paper was to perform a brute force attack to guess the right solution of a 3-SAT problem using a parallel apprach as depicted in Figure 6. The parasite node creates 2^n specially constructed messages designed to evaluate a potential solution. These messages are sent to many target servers throughout the Internet. After receiving the message, the target server verifies the data integrity of the TCP segment by calculating a TCP checksum. The construction of the message ensures that the TCP checksum fails for all messages

4 In this thesis we are not covering, neither we are interested in, the ethical or moral implication of using this technique.

5 In *voluntary computing* the user can be unaware of the actual code they are executing, but they are aware of the execution.

containing an invalid solution to the posed SAT problem. Thus, a message that passes the TCP checksum contains a correct solution. The target server will respond to each message it receives (even if it does not understand the request). As a result, all messages containing invalid solutions are dropped in the TCP layer. Only a message which encodes a valid solution *reaches* the target server, which sends a response to the *request* it received.

This approach may seem wrong or at leat not right, with respect to the user, but if one think about it notice how we are always making computation without even knowing. GWAP or application like http://www.google.com/recaptcha are examples of involuntary human computation (as in Table 1). So thay are using the same technique to perform a sort of *parasitic human computing* without complainig about the user will.

To avoid the ethic implication of doing *parasitic computing* a hybrid approach (parasitic/voluntary) can be used. If the user give the permission to run computation on its computer exchange of a return of any type, then we are able to score the best on both approaches. A similar solution was proposed in Karame, Francillon, and Čapkun, 2011. In this paper they propose a microcomputations as micropayments in web-based services. Their solution is to give the user access to online contents (such as newspaper, video, etc.) after performing small JavaScript computation.

PARASITIC JAVASCRIPT as described in Jenkin, 2008 can be considered an enhancement of the solution proposed by Barabási *et al.*, 2001, since using JavaScript and the HTML5 features to their full potential (see 1.2.1) we are able to perform any kind of computation within the browser window/tab. JavaScript offers also a standard platform for the exection of code without the need of any ad-hoc software for each platform. Furthermore all the code executed by a browser run in a sandboxed environment, keeping the user computer safe from any malicuis intent.

1.2 ENABLING WEB-BASED DISTRIBUTED COMPUTATION

Using the web (i.e. the browser) as a platform for distributing and executiong code implies that the available technologies are powerful enough to perform high-level *computation* and real-time *communication*. These are the requirements for evaluating if the web is a suitable platform for code distribution.

COMPUTATION is the key for being able to perform task within the browser. *Computation* can involve any kind of operation on the data, and the data itself can be of any type. For instance creating an application that analyze audio files is relatively simple using standard languages (e.g. C, C++, Java, etc) and until a few years ago was almost[6] impossible to do within a browser, or creating a image manipulation program that runs without external plugins.

HTML5 filled the gap that existed between any "standard" language and JavaScript giving the developers access to all the required APIs needed to create fully functional web-applications. In 1.2.1 are presented all the features, along with some technical details, that have enabled these evolution.

There are also initiatives that aim at simplify the deployment of JavaScript application. Since most of the developers have experience on languages other than JavaScript there is the need of creating application in one language and *cross comiling* it for the web. Projects like *Emscripten* and *Google Web Toolkit* offer the possibility to write the code directly in C,C++ (*Emscripten*) or Java (*Google Web Toolkit*) and compile it into pure, and optimized, JavaScript.

Emscripten, by Mozilla, is an LLVM-to-JavaScript compiler. It takes LLVM bitcode (which can be generated from C/C++ using Clang, or any other language that can be converted into LLVM bitcode) and compiles that into JavaScript. Since it is a compiler it offers miltiple grades of optimization that reduce the size of the JavaScript file and speedup the computation. The website is full of demos of the ported application, including games, 2D/3D game engines, various libraries and also SQLite.

Google Web Toolkit, by Google, is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest, and JavaScript.

COMMINICATION is being empowered, with respect to HTML4, by introducing *WebSocket*, that enable full-duplex data exchange with the server, and also Cross-origin Resource Sharing (CORS) that give the developers the possibility to make Asynchronous JavaScript and XML (AJAX) requests to "foreign" servers (other than `localhost`) without the need of a proxy for forwarding the requests.

1.2.1 HTML5

When speaking of HyperText Markup Language (HTML)5, usually, one is not only focusing on the markup language but on a set of web technologies and specifications strictly related to HTML5. This set of technologies includes the HTML5 specification itself, the Cascading

6 Without using strange interaction between Flash/Silverlight and the browser.

Figure 7.: Official HTML5 logo & unofficial CSS3 logo.

Style Sheets (CSS)3 recomendations and a whole new set of JavaScript APIs. So, first things first, let's point out the differences:

HTML5 refers to a set of semantic tag (like `<footer>`, `<header>`, `<article>`, . . .), media tags (like `<video>` or `<audio>`) and the so called Web Form 2.0 alongside with all the "old" tags inherithed from HTML4. These tags helps developers to give semantics to the website they make, so they (the websites, not the developers) can be better understood by Seach engines or HTML parsers (like those used for reading the site for blind people).

CSS3 refers to the presentation layer of the pages. Here are introduced specification including image effects, 3D transformation, new tag selectors, form element validation, etc. The specifications take care also of the new devices (like smartphones and tablets) giving the user the `media queries` to examine the media (screen, print, aural) and provide different CSS rules.

JS refers to the JavaScript with a new set of API for interacting with the new media elements and other tags, as long as API for concurrent computation, real-time communication, offline storage, etc.

With the advent of HTML5, like any new technology, many problems were resolved and many others have been created. The main issue with using HTML5 is the browser compatibility and browser-specific methods. When browser start implementing some HTML5 draft feature, since is not fully standardized [7], they prevent the pollution the DOM by prefixing the standard method (i.e. `requestAnimFrame` can became `mozRequestAnimFrame` or `webkitRequestAnimFrame`) with a browser specific prefix[8]. This prefixing is particularly common

7 In fact HTML5 (at the time of writing) is not yet standardized, its still a draft. See `http://www.w3.org/TR/html5/`

8 `o`: for Opera, `ms`: for Internet Explorer, `moz`: for Firefox, and `webkit`: for the WebKit based browser (Chrome and Safari)

in the CSS3 where thing becames awful[9].

To avoid browser inconsistency there are plenty of JavaScript frameworks for every purpose. Frameworks like *jQuery* provide a layer of abstraction between browser-specific code and the user, giving developers JavaScript fallbacks for the most common API and additional features not covered by the standard implementation. Other frameworks like *Modernizr* give developers the ability to test if some HTML5 feature is available in the currently used browser and provide a general fallback system for dynamically load polyfills[10].

Now are presented the main HTML5 features to better understand how they can be used in this System.

CANVAS Let's start with the official definition[11]

```
The canvas element provides scripts with a resolution-dependent
bitmap canvas, which can be used for rendering graphs, game
graphics, or other visual images on the fly.
```

So the `canvas` element is basically a *Canvas*, like the name says, where one can *paint* anithing. On top of this, the `canvas` element give the developers access to the underlying raw pixel data. Also in the `canvas` element you can *draw* images taken from a `<img>` tag or a frame taken from a `<video>` tag.

As one can se now we have all the tools we need to perform image analisys or video manipulation within the browser. Obviously there are plenty of JavaScript libraries that facilitate the whole process of filtering or, in general, image manipulation (like Pixastic or Camanjs), other libraries give you the tools to create diagrams or charts on the fly (like Raphaël or Processingjs).

The canvas element also provides a 3D context to draw and animate [12] high definition graphics and models using the WebGL API. This API is mantained by the Khronos Group and is based on OpenGL ES 2.0 specifications. On top of these API there are a lot of libraries[13] made to facilitate development of 3D applications, one of the the most used is the Three JavaScript library, that ca be used for creating and animating 2D or 3D scenes within the canvas element.

WEBSOCKET The WebSocket is an API interface for enabling bi-directional full-duplex client server communication on top of the Transmission Control Protocol (TCP) protocol. It enables real-time communication between clients and servers, allowing servers to **push** data to the clients and obtain *real* real-time content updates.

9 See CSS animation or gradients.
10 A polyfill is a JavaScript library or third part plugin that emulates one or more HTML5 feature, providing websites to have a consistent behaviour.
11 Got from the specs: http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element
12 Animations are not natively supported, must be coded separately.
13 For a reference see http://en.wikipedia.org/wiki/WebGL#Developer_libraries

Like many other HTML5 features on top of WebSocket was built a library that provides easy access to these functionality as long as fall-backs for old browsers. **socket** provide a single entry-point to create a connection to the server and manage the message exchange, providing fallbacks[14] to ensure cross-browser compatibility.

WEBWORKERS A problem of coding load intensive JavaScript application is the single thread nature of the language. Every script runs in the same thread of the browser window/tab, this can lead to some unwanted behaviour (like browser freezing or a warning dialog that that alerts the user).

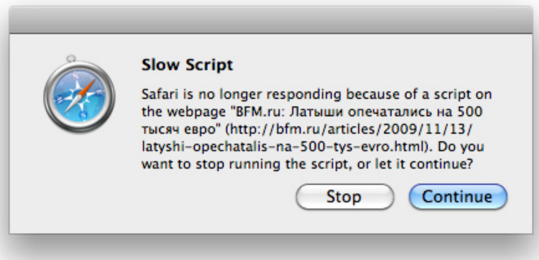

Figure 8.: The slow script dialog.

To solve this problem Jenkin, 2008 proposed a timed-based programming structure that ensures that the code runs without any browser warning or freezing and also offer the developer to tweak the performance of the script by dynamically adjusting the interval between the step execution. This method leaverage on the `setTimeout` function of JavaScript in order to split code into timestep-driven code chuncks to execute. Here is an example of loop translated into a timer-based loop:

while *condition* **do**
  ...do something...
end

procedure STEP
  ...do something...
  if *condition* **then**
    `setTimeout(`*STEP, delay*`)`
  end

Obviously this is not the solution to the problem, its a hack that trick the browser.

WebWorkers offer a simpler solution, they provide a simple, yet powerful, way of creating *threads* in JavaScript. The official definition says:

14 If WebSocket, are not avilable the library can use Adobe®Flash®Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe and JSONP Polling

```
The WebWorkers specification defines an API for running scripts
in the background independently of any user interface scripts.
This allows for long-running scripts that are not interrupted
by scripts that respond to clicks or other user interactions,
and allows long tasks to be executed without yielding to keep
the page responsive.
```

The core concept behind WebWorkers is the `Worker`. A `Worker` is a piece of JavaScript code that runs in parallel to the main thread and is able to send and recieve messages (just like normal threads).

STORAGE When web developers think of storing anything about the user, they immediately think of uploading to the server. HTML5 changes that, as there are now several technologies allowing the app to save data on the client device.

HTML5 support a number of storage techniques able to store data within the browser to be accessed later, here is a simple list withe the principal features:

WEB STORAGE is a convenient form for offline storage, it uses a simple key-value pairs like any JavaScript `Object` stored in the browser accessible every time the site need to use them.

WEB SQL DATABASE is an offline SQL database, usually implemented using SQLite, a general-purpose open-source SQL engine.

INDEXEDDB is a nice compromise between Web Storage and Web SQL Database. Like the former, it's relatively simple; and like the latter, it's capable of being very fast. It uses the same mapping as *Web storage* and index certain fields inside the stored data.

FILESYSTEM API as the name says offer the ability to manipulate the file system of the host.

OFFLINE STORAGE In this category falls `application cache`. The `application cache` is controlled by a plain text file called a `manifest`, which contains a list of resources to be stored for use when there is no network connectivity. The list can also define the conditions for caching, such as which pages should never be cached and even what to show the user when he follows a link to an uncached page.

If the user goes offline but has visited the site while online, the cached resources will be loaded so the user can still view the site in a limited form. Here is a simple cahce file:

```
CACHE MANIFEST

# This is a comment

CACHE:
/css/screen.css
/css/offline.css
/js/screen.js
/img/logo.png
```

```
http://example.com/css/styles.css

FALLBACK:
/ /offline.html

NETWORK:
*
```

1.2.2 WebCL

With the advent of General-purpose computing on graphics processing units (GPGPU), the spreading of multicore CPUs and multiprocessor programming (like OpenMP) we can see emerging an intersection in parallel computing. This intersection is known as **heterogeneus computing**. There are initiatives aimed at enabling numeric calculation, even complex, on the web client. Open Computing Language (OpenCL) is a framework for heterogeneus computing and Web Computing Language (WebCL) is a porting of this technlogy to the web.

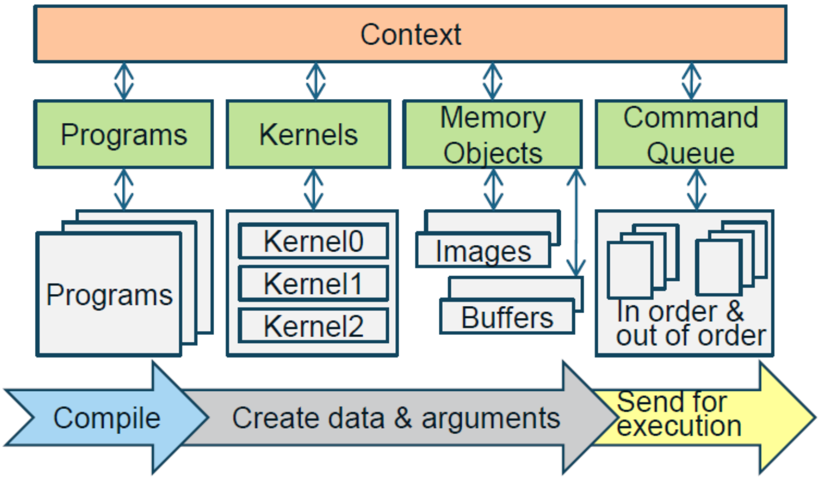

Figure 9.: OpenCL execution flow.

OpenCL uses a language based on C99[15] for writing *kernels*, functions that actually execute on OpenCL devices. Here is the list of action performed to run code on OpenCL enabled computers:

1. Query host for OpenCL devices.
2. Create a context to associate OpenCL devices.

15 A programming language dialect for the past C developed in 1999 (formal name ISO/IEC 9899:1999)

3. Create programs for execution on one or more associated devices.

4. From the programs, select kernels to execute.

5. Create memory objects accessible from the host and/or the device.

6. Copy memory data to the device as needed.

7. Provide kernels to the command queue for execution.

8. Copy results from the device to the host

The main focus when building high-end web-application like 3D games is responsiveness. Altough JavaScript can be optimized and parallelized (see 1.2.1) it cannot be fast as an application software, because JavaScript must be interpreted by the browser and then executed as machine code. WebCL provide an easy framework for building and running machine code in parallel directly from the browser.

The availability of all those technical API give the possibility to create a system capable of performing any *Human* or *Automatic* computation task without the need of external plugins. We used all the features of HTML5 for the computation side of the System, WebSocket are used for real-time task monitoring and with the CORS are used within the task to request any external data that a *Requester* need.

2 THE MODEL

In this chapter, we define the *architectural model*for our system and the reference infrastructure supporting this model. The *architectural model*is the data model on which the single components of the system are build upon. It describes the components that interact each other during the task lifecycle and embodies also the requirements and the features of the system as expressed in the introduction.

Concerning the data model we have subdivided it in 3 parts, this subdivion is made to better distinguish each of the 3 main steps used in every distribution system in order to create, distribute and process the data. **??** gives an overview of the *architectural model*that is composed by:

THE DATA MODEL: describes the data structure used to create this system.

THE ARCHITECTURAL MODEL: describes the reference architecture of the sytem.

THE EXECUTION MODEL: focuses on the execution model of the task.

PLUGGABLE STRATEGIES: here are provide some example of strategy that can be plugged to the system.

2.1 DATA MODEL

In this section, we define the *Data model* of the System. All the components used in the *Architectural model* are based on this model and all it's. As can be seen in Figure 10 the *Data model* is composed of 5 parts that together compose a basic human/automatic computation platform.

THE WORKFLOW contains the all the information about a *Work*, including how is composed, in terms of Task, and the relations that intercour between two or more Task.

THE TASK DATA MODEL contains the actual data structure for each Task or Work.

THE TASK MODEL contains all the data assocated to a Task or Work.

THE TASK EXECUTION focus on the actual execution step for each Task, providing information on wich implementation to use according to a specific Performer.

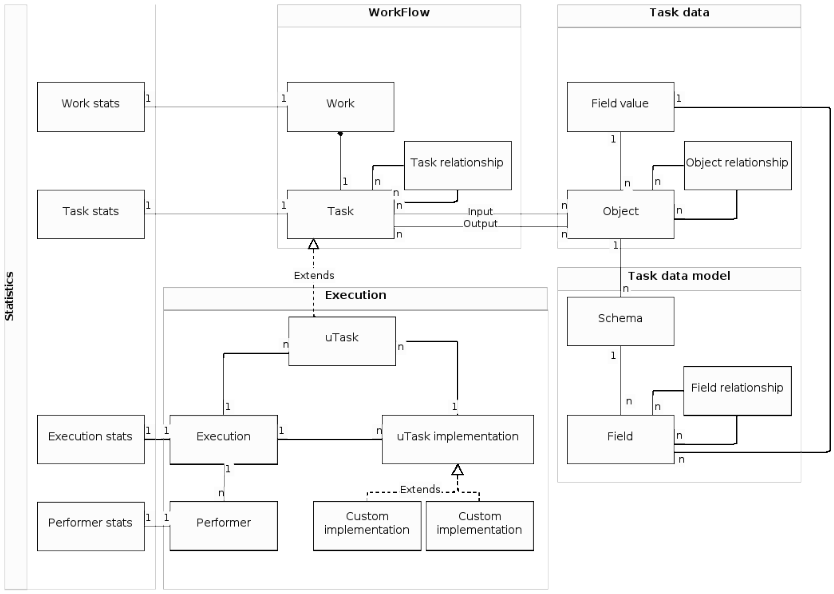

Figure 10.: Data Model.

STATISTICS provide all the statistics associated to the Work lifeclycle, from the creation to the execution.

Figure 11.: Conceptual organization of Work, Task and µTask.

2.1.1 WorkFlow

The WorkFlow embodies all the data associated to the *flow* of Task that need to be executed in order to complete a **Work**. As an example consider image tagging with tag validation as a *Work*, to complete this we need to perform a few steps:

1. Let the user tag an image

2. Gather the result tags associated to an image
3. Let some different validate the tags for the image
4. Gather the validation result
5. Output the validated tag

As one can notice this *Work* is subdivided in two main steps, the first when we gather a seet of tag from the users, the second when these tags are validated. These tow steps are human computation Task that are part of the *Work* of image tagging and validation.

The Work

The Work represent the main goal of the *Requester* and it's defined by:

- A **Name** that identifies the Work.
- **Contraints** definied by the *Requester* used to prioritize the Work among the others (e.g. Due date, Perormer skills, Max execution time).
- **Input** data, defined by a *Schema*. To keep the model as general as possible no assumption are made on the *Schema* type (relational, graph, etc.).
- **Output** data, defined as an extension of the input schema (sharing the same schema type).
- A set of **Task**. Their orchestration is made at design time, specifing a *Flow*

The Flow

The Flow describes how the *Task* are connected (organized) to fullfill the rewuirements of a *Work*. In a Flow we can use control structures and *Variables*. The control structures availabe are:

SEQUENCE: represent the normal flow of an application where one operation is executed after the previous is completed.

CHOICE: give the possibility to made choice according to one, or more, *Variables*.

LOOP: Allow to execute some steps multiple times, according to a predefined value or a *Variable*.

PARALLEL: the steps of the flow are not executed in *Sequence*, allowing the parallelization of some steps.

The *Variables* can be predefined or computed during the Flow execution to change the behaviour of the Flow itself. For instance a variable can decide wheter to execute a loop or not or even decide what control sequence to use in the next steps.

The Task

The Task is the kernel of the whole system, it represent an activity, tipically focusing on a purpose. A Task is characterized by:

- A **Name** that identifies the Task.
- **Input** data, with a *Schema*. Usually the *Schema* of a Task is a projection of the *Schema* of a *Work*.
- **Output** data, with a *Schema* that is an exetension of the input *Schema*.
- A Task **type**[1] defining, at abstract level, what kind of data manipulation will be performed by a Task. These categorization are taken from Bozzon, Brambilla, and Ceri, 2012, here are a few:
  - Like
  - Order
  - Classify
  - Add
  - ...

  Each Task type is defined by:
  - I/O relationship, defining, at abstract level, how the Task transforms the data and the schema.
  - A default implementation.
- A **Status** encoding the current state of the Task. A Task, can have only one of the following statuses at point of its lifecycle:
  - *Planning-Input*: the Task has been created, have a *Schema* and *Object data* associated and a defined Task *type*.
  - *Planning-μTask*: a set of μTask has been associated to the Task.
  - *Planning-Assignment*: a set of *Performers* has been selected to execute the μTask.
  - *Wait*: Task planned, μTask ready for executiuon.
  - *Running*: μTask are running.
  - *Ended*: all the μTask have completed their execution.
- A set of **Subscriber**s able to recieve updates on the Task execution.
- A set of **Execution constraints** used for priortizing the Task among others or to modify the standard behaviour of the task to fullfill these constraints. The availbale constraints are:
  - Maximum execution time

1 An assumption is made to make the list fit all the possible abstract task our System is able to handle.

  - Due date
  - TODO others

- **Configuration data**, provided as JavaScript Object Notation (JSON). For instance the classes we want to use in a calssification Task.
- μ**Task**s TODO????.
- An **Aggregation** function, in charge of collecting the μTask results and generating the Task output.
- μ**Task planning** strategy, in charge of defining how many μTask create for a given Task and associate the right portion of input data to such μTask. For example total disjunction, redundancy, partial overlap, etc.
- **Performer assignment** strategy able to assign *Performers* to μTask. Some strategies can be: manual, random, most reliable, etc.
- μ**Task implementation** strategy in charge of routing the correct μTask implementation for each μTask execution. The routing can be done according to the *user-agent* (e.g. Browser) or to the *user profile* or even *fixed* for all.
- A **Task planning** wich embodies the funcitonalities of μ*Task planning* strategy, *Performer assignment* strategy and μ*Task implementation* strategy deciding the logic behind the invocation of those strategies.
- A **Task control** strategy able to control the status of the Task and if needed perform corrective actions.
- An **Emission policy** specifing wich *Subscriber* need to be notified of a Task change in *Status*.

2.1.2 Task Data Model

The Task Data Model contains all the data related to the description of the *Schema* of the data of a Work/Task. This model resembles a the *metadata* of the actual data of the Work/Task defining all the fields and their type according to a *Schema*.

The Schema

The Schema contains all the information related to the data structure of a Work/Task, and its defined by:

- A **Name** that identify the Schema among the others.
- A set of **Field**s that compose the actual schema of the data.
- A list of **Object**s associated to this *Schema*, these objects represents the actual data associated to the Work/Task.

The Field

The Field represent the definition of a Field in a *Schema*, with all the properties that define if the field is calculated, derived, etc. The Field is defined by:

- A **Name** that identify the Field.
- A **type** definig the type of the data that this field contains. i.e. `string`, `number`, etc.
- A set **related fields** that defines how this field is composed. TODO ???
- A **relation** that specifies which type of relation occurs among the *related fields*.
- The list of **data** of in terms of the associated *Field values*

2.1.3 Task Data

The Task Data contains the actual data instance for each Task, defined in the *Schema*. All the data are contained in a *Object* that represent the the instance of the Task data (e.g. A row of the Task Data table). Due to the metadata-like model of the System, we need to store all these information into a separate table and use the *Object* as a simple reference table.

The Object

The Object contains the actial data value as reference to field value instances; it's composed by:

- A **Name**.
- A list of field values **data**.
- TODO ??

The Field Value

The Field Value contains the data associated to a particular field, defined in the metedata model. It's defined by:

- An **Object** that define tho what *Object* they refer to.
- The **Field** to wich the datta belongs.
- TODO ???

2.1.4 Task Execution

The Task Exection embodies all the information relative to the actual exection of the code. The majority of these data belongs to the **Execution layer** thus can be phisically located into another piece of software in charge of the execution of the code.

µTask

The µTask is the implementation of a Task that insist on a specific subset of data of the Task. Can be also considered as an activity assigned to one or more Performers. It is defined by:

- A **Name**.
- A list of **Execution**, representing the actual activities performed by a *Performer*
- A set of **Execution constraints**.
- **Input** data, as a subset of the Task input data.
- **Output** data, with the same schema as the related Task output data.
- A list of **Properties**, defined as name-value pairs, having domain specific meaning.
- One or more µ**Task implementation**.

The Execution

The Execution is related to one *Performer* that need to compute a µTask. An Execution is defined by:

- a **Status** telling the status of the execution of the µTask, the available statuses are:
  - *running*
  - *suspended*
  - *idle*
  - *ended*
- A set of **Execution data** provided as JSON object.
- A µTask **Implementation**

µTask implementation

The µTask Implementation is the actual application logic and presentation delivered to a *Performer* to run a µTask. The System provides a default implementation according to the Task type, in addition, a *Requester* can specify one or more Custom implementations, in order to obtain more control over the execution process.

Performer

A Performer is a human being able to execute one or more µTask. The performer is characterized by a set of attributes such as:

- A **Name**

- **Demographic** information
- **Performance** information
- **Trustworthiness**
- **Social properties**

2.1.5 Statistics

This Statistics model contains all the information related to the Task profiling and statistics used to tweak the performance of a Task, or used by components (like the *Task controller*) to take decision on the Task flow. In this model are contained data about *Work*, *Task*, μ*Task*, *Performer*, etc. The data contained in these tables can be:

- **Creation date**
- **Total execution time**
- **Average number of *Performer*s/h**
- **Last execution**
- etc.

2.2 ARCHITECTURAL MODEL

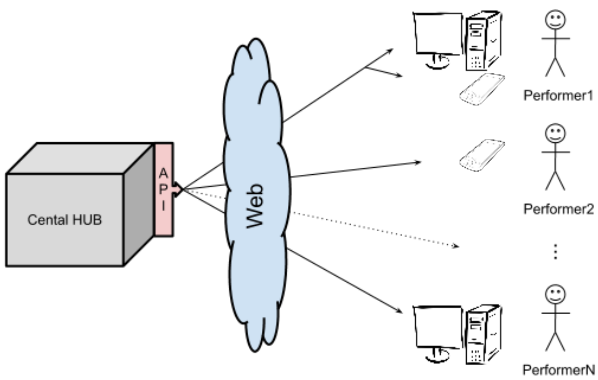

Figure 12.: Reference architecture.

The system use as a reference architecture the one depicted in Figure 12. Here we have a centralized hub that *defines* and *distribute* the workload, a pletora of clients with their browsers and the users. The clients of this model are all coherent and transparent to the execution of the code, wich is distributed to the end-user according to the platform they are using. As you can see the structure is almost the same as any other task distribution platform, the strenghts of this system are in the characterization of the actors in the system.

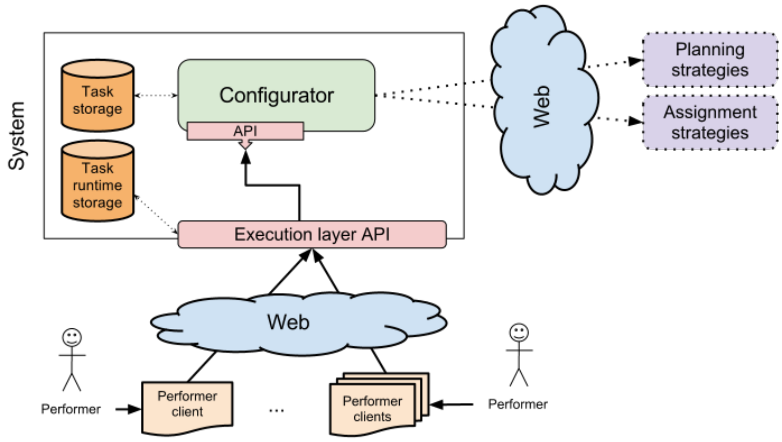

Figure 13.: Specialized architecture.

The reference model in Figure 12 has been customized to meet our needs of flexibility and pluggability, so we introduced a *configurator* and a *execution layer* in the central hub. These are the components that allow our system to cover all the dimension presented in Table 1.

The **configurator** is in charge of defining and configuring a task in the system, allowing the *requester* to add hooks to exeternal resource in order to manage the assignment cycle and the planning strategy.

The **execution layer** provides useful API for managing the µTask and communicate with the *configurator*.

2.2.1 Configurator

The **Configurator** is the component in charge of the task lifecycle management. The principal functionalities offered by the **Configurator** are:

- Allow the **creation** of a Task, also at abstract level, using either the API or the built-in UI.

- Allow a *Performer* to **execute** the Task using a standard non configurable UI, provided as-is for each Task type.

- Allow to **request information** about a Task, the information that can be requested includes:
  - Retrieve the list of µTask associated with a given Task
  - Post the result of the execution of a given µTask
  - Notify about the completion of a Task or µTask

Alongside these main functionalities it offer a *Requester* the ability to monitor the state of a Task and/or a µTask.

2.2.2 Execution layer

This component is in charge of managing the µTask implementation for each Task or for each µTask. The implementations have a fallback behaviour so, if a custom µTask implementation is not present then the system search for a custom Task implementation, if this is missing then the built-in implementation is used. On top of this fallback system the component offer the possibility to create code for a target platform.

The **Execution layer** offer the following funcionalities:

- Allow a *Requester* to configure the implementations associated to a Task and/or a µTask. The implementations are configured specifing the target platform (mobile, desktop, tablet, ...) and the executable resources used by the implementation (i.e. HTML, CSS and JS files). Wich implementation to use is configured later in the *Planning* step.
- Create a layer of abstraction between the implementation and the Configurator, creating a sandboxed environment where the implementation can run and communicate with the Configurator.
- Allow the *Performer* to execute a specific µTask implementation.

2.2.3 Task storage & task runtime storage

These are the storage areas where we put all the data associated with the Task. We used two separated storage area in order to keep the runtime configuration separated from the abstract configuration data of the Task.

The task runtime storage contains all the ad-hoc code written by the *Requester* for each platform. **This code can be reused by the other *Requester* to execute the same task (for example image tagging).**

2.2.4 Performer & Performer client

The *Performer client* represents the platform (like desktop or mobile) on wich a *Performer* executes the Task implementation. The *Performer client* make use of the *Execution layer API* to retrieve the correct implementation, communicate the status during the exection of a µTask and post the result of the execution. The *Performer* is the actual user that is using the *client*.

2.2.5 Planning strategies

Any third-part component in charge of the creation and management of the µTask associated with a Task. During the Task configuration step the *Requester* decide when this external component need to be called. The *Planning strategy* can be called only once, for example during the task creation, or ca behave like an handler to the µTask ended event, in this case the *stategy* is able to decide wheter is necessary to spawn other µTask execution to fullfill the requirements.

2.2.6 Assignment strategies

Any third-part component used to associate µTask to Performers. The binding can leverage on some skill of the Performer, for example a stategy can be: associate this set of µTask to Performers skilled German translation, and all the other to any Performer. As for the *Planning strategies* this component can be invoked only once or in response to some events (like µTask ended or created).

2.3 EXECUTION MODEL

Figure 14.: Representation of the Task execution flow.

Figure 14 represents the flow of the execution of a Task during the execution. As one can notice the flow is almost straightforward except the initial part when the *Task control* strategy tweak the execution to fulfill some predefined requirements.

The System is able to operate in different scenarios, according to the logic implemented in the *Task planning*, the *Task control* strategy may change the flow of the execution. In Table 2 are presented the different execution scenarios that the system is able to handle.

Table 2.: Task planning vs. Task Assignment.

	Static	**Dynamic**
Static	2.3.1	2.3.2
Dynamic	2.3.3	2.3.4

2.3.1 Static execution

This scenario of execution represent the simplest use-case possible, where the *Task planning* is executed only once at creation time and the μTask are planned and assigned only once. In this `mode` the *Task control* have only the role of controlling if the *constraints* are verified. Here is a list of the operation that the *Task control* strategy must perform:

- **Stop** a Task if constraints are met.
- **Invoke** the *Aggregation function* at the end of the Task execution.
- **Notify** the *Subscribers* about the Task execution.

2.3.2 Static μTask planning & Dynamic assignment

In this scenario the μTask are planned once at creation time, but the assignment is performed dynamically. In this scenario the *Task control* strategy invokes the *Performer assignment* strategy to assign *Performers* to μTask, ensuring that the constraints are verified. The *Task control* strategy can also decide to reassign *Performer*s to μTask, while ensuring constraints validity. In this scenario the *Task control* strategy:

- **Stop** a Task if constraints are met.
- **Invoke** the *Aggregation function* at the end of the Task execution.
- **Notify** the *Subscribers* about the Task execution.
- **Invoke** the *Performer assignment* strategy to bind μTask to *Performer*s.

2.3.3 Dynamic μTask planning & Static assignment

In this scenario the *Performer* assignment are performed at creation time and the μTask planning is performed during the flow of the execution. As one can notice this can lead to consistency problem due to the missing μTask during the binding step. Since this scenario can lead to consistency problems it must be used with care with respect to the others. To avoid problem we suggest to use simple *Performer* assignment such as the `fixed` one, using this strategy we do not have to take care of the consistency. Summing up, the Task Control Strategy:

- **Stop** a Task if constraints are met.

- **Invoke** the *Aggregation function* at the end of the Task execution.
- **Notify** the *Subscribers* about the Task execution.
- **Invoke** the *Task planning* strategy to *re-plan* μTask or **Create** new μTask

2.3.4 Dynamic μTask planning & Dynamic assignment

In this scenario all the assignments are performed dynamically. Here the μTask can be associated either at creation time or during the flow of the execution. The same stands for the *Performer* assignment, this can be done at any time, i.e. *Performer*s can be assigned only upon the request of a Task execution. Summing up, the Task Control Strategy:

- **Stop** a Task if constraints are met.
- **Invoke** the *Aggregation function* at the end of the Task execution.
- **Notify** the *Subscribers* about the Task execution.
- **Invoke** the *Performer assignment* strategy to bind μTask to *Performer*s.
- **Invoke** the *Task planning* strategy to *re-plan* μTask or **Create** new μTask

2.4 PLUGGABLE STRATEGIES ASSIGNMENT

In this section are covered the pluggable strategies that can be *replaced* by the *Requester* during the creation of a *WorkFlow*, first we present the standard implementation in the System, then we give an overview on the possible custom strategies that can be replaced.

2.4.1 Built-in strategies model

Here are presented the models of the default implementation for the pluggable strategies. These default models are quite flexible to allow the creation of most of the common Task that need a distributed approach, but other distributed human Task, like GWAP, must have a direct control over the whole execution flow.

μTask planning strategy

μTask planning strategy is a pluggable logic focused on the organization and spawning of the μTask in order to execute a Task. A Task planning strategy is defined by:

- A set of **Constraints** that rule the execution.
- A **Planning policy** that can be defined at:

DESIGN TIME: the assignment is made at design time during the creation phase. After the planning is done it can be modified only

DYNAMIC: the planning is done at least once, using a provided set of input *Object*s. The planning can be further invoked due to:

- *Variations* in the state of the Task. i.e. an object can be reassigned to another μTask.
- *Addition* of new *Object*s through the API.

Note that the addition of new μTask can be performed using the API but usually do not involve the invocation of a μTask planning strategy.

This strategy produce as output a set of μTask with the corresponding *Object*s.

Performer assignment strategy

The Performer assignment strategy is a pluggable logic devoted to the assignment *Performers* to μTask. A Performer assignment strategy is composed by:

- A set of **Constraints**.
- A list of **routes** that, by matching the description of a *Performer*, decide if a μTask can be assigned to a *Performer*.
- An **Assignment policy** that can be:

  ONE-SHOT: the assignment is performed according to a prdefined number of *Performer*s and μTask.

  DYNAMIC: the assignment is performed at least once and can be invoked multiple times later according to *Variables* that can change over time.

μTask implementation strategy

μTask implementation strategy is a pluggable logic in charge of selecting a suitable μTask implementation for an *Execution*. A μTask implementation strategy is characterized by:

- A set of assignment **Contraints**.
- A list of **routes** that, by matching the description of an *Execution*, decide if a μTask can be assigned to an *Execution*.
- An **Assignment policy** that can be:

  STATIC: the assignment is performed according to a prdefined number of *Performer*s and μTask.

  DYNAMIC: the assignment is performed at least once and can be invoked multiple times later according to *Variables* that can change over time.

Task planning strategy

Task planning strategy embodies the functionalities of a μTask planning strategy and of a Performer Assignment strategy, deciding the logic by which the two strategies should be invoked.

Task control strategy

The Task control strategy is a pluggable logic devoted to verifing the status of a Task, possibly against the assigned constraints. The logic can be executed:

- **Once** when the Task ends.
- According to a **temporal schedule**.
- Every time a μTask is **executed**.

The corrective actiona available to the Task controller are:

- The **re-planning** of the task, also with the creation of new μTask.
- The **re-assignment** of μTask to *Performer*s.
- **Delete** of executed μTask.
- **Change** the properties of an executed μTask.
- **Re-execution** of the entire Task.
- **Halting** the Task.
- Etc.

Aggregation function

An Aggregation function is a pluggable logic devoted to the summarization of the results of several μTask aimed at creating the final output data of a Task. Examples of aggregation functions are Sum, Avg, MajorityAgreement, etc.

Emission policy

The Emission policy is a pluggable logic in charge of notifing the *Subscribers* about the status of a Task. This logic can be executed:

- **Once** the Task ends.
- According to a **temporal schedule**.
- Every time a task is **executed**.

2.4.2 Custom strategies

TODO ???

Example 1

TODO ???

Example 2

TODO ???

3 THE USE-CASES

This chapter will cover the use-cases used to test the System. These use-cases represents the principal scenarios where there is a need of a distributed platform to spread the computation, or the computation itself is distributed (like the GWAP). These use-cases are also chosen to stess the matrix in Table 1, in particular the *voluntary* part, because the *involuntary* part can be implmented straightforward.

At the end of every use-case will be presented a benchmark/metric, if available, with the corresponding available tools.

3.1 AUTOMATIC

This use-case is the implementation of the *voluntary-automatic* scenario presented in the matrix Table 1.

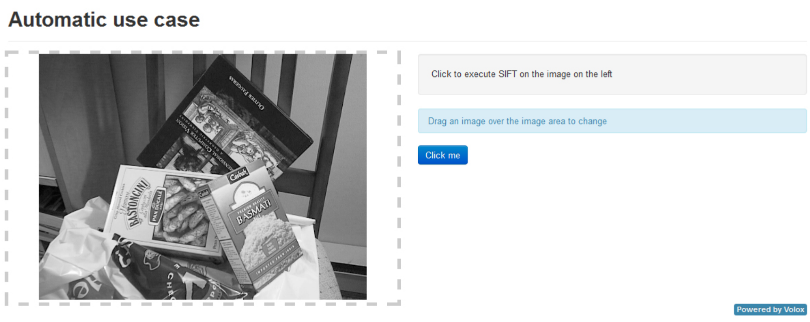

Figure 15.: Interface of the automatic use-case.

For the Automatic use case we choose to implement a widely used feature detection algorithm, the Scale-Invariant Feature Transform (SIFT). To perform this load intensive algorithm we used the power of the WebCL framework to greatly speedup the computation.

In order to build a working example of the algorithm we started with the creation of an abstraction layer over the WebCL raw implementation, then we created a small *MultiMedia* library able to compute the needed operation on the images using our *abstraction layer* eventually we implemented the algorithm in the *MultiMedia* library.

THE ABSTRACTION LAYER is in charge of the communication with the raw Nokia WebCL framework as well as creating a stateful object capable of managing of the I/O data for a WebCL *kernel*.

THE MULTIMEDIA LIBRARY is used to perform the operation required by the algorithm (such as `blur`, `scale`, `RGB to gray` etc.) either using WebCL or the built-in HTML5 functions.

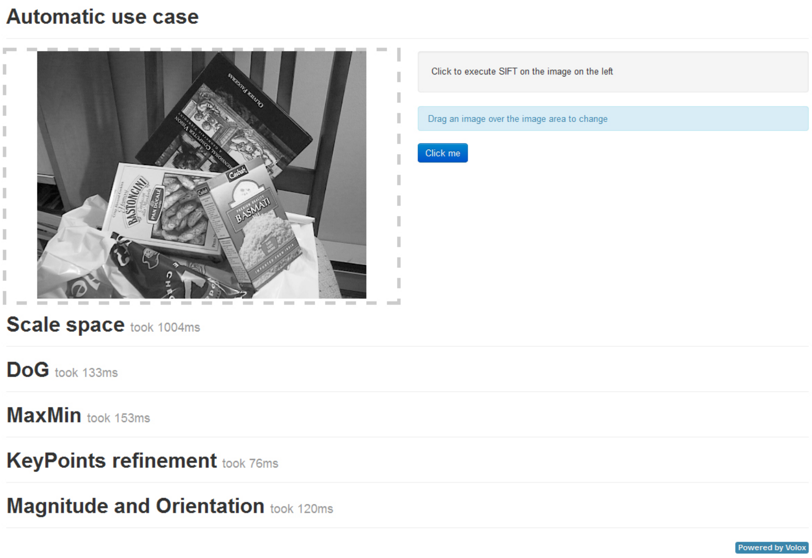

Figure 16.: Intermediate results of the algorithm.

THE ALGORITHM is composed of 4 sequential steps:

SCALE-SPACE EXTREMA DETECTION: in this step we generate the so called scale space representation of the image. In order to do this we need to convolve the image at different scales with a varing guassian kernel and then the difference of successive blurred images are taken. This step produce the Difference of Gaussian (DoG) images, the first *Keypoints* are identified as local minima/maxima of the DoG image across scales.

KEYPOINT LOCALIZATION: in this step the *keypoints* are filtered to remove unstable points and keep only the good ones. This step can be further subdivided into 3 stages

- *Interpolation* of nearby data for accurate position.
- *Discarding* low-contrast keypoints.
- *Eliminating* edge responses.

ORIENTATION ASSIGNMENT: in this step foe each keypoint is assigned an orientation and a magnitude. This step is used to achieve *invariance rotation*.

KEYPOINT DESCRIPTOR: this step generate the descriptors for the remaining keypoints.

Figure 17.: SIFT result comparison with the reference data.

Benchmark/Metric

Since the purpose of this use-case is the feasability of high load computation on the user browser, the implementation of the algorithm has not been optimized. The preformance of this implementation are not comparable to the existing implementation in C/C++, but we can leverage on the parallelism of the whole system to obtain usable results.

Nevertheless in our test cases we obtained the following results:

Table 3.: SIFT performace.

Image size	ScaleSpace + Dog	Keypoints detection	Total time
400x360	1130ms	310ms	1500ms
400x360	1130ms	310ms	1500ms
400x360	1130ms	310ms	1500ms
400x360	1130ms	310ms	1500ms

3.2 HUMAN

Dato un testo disambiguarlo usando YAGO (AIDA, https://d5gate.ag5.mpi-sb.mpg.de/webaida/), EntityPedia?, e altri *Modernizr*

Benchmark

3.3 HYBRID (AUTOMATIC+HUMAN)

This use-case is the implementation of an *hybrid* use-case that embodies both the automatic and the human scenario. In the matrix at Table 1 this use case is placed between the human and the automatic, voluntary, scenario.

Figure 18.: Interface of the hybrid use-case.

This use-case has the purspose of *detecting faces* in a picture, to accomplish this task are used an automatic face recognition algorithm[1] plus a human interacition that has the double purpose of validating the algotithm result and detect the missing faces in the image.

This scenario is implemented in 2 steps, in the first step we run the algorithm for detecting the faces (this is the *automatic* scenario), in the second step we present a subset of detected faces to the user asking to find the missing faces. We need to remove some result from the set in order to check the trustworthiness of the user.

At the end of the execution we obtain a set of automatically detected faces plus all the faces detected by the user. By intersecting this two set we could improve the performance of our algorithm by adding the faces it was unable to find to the training set.

Benchmark/Metric

With this hybrid approach we are able to stress all the matrix in with a single example. The most similar approach to this solution is GWAP.
TODO ???

1 Available at `https://github.com/liuliu/ccv/tree/unstable/js` with a demo application here: `http://liuliu.me/ccv/js/nss/`

4 IMPLEMENTATION AND EVALUATION

4.1 ARCHITECTURE

4.2 PERFORMANCE COMPARISON???

A CONCLUSION AND FUTURE WORKS

ACRONYMS

WebCL Web Computing Language

The WebCL working group is working to define a JavaScript binding to the Khronos OpenCL standard for heterogeneous parallel computing. WebCL will enable web applications to harness GPU and multi-core CPU parallel processing from within a Web browser, enabling significant acceleration of applications such as image and video processing and advanced physics for Web Graphics Library (WebGL) games.

SIFT Scale-Invariant Feature Transform

SIFT is an algorithm in computer vision to detect and describe local features in images.

OpenCL Open Computing Language

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPU, GPU, and other processors. OpenCL includes a language (based on C99) for writing *kernels* (functions that execute on OpenCL devices), plus APIs that are used to define and then control the platforms. OpenCL provides parallel computing using task-based and data-based parallelism.

WebGL Web Graphics Library

WebGL is a cross-platform, royalty-free API used to create 3D graphics in a Web browser. Based on OpenGL ES 2.0, WebGL uses the OpenGL shading language, GLSL, and offers the familiarity of the standard OpenGL API. Because it runs in the HTML5 Canvas element, WebGL has full integration with all DOM interfaces.

CORS Cross-origin Resource Sharing

Cross-origin resource sharing (CORS) is a web browser technology specification which defines ways for a web server to allow its resources to be accessed by a web page from a different domain. Such access would otherwise be forbidden by the same origin policy. CORS defines a way in which the browser and the server can interact to determine whether or not to allow the cross-origin request. It is a compromise that allows greater flexibility, but is more secure than simply allowing all such requests.

Rich Internet Applications (RIA) are web-base application taht have many of the characteristics of desktop application software.

HIT Human Intelligent Task

TCP Transmission Control Protocol

JSON JavaScript Object Notation

AJAX Asynchronous JavaScript and XML

HTML HyperText Markup Language

CSS Cascading Style Sheets

BOINC Berkeley Open Infrastructure for Network Computing

GWAP Game With A Purpose

GPGPU General-purpose computing on graphics processing units

SETI@home Search for Extra-Terrestrial Intelligence *at* home

SETI@home is an Internet-based public volunteer computing project employing the BOINC software platform, hosted by the Space Sciences Laboratory, at the University of California, Berkeley, in the United States. Its purpose is to analyze radio signals, searching for signs of extra terrestrial intelligence, and is one of many activities undertaken as part of SETI.

DoG Difference of Gaussian

Essential bibliography

Barabási, A.L. *et al.*

2001 "Parasitic computing", *Nature*, 412, 6850. (Cited on pp. 7, 8.)

Bozzon, A., M. Brambilla, and S. Ceri

2012 "Answering search queries with CrowdSearcher", in *Proceedings of the 21st international conference on World Wide Web*, ACM. (Cited on p. 20.)

Dean, J. and S. Ghemawat

2008 "MapReduce: Simplified data processing on large clusters", *Communications of the ACM*, 51, 1. (Cited on pp. xi, xii, 6.)

Group, Khronos OpenCL Working *et al.*

2008 "The opencl specification", *A. Munshi, Ed.*

Jenkin, N.

2008 "Parasitic JavaScript". (Cited on pp. 8, 12.)

Karame, G.O., A. Francillon, and S. Čapkun

2011 "Pay as you browse: microcomputations as micropayments in web-based services", in *Proceedings of the 20th international conference on World wide web*, ACM. (Cited on p. 8.)

Law, E. and L. Ahn

2011 "Human computation", *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 5, 3. (Cited on p. 3.)

Little, G. *et al.*

2010 "Turkit: human computation algorithms on mechanical turk", in *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, ACM. (Cited on p. 3.)

Quinn, A.J. and B.B. Bederson

2011 "Human computation: a survey and taxonomy of a growing field", in *Proceedings of the 2011 annual conference on Human factors in computing systems*, ACM, pp. 1403–1412.

Turing, A. M.

1950 *Computing Machinery and Intelligence*. (Cited on p. 3.)

Von Ahn, L. and L. Dabbish

2004 "Labeling images with a computer game", in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, pp. 319–326. (Cited on p. 4.)

Von Ahn, L., R. Liu, and M. Blum

2006 "Peekaboom: a game for locating objects in images", in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, ACM, pp. 55–64. (Cited on pp. xi, 4.)

WebGL-OpenGL, ES

2011 "2.0 for the Web", *Verkkodokumentti< http://www. khronos. org/webgl/>. Luettu*, 16.

Online resources

Emscripten 2012 , http://emscripten.org/. (Cited on p. 9.)

FoldIt 2012 , http://fold.it/. (Cited on pp. xi, 4, 5.)

Google Web Toolkit 2012 , https://developers.google.com/web-toolkit/. (Cited on p. 9.)

jQuery 2012 , http://www.jquery.com/. (Cited on p. 11.)

Modernizr 2012 , http://modernizr.com/. (Cited on pp. 11, 35.)

MTurk 2012 , http://www.mturk.com/. (Cited on pp. xi, xii, 3, 4.)

Socket.io 2012 , http://socket.io/.