Riccardo Volonterio

A framework for web-based human and machine 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.

1 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 *Amazon's Mechanical Turk*. 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 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` $\neq$ `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	*Amazon's Mechanical Turk*
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

1 Not web-based, but using standalone clients.

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

2 THE BACKGROUND

In this chapter are presented the main fields in which a *A framework for web-based human and machine computation* falls, providing a brief introduction to the term used and the core concepts that will be used during the exposition.

In section 2.1 will introduce the concept of *distributed computing*, focusing on *Human Computation (HC)* and *Automatic computation*, from both the theoretical point of view and to the state of the art tools that leverages on this techniques.

In section 2.2 will present the web technologies that enables the use of the *distributed computing* paradigm over the web, focusing on the computational part of the *distributed computing* process.

2.1 CROWD-BASED COMPUTATION DISTRIBUTION

Under the name of *Crowd-based computation distribution* can fall a lot of different computational model. As shown in Figure 1, distributing computation to the crowd embodies not only the HC field but also the concept of *distributed computing*, because the crowd can be composed by humans or computers. When dealing with an *automated crowd* we are speaking of **distributed computing**, otherwise we are dealing with a *human crowd*.

Generally speaking *computation distribution* is a paradigm for splitting a task into atomic subtasks that can be performed on multiple nodes, eventually the nodes send the result of the *computation* back to the central hub. Using a client-server architecture the list of operation required to have *computation distribution* is:

1. the server **splits** the workload into atomic operations
2. the server **send the task**, among with the needed data, to the clients
3. the client **performs** the atomic task
4. the client **send the results** back 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 *computation distribution* system, although they can be "implemented" in different ways

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

or can be joined together, they are always present.

Consistently with the Table 1 and with the previous subdivision we splitted the general problem of crowd-based computation distribution into two fields: *Human Computation & Game With A Purpose* and *distributed computing*

In 2.1.1 are presented the theoretical basis as long as the state of the art tools that deals with HC and the distribution of task to a human crowd.

In 2.1.2 the concept of *distributed computing* will be presented and the main tools that implements this paradigm are described.

In 2.1.3 is presented the concept of *Crowdsearching* a new paradigm for searching using the crowd.

2.1.1 Human computation & GWAP

Human Computation (HC) is a computer science technique where a computational process performs its function by outsourcing certain steps to humans. This *outsourcing* process, as explained in the introduction is mainly due to the computational complexity of Artificial Intelligence (AI) algorithms. There are some AI problems that cannot be solved by computers or are too computational intensive to be solved by computers in a reasonable amount of time.

Some of these are very simple tasks for humans, for example natural language processing and object recognition 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 computers. Other AI problems 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 (HC)* 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*". A most simple and direct definition of HC is: the point:

Some problems are hard, even for the most sophisticated AI algorithms. Let humans solve it...

— Edith Law

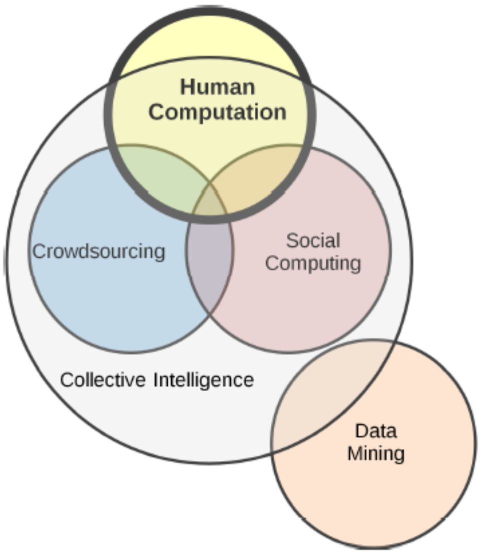


Figure 2.: Human Computation relation with CrowdSourcing, Social Computing and Collective intelligence.

Human Computation is related with other terms, such as *Crowd-Sourcing*, *Social Computing* and *Collective intelligence* as depicted in Figure 2, here we give some definition to better understand the similarities and the differences:

CROWDSOURCING is "*the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call*" Howe, 2006. So it not involves computation directly like HC.

SOCIAL COMPUTING "*describes any type of computing application in which software serves as an intermediary or a focus for a social relation*"

Schuler, 1994. So despite of the name its purpose is not computing.

COLLECTIVE INTELLIGENCE defined very broadly as "*groups of individuals doing things collectively that seem intelligent*".

When dealing with a human crowd the main issue is to engage users to perform task. A user can be motivated to perform task due to it's nature (e.g. the task helps finding the cure to some disease) or to the revenue (e.g. karma[1]) it get for doing such task. The most effective way for recruiting and motivating user is to gave them money[2]. For instance *Amazon's Mechanical Turk* is an online tool for performing Human Intelligent Task (HIT) in exchange of money rewards[3].

Figure 3.: Centralized vs Distributed execution of Human Computation.

The categorization of HC can be further specified by adding another dimension that involves how the tasks are executed by the users. As you can see in Figure 3 there are two main types of HC execution: *centralized* and *distributed*.

Centralized

In the *centralized* execution we have a central hub (i.e. a website) where users must go to perform the task. Typically the execution of a task do not involves the offload of code and data to the user, and there is no need of ad-hoc softwares to run the task.

1 Reputation points used in `www.reddit.com`.
2 Since the ancient time. TODO ???
3 The rewards for a single HIT can be as low as 0.01$.

A good example of a *centralized* HC platform is *Amazon's Mechanical Turk*. *Amazon's Mechanical Turk* is an online platform for executing task in exchange of money rewards. The platform is divided in two sections, one for the *Worker*s and one for the *Requester*s. The **Worker**s are users willing to spend time to execute a HIT and receive the reward, the **Requester**s are users that publish HIT and after getting the results pay the *Worker*s.

The lifecycle of a HIT is the following:

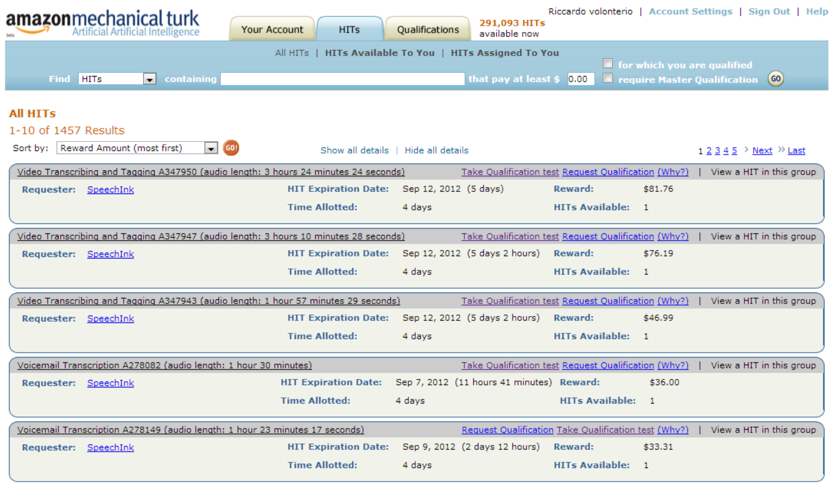
1. A *Requester* creates a HIT using one of the predefined project instances available.

2. Once the creation is completed the HIT is ready to be executed by the *Worker*s.

3. To execute a HIT a *Worker* must visit the *Amazon's Mechanical Turk* website and choose from a list of available HITs the one that he/she wants to perform (see Figure 4).

4. Once the whole HIT is completed the *Requester* check the result obtained and if it is satisfied proceed with the payment.

As one can see the whole flow of the HIT from the creation to the payment of the *Worker*s is done within the browser.


amazonmechanical turk
Artificial Artificial Intelligence
beta
Your Account
HITs
Qualifications
Riccardo volonterio | Account Settings | Sign Out | Help
291,093 HITs available now
All HITs | HITs Available To You | HITs Assigned To You
Find HITs containing that pay at least $ 0.00
for which you are qualified
require Master Qualification
GO
Sort by: Reward Amount (most first) GO!
Show all details | Hide all details
1 2 3 4 5 > Next >> Last

All HITs
1-10 of 1457 Results

Video Transcribing and Tagging A347950 (audio length: 3 hours 24 minutes 24 seconds) — Take Qualification test Request Qualification (Why?) | View a HIT in this group
Requester: SpeechInk — HIT Expiration Date: Sep 12, 2012 (5 days) — Reward: $81.76
Time Allotted: 4 days — HITs Available: 1

Video Transcribing and Tagging A347947 (audio length: 3 hours 10 minutes 28 seconds) — Take Qualification test Request Qualification (Why?) | View a HIT in this group
Requester: SpeechInk — HIT Expiration Date: Sep 12, 2012 (5 days 2 hours) — Reward: $76.19
Time Allotted: 4 days — HITs Available: 1

Video Transcribing and Tagging A347943 (audio length: 1 hour 57 minutes 29 seconds) — Take Qualification test Request Qualification (Why?) | View a HIT in this group
Requester: SpeechInk — HIT Expiration Date: Sep 12, 2012 (5 days 2 hours) — Reward: $46.99
Time Allotted: 4 days — HITs Available: 1

Voicemail Transcription A278082 (audio length: 1 hour 30 minutes) — Take Qualification test Request Qualification (Why?) | View a HIT in this group
Requester: SpeechInk — HIT Expiration Date: Sep 7, 2012 (11 hours 41 minutes) — Reward: $36.00
Time Allotted: 4 days — HITs Available: 1

Voicemail Transcription A278149 (audio length: 1 hour 23 minutes 17 seconds) — Request Qualification Take Qualification test (Why?) | View a HIT in this group
Requester: SpeechInk — HIT Expiration Date: Sep 9, 2012 (2 days 12 hours) — Reward: $33.31
Time Allotted: 4 days — HITs Available: 1

Figure 4.: *Amazon's Mechanical Turk* web interface for choosing the HIT.

This platform has been *extended* as presented in Little *et al.*, 2010 to create complete Artificial Intelligence algorithms able to use human computation as functions during the execution process. The code in Listing 1 is an example of an algorithm implemented using Turkit, here `mturk.prompt` and `mturk.vote` are Human Intelligent Task executed on the *Amazon's Mechanical Turk* platform.

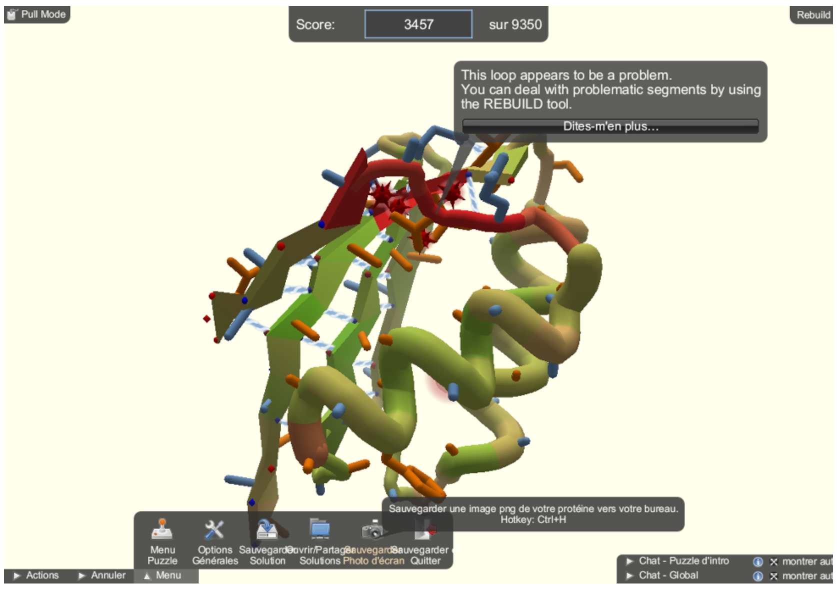
Listing 1: Example of a Turkit algorithm.

```
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
})
```

Distributed

In a *distributed* execution environment the central hub act as a distribution node in charge of offloading the task upon user request. The user can now run the task locally without the intervention of the central hub, eventually when the task is done the user contacts the hub to upload the results. The process of requesting the task to the hub, executing the task and sending the results is all done by the users, typically by a standalone piece of software installed by the user.

This solution needs the creation of ad-hoc software able to run on every platform to give users an usable tool for their purpose.



This loop appears to be a problem.
You can deal with problematic segments by using the REBUILD tool.

Sauvegarder une image png de votre protéine vers votre bureau.
Hotkey: Ctrl+H

Figure 5.: The FoldIt user interface for .

An example for this kind of code distribution is the *FoldIt* 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 analyzed by researchers, that can determine whether or not there is a native structural configuration that can be applied to the relevant proteins (see Figure 5).

Game With A Purpose

Game With A Purpose (GWAP) is "*a human-based computation technique in which a computational process performs its function by outsourcing certain steps to humans in an entertaining way*" von Ahn. GWAP a come from a simple observation of data on how many hours are spent playing games. Von Ahn and Dabbish, 2008 reported that accordingly to the Entertainment Software Association[4], more than 200 million hours are spent each day playing computer and video games in the U.S. Indeed, by age 21, the average American has spent more than 10,000 hours playing such games equivalent to five years of working a full-time job 40 hours per week.

The simple idea behind GWAP is *why not take make playing games useful*? If a task can be transformed into a game the user can be motivated to play the game so there is no need of other types of reward (i.e. money) for doing suck task. The entertainment of playing the game itself can be used as a reward for the user.

The ESP game, is a GWAP developed by Luis von Ahn to perform image tagging. The users task is to agree on a word that would be an appropriate label for the recognition of the image as described in Von Ahn and Dabbish, 2004. Another GWAP by von Ahn is Peekaboom, where users help computers locate objects in images.

2.1.2 Automatic computation

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

Distributed computing deals with the execution of code on multiple computers connected to a network. As stated in "Foundations of multithreaded, parallel, and distributed programming. 2000": "*Distributed computing is a field of computer science that studies distributed systems. A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a distributed program, and distributed programming is the process of writing such programs.*"

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[5] that, using the

4 Game data from `www.theesa.com/facts/gamer_data.php`
5 Dean and Ghemawat, 2008.

Figure 6.: General structure of a distributed computing system.

core concept of *Divide et impera* can produce highly parallelizable algorithms.

The name "*distributed computing*" refers to a wide range of different application (i.e. grid computing, cloud computing, Parasitic computing, jungle computing[6]), that implement the same paradigm with different purposes. Using the dimensions presented in Table 1 we can specialize the core concept of *distributed computing* in two: *voluntary computing* and *parasitic computing*.

Voluntary computing

Voluntary computing refers to all those *distributed computation* systems where the computation is performed on behalf of the user will. In such systems users go the website of the "project" they intend to support and, usually by installing an ad-hoc client, give the resources (i.e. CPU idle time, storage, etc.) of their machines to the chosen "project".

The first project implementing such computing paradigm was Great Internet Mersenne Prime Search (GIMPS)[7], started in 1995, then other platforms (such as distributed.net, SETI@home, etc.) has been developed to support this type of computation.

Berkeley Open Infrastructure for Network Computing (BOINC)[8] is an open-source software for Voluntary computing and grid computing. BOINC was originally developed to support the Search for Extra-

6 Winner of the coolest name 2012
7 `http://www.mersenne.org/`
8 `http://boinc.berkeley.edu/`

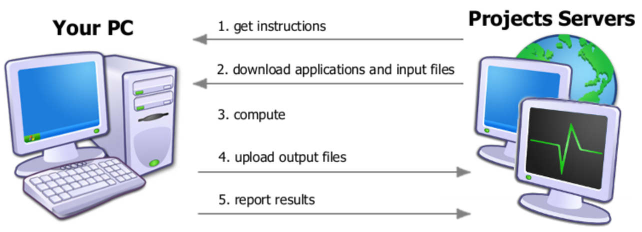


Figure 7.: The BOINC computational flow.

Terrestrial Intelligence *at* home (SETI@home) project, before it became used as a platform for other distributed computing applications. It consist on two parts, the back-end server, running on Linux platforms, and the client, cross-platform, for the end-user computing.

The client software allow users to connect to the BOINC grid, to perform computation. The flow of the execution, as depicted in Figure 7 is the standard *distributed computing*, here is the list of actions performed to run the task on the client:

1. get instruction from the server on how to get all the resources needed to perform the task
2. download all the resources from the server
3. execute the downloaded application
4. send the results obtained to the server

There are over 40[9] 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.

Parasitic computing

Parasitic computing[10] 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 parasitic computing 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[11].

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

9 At the time of writing.
10 In this thesis we are not covering, neither we are interested in, the ethical or moral implication of using this technique.
11 In *voluntary computing* the user can be unaware of the actual code they are executing, but they are aware of the execution.

"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.

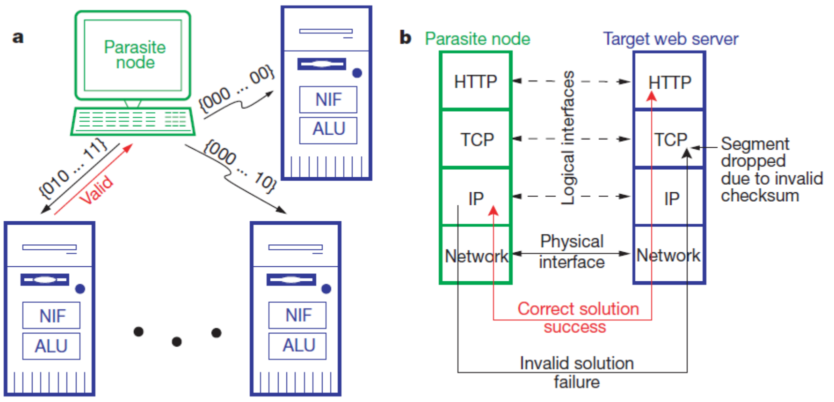


Figure 8.: Schematic diagram of the parasitic computer solving the 3-SAT problem.

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 approach as depicted in Figure 8. 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 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 least 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 reCAPTCHA are examples of involuntary human computation (as in Table 1). So they are using the same technique to perform a sort of *parasitic human computing* without complaining 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 micro-computations as micro-payments 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 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 2.2.1) we are able to perform any kind of computation within the browser window/tab. JavaScript offers also a standard platform for the execution 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 malicious intent.

2.1.3 CrowdSearcher

Web search has evolved since it was a massive link analysis system. Now the contents of a search query has been enriched with multimedia contents from the crowd. For instance a query can show results as links to other websites, user generated video, tweets or images. On top of this, a lot of tasks has been tweaked to generate more searchable data (e.g. image and video tagging), so now we are able to search for them. The crowd is becoming part of the search process, by adding information, ranking the results, moderating contents, etc.

CrowdSearch is targeted to enabling, promoting and understanding individual and social participation to search Fraternali *et al.*, 2012. CrowdSearch uses the crowds as sources for the content processing and information seeking processes; it fills the gap between generalized search systems, which operate upon world-wide information - including facts and recommendations as crawled and indexed by computerized systems and social systems, capable of interacting with real people, in real time Fraternali *et al.*, 2012. Crowd-searching can be defined as the promotion of individual and social participation to search-based applications and improve the performance of information retrieval algorithms with the calibrated contribution of humans Bozzon, Brambilla, and Ceri, 2012.

The CrowdSearch framework

The framework (see Figure 9) proposed by Bozzon, Brambilla, and Ceri, 2012, uses the humans' skills to improve some the result of a query. To perform such operation the framework must choose which

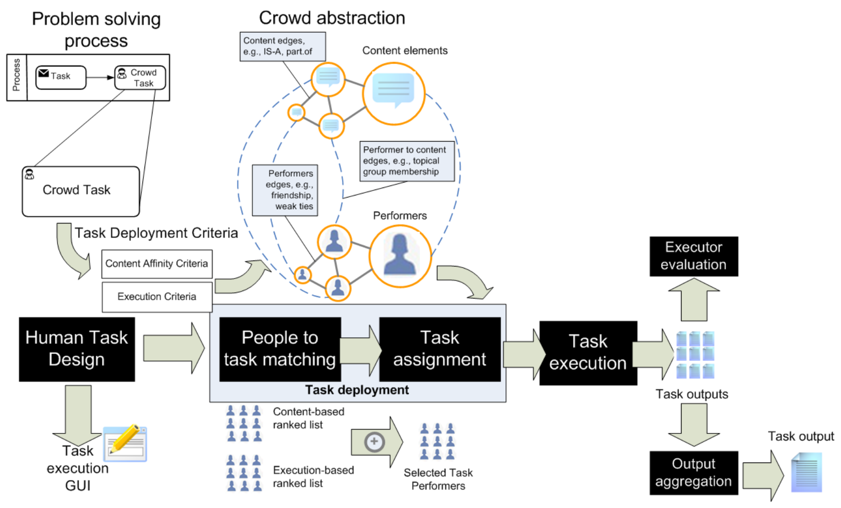


Figure 9.: The CrowdSearch framework.

task must be executed by humans and which must not. This has been addressed by mapping some task to humans and some to machines.

When a task is assigned to the crowd, the execution mode is chosen according to the *Human Task Design*. This step produces the actual design of the *Task Execution GUI*, and the *Task Deployment Criteria*. These criteria can be logically subdivided into two subject areas: *Content Affinity Criteria* (what topic the task is about) and *Execution Criteria* (how the task should be executed). The Execution Criteria could specify constraints or desired characteristics of task execution including: a time budget for completing the work, a monetary budget for paying workers, etc.

The framework provides an abstraction for the crowd and their skills, called *Crowd Abstraction*, human performers and content elements are represented as nodes connected by edges in a bipartite graph. Edges connecting performers denote a friendship, edges connecting content elements denotes a semantic relationships and finally, a connection from a performer to a content element may denote an interest.

The next step to the *Human Task Design* is the *Task Deployment*, whose goal is to assign the crowd tasks to the most suited performers. This step is subdivided in two sub-step: *People to Task Matching* and *Task Assignment*. The first sub-step is almost a query matching problem, where the workers must be ranked according to the *Task Deployment Criteria* and a set of suitable workers is selected (e.g. top-k).

The final step is the *Task Execution*. This step is the actual execution of the task by the crowd. The results are then merged to form the result of the crowd task.

2.2 ENABLING WEB-BASED DISTRIBUTED COMPUTATION

Using the web (i.e. the browser) as a platform for distributing and executing code implies that we have the available technologies to perform high-level *computation* and real-time *communication*. These are the requirements for evaluating the web as 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[12] 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 2.2.1 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 compiling* 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 multiple 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.

COMMUNICATION is being empowered, with respect to HTML4, by introducing *WebSocket*, that enables full-duplex data exchange with

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

the server, and 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.

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.

2.2.1 HTML5

Figure 10.: Official HTML5 logo & unofficial CSS3 logo.

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 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 [13], they prevent the pollution the DOM by prefixing the standard method (i.e. `requestAnimFrame` can became `mozRequestAnimFrame` or `webkitRequestAnimFrame`) with a browser specific prefix[14]. This prefixing is particularly common in the CSS3 where thing becames awful[15].

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[16].

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[17]

> `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),

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

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

15 See CSS animation or gradients.

16 A polyfill is a JavaScript library or third part plugin that emulates one or more HTML5 feature, providing websites to have a consistent behaviour.

17 Got from the specs: `http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element`

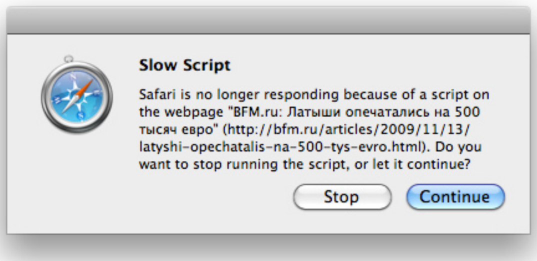
other libraries give you the tools to create diagrams or charts on the fly (like Raphaël or Processingjs).

The canvas element also provides a 3D context to draw and animate [18] 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[19] 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.

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[20] 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).



Slow Script

Safari is no longer responding because of a script on the webpage "BFM.ru: Латыши опечатались на 500 тысяч евро" (http://bfm.ru/articles/2009/11/13/latyshi-opechatalis-na-500-tys-evro.html). Do you want to stop running the script, or let it continue?

Figure 11.: 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:

18 Animations are not natively supported, must be coded separately.
19 For a reference see http://en.wikipedia.org/wiki/WebGL#Developer_libraries
20 If WebSocket, are not avilable the library can use Adobe®Flash®Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe and JSONP Polling

```
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:

```
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:
*
```

2.2.2 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.

OpenCL execution is based on *kernels*, special functions written in a language based on C99[21] with some extensions. These *kernels* can be compiled at build-time or at run-time and also have a rich setof built-in function, e.g. `cross`, `dot`, `sin`, `cos`, `pow`, `log`, etc.

An OpenCL device (like CPU or GPU) is composed by (see Figure 13):

- A collection of one or more **compute units**, this concept is similar to the *cores* of a standard CPU.

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

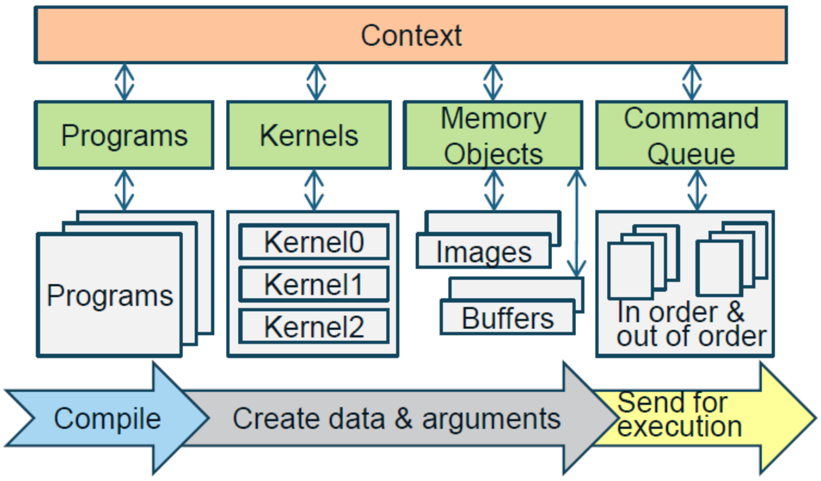


Figure 12.: OpenCL execution flow.

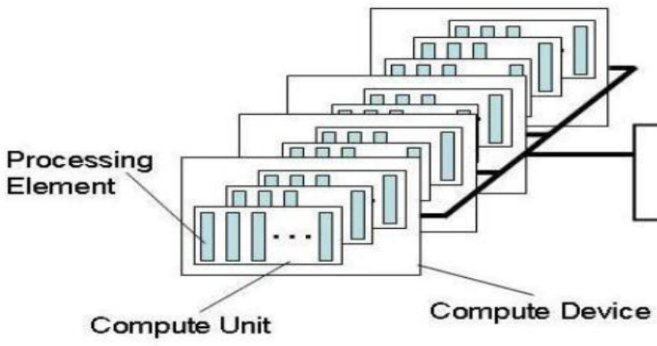


Figure 13.: OpenCL device composition.

- A *compute unit* is composed by one or more **processing elements**, this concept is similar to the *threads*.
- Processing elements execute code as Single instruction multiple data (SIMD) or Single process/program multiple data (SPMD).

Once the *kernels* have been created the flow for executing an OpenCL program can start. Here is the list of action performed to run code on OpenCL enabled computer:

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

Currently there are two main implementation of this API, one supporting Windows and Linux using Firefox (holded by Nokia), and one, holded by Samsung, that bring this interface to WebKit browsers. Speaking of performance improvements Table 2 shows the data from the TIZEN™ developer conference held in May 2012:

Table 2.: Samsung WebCL performance compared to pure JavaScript.

Demo name	JavaScript	WebCL	Speed-up
Sobel filter (with 256x256 image)	~ 200ms	~ 15ms	13x
N-body simulation (1024 particles)	5-6fps	75-115fps	12-23x
Deformation (2880 vertices)	~ 1fps	87-116fps	87-116x

3 CONCEPTUAL MODEL

In this chapter will be presented a framework for task distribution and execution able to cover all the dimension defined in **??**. First is introduced a methodology to follow in order to pass from the model to the actual application implementation, available in chapter 5. The components that will compose the framework are founded on a data model, describing the logical structure of the data processed by the application.

The data model is composed of three parts: the *Data Model*, the *Architectural Model* and the *Execution model*. Eventually are explained the pluggable strategies available for this framework.

THE DATA MODEL describes the data structures used to create this system. TODO ??

THE ARCHITECTURAL MODEL describes the reference architecture of the framework.TODO ??

THE EXECUTION MODEL focuses on the execution model of the task.TODO ??

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

3.1 DEVELOPMENT METHODOLOGY

During the development process of A framework for web-based human and machine computationthe first issue that need to be resolved was the description of a suitable model for supporting such framework. The design of a model involves the definition of the data structures that will be used by the workflow, and of course the definition of the workflow itself.

3.1.1 Data Design

In the design process for the data model we faced the problem of creating a suitable underlying model for all the application need that we have to cover, spacing from pure Human Computation task to *parasitic computing*. This wide range of possible application and the need of great flexibility demanded to the system, led us to the creation of a meta model for almost all the data structures used in the framework.

The metamodel was subdivided in four: the *Workflow*, the *Task Data Model*, the *Task Model* and the *Task Execution Model*.

THE WORKFLOW MODEL describes the flow of the execution of the Task within the framework, the relationships that needs to be taken into account to correctly execute a sequence of task (called *Work*).

THE TASK DATA MODEL describes the data model for each Task, including, if present, the field dependencies and if a field belongs to the input or the output set.

THE TASK MODEL describes the actual data of the Task, called *Objects*, for the Task.

THE TASK EXECUTION MODEL describes the actual execution step for each Task, providing information on which user is performing which task, as long as data associated to the execution itself.

3.1.2 Workflow Design

Figure 14.: The conceptual Workflow handled by the framework.

In the *Workflow Design* we must describe, at conceptual level, how our framework deal with the client and how the task are executed. During the *Data Design* we stated that the tasks can have relationships and thus our framework must be able to handle different Task types seamlessly. In Figure 14 is depicted a typical flow of execution of a Work (a composition of Tasks) with multiple task types.

Our Workflow model must be able to handle the interleaving of human and automatic task without any human intervention. The Workflow must take into account also the constraints defined during the creation step and manage the execution of the tasks accordingly.

A feature that the framework is required to handle is also the case where the application type is not strictly human or strictly automatic. We can have scenarios where the task is a blend of both human and automatic interactions. GWAP are a case where the automatic computation and the human interaction are blended together. Other scenarios can be the validation of the automatically computed results as we presented in our use case (see 4.3).

To guarantee the needed flexibility the framework must be able to include third-part logic in charge of managing certain steps of the execution (like Task planning).

3.1.3 Framework Design

In the design of the framework all the conceptual designed data and workflows as long as the requirements are merged and adapted to create the whole structure for supporting the framework.

During the **Data Design** we pointed out the data structures that need to be managed. The *Workflow Model* need to specify the relationship between the modules an the "connection points" where the third part softwares are plugged. On top of that every Task must have defined its set of input and output data. The *Task Data Model* specifies all the data structure of the task, building the metamodel for each Task. The metamodel contain all the information on the structure of the data model for a Task. In the *Task Model* are contained all the actual data for each Task, called Objects, each object represents a "row" of the data. Each object is, in turn, associated to a Task. The *Task Execution Model* contains all the information on the actual execution of a Task, keeping information on the user that performer the task and the used device, if any.

The **Workflow Design** allow us to create the software able to handle all the different types of archetype application (e.g. Human computation, Automatic computation) giving also the guidelines on how to manage the interaction between subsequent Task execution.

3.2 DATA MODEL

3.2.1 WorkFlow

The WorkFlow embodies all the data associated to the *flow* of Task that need to be executed in order to complete a set of Tasks. The tasks can belong to different archetypes (e.g. Human, Automatic) and have their own set of assigned objects. The *Workflow* is composed of:

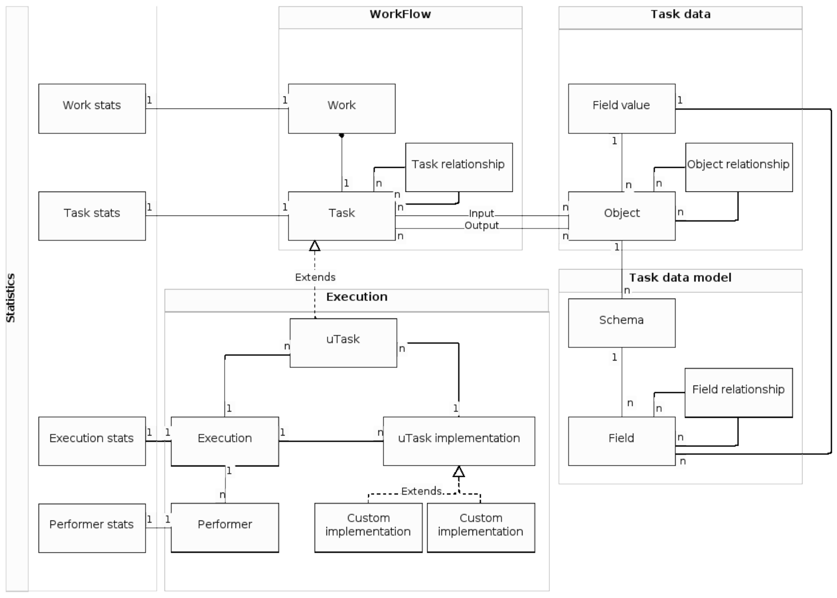


Figure 15.: Overall Data Model schema, with logical subdivision.

THE WORK: is the abstract representation of a set of Tasks related to each other.

THE FLOW: describes how the Tasks, belonging to a Work, are related to each other, also defining the type (e.g. dependency, parallel) of relation between such Tasks.

THE TASK: is the representation of a general activity that can be performed by the framework.

The Work

The *Work* represents a set of related Task finalized to the execution of some kind of data manipulation or analysis on a set of input Objects. The result of the execution produces output Objects described by a Schema. The Work is defined by:

- A **Name** that identifies the Work.
- **Constraints** used to bind some aspect of the Work or to make decision for giving less or more priority to this work with respect to the others. Example of constraints are: Due date, Performer skills, Max execution time, etc.
- **Input** data, defined by a *Schema* and the associated Objects. 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** composing the Work. Their orchestration is made at design time, by specifying a *Flow*.

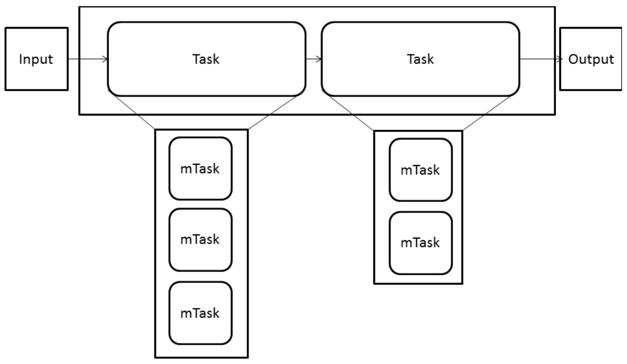


Figure 16.: Conceptual organization of Work, Task and μTask.

The Task

The Task is the central unit of computation, It represent a single activity focusing on a particular action. The activity must not be an atomic operation, like in an algorithm a single function call, but it should be focused on a single purpose. For example tagging an image can be considered as a single Task, otherwise tagging and validating an image should be divided in two separated tasks if possible. the flexibility of the framework allow the creation of any type of Task, the previous statement is only an advice to better separate different activities into different Task. As we mentioned our system can seamlessly handle complex Task involving multiple activities, such as whole GWAP game into a single Task. A Task is characterized by:

- A **Name** that identifies the Task.
- **Input** data, defined by a *Schema* and the associated Objects. To keep the model as general as possible no assumption are made on the *Schema* type (relational, graph, etc.). Usually the Schema is a projection of the Schema of a Work. Like the Schema the input Object usually are a selection of the Work input Objects.
- **Output** data, defined as an extension of the input schema (sharing the same schema type).
- 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

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

  - 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 execution.
  - *Running*: µTask are running.
  - *Ended*: all the µTask have completed their execution.

- A set of **Subscriber**s able to receive updates on the Task execution. The Subscribers are declared during the Task creation step and the updates that they will receive are about change on the Status of the Task.

- A set of **Execution constraints** used for prioritizing the Task among others or to modify the standard behavior of the Task. The constraints that can be used are the same defined for the Work in 3.2.1.

- **Configuration data** are application specific data used to configure the behavior of the Task once it is executing. For instance we can store here the classes for a classify Task type.

- µ**Task**s are the instances of Task assigned to one or more Performers on a Device, to be performed on one or more input Objects.

- An **Aggregation** function, in charge of collecting the µTask results and generate 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 objects to each 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 *fixed* for everyone, can be done according to the *user-agent* (e.g. Browser) or other conditions.

- A **Task planning** which embodies the functionalities of μ*Task planning* strategy, *Performer assignment* strategy and μ*Task implementation* strategy and whose purpose is to decide the logic behind the invocation of those strategies.

- A **Task control** strategy is able to control the status of the Task and if needed perform corrective actions.

- An **Emission policy** specifying which *Subscriber* need to be notified of a Task change in *Status*.

The **relationships** that may exist between Tasks are materialized into the *Task Relationship* object. This object contains all the information related to the flow of the task execution during the execution of a Work. An example of a Task relation can be the validation of the results obtained by an Automatic computation task previously performed. In this case the relation is the simple sequence.

In a flow we can use *Control Structures* and *Variables*. The *variables* are included into the flow to control the behavior of the Work and can be modified by some condition upon Task execution. For instance if an Automatically obtained results from Task are below a certain threshold, this check is performed after the Task execution, then a variable is changed accordingly. The *Control Structures* are common to all the Workflow managers and control hoe the Tasks are executed:

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.

3.2.2 Task Data Model

The Task Data Model contains all the information about the Task metamodel. The metamodel is organized in Fields and the Fields are grouped into a Schema. This data structure resembles the standard DBMS schema organization, where we have a Schema defined as a collection of Fields with possible relationship between them. The Task Data Model is composed by

THE SCHEMA: is the abstract representation of a table structures.

THE FIELD: contains all the information of the field type and relationships.

The Schema

The Schema is used as a container of Fields that compose the meta-model of the Task data. The Schema is defined by:

- A **Name**
- A **list of Fields** defining the structure of the Task data.
- A **list of Objects** representing the Task Data instances

The Schema is in **relation** with the actual data instances of the Task, called Objects, and the Fields it is composed of.

The Field

The Field contains information on the type of the data it contains and information on the relation between two, or more, Fields (like the type of dependency). The Field is composed by:

- A **Name** that identifies the Field.
- A **type** defining the type of the data that this field contains, for instance `string`, `number`, etc.
- A set **related fields**. the relationship between the fields is defined by the *relation* attribute.
- A **relation** that specifies which type of relationship occurs among the *related fields*.
- The **list of field values** representing the data contained within this field in the Object structure.

The **relationship** existing between the Field are materialized using the *Field Relationship* table. Here we have all the information of the fields involved and on the type of the relationship (e.g. derivate of, copy of, calculated etc.).

3.2.3 Task Data

The Task Data contains the actual data instance for each Task. The data are subdivided into *Field Values* and *Objects*, this division is made to mirror the structure presented in 3.2.2. The Objects represents the "tuple" of the table, while the Field Values contains the atomic information of a Field within a Tuple. This structure allows the direct access to Field Values without the need of parsing the whole Object element. As mentioned the Task Data is composed of:

THE OBJECT: a Task data instance, represented by a set of Field Values.

THE FIELD VALUE: represents the instance of a Field. As the Fields they can be related to each other.

The Object

The Object represents the Task data instances. They are composed of a set of Field Values that, joined together, forms the Object. The object itself is composed by:

- A **Name**.
- A **list of Field Value**s data.

A Objects can have **relationships** among them. These relationships are materialized into the *Object Relationship* entity. This object contains the needed information used to identify a set of Objects belonging to a particular Task.

The Field Value

The Field Value contains the actual data of a Field. Since this object is used to represent any kind of data, it must be able to handle different types of information (e.g. `string`, `BLOB`, `number`, `date`, etc.). The Task Field is composed by:

- The field **Value**.
- A **related Object** used to identify to which Object this Field Value belongs to.
- The **Field** to which data belongs.

The Field Value have **relationships** with the Object to which it belongs and with the Field that defines the general type of the Field Value.

3.2.4 Task Execution

The Task Execution is used to collect information on the Task execution. Since a Task must be distributed to multiple nodes[2] the Task must be splitted in μTask. Then we need to know what user is performing such μTask, and, based on the device used, we can have multiple implementation of the same μTask. All these information are subdivided in:

THE μTASK: is a subclass of the Task. The μTask is a portion of the Task executed by the end-user.

THE μTASK IMPLEMENTATION: represent the real implementation of the μTask.

THE PERFORMER: is the Human being in charge of the execution of the Task.

2 We use the broad term node to identify both the user and the device on which the Task must be executed.

THE EXECUTION: represent a bridge object connecting the µTask, the µTask Implementation and the Performer to identify the execution of a µTask using a µTask implementation by a particular user.

µTask

Since the same Task must be distributed to many users there is the need of a further subdivision of a Task into "atomic" activities (see Figure 16). The µTask is a subclass of the Task that representing this subdivision. As a subclass it inherits all the information of the data and the schema from the parent Task. Usually µTasks have associated only a portion of the parent Task data objects. The µTask is defined by:

- A **Name**.
- A set of **Execution constraints**, similar to the ones defined in 3.2.1. The difference is that this constraints bind the execution "client side". This means that with these we can control the computational load on the user machine according to some policy (like user preferences, device capabilities, etc).
- **Input** data, as a subset of the Task input Objects.
- **Output** data, with the same schema as the related Task output Objects.
- A **list of Properties**, defined as name-value pairs, having domain specific meaning. For instance .

The µTask is in **relationship** with its µTask implementations. This relation is useful when choosing a suitable implementation to execute on a particular device.

µTask implementation

The µTask Implementation represent the application logic and presentation delivered to a user to perform a Task. The Task have a default implementation and the other implementations can be supplied to customize the Task execution. Each µTask implementation is in **relation** with the corresponding µTask.

Performer

The Performer is an object used to characterize a human being in the Data Model. The user is characterized with properties that defines his/her skills on a certain field (like music, style, etc.) and some other attributes used for profiling. The Performer object is defined by

- A **Name**
- **Demographic** information

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

The Execution

The Execution is a bridge object that joins together μTask, μTask implementation and performer. This object represent a single instance execution of a Task by a user. Using this object we can identify cheaters or reward good Performers, or using it as a timeline to see how the Performers respond to the Task and change it accordingly. An Execution object is defined by:

- A **Status** that identifies the current status of a μTask implementation, the available statuses are:
  - *running*
  - *suspended*
  - *idle*
  - *ended*
- A **set of Execution data**. TODO ??

Since it is a bridge object the Execution have **relationships** with μTask, μTask implementation and the Performer. These relationships identifies the Execution itself.

3.2.5 Statistics

This Statistics model contains all the statistics and data related to the profiling of a Task. These data can be used by other components, such as the Task controller to take decision during the flow of a Task. The Statistics are about *Work*, *Task*, *μTask*, *Performer*, etc. The data contained in these objects are:

- **Creation date**
- **Total execution time**
- **Average number of *Performer*s per hour**
- **Last execution**
- etc.

3.3 ARCHITECTURAL MODEL

During the Design Process we faced the problem of finding a suitable Architectural Model able to support all the requirements, in terms of flexibility and pluggability, raised during the Process Design. In our

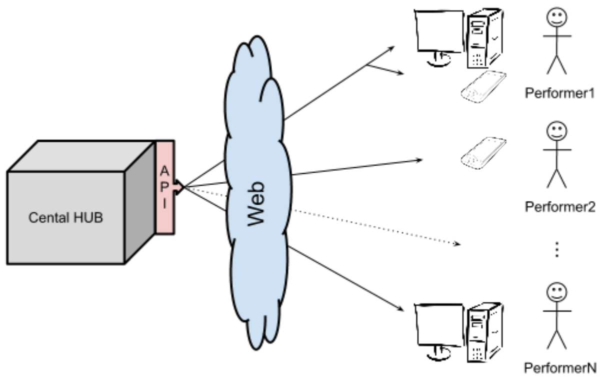


Figure 17.: Reference architecture.

model we use as a reference architecture the one depicted in Figure 17. Here we have a central hub in charge of distributing the Tasks to the nodes[3] and an abstraction layer.

The *Central Hub* is used to manage all the data exchange between to the nodes and the hub itself, orchestrating all the communication flow.

The *Abstraction Layer* is used to normalize the differences between the nodes, creating a coherent representation of nodes.

As described in 3.1.2, our framework has multiple configuration points used to customize the Task behavior. As described in 3.2.1 on page 25, for each Task we identified seven configuration points:

µTASK PLANNING STRATEGY: defines how many µTask to create for each Task and associate the right portion of Objects to each of them.

µTASK IMPLEMENTATION STRATEGY: defines the logic behind the choice of a µTask implementation sent to a user.

PERFORMER ASSIGNMENT STRATEGY: in charge of choosing the users who are suitable to execute a certain Task.

TASK PLANNING STRATEGY: orchestrates the invocation of the µTask planning strategy, the µTask implementation strategy and the Performer assignment strategy.

TASK CONTROL STRATEGY: defines a controller for the Task, able to check the status, and if needed, perform corrective actions.

AGGREGATION FUNCTION: is in charge of joining the results obtained from the µTasks execution and create a Task output.

EMISSION POLICY: is used to rule notifications sent to the Subscribers.

All these configuration points are described in 3.5.

3 We refer to nodes because we want to enclose both humans and devices.

3.4 EXECUTION MODEL

Figure 18.: Representation of the Task execution flow.

During the Framework Design stage we found how our framework processes a Task and how this behavior can be configured. In Figure 18 is depicted the typical flow of a Task execution. As one can notice all the controlling strategy is handled by the *Task Control strategy* that, accordingly to the *Task Planning strategy*, tweaks the execution of a Task. By changing the Task Planning strategy we can obtain four different execution modalities, summarized in Table 3:

STATIC EXECUTION: where both the µTask planning and the Performer assignment is performed statically at Task design time.

STATIC µTASK PLANNING & DYNAMIC ASSIGNMENT: in this scenario the µTask are planned at design time and the Performer assignment can be made at "any" time after the creation of the Task.

DYNAMIC µTASK PLANNING & STATIC ASSIGNMENT: in this scenario we first Assign Performers to µTask and then we Plan the µTasks[4].

DYNAMIC µTASK PLANNING & DYNAMIC ASSIGNMENT: where both the strategies can be specified after the Task creation[4].

STATIC EXECUTION This execution mode represents the simplest scenario possible. Static execution means that both the *Task Planning strategy* and the *Performer Assignment strategy* are executed once at Task

4 This scenario presents some pitfalls, see 3.4 to further explanations.

Table 3.: Task planning vs. Task Assignment.

	Static	**Dynamic**
Static	static-static	static-dynamic
Dynamic	static-dynamic	dynamic-dynamic

creation time. The Task Control strategy duty is to control if the *Constraints* are verified and monitor the Task status. Here is the list of the default actions performed by 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.

STATIC μTASK PLANNING & DYNAMIC ASSIGNMENT In this execution mode the user wants to define all the μTask and then assign Performers later, usually with a custom Performer Assignment strategy. In this scenario the μTask Planning strategy is are planned once at creation time and the assignment can be performed at any time. In this scenario the *Task Control* strategy invokes the *Performer Assignment* strategy ensuring that the constraints are met. The *Task control* strategy can also decide to reassign *Performer*s to μTask later during the execution. Here is a list of the default actions performed by 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.

DYNAMIC μTASK PLANNING & STATIC ASSIGNMENT In this scenario we want the user requesting the Task to be able to dynamically assign the μTask to the Performers. This scenario can lead to some pitfalls. For instance if the Performer assignment is executed and there are no μTask defined then we can have unhandled behavior[5]. Supposed we not fall into these pitfalls, in this scenario the *Performer Assignment* strategy is called at creation time and the μ*Task Planning* strategy can be called at any time. 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

5 This behavior is not managed by the framework. A custom logic must be supplied to handle such execution.

DYNAMIC µTASK PLANNING & DYNAMIC ASSIGNMENT This scenario covers the most flexible use-case. Here the user can create an "empty" Task and later assign the µTasks and Performers. Also the strategies can be called more than once to adapt to the user preferences. For instance if the user that created the Task notice that the results are biased then he/she can decide to expand the Performers set and re-plan the µTasks. 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

3.5 PLUGGABLE STRATEGIES ASSIGNMENT

As pointed out during the Process Design and briefly introduced in the Architectural Model, our framework must handle custom strategies. These strategies are used to grant to our model enough flexibility to handle all the application archetypes defined in Table 1. Here is the list of the configurable strategies:

µTASK PLANNING STRATEGY: defines how many µTask to create for each Task and associate the right portion of Objects to each of them.

µTASK IMPLEMENTATION STRATEGY: defines the logic behind the choice of a µTask implementation sent to a user.

PERFORMER ASSIGNMENT STRATEGY: in charge of choosing the users who are suitable to execute a certain Task.

TASK PLANNING STRATEGY: orchestrates the invocation of the µTask planning strategy, the µTask implementation strategy and the Performer assignment strategy.

TASK CONTROL STRATEGY: defines a controller for the Task, able to check the status, and if needed, perform corrective actions.

AGGREGATION FUNCTION: is in charge of joining the results obtained from the µTasks execution and create a Task output.

EMISSION POLICY: is used to rule notifications sent to the Subscribers.

This section explains the purposes of each strategy and what are the main properties for each strategy.

3.5.1 µTask Planning strategy

The µTask Planning strategy is a pluggable logic devoted to the creation, or deletion, of µTask. Eventually this strategy must, during the creation time of a µTask, bind a corresponding set of Task Objects to the µTask. This binding produce a set of µTask with the corresponding Task Objects. A Task planning strategy is defined by:

- A set of **Constraints** that rules the execution. TODO ??
- 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.

3.5.2 Performer Assignment strategy

The Performer assignment strategy is a pluggable logic devoted to the assignment *Performers* to µTask. Once we have a set of µTask we can assign to them the suitable Performers. The Performers are chosen according to some criteria like their skills or the place they live. A Performer assignment strategy is composed by:

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

  ONE-SHOT: the assignment is performed according to a predefined 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.

3.5.3 µTask Implementation strategy

µTask Implementation strategy is a pluggable logic in charge of selecting a suitable µTask implementation for an *Execution*. This strategy is invoked before the execution of a Task on a device. Based on the

Performer preferences or on the device characteristics a suitable µTask implementation is routed to the user. A µTask implementation strategy is characterized by:

- A **set of assignment Constraints**. TODO ???
- 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 predefined 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.

3.5.4 Task Planning strategy

The Task Planning strategy embodies the functionalities of a µTask Planning strategy and Performer Assignment strategy, deciding the logic by which the two strategies should be invoked. This strategy can be used to manage the re-planning of the µTasks or to call the Performer Assignment strategy. The invocation of this strategy is controlled by the Task Control strategy that can call this strategy upon changes in the Task status.

Task control strategy

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

- **Once** when the Task ends. For instance when all the µTasks are executed to re-plan the execution.
- According to a **temporal schedule** (i.e. every x minutes, once a day, at noon, etc.).
- Every time a µTask is **executed**.

Among the corrective actions available to the Task controller we have:

- The **re-planning** of the task, also with the creation of new µTask.
- The **re-assignment** of µTask to *Performer*s.
- **Deletion** of executed µTask.
- **Change** the properties of an executed µTask. For instance we can set the results as invalid if we have spotted a cheater.
- **Re-execution** of the entire Task.
- **Halting** the Task.
- etc.

3.5.5 Aggregation function

An Aggregation function is a pluggable logic devoted to joining the results obtained with the μTasks execution. These results are merged according to a Task specific logic in order to produce the Task output result. An aggregation function can be as simple as a *Sum* or an *Average* but can can also be more complex. For instance we can gather the results obtained, perform filtering operation and eventually produce an image.

Emission policy

The Emission policy controls how the *Subscribers* are notified about the status changes in a Task. This logic can be executed:

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

4 THE USE-CASES

In the previous chapter we introduced a framework for web-based human and machine computation, able to handle different types of application archetypes. In this chapter we present the use-cases use to test this framework under different scenarios. Since we wanted to test the framework with all the possible application archetypes we implemented three use-cases: *Automatic*, *Human* and *Hybrid*:

AUTOMATIC: the automatic use-case is used to simulate a *Distributed Computing* application. In this scenario we compute the SIFT algorithm on an image and return the obtained keypoints to the server.

HUMAN: the human use-case is used to test if we can perform Human Computation with our framework. To test this scenario we implemented a text disambiguation application.

HYBRID: the hybrid use-case is used to test both the automatic and the human scenarios. Here we implemented a GWAP on top of a face recognition algorithm.

We focused on the implementation of the *Voluntary* scenario, see Table 1, because the *involuntary* case is almost straightforward to obtain. At the end of every use-case will be presented, if possible, a simple benchmark/metric where the use-case results are compared with the ones obtained with the available tools.

4.1 AUTOMATIC

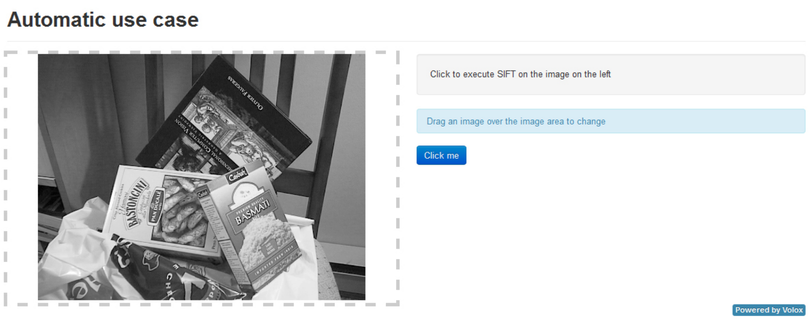

Automatic use case
Click to execute SIFT on the image on the left
Drag an image over the image area to change
Click me
Powered by Volox

Figure 19.: Interface of the automatic use-case.

The Automatic use-case aim at executing computationally intensive applications on the user device (e.g. browser). This scenario allow us to test if our framework is able to act as a *Distribute Computing* platform, as described in 2.1.2.

For this use-case we choose to implement an image recognition algorithm. We used an image recognition algorithm because they are commonly used by search engines to find images with similar features. Also these algorithms usually have high requirements in terms of CPU load and resources usage, so they are the perfect candidate for our purposes. The algorithm we choose to implement was Scale-Invariant Feature Transform (SIFT), one of the most widely adopted for this purpose.

4.1.1 SIFT algorithm

The SIFT algorithm is composed of four sequential steps: Scale-space extrema detection, Keypoint localization, Orientation assignment and Keypoint descriptor.

SCALE-SPACE EXTREMA DETECTION: is the stage where the interest points are detected.

KEYPOINT LOCALIZATION: is used to filter the unstable keypoints and keep only good keypoints.

ORIENTATION ASSIGNMENT: compute orientation and magnitude for each keypoint.

KEYPOINT DESCRIPTOR: generates the keypoints descriptors.

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 $I(x, y)$ at different scales $k\sigma$ with a varying Gaussian kernel $G(x, y, k\sigma)$ obtaining:

$$L(x, y, k\sigma) = G(x, y, k\sigma) * I(x, y)$$

Then the difference of successive blurred images are taken

$$D(x, y, \sigma) = L(x, y, k_i\sigma) - L(x, y, k_j\sigma)$$

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.

INTERPOLATION OF NEARBY DATA FOR ACCURATE POSITION The interpolation is done using the quadratic Taylor expansion of the DoG $D(x, y, \sigma)$ scale-space function, with the candidate keypoint as the origin. This Taylor expansion is given by:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} + \frac{1}{2} \cdot \mathbf{x}^T \cdot \frac{\partial^2 D}{\partial \mathbf{x}^2} \cdot \mathbf{x}$$

where D and its derivatives are evaluated at the candidate keypoint and $\mathbf{x} = (x, y, \sigma)$ is the offset from this point.

DISCARDING LOW-CONTRAST KEYPOINTS To discard the keypoints with low contrast, the value of the second-order Taylor expansion $D(\mathbf{x})$ is computed at the offset $\hat{\mathbf{x}}$. If this value is less than 0.03, the candidate keypoint is discarded. Otherwise it is kept, with final location $\mathbf{y} - \hat{\mathbf{x}}$ and scale σ, where $\mathbf{y}$ is the original location of the keypoint at scale σ.

ELIMINATING EDGE RESPONSES The DoG function will have strong responses along edges, even if the candidate keypoint is not robust to small amounts of noise. Therefore, in order to increase stability, we need to eliminate the keypoints that have poorly determined locations but have high edge responses. For poorly defined peaks in the DoG function, the principal curvature across the edge would be much larger than the principal curvature along it. Finding these principal curvatures amounts to solving for the eigenvalues of the second-order Hessian matrix, $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

The eigenvalues of $\mathbf{H}$ are proportional to the principal curvatures of D. The trace of $\mathbf{H}$, gives us the sum of the two eigenvalues, while its determinant yields the product. The ratio $\mathbf{R} = \operatorname{Tr} \mathbf{H}^2 / \operatorname{Det} \mathbf{H}$ can be shown to be equal to $(r + 2)^2/r$, which depends only on the ratio of the eigenvalues rather than their individual values. It follows that, for some threshold eigenvalue ratio r_{th}, if $\mathbf{R}$ for a candidate keypoint is larger than $(r_{th} + 1)^2/r_{th}$, that keypoint is poorly localized and hence rejected.

Orientation assignment

In this step for each keypoint is assigned an orientation and a magnitude. This step is used to achieve *invariance rotation*. The magnitude $m(x, y)$ and orientation $\theta(x, y)$ are calculated as follows:

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$
$$\theta(x, y) = \tan^{-1}\left(\frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)}\right)$$

Keypoint descriptor

Previous steps found keypoint locations at particular scales and assigned orientations to them. This ensured invariance to image location, scale and rotation. Now we want to compute a descriptor vector for each keypoint such that the descriptor is highly distinctive and partially invariant to the remaining variations such as illumination, 3D viewpoint, etc. This step is performed on the image closest in scale to the keypoint's scale.

First a set of orientation histograms are created on 4x4 pixel neighborhoods with 8 bins each. These histograms are computed from magnitude and orientation values of samples in a 16x16 region around the keypoint such that each histogram contains samples from a 4x4 subregion of the original neighborhood region. The magnitudes are further weighted by a Gaussian function with equal to one half the width of the descriptor window. The descriptor then becomes a vector of all the values of these histograms. Since there are 4x4 = 16 histograms each with 8 bins the vector has 128 elements. This vector is then normalized to unit length in order to enhance invariance to affine changes in illumination.

4.1.2 Benchmark/Metric

Since the purpose of this use-case is the feasibility of high load computation on the user browser, the implementation of the algorithm has not been optimized. Then the performance of this implementation are not comparable to the existing C/C+ implementation+, but we can leverage on the parallelism of the whole framework to obtain an higher throughput. In our test cases we obtained the results presented in Table 4.

Table 4.: SIFT algorithm performances.

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

4.2 HUMAN

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

4.2.1 Benchmark

4.3 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.

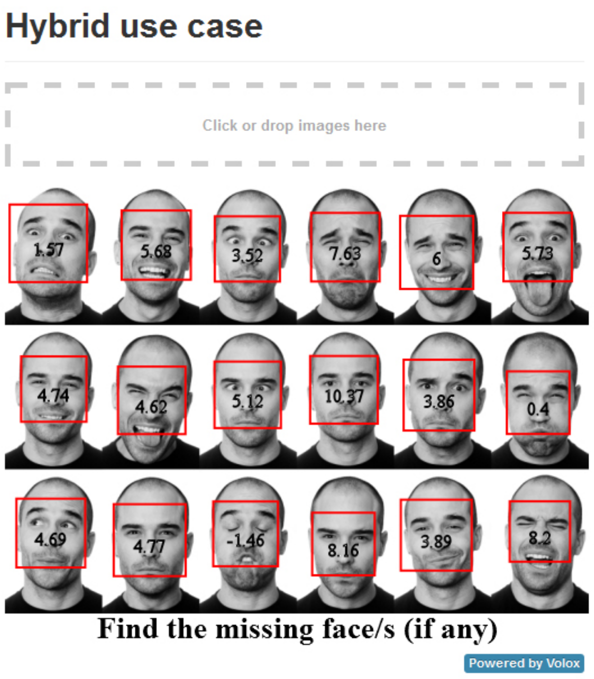


Figure 20.: Interface of the hybrid use-case.

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), the second step is implemented as a GWAP.

The game, under the name **ThemAmongUs** has been inspired by the 1988 film "*They Live*" directed by John Carpenter. *ThemAmongUs* is a single player arcade shooter in which the player assumes a role of an agent that fights against an alien race disguised as human beings. Equipped with a special camera able to distinguish between human beings and non humans, the agent is asked to shoot at the head of the beings that have not been identified by the camera software. The

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

camera may fail in some occasion, so the agent has to use his judgment to fire only at the right targets.

4.3.1 Task implementation

Objective of the game is being able to identify the visage of the subject portrayed in a photo. Automatic algorithms for face detection may fail in this task, by recognizing objects that are not the required ones (e.g. knees) or not recognizing at all the face of a person. For this reason the game will be able to perform two different tasks: identify faces that have not been recognized and remove elements that have been marked improperly.

The game present the image with the bounding boxes of the faces that has been identified by the first step to the player. The faces that have not been recognized are the ones that the player needs to mark during the game by clicking over them with the left mouse button and the area interested by the click will be circumscribed in order to obtain the bounding box for the identified face.

Possible wrong bounding boxes will be removed by the player with the right mouse button, in order to remove the false positives that may be identified by automatic algorithms. Bounding boxes with high level confidence are kept into the image, while bounding boxes with low level confidence values will be removed: it will be a player duty to mark those faces by playing the game.

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.

4.3.2 Introducing Score Degradation

The new game mechanic that is presented to manage the task is called Score-Degradation. This technique may be used in scenarios in which there are not the possibility to compare the results provided by the players with techniques such as the one provided in output-aggregation or inversion-problems games, because the game that is being taken into consideration is a not a multiplayer game but a single player one.

Goal of the technique is to force the user to always provide the right answer with game mechanics that involve low reaction times , high penalties for mistakes (such as early game termination) and incentives to achieve the best results compared to all the other players.

Players are first evaluated based on well known trial examples tasks to understand their reliability level. Failing the required task in these training examples usually ends the gaming experience for the player, forcing him to start the game from the beginning.

Once a sufficient level of trust for the player has been reached, the player is then provided with a sequence of mixed tasks, some of them with an already well established knowledge of the expected results, some of them with completely unknown expected results. While the results of the first kind of tasks will still be checked against the right results, for the second kind of tasks the results provided by the players will always be considered good results.

The players will not be able to distinguish which of the instances of the tasks are being checked against their provided results and which results are simply considered "true as provided" without any further checks. This behaviour is also enforced by the fact that the player is not able to understand the moment in which the "trial phase" will end and the fast reaction times force him to not even have the time to think about providing misleading results, with the risk of having to start the game from the beginning again.

In this way the player are always forced to try to give the best possible solution for a specific task. The collected results can be further improved by using traditional aggregation techniques such as majority voting or similar, depending on the task that has to be solved.

Figure 21.: Hybrid implementation gameplay.

4.3.3 Gameplay

Goal of the game is to obtain the highest possible score given a limited amount of time (1 minute). The player is provided with a series of images that present bounding boxes of the face of human beings automatically identified by the special camera of the agent. Each provided image will constitute a round of the game. If in the image some face has not been surrounded by a bounding box, it means that the portrayed subject is an alien and must be shot at by pressing the left click button.

The player has a limited amount of time, typically 5 seconds, to shoot at all the faces that have not been recognized, in order to obtain a certain amount of points. During a round the player may also find improper bounding boxes, such as knees or other part of the body that have been recognized as a face.

The player may right click on these boxes to remove them and obtain additional points. When the player has shoot to all the unboxed faces, he may shoot at a button on the right lower corner of the screen to play the next round of the game. The game will end if the player will shoot at a recognized face by mistake.

At the end of each round (after the 5 seconds have passed or when the player has pressed the end button), the system checks if the player has missed any face. If it is the case and the image was a trial one or one for which the results were known, the player will lose the game with a score equal to the number of points he achieved so far. Otherwise the score for the current round are calculated in the following way:

$$\begin{aligned} \mathsf{Score} &= (\mathsf{RoundNumber} * 10) * (\mathsf{NumberOfAliensKilled}) \\ &\quad + (100 * (\mathsf{FalseBBRemoved})) \end{aligned} \tag{4.1}$$

At the end of the global gaming time, a player who has not made any mistake will receive 1000 additional points. The points are used to provide an incentive to improve and beat other players by improving the score on further matches.

4.3.4 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.

5 IMPLEMENTATION AND EVALUATION

In this chapter the architecture of the whole system is explained as long as the motivation that led us to these choices.

When chosing a suitable backend for our System we needed to take care of all the features it must be able to perform (see Table 1). Without all these needs fullfilled the System cannot bring any original contributions.

5.1 ARCHITECTURE

The architecture is divided in two main parts, the task creation (managed by the Configurator) and the task execution (managed by the Execution layer). As shown in Figure 24 in the Model this two parts can be safely separated into standalone components.

5.1.1 Configurator

Figure 22.: The Crowdsearcher interface.

The configuration and the management of the Task/Work lifecycle is demanded to a third part component, the **CrowdSearcher**. As descibed in **??** the *CrowdSearcher* is a *centralized* human computation platform able to execute the task once the user reached the *CrowdSearcher* website. For the execution the *CrowdSearcher* give its default implementation for the most common Task.

The *CrowdSearcher* has been implemented using the WebRatio tool and so is a Java standard web-application.

5.1.2 Execution Layer

Figure 23.: Official NodeJS logo.

The execution layer is being developed using the great flexibility of *NodeJS*. Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. This platform was chosen due to the great speed of the request-response cycle when dealing with relatively small files (like in our scenario).

Due to the great hype around *NodeJS* and the ease of developing libraries for this platform, there are thousands of these being developed and made available via the built-in package manager (`npm`).

For the implementation of the sever we used the *Express* web application framework. This framework allow to create easy routing on pages and provide a robust templating system. With all these features we were able to create a REST web-server that interact with the *Configurator* in order to recieve µTask, execute the code and post the results back to the Configurator.

The *execution layer* allow the µTask implementation to inherith a JavaScript *Class* that give a set of API for retrieving Task configuration, gathering data and eventially post the µTask results back to the *Configurator*.

The storage of the µTask implementations is made using the filesystem because the resources needed to execute a µTask are almost text or binary files like images or JavaScript/CSS files. If for a certain task a custom default[1] implementation is provided than a `default` folder is created for that task containing all the files. If the *Requester* provides a platform-specific implementation than a folder with the platform name (e.g. `mobile`, `tablet`, etc.) is created and the code qill be uploaded there. When a user visit the page containig the logic to run a task, then

1 By default we mean a single implementation for all the platforms/devices, see **??**

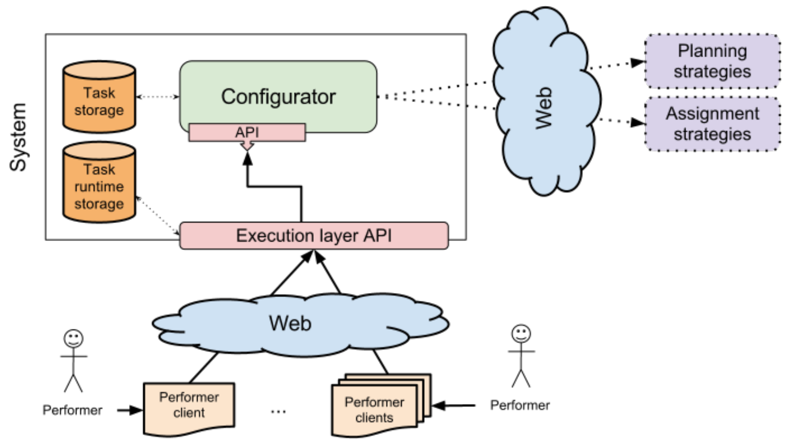


Figure 24.: Specialized architecture.

according to the strategy configured the right implementation will be sent to the device.

5.2 PERFORMANCE COMPARISON???

TODO ??

The reference model in Figure 17 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*.

5.2.1 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.

5.2.2 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.

5.2.3 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).**

5.2.4 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*.

Now the built-in implementation of task creation, planning and execution are described to better understand how the whole system works.

The description is focused on the built-in implementation, because it is the default behaviour of the system. By pluggin-in custom strategies one can completely change how the system behaves, thus this case will not be covered here. For examples on how the pluggable strategies works see 3.5.

5.2.5 Task creation

The task creation if performed by the *Configurator* either by using its web-interface or via API calls. The creation of a Work/Task can be done by providing a JSON file, containing all the data definition as long as the data instances, or "manually" following a step by step procedure within the *Configurator*. As shown in Figure 25 the manual Task creation involves the definition of a **Schema** for the data. The schema is composed of **Field**s, for each filed the *Requester* must specify a type. The data instances can be added to the *Schema* either during the definition of the *Fields* or at the end of the *Schema* definition.

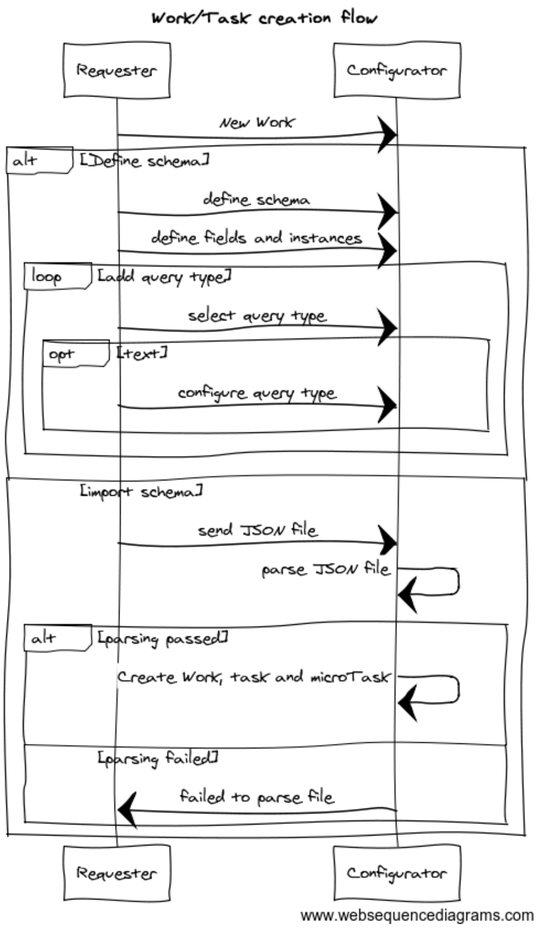


Figure 25.: Work/Task creation flow.

5.2.6 Task planning

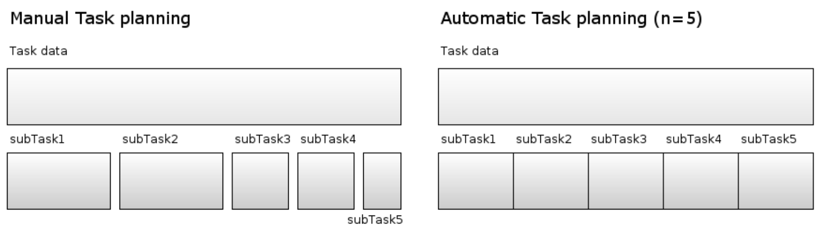


Figure 26.: Manual Task planning vs Automatic Task planning.

The planning of a Task involves the creation of subtasks with associated data that need to be executed. The assignment can be performed automatically or manually. The automatic plan assignment uses a simple subdivision based on the number of instances to assign to each subTask.

As depicted in Figure 27 manual planning involves the *Requester* interaction in orer to create **subTasks**. After creating the subTask the *Requester* has to select the instances belonging to this subTask. Eventually the *Requester* is able to select and, if needed, configure the type of the subTask, based on the task types of the parent Task.

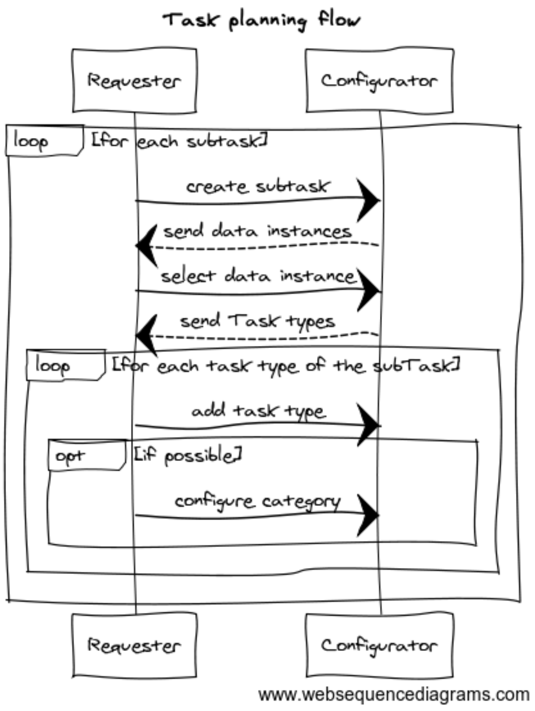


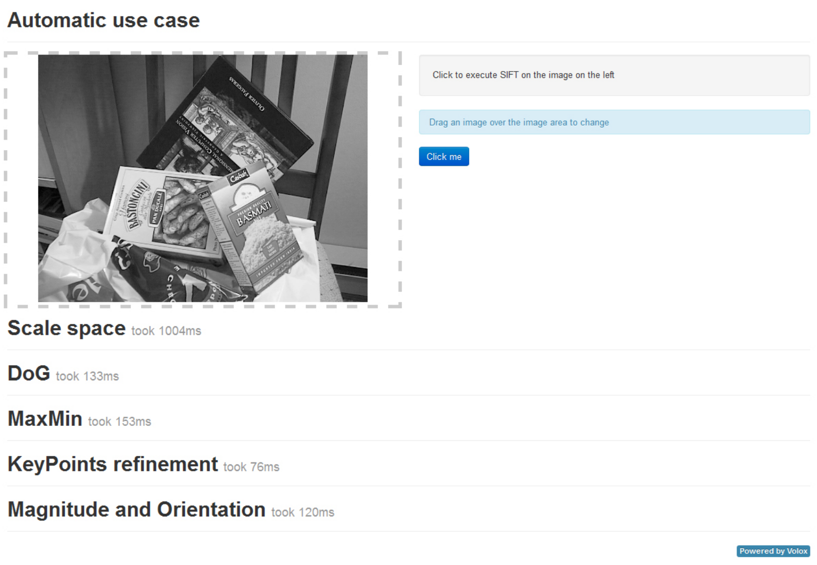
Figure 27.: Task planning flow.

5.2.7 Task execution

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.


Automatic use case

Click to execute SIFT on the image on the left

Drag an image over the image area to change

Scale space took 1004ms

DoG took 133ms

MaxMin took 153ms

KeyPoints refinement took 76ms

Magnitude and Orientation took 120ms

Figure 28.: Intermediate results of the algorithm.

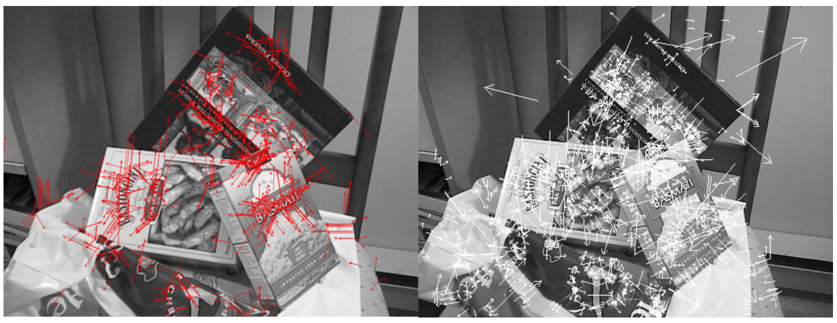


Figure 29.: SIFT result comparison with the reference data.

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 that have many of the characteristics of desktop application software.

HIT	Human Intelligent Task
HC	Human Computation
AI	Artificial Intelligence
GIMPS	Great Internet Mersenne Prime Search
TCP	Transmission Control Protocol
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
SPMD	Single process/program multiple data
SIMD	Single instruction multiple data
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

Andrews, G.R.

nodate “Foundations of multithreaded, parallel, and distributed programming. 2000”, *Wesley, University of Arizona, USA*. (Cited on p. 7.)

Barabási, A.L. *et al.*

2001 “Parasitic computing”, *Nature*, 412, 6850. (Cited on pp. 9, 11.)

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 pp. 11, 25.)

Dean, J. and S. Ghemawat

2008 “MapReduce: Simplified data processing on large clusters”, *Communications of the ACM*, 51, 1. (Cited on pp. xiii, xiv, 7.)

Fraternali, P. *et al.*

2012 “CrowdSearch: Crowdsourcing Web search”. (Cited on p. 11.)

Group, Khronos OpenCL Working *et al.*

2008 “The opencl specification”, *A. Munshi, Ed*.

Howe, J.

2006 “The rise of crowdsourcing”, *Wired magazine*, 14, 6, pp. 1–4. (Cited on p. 3.)

Jenkin, N.

2008 “Parasitic JavaScript”. (Cited on pp. 11, 16.)

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. 11.)

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. 5.)

Parameswaran, M. and A.B. Whinston

2007 “Research issues in social computing”, *Journal of the Association for Information Systems*, 8, 6, pp. 336–350.

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.

Schuler, D.

1994 "Social computing", *Communications of the ACM*, 37, 1, pp. 28–29. (Cited on p. 4.)

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. 7.)

2008 "Designing games with a purpose", *Communications of the ACM*, 51, 8, pp. 58–67. (Cited on p. 7.)

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 p. xiii.)

Wang, F.Y. *et al.*

2007 "Social computing: From social informatics to social intelligence", *Intelligent Systems, IEEE*, 22, 2, pp. 79–83.

WebGL-OpenGL, ES

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

Online resources

Amazon's Mechanical Turk 2012 , `http://www.mturk.com/`. (Cited on pp. xiii, xiv, 4, 5.)

Emscripten 2012 , `http://emscripten.org/`. (Cited on p. 13.)

Express 2012 , `http://expressjs.com/`. (Cited on p. 48.)

FoldIt 2012 , `http://fold.it/`. (Cited on pp. xiii, 6.)

Google Web Toolkit 2012 , `https://developers.google.com/web-toolkit/`. (Cited on p. 13.)

jQuery 2012 , `http://www.jquery.com/`. (Cited on p. 15.)

Modernizr 2012 , `http://modernizr.com/`. (Cited on pp. 15, 42.)

NodeJS 2012 , `http://nodejs.org/`. (Cited on p. 48.)

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