Riccardo Volonterio

# Web-based Human- and Machine-Driven computation

Tesi di laurea specialistica


Relatore: Alessandro Bozzon
Co-relatore: Luca Galli

Politecnico di Milano
Polo regionale di Como
Dipartimento di Elettronica e Informazione
Settembre 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, Settembre 2012* L. P.

# INTRODUCTION

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

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

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

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

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

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

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

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

1 Not web-based, but using standalone clients.

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 | *MTurk* |
| Involuntary | Parasitic computing | GWAP |

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

## ORIGINAL CONTRIBUTION

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

- ease of access to the tasks
- usage of standardized protocols/languages

- ease of implementation by the *requester*
- ease of execution by the users

The original contributions are:

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

## OUTLINE

The thesis is organized in four main parts.

THE FIRST CHAPTER

NEL SECONDO CAPITOLO

NEL TERZO CAPITOLO

NELL'ULTIMO CAPITOLO

# 1 THE BACKGROUND

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

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

## 1.1 CROWD-BASED COMPUTATION DISTRIBUTION

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

Write something about the crowd based distribution of the tasks, use references to (Mechanical turk Little *et al.*, 2010) if possible.

The online tool *MTurk* provide a framework for the creation distribution, execution and result gathering of task (called Human Intelligent Task (HIT)). Diring the creation a *Requester* The *Requester* can push request for executing HIT, these are

### 1.1.1 Human computation & GWAP

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

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

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

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

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

*Centralized*

Centralized Mturk

*Distributed*

Distributed FoldIt

### 1.1.2 Automatic computation

*Voluntary computing*

BOIC + SETI

*Parasitic computing*

Parasitic computing[1] is a technique that, using some exploits and ad-hoc code, permits to execute computation on unaware host computer. This approach was first proposed by Barabási *et al.*, 2001 to solve the NP-complete 3-SAT problem using the existing TCP/IP protocol and its error handling routines.

Parassitic compiting has a strong relationship with *distributed computing*, in fact it is like a specialization of the general class of *distributed computing* where the user is unaware of the execution[2]. Given that we can list the main steps used to perform distributed computing:

- Split task into atomic operations executable by any host
- Send the code to all the host computers
- Execute the code
- Gather the results from the hosts
- Join all the hosts result and compute the task output

Distributed computing leverage on the idea of *divide and conquer* like the programming model of MapReduce[3]. Frameworks as BOINC and SETI@home implement distributed computing paradigm to perform large scale operations (such as signal analisys) among the volunteers that installed the clients. These volunteers choose the project they are interested in and give the idle time of their machines to perform the computation.

Parasitic computing performs the same kind of task in the same *distributed* fashion but the main difference is that the users are unaware of the computation that is being executed on their pc.

1 In this thesis we are not covering, neither we are interested, in the ethical or moral implication of using such programming model.
2 In *distributed computing* the user can be unaware of the purpose computation is for or what actial code they are executing, but they are aware of the execution.
3 Dean and Ghemawat, 2008.

- **Parlare di quante volte effettuiamo computazione parassitica senza sperlo.**
  Esempi?

- **Parasitic computing può anche essere fatto in un modo conscio.** Notificando all'utente la possibilità di eseguire del codice (senza sapere quale) in cambio di un ritorno di qualche tipo (Karame *et al.*, 2011).

- Using the same model of unaware host we can perform high level computation using JavaScript.*Modernizr*

The main drawback of distributed computing is the portability and distribution. The installation of some kind of client to execute the code can be seen as a problem for some user, as an example some users simply cannot install software on their workstation, due to security restriction or missing disk space. The other problem is distribution, the main purpose of these frameworks is to perform massive parallel computation, but for the computation to be really massive we need a lot of volunteers that installed the client on their pc and are online to execute the code.

**Grafico con insiemi per distributed computing and parasitic computing?**

PARASITIC JAVASCRIPT can lead to a solution of these problems using a widespread and standard technologies. Using the Web as the distribution platform the audiance can scale rapidly from to thousands to hundred thousands of users. Regarding the need of third part software installation and security issues, using JavaScript these problems are avoided, because all the code the browsers runs is executed into a sandboxed execution environment so it cannot harm the users pc. The same stands for the portability of the code, bacause almost all bowsers[4] support JavaScript with all the HTML5 features (see 1.2.1), so the porting of the code is guaranteed on every system that can run a browser.

Let make an example **CREARE ESEMPIO CON BOINC E UN SITO DA 500.000 VISITE**

Using parasitc JavaScript can lead to some **hybrid** solution between distibuted and parassitic computing. Using the browser we can ask to user if it is willing to run some code [5] then we can proceed downloading all the required resource to run the code. This approach make possible to have a proactive approach to volunteer computing, so there is no more the need of waiting until the users are willing to spend some time running a task.

This **hybrid** approach is proposed in Karame *et al.*, 2011 as long as a µPayment model for task execution.

- problema del distributed computing (installazione del client | distribuzione) - FATTO

- soluzione: piattaforma standard condivisa da tutti Javascript - FATTO

- problema HTML4 -> HTML5 collegamento - FATTO

- permette una soluzione idriba (avviso che può essere eseguita della computazione, l'utente sceglie) - FATTO

4 **COUGH** IE **COUGH**

5 **mettere una nota in cui si parla del revenue dell'utente e alla sezione in cui viene discusso meglio il tutto**

## 1.2 ENABLING WEB-BASED DISTRIBUTED COMPUTATION

Web-based computation implies that a client is able to perform almost any kind of task that usually is done by an application software, as an example think about image analisys, audio/video playback or socket connection; these operations are available to developers without the need of additional libraries or external *plugins*.

When building Rich Internet Application (RIA) developers have to face the problem of building *rich* web application without the required tools for **communication**, **data access** and **data storage**. Access to raw data of images or audio, API for file management, data storage and full-duplex communication are all problems that could not be solved without using plugins like Flash or Silverlight.

The advent of HTML5 has brought a breath of fresh air to the Web. HTML5 specifies all these features as part of the language specifications so they are being implemented in all mayor javascript engines (Presto, V8, SquirrelFish, JägerMonkey). This means that almost all the required tools to build real *rich* internet application are built-in in the JavaScript language.

**COMMUNICATION** is being empowered by the introduction of *WebSocket* that enable full-duplex data exchange with the server. Also the introduction of Cross-origin Resource Sharing (CORS) give the developers the possibility to contact foreign servers using Asynchronous JavaScript and XML (AJAX) without the need of a proxy for forwarding the requests.

**DATA ACCESS** is obtained using HTML5 media elements (`<video>` and `<audio>`) or the File API.

**DATA STORAGE** is available through the `localStorage` and `sessionStorage` global variables or using IndexedDB or even a built-in WebSQL database.

With the introduction of all these features developers can use the power of JavaScript to perform image analysis, audio/video palyback (without any external plugin installed), create 2D/3D games and so on.

These features make possible to create tools like *Emscripten* that is a LLVM-to-JavaScript compiler. Basically allow developers to convert their C/C++ code into standard JavaScript, obviously the performance are not comparable but different level of code optimization lead to good performance gains in terms of code size and execution speed.

Additionally specification like CORS, not strictly related to JavaScript, allow the users to make cross-site request, that was a great limitation in JavaScript develpment.

### 1.2.1 HTML5

In this thesis when i refer to HTML5 i'm not speaking only about the HTML5 tag reference. I am speaking about a set of thechnologies and specifications related to HTML5. It includes the HyperText Markup Language version 5 (HTML5) specification itself, the Cascading Style Sheets (CSS3) recomendations and a whole new set of JavaScript APIs. So, first things first, lets make some clarification:

**HTML5** refers to a new set of semantic tag (like `<footer>`, `<header>`, `<article>`, ...), media tags (like `<video>` or `<audio>`) and the so called Web Form 2.0.

**CSS3** refers to the presentation layer specification including image effects, 3D transformation, tag selectors and form element validation.

**JS** refers to the new set of API provided, that enable interaction with all these new elements, and additional, non tag-related, functionalities (like WebSockets or WebWorkers).

With the advent of HTML5, like any new web-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. Every borowser has its own implementation of the HTML5, this is mainly due to the early implementation of draft specification[6].

To avoid browser inconsistency we could use JavaScript frameworks. 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 tools like *Modernizr* give developers the ability to test if some HTML5 features are supported or not and provide a general fallback system for dynamically loading polyfills[7].

Now i will analyze in detail the main features of HTML5 to better understand their usefullness.

**CANVAS** Let's start with the official definition[8]

```
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 basically is a *Canvas*, like the name says, but give the developer the access to the raw pixel data of the canvas contents. Also in the canvas element you can draw the image taken from an `<img>` tag or a frame from a `<video>` tag. As you can se now we have the capability to manage image data directly and perform client-side task like image analisys or video manipulation. Obviously there are plenty of JavaScript libraries that give you methods to perform image filtering or generally image manipulation (like Pixastic or Camanjs), other libraries give you the possibility to create images on the fly (like Raphaël or Processingjs).

The canvas element also provide a 3D context to draw and animate[9] 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[10] created for easy development, the most used is the Three JavaScript library, that ca be used for creating and animating 2D or 3D scenes in the canvas element.

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

7 A polyfill is a JavaScript library or third part plugin that emulates one or more HTML5 features, providing websites to have the same *look and feel* also on older browser.

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

9 Animation is not natively supported, you must code it yourself.

10 For a reference see http://en.wikipedia.org/wiki/WebGL#Developer_libraries

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

Like other HTML5 features, WebSocket has a library, build on top of the API, that provides easy access to these functionality as long as a couple of fallbacks. **socket** provide a single entry-point to create a connection to the server and manage the message exchange, it also provide a few fallbacks[11] to ensure cross-browser compatibility.

WEBWORKERS A problem you have to face when you are building computationally heavy JavaScript code is its single thread nature. Every script runs in the same thread, this can lead to some unwanted behaviour like browser freezing or the newly introduced warning dialog "*A script is slowing the browser*". The browser shows the dialog to prevent freezing of crashing of the whole bowser application, but this dialog prevent the script to fullfill their task. So how can we execute long running JavaScript computation if the browser stop the code?

Jenkin, 2008 proposed a timed-based programming structure that ensure the code to be run without any browser warning and also offer the developer to tweak the performance of the script by dynamiccaly 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 time-based loop:

```
while condition do
    ...do something...
end
```

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

Obviously this is not a solution it is a way to hack the browser JavaScript performance monitor and avoid the warning dialog. WebWorkers provide a standard way to create *Workers* that execute in background, also performing heavy computation without harming the browser flow. Let's provide an official definition:

> 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.

So basically fills the gap of parallel code execution in JavaScript.

### 1.2.2 WebCL

With the advent of General-purpose computing on graphics processing units (GPGPU), the spreading of multicore CPUs and multiprocessor programming (like OpenMP) we can see emerging an intersection in parallel

11 WebSocket, Adobe® Flash® Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe,JSONP Polling

computing. This intersection is known as **heterogeneus computing**. Open Computing Language (OpenCL) is a framework for heterogeneus compute resources and so Web Computing Language (WebCL) is a porting of this technlogy to the web.

OpenCL uses a language based on C99[12] for writing *kernels*, functions that actually execute on OpenCL devices.

The main focus when building high-end web-application like 3D games is responsiveness. Altough JavaScript can be optimized and parallelized (see 1.2.1 on page 4) it cannot be fast as an application software, because JavaScript must be interpreted by the browser and then executed as machine code. WebCL provide an easy framework for building and running machine code in parallel directly from the browser.

- Come usiamo noi queste tecnologie
- task monitoring
- SIFT??

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

# 2 THE MODEL

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

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

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

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

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

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

## 2.1 DATA MODEL

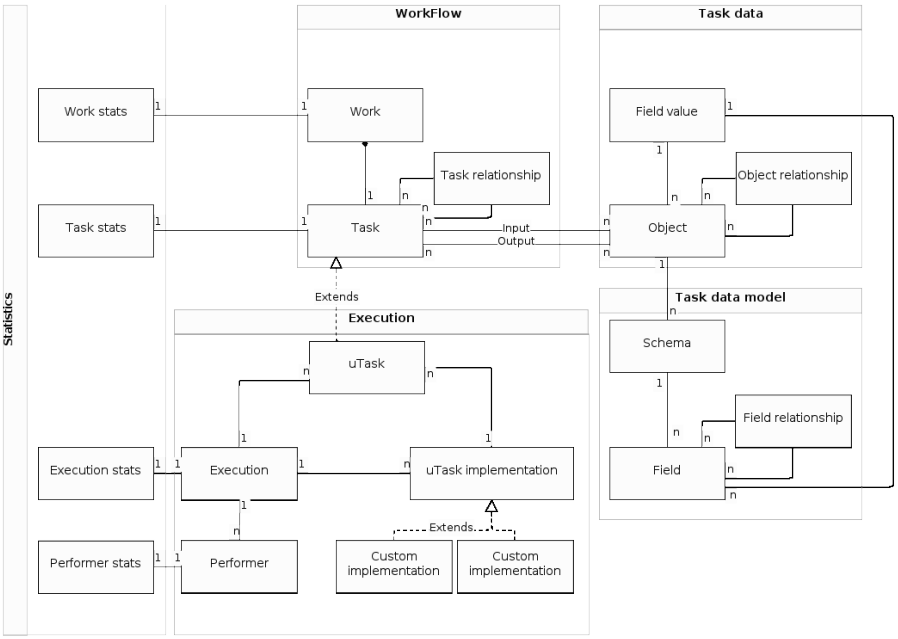


**Figure 1.:** Data Model.

In this section, we define the *Data model* of the System. All the components used in the *Architectural model* are based on this model and all it's. As can be

seen in Figure 1 the *Data model* is composed of 5 parts that together compose a basic human/automatic computation platform.

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

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

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

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

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

**Figure 2.:** Conceptual organization of Work, Task and μTask.

### 2.1.1 WorkFlow

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

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

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

*The Work*

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

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

*The Flow*

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

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

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

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

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

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

*The Task*

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

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

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

  - Order
  - Classify
  - Add
  - ...

  Each Task type is defined by:

  - I/O relationship, defining, at abstract level, how the Task transforms the data and the schema.
  - A default implementation.

- A **Status** encoding the current state of the Task. A Task, can have only one of the following statuses at point of its lifecycle:
  - *Planning-Input*: the Task has been created, have a *Schema* and *Object data* associated and a defined Task *type*.
  - *Planning-*µ*Task*: a set of µTask has been associated to the Task.
  - *Planning-Assignment*: a set of *Performers* has been selected to execute the µTask.
  - *Wait*: Task planned, µTask ready for executiuon.
  - *Running*: µTask are running.
  - *Ended*: all the µTask have completed their execution.
- A set of **Subscriber**s able to recieve updates on the Task execution.
- A set of **Execution constraints** used for priortizing the Task among others or to modify the standard behaviour of the task to fullfill these constraints. The availbale constraints are:
  - Maximum execution time
  - Due date
  - TODO others
- **Configuration data**, provided as JavaScript Object Notation (JSON). For instance the classes we want to use in a calssification Task.
- µ**Task**s TODO????.
- An **Aggregation** function, in charge of collecting the µTask results and generating the Task output.
- µ**Task planning** strategy, in charge of defining how many µTask create for a given Task and associate the right portion of input data to such µTask. For example total disjunction, redundancy, partial overlap, etc.
- **Performer assignment** strategy able to assign *Performers* to µTask. Some strategies can be: manual, random, most reliable, etc.
- µ**Task implementation** strategy in charge of routing the correct µTask implementation for each µTask execution. The routing can be done according to the *user-agent* (e.g. Browser) or to the *user profile* or even *fixed* for all.
- A **Task planning** wich embodies the funcitonalities of µ*Task planning* strategy, *Performer assignment* strategy and µ*Task implementation* strategy deciding the logic behind the invocation of those strategies.

- A **Task control** strategy able to control the status of the Task and if needed perform corrective actions.
- An **Emission policy** specifing wich *Subscriber* need to be notified of a Task change in *Status*.

### 2.1.2 Task Data Model

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

#### *The Schema*

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

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

#### *The Field*

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

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

### 2.1.3 Task Data

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

#### *The Object*

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

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

*The Field Value*

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

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

### 2.1.4 Task Execution

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

*µTask*

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

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

*The Execution*

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

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

#### μ*Task implementation*

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

#### *Performer*

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

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

### 2.1.5 Statistics

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

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

## 2.2 ARCHITECTURAL MODEL

The system use as a reference architecture the one depicted in Figure 3. Here we have a centralized hub that *defines* and *distribute* the workload, a pletora of clients with their browsers and the users. The clients of this model are all coherent and transparent to the execution of the code, wich is distributed to the end-user according to the platform they are using. As you can see the structure is almost the same as any other task distribution platform, the strenghts of this system are in the characterization of the actors in the system.

The reference model in Figure 3 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.

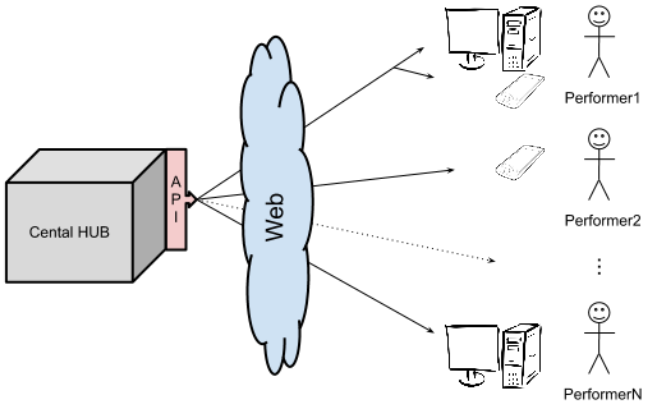


**Figure 3.:** Reference architecture.

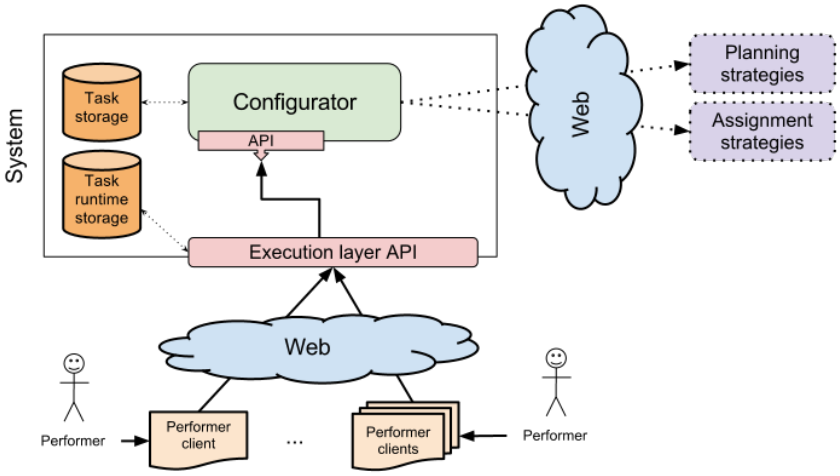


**Figure 4.:** Specialized architecture.

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

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

### 2.2.1 Configurator

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

- Allow the **creation** of a Task, also at abstract level, using either the API or the built-in UI.
- Allow a *Performer* to **execute** the Task using a standard non configurable UI, provided as-is for each Task type.
- Allow to **request information** about a Task, the information that can be requested includes:
  - Retrieve the list of μTask associated with a given Task

  - Post the result of the execution of a given μTask
  - Notify about the completion of a Task or μTask

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

### 2.2.2 Execution layer

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

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

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

### 2.2.3 Task storage & task runtime storage

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

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

### 2.2.4 Performer & Performer client

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

### 2.2.5 Planning strategies

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

### 2.2.6 Assignment strategies

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

## 2.3 EXECUTION MODEL

**Figure 5.:** Representation of the Task execution flow.

Figure 5 represents the flow of the execution of a Task during the execution. As one can notice the flow is almost straightforward except the initial part when the *Task control* strategy tweak the execution to fulfill some predefined requirements.

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

**Table 2.:** Task planning vs. Task Assignment.

| | **Static** | **Dynamic** |
|---|---|---|
| **Static** | 2.3.1 | 2.3.2 |
| **Dynamic** | 2.3.3 | 2.3.4 |

### 2.3.1 Static execution

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

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

### 2.3.2 Static μTask planning & Dynamic assignment

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

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

### 2.3.3 Dynamic μTask planning & Static assignment

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

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

### 2.3.4 Dynamic μTask planning & Dynamic assignment

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

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

## 2.4 PLUGGABLE STRATEGIES ASSIGNMENT

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

### 2.4.1 Built-in strategies model

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

#### μ*Task planning strategy*

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

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

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

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

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

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

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

#### *Performer assignment strategy*

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

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

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

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

*μTask implementation strategy*

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

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

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

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

*Task planning strategy*

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

*Task control strategy*

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

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

The corrective actiona available to the Task controller are:

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

*Aggregation function*

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

*Emission policy*

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

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

### 2.4.2 Custom strategies

TODO ???

*Example 1*

TODO ???

*Example 2*

TODO ???

# 3 THE USE-CASES

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

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

## 3.1 AUTOMATIC

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

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

**Figure 6.:** The interface of the automatic use-case.

**Figure 7.:** Step results of the algorithm.

*Benchmark*

## 3.2 HUMAN

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

*Benchmark*

## 3.3 HYBRID (AUTOMATIC+HUMAN)

Hybrid (Face recognition)

*Benchmark*

**Figure 8.:** Comparison with the reference data.

# 4 IMPLEMENTATION AND EVALUATION

## 4.1 ARCHITECTURE

## 4.2 PERFORMANCE COMPARISON???

# A CONCLUSION AND FUTURE WORKS

## ACRONYMS

**HTML5** HyperText Markup Language version 5

HTML5 is a markup language for structuring and presenting content for the World Wide Web, and is a core technology of the Internet originally proposed by Opera Software.

**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.

**RIA** Rich Internet Application

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

**HIT** Human Intelligent Task

**TCP** Transmission Control Protocol

**JSON** JavaScript Object Notation

**AJAX** Asynchronous JavaScript and XML

**CSS3** Cascading Style Sheets

**BOINC** Berkeley Open Infrastructure for Network Computing

**GWAP** Game With A Purpose

**GPGPU** General-purpose computing on graphics processing units

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

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

**Essential bibliography**

Barabási, A.L., V.W. Freeh, H. Jeong, and J.B. Brockman
2001 "Parasitic computing", *Nature*, 412, 6850. (Cited on p. 2.)

Bozzon, A., M. Brambilla, and S. Ceri
2012 "Answering search queries with CrowdSearcher", in *Proceedings of the 21st international conference on World Wide Web*, ACM. (Cited on p. 11.)

Dean, J. and S. Ghemawat
2008 "MapReduce: Simplified data processing on large clusters", *Communications of the ACM*, 51, 1. (Cited on pp. xi, xii, 2.)

Group, Khronos OpenCL Working *et al.*
2008 "The opencl specification", *A. Munshi, Ed*.

Jenkin, N.
2008 "Parasitic JavaScript". (Cited on p. 6.)

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. 3.)

Law, E. and L. Ahn
2011 "Human computation", *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 5, 3. (Cited on p. 1.)

Little, G., L.B. Chilton, M. Goldman, and R.C. Miller
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. 1.)

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

Turing, A. M.
1950 *Computing Machinery and Intelligence*. (Cited on p. 1.)

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. xi.)

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

**Online resources**

*Emscripten* 2012 , `http://emscripten.org/`. (Cited on p. 4.)

*FoldIt* 2012 , `http://fold.it/`. (Cited on p. xi.)

*jQuery* 2012 , http://www.jquery.com/. (Cited on p. 5.)

*Modernizr* 2012 , http://modernizr.com/. (Cited on pp. 3, 5, 24.)

*MTurk* 2012 , http://www.mturk.com/. (Cited on pp. xi, xii, 1.)

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