Edit, Compile, Execute and Debug C++ on the Web

Degree in Informatics Engineering
Fundamentals of Computing

Final Project

Albert Lobo Cusidó

Advisor
Jordi Petit Silvestre

January, 2017

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - BarcelonaTech

Abstract

The aim of this project is to create a web application to edit, compile, and debug C++ code. This application can be used by instructors to make introductory programming courses more engaging. The first phase of this project provides the planning and design of a software solution to build the application. The main phase describes the implementation of the solution using rapid application development methodology. In the final phase, the implemented solution is evaluated, concluding it is a good candidate for the aforementioned application.

Resum

El propòsit d'aquest projecte és crear una aplicació web per editar, compilar, i depurar codi C++. Els professors poden servir-se d'aquesta aplicació per fer més atractius els primers cursos de programació. La primera fase del projecte aporta la planificació i el disseny d'una solució de software per a construir l'aplicació. La fase principal del projecte descriu la implementació de la solució usant la metodologia de desenvolupament RAD. En la fase final del projecte, la solució implementada s'avalua, concloent que és una bona candidata per l'aplicació mencionada.

Contents

Chapter 1

Introduction

This project details the design and implementation of a web application to edit, compile, and debug C++ programs. The goal is to use this application in introductory programming courses in the *Facultat d'Informàtica de Barcelona*, and is available at:

`https://ide.jutge.org`

The solution presented in this project is what is known as a Web IDE (Integrated Development Environment). This chapter presents a brief introduction to IDEs, states the personal motivation of the author, and describes the structure of this report.

1.1 Brief introduction to IDEs

An IDE is essentially a program in which development is done. This contrasts with software development using unrelated tools, such as a text editor and a terminal. IDEs typically provide features for editing source files and compiling them, as well as deploying and inspecting the resulting software.

IDEs are designed to maximize programmer productivity by providing tight-knit components with a common user interface. They take care of the configuration necessary to piece together multiple development utilities, and present the same set of capabilities as a cohesive unit. This increases developer productivity because learning to use the IDE is faster than manually integrating all of the individual tools (see [43] and [55]).

When programmers started using the console to develop, IDEs natu-

rally appeared. A good example of this is GNU Emacs [21], an extensible editor that is commonly used as an IDE on Unix-like systems. Since then, many IDEs have been created; some of them are dedicated to a particular language (such as Pharo [57] for Smalltalk [62]), and others support multiple languages (such as Eclipse [28]). According to [4], the most popular IDEs as of today (December 2016) are Visual Studio [54] and Eclipse [28].

With the advent of cloud computing, many IDEs are available online and run within web browsers. According to [5], the most popular Web IDEs are Cloud9 [6] and JSFiddle [46]. However, Web IDEs have a much more limited feature-set because they are operating in a web browser; in particular, there is currently no Web IDE that allows debug of C++ programs.

1.2 Personal motivation

I have always used IDEs to do my programming projects. The first programming language I learned was C++, and I used an IDE called Turbo C++. One of the first things I was taught was that I could see what my program was doing at any moment, using a feature called the debugger. With just a few button clicks, I managed to stop execution in the middle of the program, and saw what variables I had and what their value was. I could even see which functions had been called before the program paused!

The value of using an IDE to learn was not so clear to me until I actually came to university. Surely using a text editor and a terminal to code C++ was an act of masochism! Before I could even get the program running, I had to find and install a C++ compiler, and figure out the correct command options. On top of that, there was still no way to pause the program and see the variables —I had to get a debugger and learn how it worked. In practice, though, students ended up plaguing their code with unnecessary print statements. Why all this complication, I wonder, when all students need is to learn the basics of C++?

One of my passions is programming —learning more and honing my skills is the reason I started the Degree in Informatics Engineering in the first place. Being offered this project was perfect for me, because I would be able to use my programming abilities, and mindset, to create something genuinely useful, as well as learning the wide range of technologies involved in the process, and applying knowledge gained in the Fundamentals of Computing specialization.

1.3 Report structure

This report consists of four different parts. Each part describes an essential phase of the project's development.

Formulation: Identifies and analyzes the problem to solve, specifies the scope of the project, and shows the design of the solution.

Planning: Describes the time plan and budget to develop the project.

Implementation: Describes the development of the different components that integrate the solution.

Evaluation: Describes the validation methodology, reviews time management and costs, and discusses the sustainability and legality issues of the project.

Part I

Formulation

Chapter 2

Analysis

This chapter describes the problem that this project aims to solve, and decides whether there is an existing solution that may apply to solve such problem.

2.1 Context

Jutge.org [45] is a programming judge specially built with an educational aim. For the past ten years, this tool has played an essential role in the task of improving the teaching in the programming courses at UPC [44].

The advisor of this project is a founder of the *Jutge.org* platform. The work produced in this project intends to provide an enhancement to the platform: to build a web application for students to inspect their C++ programs in an easy and practical way (C++ is the most used programming language amongst *Jutge.org* users [44]).

Indeed, this new web application for *Jutge.org* hopes to help teachers make the introductory programming courses more engaging by providing an environment in which students can write C++ code and immediately see the results of its execution.

2.2 The problem

The inspection of programs is typically done with a utility called the *debugger* (debug: to identify and remove errors from software). This is a key part of

software development, although it proves to be difficult for newer students.

The first step for programming students is to find and install both the compiler and debugger programs. This can be tricky, depending on which operative system they are using. But the hardest problem lies in the interaction with said utilities —the debugger in particular—, which requires learning and combining a lot of commands that will only make sense to someone who has a deeper knowledge of what is going on under the hood.

2.3 The requirement

To help students focus on the more important aspects of programming, such as learning basic techniques (for example, recursion and iteration), this project aims to build an application for students to debug their programs using only a modern web browser.

2.4 State of the art

There are quite a few Web IDEs that support the C++ language. Some of the most popular free ones are:

- Cloud9 [6]
- TutorialsPoint - Coding Ground [69]
- ideone [51]

As much as these IDEs allow editing and executing C++ code, none of them currently provide debugger features.

Given that there are no readily-available tools that fulfill the project's requirement, a new brand solution is necessary.

2.5 Stakeholders

Given the previously described requirement, the different individuals who will use the platform are considered:

Students: They want to code and debug their programs easily, without having to worry about details on how the debug process actually works.

UPC staff: They will supervise and, potentially, improve upon the project platform by making plugins for other languages or updating the C++ plugin.

Project staff: The author of this project will have to take the roles of a *project manager*, a *software engineer*, and a *software developer*, in order to complete it.

Chapter 3

Objectives

This chapter describes the different objectives that the project must fulfill in order to satisfy the requirement stated in the previous chapter.

3.1 Main objectives

The main objective of this project is to build a Web IDE for students to create and inspect their C++ programs. The name of this tool will be, appropriately enough, *ide.jutge*.

This objective involves creating all the individual components or utilities that will make up the IDE. More details about each one of these components, and how they interact, will be provided in chapter 4, Design.

3.2 Secondary objectives

This project also pursues the following secondary objectives:

3.2.1 Create an extensible plugin system for customizing the environment

Much like Eclipse, or IntelliJ, the *ide.jutge* should allow extra features and functionality to be added through a plugins system. This way, the application

would be able to support other languages such as Java, or help to solve *Jutge.org* problems by automatically executing public test cases.

3.2.2 Make the infrastructure scalable, stable and secure

The underlying infrastructure must be able to handle a large amount of users without hindering performance. Moreover, the solution must be secure and fault-tolerant; user programs must not be able to break or affect the solution negatively.

3.2.3 Create a step-by-step debug mode

The IDE must be able to execute programs one instruction at a time, with a brief pause of a few seconds between instructions. This execution mode will be referred to from here on as *Slow motion debug*.

3.2.4 Detect memory errors in the student's programs

The IDE must be able to pinpoint memory-related errors in the programs, such as an illegal memory access, or memory leaks.

3.2.5 Import and export code from GitHub.com

Support to read and write Gists [37] must be provided. Many programmers, including students, are currently using GitHub to manage their programming projects -even the source code of this project is hosted in a private repository at GitHub.com.

Chapter 4

Design

This chapter describes the design and general workflow of the solution, states the security concerns taken into account, and details the architecture on which *ide.jutge* is built.

4.1 Security concerns

The architecture of *ide.jutge* is largely dependant on the possible attacks such a system must deal with, the requirement to debug C++ programs in real time, and the need to make such a system scalable.

As any IT system, *ide.jutge* will have to cope with several threats. Unlike many other systems, *ide.jutge* will welcome students to submit their potentially malicious codes and execute them on their behalf. An analysis of the risks in such environments can be found in [18]. These include: accessing restricted information; misusing the network; modifying or harming the environment; exploiting covert channels; misusing additional services; and exploiting bugs in the OS. Consequently, this system must provide a secure execution environment to execute arbitrary programs without comprising the stability of the system nor its confidential information.

4.2 Debugger

A debugging tool is going to be necessary to allow programs to be inspected. It will be the underlying utility that *ide.jutge* will operate to provide debugger

features.

Therefore, it is convenient to look for a debugger which offers support to front-ends (applications which take the debugger's output and presents the state of the program being debugged). For this reason, the GBD [14] debugger will be used, as it features a machine-oriented text interface to communicate with *ide.jutge*.

4.3 Architecture

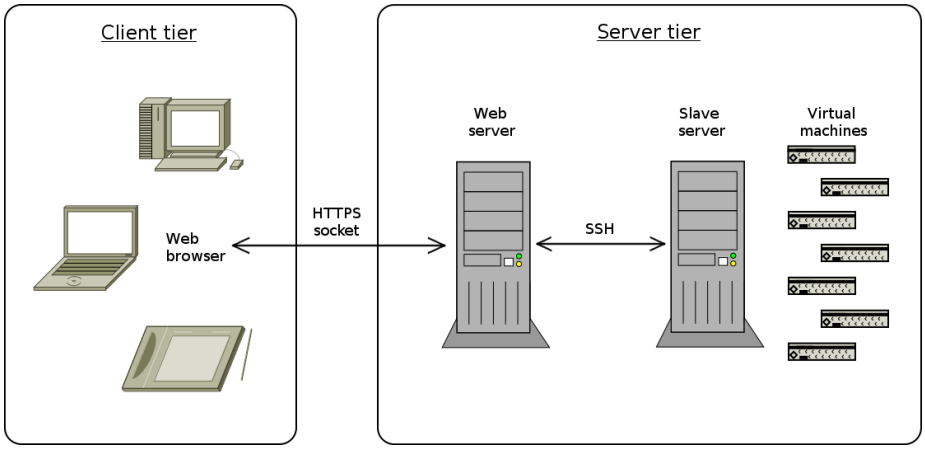

Figure 4.1: Solution architecture

A figure 4.1 diagrams the solution's architecture. As it is shown, *ide.jutge* is divided into two main tiers.

4.3.1 Client tier

The client tier consists of application clients that access the *ide.jutge* web server. In particular, these clients are C++ applications running in a web browser (ie. *Google Chrome*, *Mozilla Firefox*).

The *ide.jutge* C++ client application will provide a GUI (Graphical User Interface) for the students to code and debug their programs. It is worth noting that, since web browsers are available for a wide range of devices, students will be able to control the IDE with mobile devices such as tablets, as well as laptops and PCs.

4.3.2 Server tier

The server tier harbors the engine of the *ide.jutge*. Its purpose is to manage the compilation, execution, and debugging of the C++ programs. In essence, it must supply all the core functionality of the IDE to the client application.

In order to protect the integrity of the system, the compilation, execution and debug of the student's programs is performed inside sandboxed environments —virtual machines. These sandboxes are given a low-privilege user with limited file system access, and have no network access (ie. no Internet). Also, memory usage is restricted, as well as the number of concurrent processes (as suggested in [73]).

A set of servers working together compose the server tier:

Web server

The Web server acts as an intermediary between the client applications and the backing sandboxes. It serves the C++ client application, and handles its incoming requests (such as executing a program, or debugging). Because the underlying utilities work asynchronously, ie. the debugger, and responses must arrive quickly, most of the communication with the client application will use web sockets.

To enforce security, all connections with the client tier go over HTTPS, and connections with the slave servers use SSH.

Slave servers

Their only purpose is to run the sandboxes. Having separate slave servers helps enforce security by further isolating the execution of the student's programs. It also alleviates work from the web server and adds horizontal scalability to the system.

4.4 General application workflow

Students will access the *ide.jutge* website. The web server will respond by serving the client application in the form of HTML and JavaScript files. When the application is fully initialized in the client, it will request a sandbox. This request will be passed from the web server to the slave server, which will get a

new virtual machine ready. Once that is done, the client will be notified, and the student can start executing or debugging.

As previously stated, once execution or debug starts, there is no way to anticipate when the C++ program will finish, or pause at a given breakpoint. Hence, a socket between the client app and the web server is opened, granting bidirectional, instantaneous communication. In other words, when something happens in the executing program, the *ide.jutge* GUI will react accordingly, notifying the student in the process.

Finally, when the student closes the connection with the web server (for instance, by closing the browser, or navigating to another site), the virtual machine will be destroyed, thus freeing all allocated resources.

Chapter 5

Licenses

All code and documentation for *ide.jutge* will be hosted in a public repository at `https://github.com/llop`.

5.1 Code

The code for *ide.jutge* will be released under the the MIT licence [67].

5.2 Documents

This report and additional documentation related with the *ide.jutge* will be released under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence [8].

Part II

Planning

Chapter 6

Time Plan

This chapter describes the project's time plan, and gives an estimation of the total cost of production.

6.1 Estimated project duration

The project starts on February 2, 2016 and will be finished by June 2, 2016. Hence, its duration will be of 4 months —however, given that its presentation will be in January 2017, additional upgrades will make it into the final delivery. These improvements will be noted where appropriate in Part III and Part IV of this report.

6.2 Tasks

This section describes the various tasks and subtasks that need to be accomplished in order to complete the project. The following details are provided for each task: its description, expected duration and assignee, resources used, possible complications, and task dependencies —if any exist. The resources needed will also be listed.

6.2.1 Analysis and design

To satisfy the project requirements, a choice of technologies and software will be made. It is important to establish the architecture of the solution, and determine how all the components will connect.

Expected duration: 20 hours.

Assignee: Software engineer.

Resources: Extensive information about the available technologies and software, requiring in-depth investigation to find the most suitable utilities.

Possible complications: Some components could change during the implementation of the project, especially on the web front-end.

Task dependencies: None.

6.2.2 Environment setup

A development computer must be prepared to carry out the project. All the required software must be installed, and the machine must be configured to run the solution's architecture.

Security enforcement will require that development SSH keys and SSL certificates be generated.

Expected duration: 20 hours.

Assignee: Software engineer.

Resources: A laptop computer.

Possible complications: None.

Task dependencies: Analysis and design.

6.2.3 Sandbox

A virtual machine must be created and configured with the appropriate software to run the students' C++ programs.

Expected duration: 20 hours.

Assignee: Software engineer.

Resources: A laptop computer.

Possible complications: None.

Task dependencies: Environment setup.

6.2.4 Drivers

Drivers for the SSH communication and debugger must be created. These will allow the web server application to interact with the debugger. Extensive study of debugger IO syntax will be required to code the second driver.

Expected duration: 100 hours.

Assignee: Software developer.

Resources: A laptop computer.

Possible complications: This task can is critical to the project; special care must be taken to ensure:

- *SSH communication is fast.* For that, a test will be made to verify the SSH session doesn't close between messages.
- *Changes in the debugger's output format do not break the driver.* To avoid this, the driver will be coded and tested using the same debugger utility that is installed in the sandbox.

Task dependencies: Environment setup. This task can be done at the same time as the *Sandbox*.

6.2.5 Server application

In this task, the application that will handle all HTTPS communication with the students must be created. This app will be running, eventually, in the *Jutge.org* servers.

Expected duration: 100 hours.

Assignee: Software developer.

Resources: A laptop computer.

Possible complications: Complications can arise due to buggy code; the author will have to learn the chosen web application framework.

Task dependencies: Drivers.

6.2.6 Client application

This task requires to build the application that will be running in the students' browser, and will allow them to debug their C++ programs.

Expected duration: 100 hours.

Assignee: Software developer.

Resources: A laptop computer.

Possible complications: Complications can arise due to buggy code; the author will have to learn the chosen JavaScript components.

Task dependencies: Server application.

6.2.7 Testing and polishing

In this task, extensive testing will be conducted to ensure the whole system works as expected.

Expected duration: 40 hours.

Assignee: Software developer.

Resources: A laptop computer.

Possible complications: Too many bugs in the tested components may increase the time to complete the task. To avoid this, all the previously created components' API must be well defined, and each one individually tested.

Task dependencies: Client application.

6.2.8 Project management course

The GEP (Project management) course aims to help lead the project in the right direction. Different parts of the project will be outlined here: context and scope, time planning and economic viability.

Expected duration: 75 hours.

Assignee: Project manager.

Resources: A laptop computer.

Possible complications: None. The course aims to eliminate those by providing the author with feedback.

Task dependencies: None.

6.2.9 Project report

Writing a document explaining how the project was carried out.

Expected duration: 40 hours.

Assignee: Project manager.

Resources: A laptop computer.

Possible complications: None. The GEP course will have provided the necessary indications to successfully finish it.

Task dependencies: Project management course.

6.2.10 Project presentation

An oral presentation will conclude the project.

Expected duration: 10 hours.

Assignee: Project manager.

Resources: A laptop computer.

Possible complications: Live examples have to be well prepared.

Task dependencies: Project report.

6.3 Time table

Table 6.1 shows the duration of every task described in the previous section. The total duration of the project adds up to 525 hours. For the larger part of the project (up until the *Testing an polishing* task), the author will need to work for $\frac{400hours}{16weeks} \simeq 25$ hours a week. This seems like a reasonable workload for four months. The rest of the tasks can also be accomplished because the presentation will be in January 2017, and that leaves the author 3 months to do 125 hours of work, which is also reasonable.

Task	**Expected duration (h)**
Analysis and design	20
Environment setup	20
Sandbox	20
Drivers	
Learning	30
Implementation	20
Testing	40
Integration	10
Server application	
Learning	35
Implementation	30
Testing	30
Integration	5
Client application	
Learning	35
Implementation	30
Testing	30
Integration	5
Testing and polishing	40
Project management course	75
Project reoprt	40
Project presentation	10
Total	525

Table 6.1: Task durations

6.4 Gantt chart

Figure 6.2 shows the expected time-line of the project, taking into consideration task dependencies.

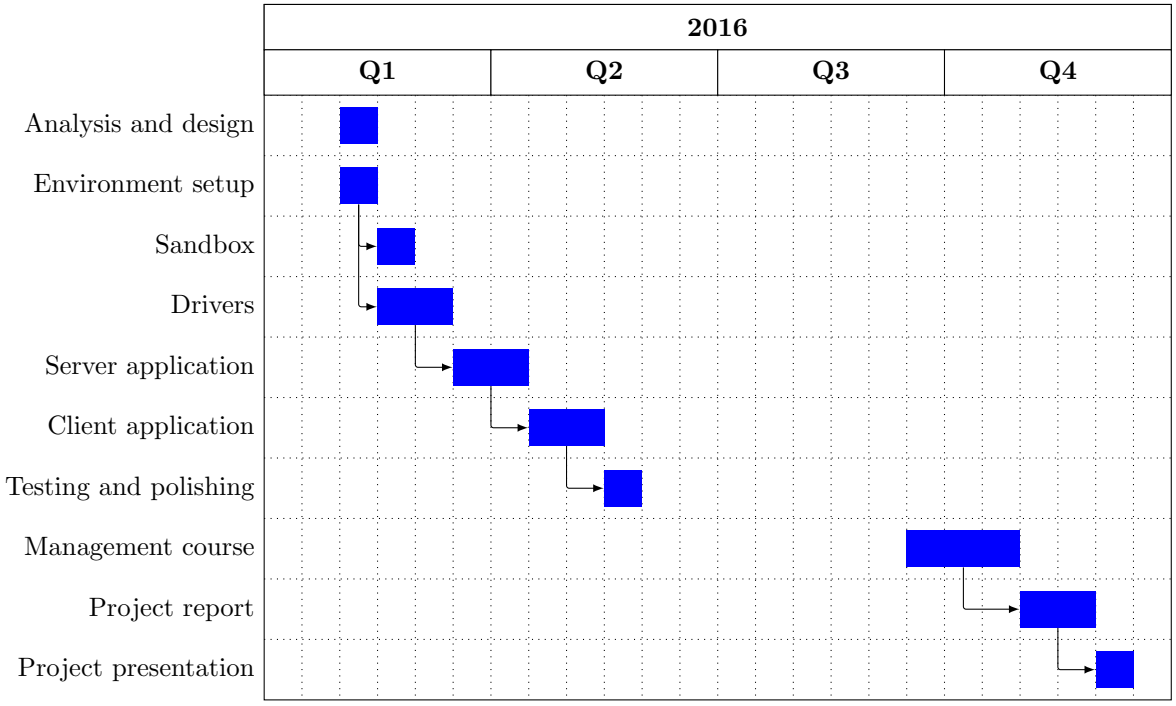

Table 6.2: Gantt chart

Chapter 7

Budget

This chapter presents a detailed estimate of all the costs required to complete project tasks. It specifies costs for staff labor, materials procurement, ongoing operating costs and other direct costs such as Internet connection.

7.1 Hardware

Only hardware required for development has been taken into account. The application is expected to run on the *Jutge.org* servers, but these have not been considered as *ide.jutge* will go live after development is finished.

This project will require a laptop computer. No other hardware is needed.

Hardware	**Cost(€)**	**Useful life (years)**	**Amortized cost(€)**
ASUS VivoBook S301L	800.00	4	66.66
Total			66.66

Table 7.1: Hardware budget

7.2 Software

The software chosen to develop this project is available for use at no monetary cost. After the *Analysis and design* phase, several utilities are selected, and Table 7.2 shows them along with their corresponding licence.

Software	License
Ubuntu	Free software licenses (mainly GPL) `https://www.ubuntu.com/about/about-ubuntu/licensing`
LaTeX	LaTeX Project Public License (LPPL) `https://www.latex-project.org/lppl`
git	GNU GPL v2 and GNU LGPL v2.1 `https://git-scm.com/about/free-and-open-source`
gedit	GNU GPL v2 or later `https://www.gnu.org/licenses/gpl-2.0.txt`
Google Chrome	Freeware under Google Chrome Terms of Service `https://www.google.com/intl/en/chrome/browser/privacy/eula_text.html`
LXC	GNU LGPL v.2.1 (some components under GNU GPL v2 and BSD) `https://linuxcontainers.org/lxc/introduction`
GCC	GNU GPL 3+ with GCC Runtime Library Exception `https://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html`
GDB	GNU GPL `https://www.gnu.org/licenses/gpl-3.0.txt`
Node.js	MIT `https://github.com/nodejs/node/blob/master/LICENSE`
Express.js	MIT `https://github.com/expressjs/express/blob/master/LICENSE`
socket.io	MIT `https://github.com/socketio/socket.io/blob/master/LICENSE`
AngularJS	MIT `https://github.com/angular/angular.js/blob/master/LICENSE`
Bootstrap	MIT `https://github.com/twbs/bootstrap/blob/master/LICENSE`

Table 7.2: Software budget

7.3 Staff

The primary cost in this budget is project staff. Only one person is required to complete the project, but they have to be in the role of a project manager, a software engineer, and a developer. Table 7.3 shows their corresponding salaries.

The total cost has been calculated by adding the cost of individual tasks, and the task durations used have been detailed in the previous chapter *Time plan*. Table 7.4 shows the expected cost of human resources according to project roles and their respective tasks.

Role	Salary(€/h)
Project manager	50.00
Software engineer	40.00
Software developer	30.00

Table 7.3: Salary per role

Role	Task	Time (h)	Cost(€)
Project manager	Project management course	75	3750.00
	Project report	40	2000.00
	Project presentation	10	500.00
Software engineer	Analysis and design	20	800.00
	Environment setup	20	800.00
	Sandbox	20	800.00
Software developer	Drivers	100	3000.00
	Server application	100	3000.00
	Client application	100	3000.00
	Testing and polishing	40	1200.00
Total		525	18850.00

Table 7.4: Staff budget

7.4 Other costs

Electric power is required for the laptop computer to work. Assuming 0.11 €/kWh in Spain [42], a consumption of 100 W, and a usage time of about 525 hours, its estimated cost adds up to 5.77€.

A working Internet connection is needed too. The price of this is 42.2€/month in Spain [7] $\simeq$ 0.06€/hour. It is expected to use the Internet connection during 30% of the project's total duration. Thus, the estimated budget for the Internet connection is $525\text{h} \cdot 0.06\ €/\text{h} \cdot 0.3 = 9.45\ €$.

7.5 Total

Table 7.5 depicts the total budget needed to develop the project. 10% of the total cost is added to face any unforeseen contingencies.

Resource	**Total cost(€)**
Hardware	66.66
Software	0.00
Staff	18850.00
Electricity	5.77
Internet	9.45
Subtotal	18931.88
Contingency (10%)	1893.19
Total	20825.07

Table 7.5: Total budget

Chapter 8

Sustainability Analysis

This chapter aims to document the various aspects of sustainability regarding this project.

8.1 Economic sustainability

Costs for all kinds of resources have been accounted for (hardware, staff, etc.). There will be no need for software or hardware updates, and in any case, a 10% additional amount has been added to the total cost in case contingencies arise.

The largest part of the total cost of this project comes from human resources, since all the software is free, and minimal hardware is required. The time estimation cannot realistically be much improved, since building the software in this project is a very delicate process. Wherever possible, existing technologies and frameworks have been used, such as LXC, Node.js and AngularJS. Hence, this would be a viable, competitive project in the real world.

Collaboration with *Jutge.org* is planned -this project will be made available to students as a learning tool in the introductory programming courses. No hardware or hosting costs will be incurred after project development, as *ide.jutge* will be hosted in the UPC's *Jutge.org* servers.

8.2 Social sustainability

Currently, students have to use GUIs such as Eclipse, or NetBeans, to debug their C++ programs. Since setting up the C++ debugging tools is rather complicated, many students end up using workarounds to debug, such as filling their programs with print command (ie. `cout`).

This project will provide a tool for everyone to debug their programs in a more visual way, which is also the preferred way to carry out this task in most work places. There will be no-one harmed by the existence of this project.

8.3 Environmental sustainability

This project will have a minimal impact on the environment. The laptop will be used for many other things, and will be sent to a recycling facility after its life ends. Also, the amount of power consumed will be negligible.

After project development, *ide.jutge* will remain environmentally friendly; it will be running in the *Jutge.org* servers, which comply with European sustainability standards regarding power consumption, and equipment disposal.

8.3.1 Sustainability matrix

Figure 8.1 represents the project's sustainability matrix.

	Project Development	Exploitation	Risks
Environmental	Consumption Design	Ecological footprint	Environmental risks
	10	20	0
Economic	Project bill	Viability plan	Economic risks
	10	20	0
Social	Personal impact	Social impact	Social risks
	10	20	0
Sustainability range	30	60	0
	90		

Figure 8.1: Sustainability matrix

Part III

Implementation

Chapter 9

Methodology

Developing various pieces of software that communicate with each other is difficult, more so if those pieces are being developed at the same time. Changes made to one component's public interface may break other components. So, it is essential to detect when something is not working in order to remain productive.

9.1 Rapid Application Development

The project is developed using the RAD methodology [53]. This approach has been chosen over more classic methodologies, such as the Structured Systems Analysis and Design Method [2] and other Waterfall models, in which a rigorously defined specification needs to be established prior to entering the development phase.

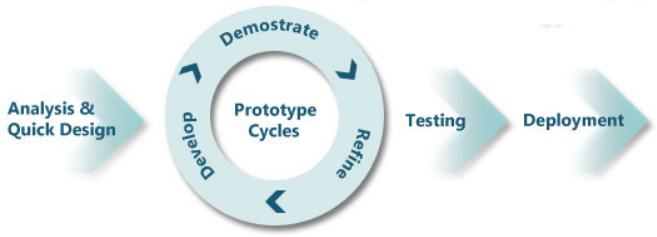

Figure 9.1: RAD methodology

RAD focuses on gathering requirements, and early testing of the prototypes using iterative design and continuous integration. A figure 9.1 diagrams the RAD model —analysis, design, build, and test phases are distributed into a series of short, iterative development cycles.

This methodology has many advantages. It allows swift detection of bugs and provides working prototypes even in the early stages of development. Stakeholders can try the software while it is being developed and give feedback. In this case, the prototype was available to the project advisor to assess and follow any progress closely, and an almost complete prototype was presented to the UPC staff for the final polishing.

The RAD model can be applied successfully to this project. *ide.jutge* is a solution in which some requirements will change during the course of the project. The impact of changing needs will be minimized by early testing of components in quick cycles. Also, *ide.jutge* can be modularized to be delivered in an incremental manner -prototype iterations will involve creating components and integrating them into the system.

9.2 Version control

Git is the used version control system. GitHub.com hosts the solution's code, tests and related documents, in a repository at `https://github.com/llop`.

Chapter 10

Debugger Prototype

The first step of the implementation process was to build a simple prototype of the solution as proof of concept. This initial prototype must implement the most basic feature of the solution: it must allow interaction with the debugger.

A choice of technologies must be made in order to create this prototype. At this point, the only concern is to find:

1. The tools to compile and debug C++.
2. The software that will interact with these C++ tools.

10.1 C++ utilities

The GNU Project [19] provides useful tools to compile and debug C++ code. These tools are the optimal choice because they work well in most UNIX variants (this project will be developed on an Ubuntu OS), and present all the benefits of *Free Software* [22].

This choice is also motivated by a will to maintain consistency with the rest of the *Jutge.org* platform, which uses the GNU Project's `gcc` and `g++` utilities to compile C and C++ code.

10.1.1 GCC

The GNU Compiler Collection includes a front end for C++ that supports the C++11 and C++14 standards. It is also the same compiler *Jutge.org* uses to compile C++ in problem corrections.

10.1.2 GDB

GDB is the GNU Project's debugger. The main benefit of GDB is its line-based machine-oriented text interface, GDB/MI. GDB/MI is specifically intended to support the development of systems which use the debugger as just one small component of a larger system [20], by specifying a standard syntax for input and output (very similar to JSON [47]).

10.2 The solution stack

The stack is the set of components needed to create the complete solution such that no additional software is required. To determine the most appropriate technologies, the following key factors are considered:

Componentization: There are going to be several components, and each one will handle a related set of functions.

Ease of development: The technologies must be user-friendly for the developer; learning and coding them has to be quick.

Community: A large community of contributors and users guarantees that any problem can be solved promptly, and help is readily available.

10.2.1 Full-stack JavaScript

JavaScript is the optimal programming language to be used across the stack, since the solution requires building components on both the *Client* and the *Server* tiers (see chapter 4, *Design*).

Client tier

ide.jutge will have to provide a dynamic and interactive interface within the web browser. Platforms like *Adobe Flash* or *Microsoft Silverlight* could deliver

such interactivity. However, JavaScript is a more suitable choice because all web browsers include a JavaScript interpreter, and these other technologies have very limited support (especially in mobile devices).

Server tier

The components in this tier are required to serve the HTML and JavaScript files for the client application, and interact with the C++ utilities on behalf of the students.

There are many languages that could work well -ie. Ruby, Python, Go, Elixir. Nevertheless, the appearance of the *Node.js* JavaScript runtime [23], the *npm* JavaScript package manager [40], and compliance with the previously stated key factors make JavaScript the most suitable candidate:

- Having the same language on both client and server simplifies communication between the tiers.
- *Node.js* currently has a large community, and over 150 thousand tagged questions in *stackoverflow.com* [64].
- Components can be reused across tiers. They can be defined on the server using *Express.js* middleware [26].
- *npm* hosts many useful packages which can provide part of the core functionality —for instance, the socket.io package enables real-time bidirectional event-based communication [12].

10.3 Design

A figure 10.1 diagrams the design of the first prototype. Two modules (in *Node.js*, components are called modules) must be created:

10.3.1 gdb-mi-parser

Parses GDB/MI output into JavaScript objects.

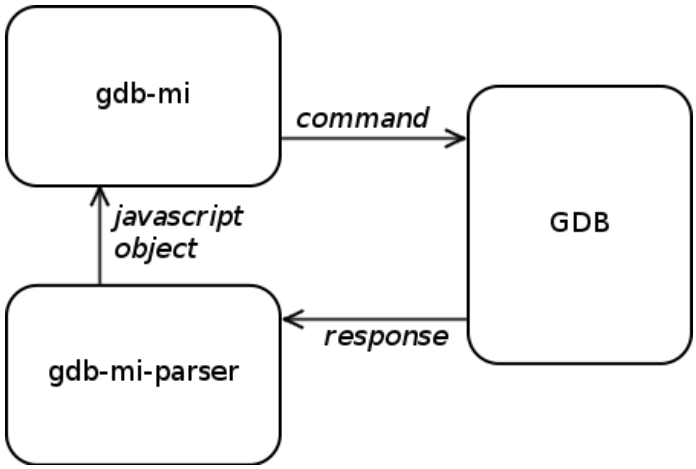

Figure 10.1: First prototype design

10.3.2 gdb-mi

This module is capable of sending commands to the debugger, and relies on the *gdb-mi-parser* module to process the response. Available commands are:

- **load:** starts debugging a given program with GDB.
- **run:** resumes the program's execution.
- **pause:** pauses the program's execution.
- **stop:** kills the program.
- **step over:** advances the program's execution to the next line of code, ignoring function calls.
- **step into:** advances the program's execution to the next line if it is not a function call. If it is, it stops at the first instruction of the called function.
- **step out:** resumes the execution of the program until the current function is exited.
- **insert breakpoint:** inserts a breakpoint at a given line.
- **delete breakpoint:** removes a breakpoint from a given line.
- **list variables:** displays the names of local variables and function arguments.
- **call stack:** lists the frames currently on the stack.
- **evaluate expression:** evaluates a given expression, and returns the result.

10.4 Configuring the C++ tools

This section describes the configuration of the C++ utilities.

10.4.1 g++

To prepare the C++ code for execution, the `g++` compilation tool must be executed using the following command:

```
g++ -g -std=c++14 <code-file>
```

Note that there are no optimization option flags (ie. `-O2`, `-O3`); that is because, oftentimes, optimized programs' execution does not match the source code. The `-g` option enables use of extra debugging information that GDB can use.

10.4.2 gdb

Once the code has been appropriately compiled, GDB is ready to debug the resulting executable file:

```
gdb -i mi <executable-file>
```

The `-i` option causes GDB to use the GDB/MI interface.

10.5 Testing

To test the prototype, a simple C++ program was compiled for execution (see Figure 10.2). All the *gdb-mi* module's commands were evaluated, and the responses checked for correctness.

Right away, it was discovered that GDB would require additional configuration to run smoothly. GDB operates by default in asynchronous mode and it will not accept a new command until the previous command is done. As a consequence, execution cannot be explicitly paused nor killed once it has started, and it is impossible to set breakpoints while the program is running. This is a problem because it would make the IDE unresponsive for indefinite amounts of time —something unacceptable for *ide.jutge*.

Fortunately, it is possible to have GDB operate in asynchronous mode

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void func(int x) {
6    string s = "Hello";
7    cout << s << '␣' << x << endl;
8  }
9
10 int main() {
11   vector<int> v(1);
12   cin >> v[0];
13   func(v[0]);
14   return 0;
15 }
```

Figure 10.2: main.cc

by calling the appropriate gdb commands in the .gdbinit file (.gdbinit is a file to specify commands that will get executed when GDB starts).

Another problem had to do with how STL data structures are printed (the `vector`, in this case). The default output format is not legible, but it is possible to get GDB to pretty-print by, again, modifying .gdbinit.

Figure 10.3 shows the .gdbinit file with all necessary configuration options.

```
python
import sys
sys.path.append('/usr/share/gcc-6/python')
from libstdcxx.v6.printers import register_libstdcxx_printers
register_libstdcxx_printers (None)
end
set pagination off
set non-stop on
set target-async on
```

Figure 10.3: main.cc

10.6 Integration with *npm*

To gain more knowledge on how the JavaScript packages work, the *gdb-mi* and *gdb-mi-parser* modules were published in npm. Registration is required, and publishing a package can be done with a single command.

10.6.1 gdb-mi

- npmjs.com: `https://www.npmjs.com/package/gdb-mi`
- github.com: `https://github.com/llop/gdb-mi`

10.6.2 gdb-mi-parser

- npmjs.com: `https://www.npmjs.com/package/gdb-mi-parser`
- github.com: `https://github.com/llop/gdb-mi-parser`

Both packages are available under the MIT licence.

Usage statistics of the packages can be found at `http://www.npm-stats.com/` (see Figures 10.4 and 10.5).

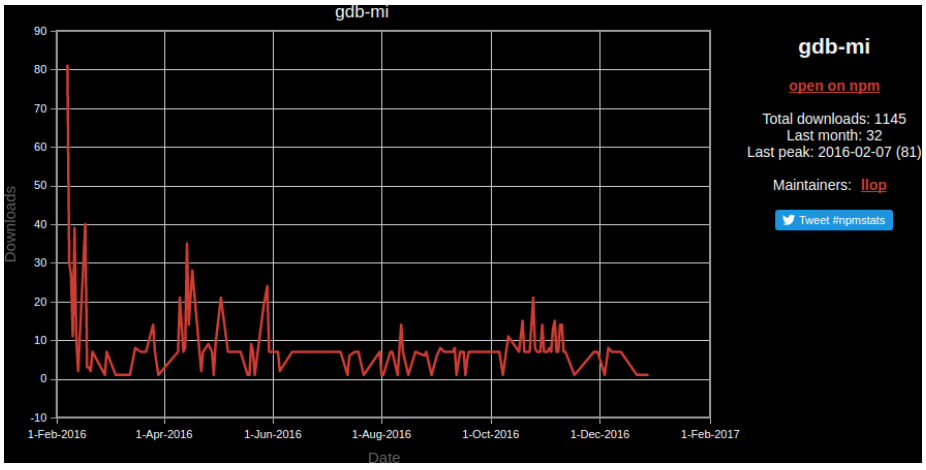

Figure 10.4: gdb-mi usage statistics

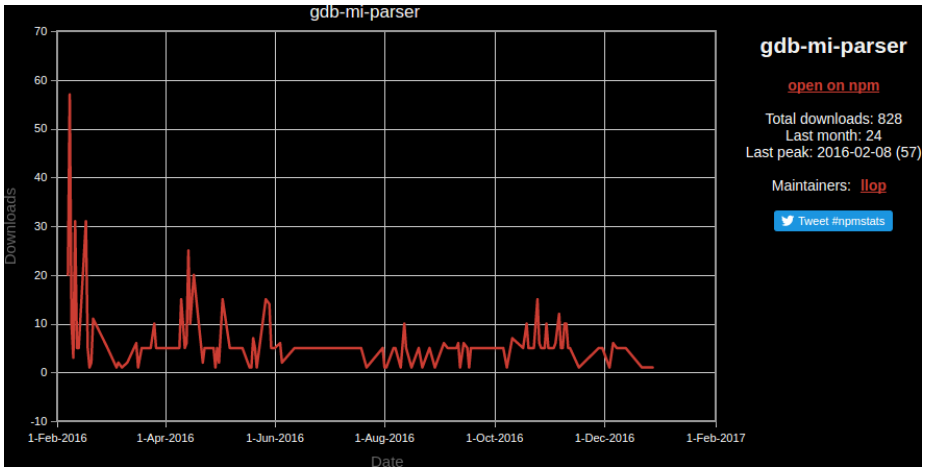

Figure 10.5: gdb-mi usage statistics

10.7 Summary

A description of how the first prototype was built has been given. This prototype successfully implemented the most basic feature of the IDE: interaction with the debugger. During the analysis phase, the required utilities and software were defined. The testing phase allowed us to identify problems early on, and ensured that the design is valid and the prototype is working correctly.

Chapter 11

Express Prototype

Students will interact with the debugger using a web browser, so it is essential to have a server application to handle communication.

The Express Prototype provided the web framework (a set of software components that is designed to support the development of web server applications) for the *Jutge.org*. This prototype had to feature:

1. A secure HTTP endpoint to serve the client the application (HTML and JavaScript files).
2. Instant bidirectional client-server communication to allow real-time interaction with the debugger.
3. A way to define components in the server so that the solution is robust and scalable.

11.1 Web framework

As previously stated, the server-side JavaScript runtime is *Node.js*. *Node.js* allows the creation of Web servers using JavaScript and a collection of modules that handle the core functionality [13].

There are a lot web frameworks available for *Node.js*. To name a few: *Express.js* [24], *Socket.io* [12], *Hapi.js* [11], *Mojito* [56], *Meteor* [39], *Derby* [10].

The decision to use an existing web framework was made to help develop the solution better and faster. It ensured that the application is well-

structured, maintainable and upgradable. Moreover, by re-using generic modules, the developer was able to focus on more relevant areas of the project.

11.2 *Node.js*, *Express.js* and *Socket.io*

Express.js is the solution's main web framework. This particular choice was motivated by how componentization is addressed with *Express*.

Express.js allows definition and use of components through *middleware* —functions that have access to the request and response objects, and the next middleware function in the application's request-response cycle. The basic idea is that each component is part of a pipeline that processes a request and generates a response, but individual components are not responsible for the entire response. Instead, a component modifies only what it needs to, and then delegates to the next piece in the pipeline. When the last piece of the pipeline finishes processing, the response is sent back to the client.

Express also supports the HTTPS protocol, which is one of the solution's security-enforcement requirements.

However, *Express* does not support bidirectional client-server interaction. To implement this feature, the *Socket.io* framework was integrated into the solution. *Socket.io* is a web framework that uses WebSockets, and enables real-time communication between the client and the server applications.

11.3 Design

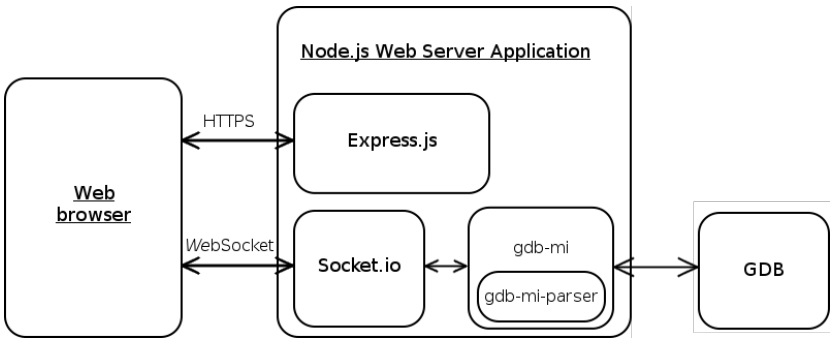

Figure 11.1: Express Prototype design

As figure 11.1 shows, *Express.js* will handle HTTPS requests, and

Socket.io will handle WebSocket connections.

In this prototype, *Express.js* has only one endpoint that serves a plain HTML file and some JavaScript files.

Socket.io allows the web browser to operate the debugger by sending events:

- **gdb-run:** starts debugging the given program.
- **gdb-stop:** kills the debugged program.
- **gdb-pause:** pauses execution of the program.
- **gdb-continue:** resumes execution of the program.
- **gdb-step-over:** advances the program's execution to the next line of code, ignoring function calls.
- **gdb-step-into:** advances the program's execution to the next line if it is not a function call. If it is, it stops at the first instruction of the called function.
- **gdb-step-out:** resumes the execution of the program until the current function is exited.
- **gdb-vars-stack-exprs:** requests the debugger to print the frames stack and the variables in the current context, and evaluate expressions.
- **gdb-insert-break:** adds a breakpoint at a given line.
- **gdb-delete-break:** removes a breakpoint from a line.
- **gdb-app-in:** writes text to the program's standard input channel.

11.4 Testing

To verify the *Express.js* HTTPS endpoint was working, a self-signed development SSL certificate had to be created using OpenSSL [27]. This test presented no complications —*Express* could successfully serve HTML and JavaScript files over a secure connection.

Socket.io also worked as expected —it provided real-time client-server communication, and each event triggered the appropriate action in the GDB components. However, this test unveiled a flaw in the design of the GDB component.

The problem stems from the fact that the debugged program uses the same input/output channels as GDB. As a consequence, it is not possible to separate the GDB output from the program's output. Likewise, there is no guarantee that GDB commands will work, because they may get sent to the program if it happens to be expecting some input. Therefore, the GDB component must be redesigned.

11.5 GDBServer

To separate GDB's IO channels from the debugged program's, the use of GDBServer was required. GDBServer is a control tool which allows to connect the program with a remote GDB. GDBServer also handles the program's input and output, thus solving the aforementioned problem.

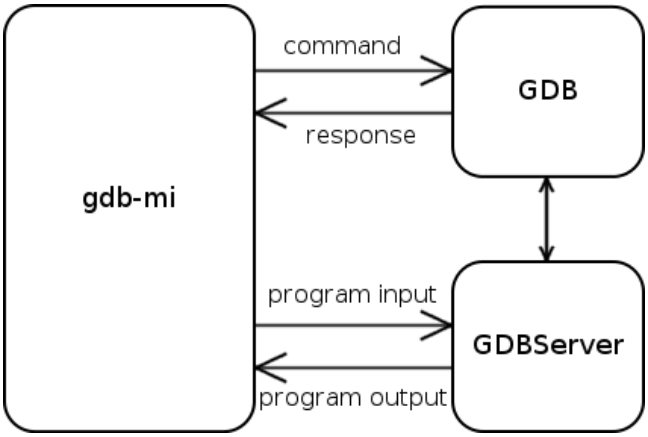

Figure 11.2: GDB component design

Figure 11.2 diagrams the new design for the GDB component, illustrating how *gdb-mi* creates an additional process for GDBServer that will handle the program's IO. The GDB process only has to take GDB commands, and return the corresponding responses. After this correction, the Express Prototype is complete.

11.6 Summary

This chapter has described the creation of the *ide.jutge* server application. At this stage, the application was far from complete —more functionality would be

added in later prototypes— but it was a solid and scalable starting point. All the targeted features had been implemented; *Express.js* provided the HTTPS endpoint and a scalable architecture, and *Socket.io* provided the real-time communication. Thanks to the testing phase, a short-coming in the *gdb-mi* component was repaired.

Chapter 12

Sandbox Prototype

This prototype aims to address the security concerns expressed in section 4.1.

The Sandbox Prototype provided a secure environment for *ide.jutge* to execute the C++ programs. Primary features of the Sandbox Prototype are:

1. Provide an isolate environment to execute programs.
2. Apply all necessary measures to secure the environment.

Once the secure environment is created, the server application must be accommodated to to support the prototype's design. The following secondary features have to be implemented:

1. Communication between the server app and the sandboxes must be fast and secure.
2. The server app must be able to save, compile, and debug the C++ programs in the sandboxes.

12.1 The sandbox

Execution of the C++ programs must be performed inside sandboxed environments, that is, virtual machines. Examples of available technologies include *Docker* [35], *OpenVZ* [9] and *Linux-VServer* [15].

12.2 Linux Containers

LXC [52] was chosen to provide the sandboxes. This is consistent with the *Jutge.org* platform, which uses the same technology to encapsulate problem corrections. Additionally, the project advisor has experience with LXC, and is able to provide valuable information on the diverse configuration options.

Each IDE instance will be assigned an LXC sandbox to compile and debug the student's C++ programs. The fastest and most practical way to supply sandboxes is to make clones of a parent sandbox —needless to say, each clone will have the same software and configuration as the parent.

To automate the parent sandbox's creation and cloning, two `bash` scripts are created. A third script is created to automatically destroy all clones.

vm-setup.sh: Creates the parent LXC container, and installs `g++`, `gdb` and `gdbserver`.

clone-box.sh: Creates a clone of the parent container.

destroy-children.sh: Forces destruction of all clone sandboxes.

12.3 Design

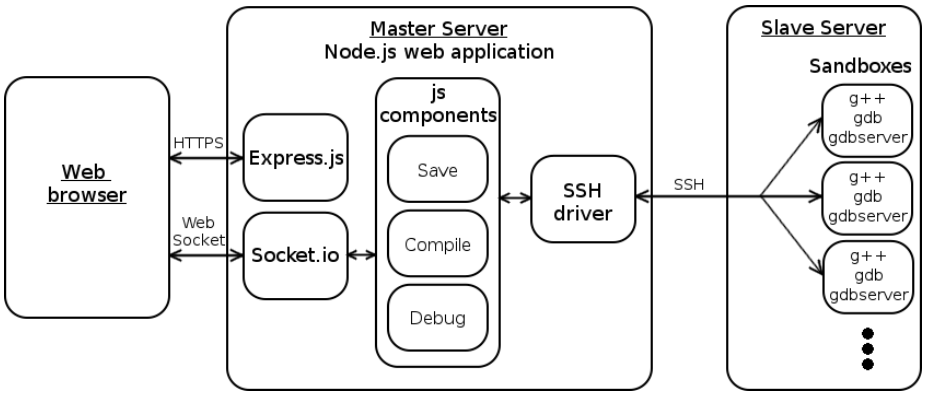

Figure 12.1: Sandbox Prototype design

12.3.1 Master and Slave servers

To further isolate execution of the programs, *Slave Servers* are used to host and serve the sandboxes. The *Master Server* will only contain the server application, considerably relieving its workload.

12.3.2 JavaScript components

Several server application modules support the Sandbox Prototype design.

SSH driver: This component provides SSH communication with the *Slave Servers*.

Save component: Connects to the *Slave Servers*, and writes C++ source code to a file in a sandbox.

Compile component: Connects to the *Slave Servers*, and orders the compilation of a C++ source file in a sandbox.

Debug component: Connects to the *Slave Server*, and starts debugging a C++ program in a sandbox. This module is just a refactoring of `gdb-mi`. Naturally, the connection stays open until the debug process is complete.

12.4 Security measures

12.4.1 SSH

Master and *Slave* servers must communicate through SSH. So, the pertinent SSH keys are generated using the `ssh-keygen` tool. A JavaScript driver is created to allow the server app to SSH the *Slave Servers*.

12.4.2 Low-privilege user

The server application will use a low-privilege user to operate each sandbox. That way, read/write permissions are restricted. The creation of this low-privilege user is added to the *clone-box.sh* script.

12.4.3 Memory-usage and network restriction

Memory usage is limited to 1GB per clone sandbox. Also, clones have no network access. These restrictions are set in the *clone-box.sh* script.

12.4.4 Limited number of processes

Malicious code could create a process that continually replicates itself, crashing the system due to resource starvation. Such attack are known as a *fork bombs* (figure 12.2 shows a C++ fork bomb). Therefore, the number of concurrent processes is limited to 200. This is accomplished running the `ulimit` command every time a SSH connection is established.

```
1 #include <unistd.h>
2 int main() {
3   while (1) fork();
4   return 0;
5 }
```

Figure 12.2: Fork bomb written in C++

12.5 Testing

Development is carried out using a single laptop computer, so this machine operates as *Master* and *Slave Servers*.

Individual components are tested to verify they work as expected. However, a client front-end is required to thoroughly validate the prototype.

12.6 Summary

This prototype features all the mechanisms to deal with potentially malicious programs. The *Slave Servers* that manage the sandboxes provide an isolate environment to execute the programs, and the `clone-box.sh` script applies the appropriate security measures.

Such measures are also enforced in the server application by creating an SSH driver which supplies fast, secure communication with the *Slave Servers*.

Also, a series of components enable the server app to trigger the three most basic actions in a sandbox: save, compile, and debug C++ programs.

Chapter 13

Angular Prototype

The Angular Prototype provided the client front-end for *ide.jutge* —a JavaScript application. This front-end is the very tool that students will use to write C++ and debug. For the Angular Prototype, only basic features were included:

1. A text editor to write C++ code.
2. A component to display compilation errors.
3. A terminal to interact with the debugged program.
4. A set of buttons to control program execution (start, stop, pause...).
5. A set of components to display the frames stack and the variables of the program when it is paused.

13.1 SPA

With the goal of providing a user experience similar to that of a desktop application, the *ide.jutge* front-end must be a SPA (Single Page Application). A SPA is a web application that fits on a single web page. *ide.jutge* will load a single HTML page and dynamically update that page as the student interacts with the IDE. WebSockets and HTML5, in combination with JavaScript, will be used to create a fluid and responsive application, without constantly reloading the page.

13.2 Web browser JavaScript framework

It is necessary to use an existing JavaScript framework in the IDE front-end. This is a practical decision; the motivation is the same as using *Express.js* in the server application, to help the developer work better and faster. Some of the available frameworks are *ExtJS* [41], *React* [36] and *Knockout* [49].

13.3 AngularJS

The chosen JavaScript framework is *AngularJS* [33]. To build an Angular application, components must be created to control the application logic and UI (User Interface) elements.

It is beneficial to use *Angular* because it manages all the components, acting as a pipeline that connects them. Since *Angular* acts as the mediator, the developer will not feel tempted to write shortcuts between components. Instead, the developer will be forced to write code in a very organized manner. *Angular* also has a large community, and help is easy to obtain.

To structure and write the *Angular* application so that it is easy to maintain, debug and scale, John Papa's *Angular* Style Guide is used [59]. This guide is endorsed by the very *AngularJS* team, and is often checked during development to produce a high-quality application.

13.4 CSS styling and UI elements

It is convenient to use CSS styling in *ide.jutge* to make the UI look good. Also, libraries to create custom elements such as the terminal and the editable datagrid are required. The list of additional packages for this prototype:

jQuery [30]: Simplifies HTML document manipulation.

jQuery UI [29]: Provides some basic UI components —for instance, tabs.

Ace Editor [1]: An embeddable code editor that supports C++ syntax highlighting.

Bootstrap [60]: It was chosen as this prototype's main styling framework to keep consistency with *Jutge.org*, which also uses *Bootstrap*.

Xterm.js [63]: A terminal emulator written in JavaScript.

EditableGrid [70]: Turns tables into advanced editable components.

UI Layout [17]: A complete page layout manager.

PNotify [61]: A notification system. Events such as a successful compilation get displayed on-screen.

Figure 13.1 shows a screenshot of the Angular Prototype's front-end application.

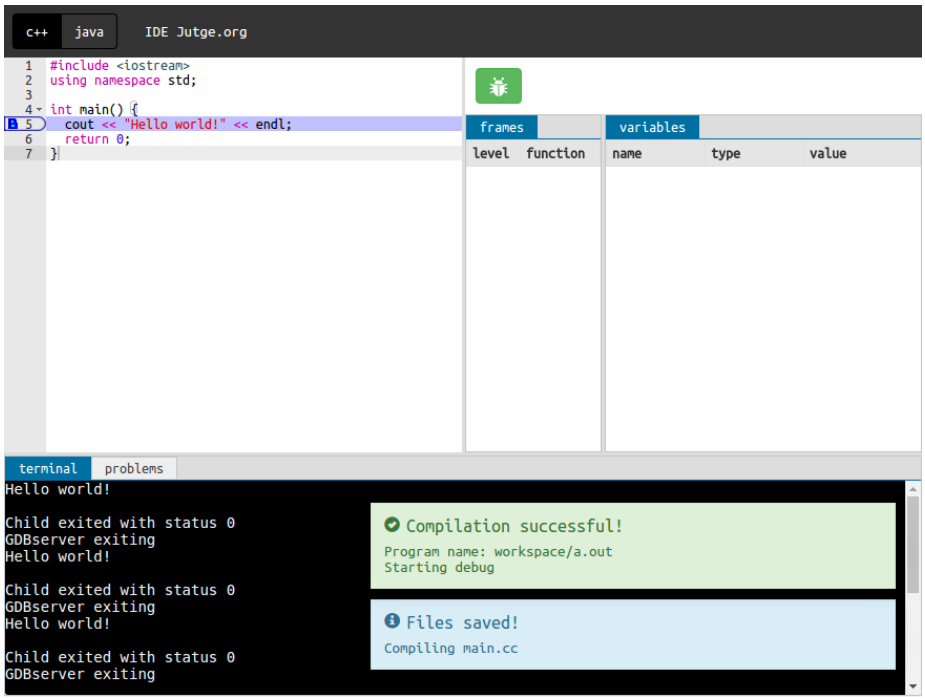

Figure 13.1: Angular Prototype IDE

13.5 Bower

In order to handle and control all the web frameworks and libraries for *ide.jutge*, the *Bower* [65] package manager is used. This tool works similar to *npm* — packages can be installed with a single command.

13.6 Design

The AngularJS Prototype has the same general design as the previous Sandbox Prototype; the only addition is a client front-end application.

13.6.1 AngularJS architecture

In the *AngularJS* parlance, a *module* is a container for the different components of an application —ie. controllers, services, views. Thanks to the *Dependency Injection* design principle [31], these components can have access to other components. *AngularJS* also features a powerful event notification system.

The client application consists of one module named *workbench*. Figure 13.2 shows all *workbench* components.

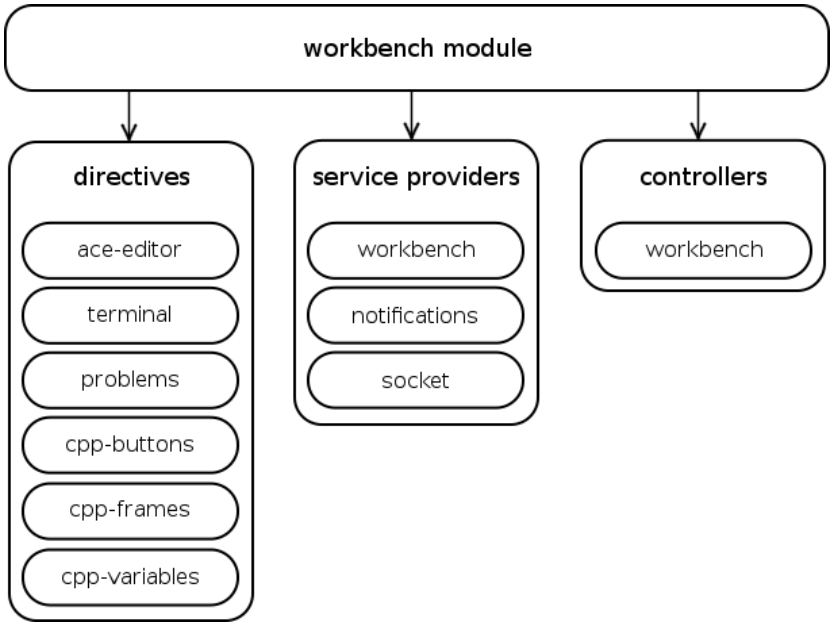

Figure 13.2: AngularJS workbench module

13.6.2 Controllers

The workbench controller is responsible for the interaction with the debugger. Using the WebSocket, it is able to trigger compilation of the C++ code, and

invoke all debug-related commands (start, stop, pause, insert breakpoint, step over, etc.). It also handles whatever messages it receives from the server, for instance, termination of the program, or pause at a breakpoint.

Furthermore, the workbench controller keeps state information about the compilation and debug functions. It also holds a list of the breakpoints, program variables and stack frames.

13.6.3 Directives

Directives are used to extend HTML with custom attributes and elements.

ace-editor: This directive provides an *Ace* editor to write C++ code. Additionally, it allows breakpoints to be added or removed, by clicking on the editor's gutter (the leftmost part of the line).

When a breakpoint is hit, the line at which execution stopped is highlighted. Also, if a program has compilation errors, an error icon and message appears in the gutter of each faulty line.

terminal: This directive uses *xterm.js* to provide a terminal. The terminal displays output from the C++ program. It can also write to said program's input channel.

problems: Provides a table which will display compilation errors, if any are found. Clicking on an error row will highlight the corresponding line in the Ace editor.

cpp-buttons: Provides a set of buttons that trigger most of the debug commands. They are:

- *play:* Compile the C++ code, and start debug if no error were found. If debug has already started, resume program execution.
- *pause:* Pause execution of a running program.
- *stop:* Kill the program.
- *step-over:* Advance to the next line of code.
- *step-into:* Advance program's execution to the next line if it is not a function call. Otherwise, stop at the first instruction of the called function.
- *step-out:* Resume execution of the program until the current function is exited.

cpp-frames: Provides a table to show the program's call stack —that is, the chain of function calls that have been invoked. This table is filled when the program pauses (usually because of a breakpoint being hit), and cleared when it resumes.

cpp-variables: Provides a datagrid to show the program's variables in the current execution context. This datagrid is filled when the program pauses, and cleared when it resumes. During a pause, program variables of basic types (`int`, `long`, `float`...) may be altered by editing the corresponding cell in the datagrid.

13.6.4 Service providers

These components are meant to provide reusable business logic independent of views.

workbench: This service is used to pass data from the controller to the directives.

notifications: This service is used to display notifications.

socket: This service allows interaction with the WebSocket.

13.7 Testing

To verify that the system works as expected, extensive testing of every feature is performed. Examples of the conducted tests include the following assertions:

- Compilation of faulty code fails, and errors are displayed in the corresponding table.
- The terminal effectively reads and writes to the debugged program's input/output channel.
- Execution pauses at the inserted breakpoints, and removed breakpoints do not pause execution.
- The control buttons send the corresponding commands to the debugger.
- Variables and frames are displayed when the program pauses.
- Basic-type variable values are assignable.

13.8 Summary

The *AngularJS* framework, in combination with the JavaScript packages listed in section 13.4, made it possible for the Angular Prototype to implement the

client application, and supply all the features described at the beginning of the chapter.

Chapter 14

Plugins Prototype

The architecture of the system must be accommodated to allow the addition of new functionality through a plugins system. The Plugins Prototype must feature an infrastructure to support the activation and operation of a set of plugins working together.

This chapter describes how the Plugin Prototype integrated such changes in design without breaking the existing functionality. It also provides detailed instructions on how to add new plugins with an example: adding the C++ plugin.

In general terms, a plugin in *ide.jutge* is a component that provides a certain type of service within the context of the *ide.jutge workbench*.

14.1 The workbench

The term *workbench* refers to the *ide.jutge* development environment, that is, the IDE. The *workbench* aims to achieve seamless tool integration by providing a common paradigm for the creation and management of resources (see section 14.5).

14.2 Web application design

Most of the changes in design affect the client front-end application. Figure 14.1 diagrams the Plugins Prototype web app's design.

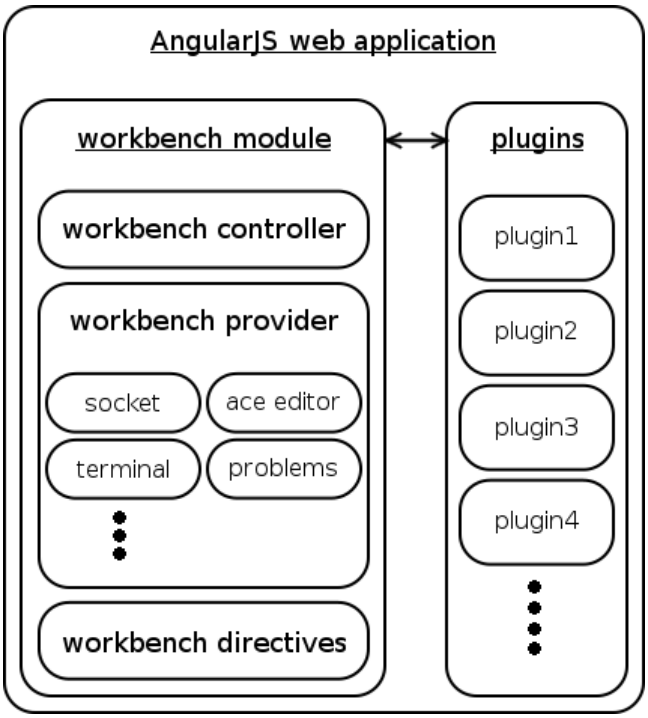

Figure 14.1: Web application design

14.2.1 The workbench module

This module manages the plugins, and also the elemental components of the IDE (socket, *Ace* editor, terminal, etc.). It provides access to said components, and executes the basic initialization procedures to set up the environment.

As in the previous prototype, the workbench module has three types of components: controllers, service providers, and directives.

14.2.2 The workbench controller

The *workbench controller*'s purpose:

- Send the server application a request for a sandbox on application startup.
- Broadcast an event when the sandbox is ready, to notify other components.

14.2.3 The workbench provider

The *workbench service provider* is used to manage the plugins, share data amongst components, and expose some basic functionality. This component allows:

Registration of plugins: New plugins must register themselves in the *workbench*. This mechanism allows the *workbench* infrastructure to automatically initialize and integrate plugins in the IDE.

Activation of plugins: In the *workbench*, only one plugin is active at a time. In general, the UI components of a plugin shall be visible when it is active, and hidden otherwise.

Saving the contents of the editor in a file inside the sandbox: This function becomes available when the sandbox is actually ready.

Accessing the basic components of the workbench: The socket, *Ace* editor, terminal, etc., are available as properties of the workbench service. This allows plugins to interact with said components (for instance, the C++ plugin sends compilation errors to the problems component).

14.2.4 The workbench directives

A smaller set of directives is required this time. Each directive will add the corresponding component into the *workbench service*.

Ace editor: Directive for the code editor.

Terminal: Directive for the terminal emulator.

Problems: Directive for the problems table.

Plugin loader: Directive to initialize plugins. This is the last *workbench* directive *AngularJS* processes; therefore, all *workbench* UI components are ready when plugins get initialized.

14.2.5 Plugins

Since the web app is built on *AngularJS*, the plugins system can take advantage of the *AngularJS* infrastructure. Plugins act as dynamically-loaded *Angular* application.

Before going any further, it is important to understand what *scope* is in *Angular*. The *scope* is the JavaScript object that glues the HTML view to the controller. *Scopes* are arranged hierarchically —every application has a single root *scope*, and all other *scopes* are descendants of that root *scope*.

Essentially, plugins must have a *scope* if they want to use *AngularJS* directives. Hence, on plugin initialization, a child *scope* of the *workbench scope* is created and assigned to each plugin.

The building blocks of a plugin are a module with a controller, and possibly some HTML templates and other components.

Module

A module must be declared to group all the plugin's working pieces: controller, directives, etc. It is necessary for plugin modules to list the *workbench* module as a dependency; this allows plugins to use the *workbench* service provider to interact with the IDE.

This module is also responsible for registering the plugin in the *workbench*. This is easily accomplished by performing registration in a *run block* —in *AngularJS*, the module's main method.

Controller

In *AngularJS*, a *controller* is simply an object that augments the *scope*. A plugin controller, however, must also implement a series of functions, which will be invoked by the *workbench* infrastructure at certain points in the plugin's life-cycle:

onLoad: Invoked after the controller has been instanced, but before any plugin HTML templates are loaded.

postLink: Invoked after all plugin HTML templates have been processed.

activate: Invoked when a plugin becomes the active plugin in the *workbench*.

deactivate: Invoked when a plugin ceases to be the active plugin in the *workbench*.

At plugin initialization time, the plugin controller is instanced. A child *scope* of the *workbench scope* is created and made available to the controller's constructor function.

HTML templates

Plugins typically include UI components to exhibit their particular functionality and allow interaction (for instance, the C++ plugin needs to add a set of buttons to control the program's execution). Within the web application's plugin system, a plugin can define several HTML templates along with their location in the DOM. The workbench infrastructure will automatically download and insert each template in the IDE.

The HTML templates can contain *AngularJS* directives. In order for directives to work, the *workbench* infrastructure instructs the *AngularJS* framework to:

1. *Compile* the HTML template —in the *AngularJS jargon*, compilation is a process of walking the DOM tree and matching DOM elements to directives [32].
2. *Link* the template to the plugin's scope -the plugin's scope becomes the directives' working model.

Hence, all HTML templates for a plugin share a common *scope* —even if they are placed in different parts of the DOM. This contrasts with typical *AngularJS* applications, where *scopes* are arranged in a hierarchical structure that mimics the DOM tree.

Other components

A plugin may contain many other components besides the controller and the HTML templates. In practice, plugins will usually require custom directives and service providers. Figure 14.2 diagrams the structure of a plugin.

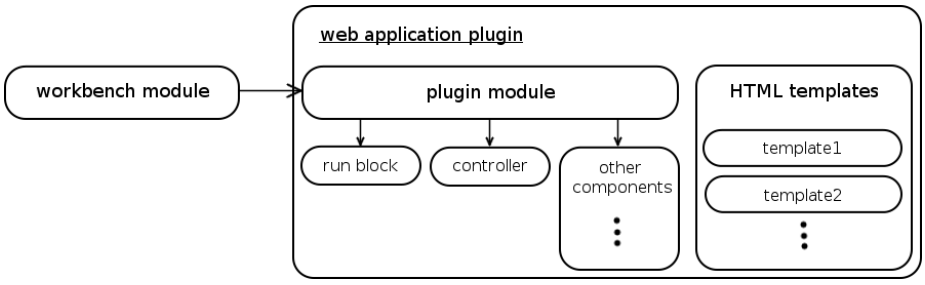

Figure 14.2: Web application plugin structure

14.3 Server application design

The plugins system is much less complicated in the server application. As in the client application, there is a base *workbench* infrastructure that manages the plugins.

A plugin in the server application is a component that links an IDE to a sandbox, and exposes its functionality to the IDE through registration of event handlers on the socket. Figure 14.3 diagrams the design of the plugin system within the server application.

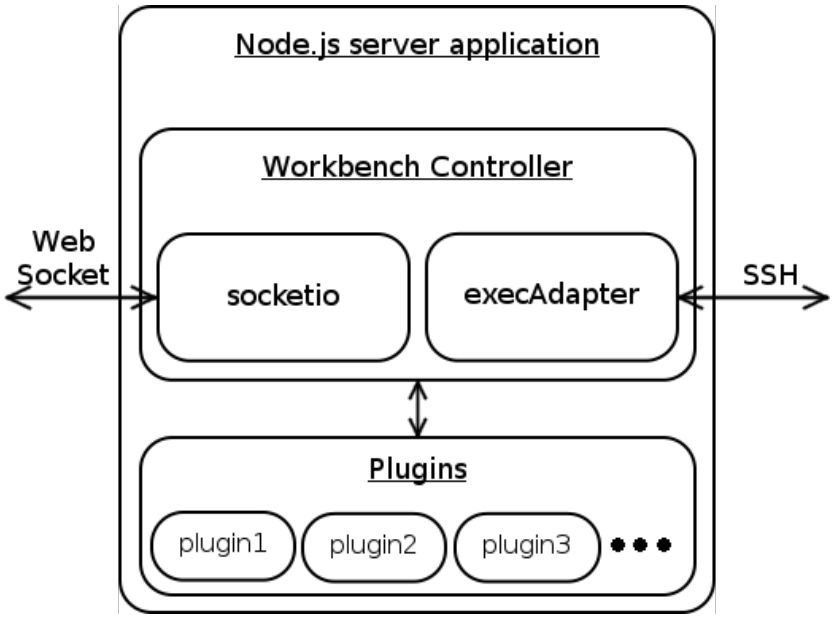

Figure 14.3: Server application plugins system

14.3.1 The Workbench Controller

This is the *Node.js* module that manages the plugins. The *Workbench Controller* detects establishment of socket connections with *ide.jutge* IDEs, and performs the following steps:

1. Allow the IDE to request only one sandbox.
2. Upon sandbox availability, expose a function to allow the IDE to save a file in the sandbox.

3. Load the plugins —this will make all the plugin's features available to the IDE.

14.3.2 Plugins

A plugin needs to communicate in realtime with the IDE, and it will do so via WebSocket. Realtime communication with the sandbox is also required, and it is provided by the use of SSH. The *Workbench Controller* passes these communication mechanisms as constructor parameters to every plugin:

```
function plugin(socket, execAdapter)
```

socket

The *Socket.io* socket instance is passed as the first plugin constructor parameter.

execAdapter

The second plugin constructor parameter is an object that enables SSH communication with the sandbox using the following methods:

spawn: Spawns a new process in the sandbox using a given command.

exec: Executes a given command in the sandbox.

14.4 Sandboxes

It is likely that the features of new plugins demand that additional software be installed in the sandboxes -ie. a Java plugin will require certain Java tools to be available.

The addition of new software in the sandboxes is simple, because only the parent sandbox must be updated —subsequent clones will automatically have the new software at their disposal. It is also convenient to update the `vm-setup.sh` script described in section 12.2, in case the whole parent sandbox needs to be regenerated.

14.5 Creation of the C++ plugin

This section describes the pattern to follow for creating a new *ide.jutge* plugin. An account of how the C++ plugin was created is given to illustrate the process.

14.5.1 Web application plugin

1. Module

Create a module for the plugin (figure 14.4), and add it to the *ide.jutge* application module (figure 14.5).

```
1 (function() {
2   'use␣strict';
3   angular.module('ide.jutge.plugins.cpp', [
4     'ide.jutge.workbench'
5   ]);
6 })();
```

Figure 14.4: cpp.module.js

```
1 (function() {
2   'use␣strict';
3   angular.module('ide.jutge', [
4     'ide.jutge.workbench',
5     'ide.jutge.plugins.cpp'
6   ]);
7 })();
```

Figure 14.5: app.module.js

2. Run block

The plugin needs register itself in the *workbench* (figure 14.6).

3. Controller

A plugin controller is needed to provide the plugin life-cycle hooks, and augment the scope according to the features of the plugin. Figure 14.7 shows a

```
1  (function() {
2    'use strict';
3    angular
4      .module('ide.jutge.plugins.cpp')
5      .run(run);
6    run.$inject = ['workbench'];
7    function run(workbench) {
8      workbench.registerPlugin({
9        id: 'cpp',
10       name: 'C++',
11       controller: 'CppController as cpp',
12       components: {
13         'jtg-menu-panel': '/plugins/cpp/components/
             menu-bar.html',
14         'jtg-plugin-panel': '/plugins/cpp/components/
             panel.html'
15       }
16     });
17   }
18 })();
```

Figure 14.6: cpp.config.js

basic template of a plugin controller. This template was completed with the C++-related code from the previous prototype's *workbench controller*, and only minimal adjustments had to be made for the component to work.

4. HTML templates

This step involves the creation of the HTML templates specified when registering the plugin (the `components` property in 14.6). The HTML code was taken from the previous model's index page.

5. Other components

The particulars of creating the rest of the plugin components are plugin-specific, as each one will provide a different set of features. In the case of the C++ plugin, only directives had to be created —their code was adapted from the previous prototype's directives.

```
1  (function() {
2    'use strict';
3    angular
4      .module('ide.jutge.plugins.cpp')
5      .controller('CppController', cppController);
6    cppController.$inject = ['$scope', 'workbench'];
7
8    function cppController($scope, workbench) {
9      var me = this;
10
11     // life-cycle hooks
12     me.onLoad = onLoad;
13     me.postLink = postLink;
14     me.activate = activate;
15     me.deactivate = deactivate;
16
17     function onLoad() {/*...*/}
18     function postLink() {/*...*/}
19     function activate() {/*...*/}
20     function deactivate() {/*...*/}
21   }
22 })();
```

Figure 14.7: cpp.controller.js

6. Index page

The final step is to include all the new JavaScript files in the index HTML page.

14.5.2 Server application plugin

1. Plugin component

A plugin *Node.js* module must be created. Figure 14.8 show the basic template used to create the C++ server plugin; this template was completed using the *compile* and *debug* JavaScript components from the previous prototype.

This plugin module must be registered in a file called `plugins.js` in order for the workbench infrastructure to pick it up (see figure 14.9).

```
1 function cppPlugin(socket, execAdapter) {
2   /*...*/
3 }
4 module.exports = cppPlugin;
```

Figure 14.8: cpp-plugin.js

```
1 function plugins() {
2   return [
3     require('./cpp/cpp-plugin.js')
4   ];
5 }
6 module.exports = plugins;
```

Figure 14.9: plugins.js

2. HTTPS endpoints

Plugins may want to use ajax (HTTPS) instead of the socket to interact with the server application. This is convenient for features that do not require a backing sandbox (for example, user authentication). The best way to accomplish this is to add new HTTPS endpoints using the traditional *Express.js* routing mechanism [25].

The C++ plugin did not require additional HTTPS endpoints to be created.

14.5.3 Sandboxes

No changes need to be made in the sandboxes, as the C++ tools have already been installed in previous prorotypes. Otherwise, the general procedure to follow is described in section 14.4.

14.6 Testing

To validate the prototype, an additional test plugin was created. This test plugin contained one HTML template, and added no extra functionality. Both plugins were automatically integrated into the IDE, and presented no operation conflicts.

The same set of tests from the Angular Prototype were also conducted to ensure the C++ plugin maintained functionality.

14.7 Summary

A detailed description of the plugin infrastructure for each of the *ide.jutge* tiers has been provided. Also, an example of how a fully functional plugin (the C++ plugin) is created has been given. The testing phase has verified that the plugin infrastructure supports the activation and operation of a series of plugins working together.

Chapter 15

EasyUI Prototype

Around this time, the advisor of this project offered the possibility to use a web framework for UI components —*jQuery EasyUI* [72].

The EasyUI Prototype would furnish *ide.jutge* with an appealing desktop-application appearance, and a collection of practical UI components. Even though it would involve rewriting a large part of the IDE user interface, using *EasyUI* would yield major benefits:

- One powerful web UI framework instead of several disparate UI components —*jQueryUI*, *Bootstrap*, *EditableGrid*, *UI Layout*, *PNotify*.
- Whereas the previous prototypes required abundant CSS to homogenize the different UI components' *look'n'feel*, minimal CSS styling is required with *EasyUI* as it provides several themes, all of them with very elegant finishes.

15.1 EasyUI

EasyUI is a compendium of highly-sophisticated UI JavaScript components for HTML5. The *EasyUI* website provides comprehensive documentation and examples on how to operate the framework. However, external online resources are scarce, and the complexity of the framework makes advanced use of *EasyUI* a complicated task, which requires extensive analysis of the *EasyUI* API.

15.1.1 Additional features

The availability of a wide array of customizable UI components motivated the addition of the following features to the EasyUI Prototype:

1. Open a local text file in the editor.
2. Save the text in the editor to a local file —ie. download the code.
3. Change the editor's font size.
4. Change the IDE and the editor's theme.
5. Provide operating instructions within the IDE.

15.1.2 Changes in the client application

To incorporate *EasyUI* in the client app, all custom *AngularJS* directives had to be rewritten.

The index HTML page, and the C++ plugin's HTML templates also had to be modified. In particular, the index HTML page would now contain a menu bar to accommodate the additional features, and a status bar which would replace the former notifications system.

15.2 Menu bar

The menu bar, at the top of the page, provides access all additional features by means of the following menus:

15.2.1 File menu

The *File* menu accomplishes additional features 1 and 2 by displaying functions:

New: Clears the code editor.

Open: Opens a local file in the editor.

Save: Downloads the contents of the text editor into a local file (in the default *Downloads* folder).

15.2.2 Edit menu

A few of the traditional file-edition commands:

Undo: Undoes the last change in the editor.

Redo: Redoes the last undone change in the editor.

Select all: Selects the entire text in the editor.

The classic *Cut*, *Copy*, and *Paste* edit commands have not been supported yet. Accessing the system clipboard via JavaScript is considered an unsafe operation: scripts could erase and replace the contents of the clipboard (data loss issue), and also read the contents of the clipboard (security and privacy issue). Implementing the *Cut*, *Copy*, and *Paste* commands requires browser-specific hacks, so their addition is left for a future update of *ide.jutge*. Nevertheless, said commands are still available via the traditional keyboard shortcuts.

15.2.3 View menu

The *View* menu accomplishes additional features 3 and 4:

Font size: Presents a sub-menu offering 5 different font sizes for the code editor.

GUI themes: The general IDE's theme, or *look'n'feel*, can be chosen from a sub-menu. 15 different themes are available, thanks to *EasyUI*.

Editor themes: The code editor's theme can be picked from a sub-menu —*Ace* editor provides 34 different themes.

15.2.4 Help menu

The *Help* menu provides the last additional feature:

Help: Displays a link to the documentation for the IDE.

About: Displays the typical *about* information: application icon, application name, version, and copyright notice.

15.3 Documentation HTTPS endpoint

Since the *Help* menu provides a link to the documentation for the IDE, an HTTPS endpoint must be exposed in the server application:

GET /doc: Returns a document containing instructions on using the IDE —the *ide.jutge* end-user guide.

This endpoint will be created using the regular *Express.js* routing system, as it will just serve a static HTML file.

15.4 Status bar

As already stated, this component, located at the bottom of the page, replaces the alert-like notification service the previous prototype was using. The status bar improves usability by removing the potentially obtrusive notification boxes the former prototype displayed (see figure 13.1).

15.5 Adapting the IDE

Some additions and modifications need to be made to fully integrate the updates into the client application.

15.5.1 Custom AngularJS directives

Several custom *AngularJS* directives are created for the menu bar, the menus and the status bar. Incidentally, this again shows why a framework like *Angular* is beneficial:

- **Smaller, more readable HTML files:** The component's HTML and JavaScript code are moved into the directive file.
- **Each directive handles a common set of functions:** This honors the separation of concerns (SoC) design principle, simplifying development and maintenance.

15.5.2 The workbench

One of the key functions of the *ide.jutge workbench* is to provide access to the IDE's core components. So, the menu bar, the status bar, and each one of the menus are made available, along with the rest of the UI components, via the *workbench service*.

Also, since the status bar has replaced the former notifications system, the *workbench service* must remove notification functions from its API.

15.6 Client C++ plugin

Taking advantage of the *EasyUI* framework, two additional features were added to the C++ plugin:

1. C++ code templates that portray basic C++ features and programming techniques.
2. Evaluation of C++ expressions while debugging.

15.6.1 Selected templates

The following C++ code templates were produced for the first feature:

Hello world! - The all-too-familiar "hello world" program.

Fill in a vector - About C++ STL (Standard Template Library) `vector`, and standard input/output.

Iterative GCD - Calculate the greatest common denominator of two numbers, iteratively.

Correct parenthesization? - Answer the question using an STL `stack`.

Fibonacci in a vector - Calculate Fibonacci numbers using an iterative technique.

Recursive factorial - Calculate a factorial using a recursive technique.

15.6.2 Templates menu

The most appropriate way to integrate the C++ templates feature in the IDE is through a sub-menu containing all the templates (see figure 15.1). Selecting a template will simply set the corresponding C++ code in the editor. As usual, a custom *AngularJS* directive needs to be created.

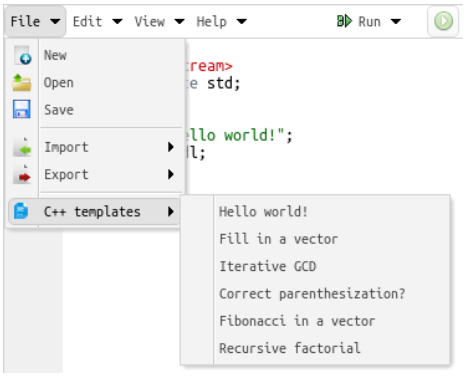

Figure 15.1: C++ templates sub-menu

The purpose of the *workbench service* exposing the UI components is to give plugins the ability to customize parts of the IDE. This fact is utilized by the C++ templates directive, which inserts the new option in the *File* menu (the menus are *EasyUI* components, and provide methods to manage menu options).

15.6.3 C++ expressions

ide.jutge has had the ability to evaluate expressions since the Debug Prototype (see section 10.3.2), and this function has been available to the client since the Express Prototype (see section 11.3).

The EasyUI Prototype will incorporate this function into the IDE by providing a datagrid where C++ expressions can be written. Much like the variables datagrid, expressions in the datagrid are evaluated and their results shown when the program pauses. When the program resumes, the results in the grid are cleared. Figure 15.2 shows a snapshot of the expressions tab at work.

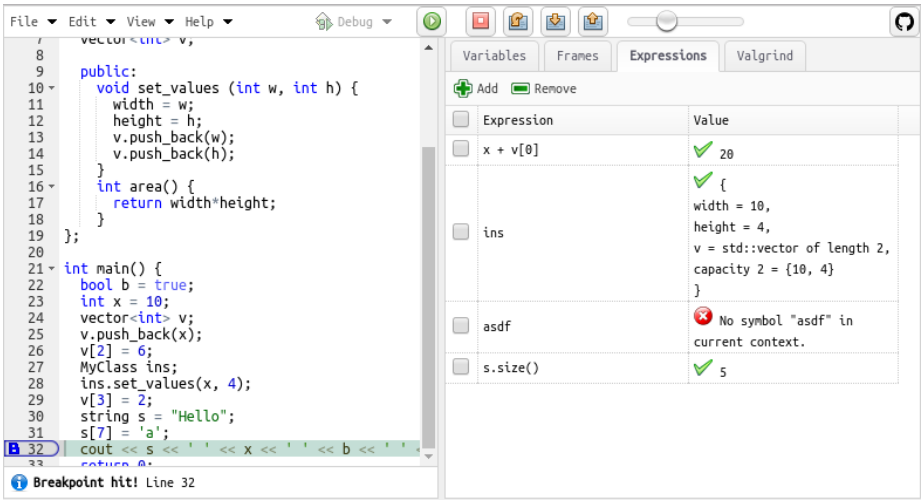

Figure 15.2: C++ Expressions tab

15.7 Testing

Validation of the prototype was performed in the usual fashion in the RAD methodology; components were continually tested during their implementation. In particular, it was ensured that:

- Refurnished UI components provided at least the same set of features as their equivalent UI components in the previous prototype.
- New UI components worked as expected and did not break any existing functionality.
- The documentation HTTPS endpoint served the end-user guide HTML page.

15.8 Summary

In the EasyUI Prototype, the *ide.jutge* IDE has gained a beautifully-designed user interface. The addition of *EasyUI* has also motivated the implementation of several supplementary features:

1. Menu bar; File, Edit, View, and Help menus.
2. IDE user-guide.

3. Status bar.
4. C++ templates, and a grid to evaluate expressions for the client C++ plugin.

Chapter 16

ide.jutge Prototype

Before proceeding with the final prototype, a quick review of the current state of the solution was made. With the previous EasyUI Prototype, *ide.jutge*:

1. Provided a web IDE that allows edition, compilation, execution, and debug of C++ programs.
2. Supplied all the security measures to deal with potentially malicious user programs.
3. Featured an extensible plugins system. By making plugins, further functionality could be incorporated into the IDE without having to modify the core components of the solution.

These features achieved the project's main objectives and the first two secondary objectives (*create an extensible plugin system for customizing the environment* and *make the infrastructure scalable, stable and secure*). So, the ide.jutge Prototype would need to accomplish the three remaining secondary objectives:

- Create a step-by-step debug mode.
- Detect memory errors in the student's programs.
- Import and export code from *GitHub.com*.

16.1 C++ execution modes

The C++ plugin needs further enhancements to help complete the secondary objectives —slow-motion debug, and memory error detection. In an effort to integrate all this functionality into the IDE in as natural a way as possible, the following execution modes are defined:

Run: Execute the program without debugging.

Debug: Debug the program (execution will pause at breakpoints).

Slow motion: Debug mode where execution advances one line at a time every few seconds.

Valgrind: Run mode where a report of memory-related errors is provided once execution completes.

The C++ plugin will provide a drop-down button so students can choose in which mode their program will run.

Figure 16.1 diagrams the design of the different parts of the C++ plugin within the *ide.jutge* architecture.

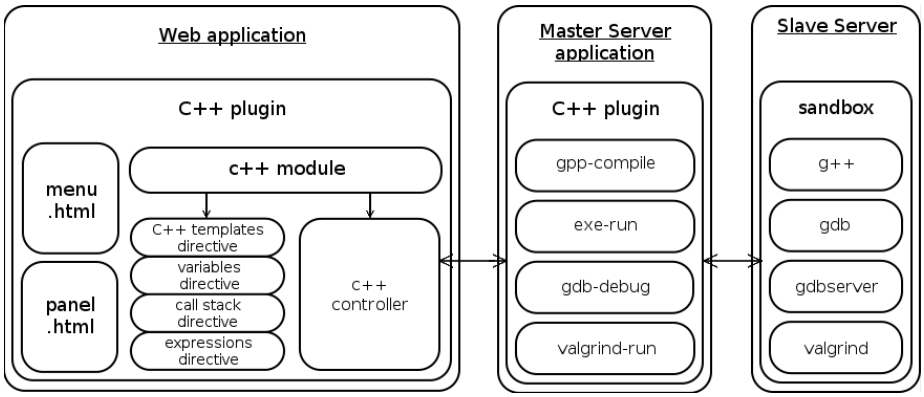

Figure 16.1: C++ plugin design

16.2 Slow motion

The *Slow-motion* debug mode is fairly simple to implement as it requires no server-side changes. All the IDE does in *Slow-motion* is start debugging the program, and issue a *step into* command at regular intervals.

The interval between steps can be changed in the IDE via a slider control that the C++ plugin provides. This slider is located to the right of the execution buttons, and allows the interval to range between one and ten seconds.

Slow-motion execution can also be paused, resumed and stopped using the appropriate execution buttons.

16.3 Run

The *Run* mode just executes the program; none of the debug or valgrind features are available —that means breakpoints have no effect, and no check is made for memory errors. This execution mode is added for the sake of completeness —all IDEs provide a plain execution mode.

The *Run* mode requires changes be made on the client and server applications:

16.3.1 Client application

When in *Run* mode, the C++ plugin needs to adjust the execution buttons' behaviour according to the following guidelines:

- Execution cannot be paused or resumed, only stopped.
- *Step* commands are not available either.

16.3.2 Server application

A new component to execute the program must be tied into the the server-side C++ plugin. Its competences:

1. Send the run (or kill) command to the sandbox when the IDE makes the pertinent request.
2. Manage the IDE-program input/output exchange.
3. Notify the IDE when program finishes execution.

16.4 Valgrind

This mode is named after the popular utility *Valgrind* [16]. In a nutshell, this tool executes the program, and produces a report in XML format containing all memory-related errors found (if any). This execution mode requires changes on both client and server tiers.

16.4.1 Client application

The C++ plugin again needs to adjust the execution buttons' behaviour following same guidelines as in the Run mode. Additionally, the C++ plugin UI needs to provide a table to display the errors reported by *Valgrind*. This table will be cleared when the program runs again.

16.4.2 Server application

A new component is created and woven into the server-side C++ plugin. This component works in many ways like the one for the *Run* mode; the valgrind component is required to:

1. Send the run (or kill) command to the sandbox when the IDE makes the pertinent request.
2. Manage the IDE-program input/output exchange.
3. After execution finishes, parse the resulting XML report and send it to the IDE.

16.4.3 Sandbox

The *Valgrind* utility must be installed in the parent sandbox, and the `vm-setup.sh` script needs to include the commands to install *Valgrind*.

16.5 Gists

The ability to read and write *Gists* accomplishes the last secondary objective, and is the icing on the ide.jutge Prototype.

Of course, students will need a *GitHb.com* account to use this service. *ide.jutge* will request permission from students to access their *gists*. Once students accept, *ide.jutge* will be able to read and write *gists* on behalf of the students.

OAuth2 is the protocol that will allow *ide.jutge* to request access to *gists* in a student's *GitHub* account, without getting their password [38], and using an access token instead. *GitHub.com* requires *ide.jutge* to be registered as an OAuth application.

The server application will use the *Gist* API [38] to interact with *GitHub* and perform the read and write actions.

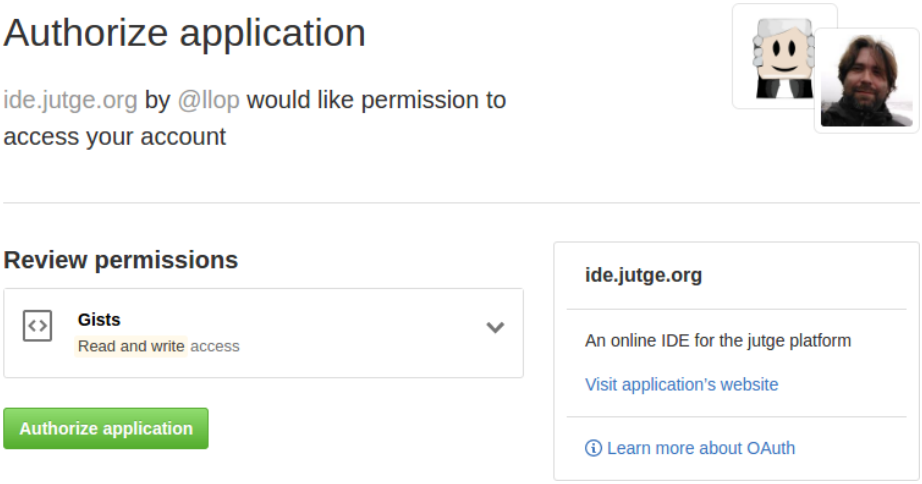

Figure 16.2: GutHub OAuth authorization page

16.5.1 Client application

The following components come into play on the client front-end:

1. In the File menu, two additional options: *Import gist* and *Export gist*. If students have not yet authorized *ide.jutge*, they will be asked to do so in a pop-up window that will open the *GitHub* OAuth access page (similar to figure 16.2).
2. A list of the student's *gists*; it allows students to select the *gist* that they want to import into the editor.
3. A form for students to save *gists*, which allows them to enter the *gist file name*, the *description*, and whether the *gist* is created as *public* or not.

4. A button for students to log out of *GitHub* within *ide.jutge*.

16.5.2 Server application

This feature does not require a sandbox, so only the server application needs to be modified. The role of the server application is to handle communication with *GitHub*, and pass results down to the IDE. The following HTTPS endpoints are exposed:

GET /auth: Redirects to the *GitHub* authorization page.

GET /callback: The URL to which *GitHub* will redirect after authorization.

POST /gists: Lists the student's *gists*.

POST /gist: Fetches a particular *gist*.

POST /update-gist: Updates a *gist*.

POST /create-gist: Creates a new *gist*.

16.6 Testing

Tests for the *Run* and *Slow-motion* modes basically verified correct interaction with the IDE's execution buttons and terminal with C++ from previous tests. However, the *Valgrind* mode required creating C++ programs with deliberate memory-related errors (see an example in figure 16.3), and adding them to the tests.

```
1 #include <vector>
2 using namespace std;
3 int main() {
4   vector<int> v(3);
5   v[4] = 0;  // invalid write
6   return 0;
7 }
```

Figure 16.3: C++ program featuring a memory error

Verification of the import/export gist features required registering the ide.jutge Prototype application to use the *GitHub* API; this was done at the *GitHub.com* website. The next tested feature was the *GitHub* OAuth2 flow,

which would authorize *ide.jutge* to access students' *gists*. Once that was done, the remaining features were validated —logging in and out; and reading, writing and listing user *gists*. All tests were conducted using the project author's *GitHub* account.

16.7 Summary

This chapter described the implementation and testing of the last features of the *ide.jutge*. A figure 16.4 displays a snapshot of the IDE. The ide.jutge Prototype provided the solution with:

1. Four execution modes —*Run*, *Debug*, *Slow-motion*, and *Valgrind*.
2. The ability to import/export code from *GitHub.com gists*.

The project's primary and secondary objectives were accomplished — therefore, *ide.jutge* was complete.

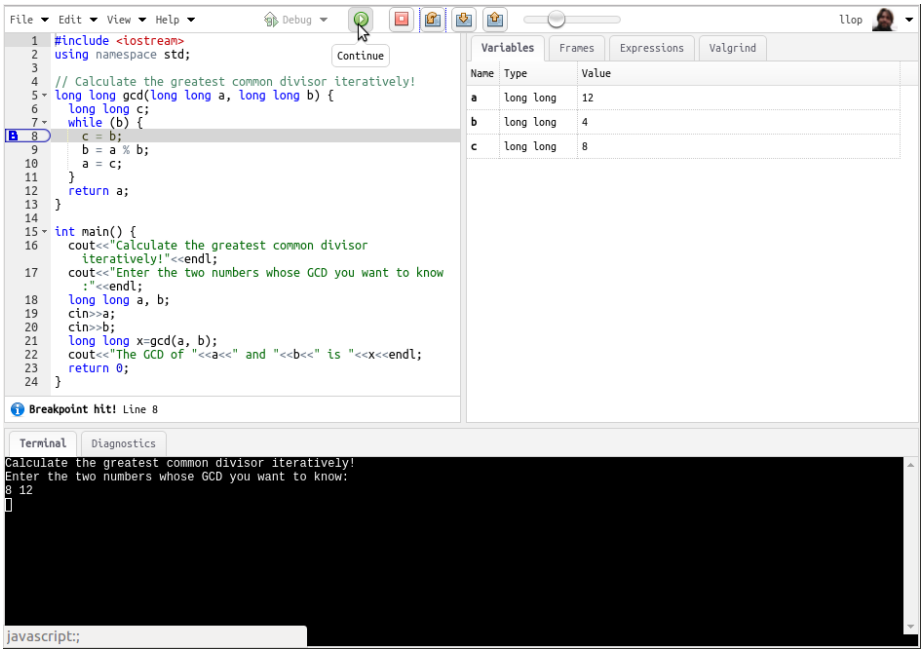

Figure 16.4: *ide.jutge* Web IDE

Part IV

Evaluation

Chapter 17

Usability

An emphasis must be placed on usability in order to make *ide.jutge* a user-friendly tool. Usage of the IDE aims to be so intuitive that no instruction manual is required; simple features must be self-explanatory, and documentation for more complex features must be promptly available within the IDE (in the line of [50]).

As figure 18.2 shows, the IDE becomes responsive in less than three seconds. According to [58] and [48], ten seconds is the maximum response time for any web application to become responsive before users lose patience and become angry. For longer delays, users will want to perform other tasks while waiting for the computer to finish. Therefore, the *ide.jutge* client application has an acceptable load-time.

The first effective step to make the IDE more usable was to take advantage of the *jQuery EasyUI* framework. Its sophisticated yet practical user interface components procured for the IDE user an experience similar to that of a desktop application.

However, during most of the development of this project, the IDE was only available to the author and the project advisor. With such limited feedback, it is probable that undetected usability problems could be lurking in *ide.jutge*. Therefore, the ide.jutge Prototype was made available to eleven more persons for a usability test.

17.1 Usability test

The aim of the test is to get direct input on how real users experience the system. Eleven persons, including UPC staff and students, and a usability expert (Pere-Pau Vázquez), were presented with *ide.jutge*, and asked to try it out.

17.1.1 Feedback

Initial feedback unveiled some bugs in the IDE, which were promptly fixed. For example, a UPC faculty member explained:

> *- Agafo un programa qualsevol (p.e. de les plantilles) i l'executo. Tot va be.*
> *- Modifico lleugerament el programa, per fer alguna cosa diferent, i al fer-ho, introdueixo un error (p.e. falta un ";")*
> *- Quan faig "run", obtinc un error de compilacio, com es d'esperar.*
> *- Arreglo l'error*
> *- Un cop arreglat l'error, no puc tornar a executar el programa, perque el boto de "run" ha quedat desactivat.*

The usability expert provided a comprehensive usability report which pointed out several functional and aesthetic problems in *ide.jutge*. To quote a few:

> *Hi ha un comportament incoherent del sistema pel que respecta a com es desen els fitxers nous: No està clar quan es desa i quan no. Si es fa clic a desar, es demana la creació d'un fitxer, però si no es fa clic, quan s'intenta depurar, diu que està desant quelcom. Però a més, si es fa Ctrl+S, que és el mètode clàssic de desar en molts editors, s'accedeix a l'opció de desar del Firefox.*

> *Els missatges desapareixen. Si s'introdueix un error i s'intenta depurar, apareix un missatge a la barra inferior, però en una estona desapareix, cosa que resulta inesperada i incòmoda, perquè potser la persona ha deixat de mirar la finestra una estona.*

> *El botó de slow motion sembla un botó d'informació. I quan està corrent, la versió d'aturar és taronja, no vermella. Si l'objectiu és distingir una pausa d'un stop, caldria dir-li pausa, però això s'hauria de poder fer amb el botó de play.*

> *En tota la gestió d'aquests menús, caldria adoptar una estratègia que fes que tot fos molt més coherent: tant en etiquetatge com en funcionament, per exemple el play que fos play sempre independentment del mode, i es digués play/continuar/executar... el que sigui que indiqui que s'està tirant endavant. També caldria evitar duplicats, etc, que el botó stop permetés parar sempre...*

All the given suggestions were taken into account to enhance the IDE. The last piece of feedback received was very positive indeed:

> *Realment, tot ho he trobat bé, útil, intuïtiu i funcionant.*

17.1.2 Results

The most significant usability problem in *ide.jutge* was that it provided no instructions on how to use the more complex features, especially debugging. Initially, there was no way for new users to immediately know how to set breakpoints in the code. Likewise, there was was no explanation for all the C++ execution modes (*Run*, *Debug*, *Slow-motion* and *Valgrind*), and C++ execution control buttons did not present a homogeneous behavior in different modes.

In general, more in-application documentation was required. So, a series of enhancements were designed.

17.2 Enhancements

In response to the feedback received, the following features were added to the ide.jutge Prototype:

- Automatically focus the editor, the terminal, or other input components depending on the previous action (*Save file*, *Debug*, etc.).
- The terminal scrolls with program output, just like a real terminal.
- Download the code in the editor using the Ctrl+S key shortcut.
- The status bar is placed underneath the code editor to improve its visibility. Also, more descriptive messages are added, and text in the status bar stays on-screen until another message comes in.
- Unified C++ execution buttons behaviour throughout the *Run*, *Debug*, *Slow-motion*, and *Valgrind* modes.

- The *Problems* tab is renamed to *Diagnostics*.
- The *Diagnostics* table shows a message indicating no errors were found, instead of just being empty. The *Valgrind* results tables does the same.
- Menu buttons relabeled to better describe their function.

17.3 Summary

The efforts directed at making *ide.jutge* a highly usable IDE were detailed in this chapter. As expected, the usability test proved to be highly instructive. Constructive feedback allowed bug detection and provided fresh ideas on how to refurnish the IDE to favor usability. Suggestions were taken to polish *ide.jutge* by adding and modifying features, and relocating UI components. Nevertheless, usability can be further improved by taking feedback from a larger set of users: the UPC students.

Chapter 18

Performance

It is essential to know the boundaries of *ide.jutge*'s operating capabilities. Indeed, if *ide.jutge* is to be accepted as an educational tool in classes, and its use permitted in exams, safe-usage limits must be established to avoid system failure.

This chapter provides a description of the memory, network and CPU-usage analysis performed. It then details the stress tests conducted to determine the stability and availability of *ide.jutge* solution. Analysis and testing took place in `jutge3`, one of the *Jutge.org* servers.

18.1 Memory usage

This section analyzes RAM and storage (disk-space) consumption of *ide.jutge*. It is anticipated that memory usage will set a high limit to the number of concurrent users the system will support.

18.1.1 Storage

The server application requires only 74.4MB of free disk-space for installation.

The sandboxes are given a limited amount of disk-space (1GB), therefore the maximum amount of sandboxes is determined by the available storage in the *Slave Servers*. `jutge3` has over 2TB of free disk space, so it could host at least 2000 sandboxes.

18.1.2 RAM

The following results were sourced from the `htop` command. Execution of the server application consumed 56.7MB RAM. And after opening 100 connections, only an additional 13MB were used.

The sandboxes, on the other hand, were more demanding. Starting 100 sandboxes took up 1.46GB of RAM —an average of 14.6MB per sandbox. Such results are impressive considering that, to the users, the sandbox feels like an isolated computer.

The sandboxes are so memory-efficient because they are LXC containers, which use *operating-system-level virtualization*. This method allows all sandboxes to share the same operative-system kernel, and provides file-level copy-on-write (CoW) mechanisms. In effect, only new and modified resources require memory allocation.

18.2 Network

This section discusses *ide.jutge*'s network usage. Measurements were taken with *Google Chrome DevTools* [34] on the development computer, using a 300MB fiber-optics Internet connection.

It is unlikely that network usage will be a problem, since all modern browsers provide cache mechanisms to avoid repeatedly downloading unmodified assets (HTML and JavaScript files, images, etc.). Still, the whole client application must be served the first time a browser loads *ide.jutge*.

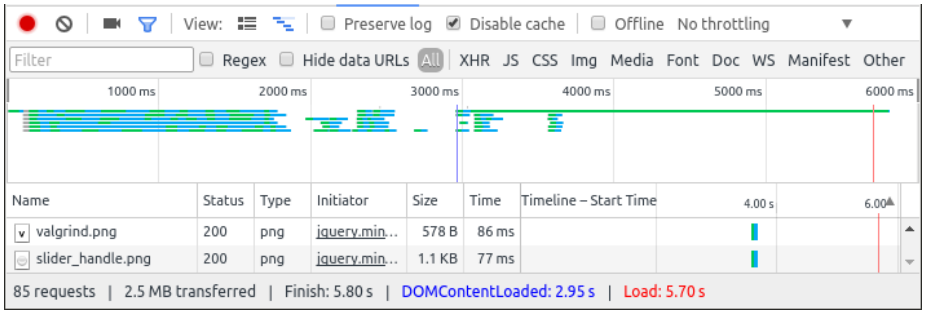

Figure 18.1: Network usage loading *ide.jutge*

Figure 18.1 shows network statistics of accessing `https://ide.jutge.org` on a clean cache. 85 requests are made, 30 of which are images, 30 are custom JavaScript files (the client application code), and the rest are CSS files

and JavaScript components. Most of the 2.5 MB transferred are third-party JavaScript files, the largest three being the *AngularJS* framework (1.2MB), *jQuery EasyUI* (410KB), and the *Ace* editor (346KB).

ide.jutge uses the minified (compressed) version of all its third-party libraries, so the there is only so much room for improvement in size. The number of requests, however, can be drastically reduced by concatenating JavaScript and CSS files. In an effort to lower network usage, the *ide.jutge* client application code is minified and concatenated into a single JavaScript file, using *Grunt* [66].

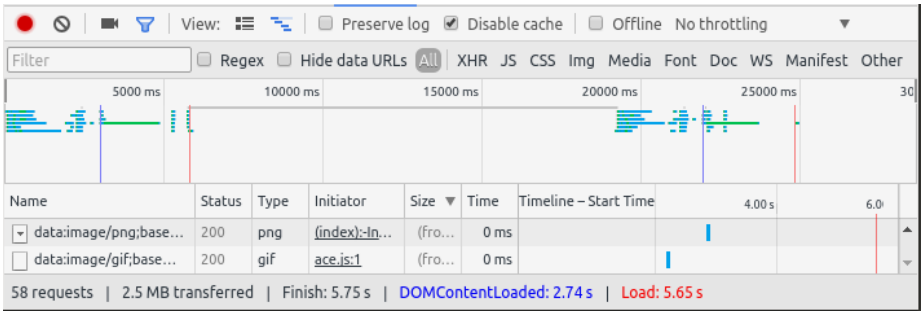

Figure 18.2: Network usage loading *ide.jutge* with minified client application

As figure 18.2 depicts, even though the number of requests is significantly diminished, overall load time does not perceivably improve. This is because the client application is too small in relation to the rest of the assets served (54.2KB compressed; twice as much otherwise), and the fiber-optics connection allows requests to be processed very quickly once the connection is open.

Speed tests indicate that `jutge3` has a 500Mbit/s upload bandwidth; that is enough to serve the *ide.jutge* client application 25 times per second. Nonetheless, server network usage could be lowered by serving CSS and JavaScript assets from a CDN (Content Delivery Network), an external server. This option could be implemented in a future iteration of *ide.jutge*, but it is discouraged because the IDE will be available to students during exams, and these take place in a secure environment with no Internet access.

18.3 CPU

This section analyzes CPU usage of *ide.jutge*. CPU usage is expected to be the main performance bottleneck in the *ide.jutge* solution. To verify this, tests are conducted to identify CPU usage peaks during regular *ide.jutge* operation.

18.3.1 Client tier

To ensure CPU-usage is reasonable in the web application, *ide.jutge* is accessed from the development computer, and different actions are performed while recording CPU-usage levels with the *Ubuntu System Monitor*. These actions involve running a C++ program, and performing common tasks such as writing code in the editor and switching the IDE themes. Figure 18.3 depicts CPU consumption during the test. The initial large peak is caused by application load, and the three smaller peaks are triggered when opening the top menus for the first time.

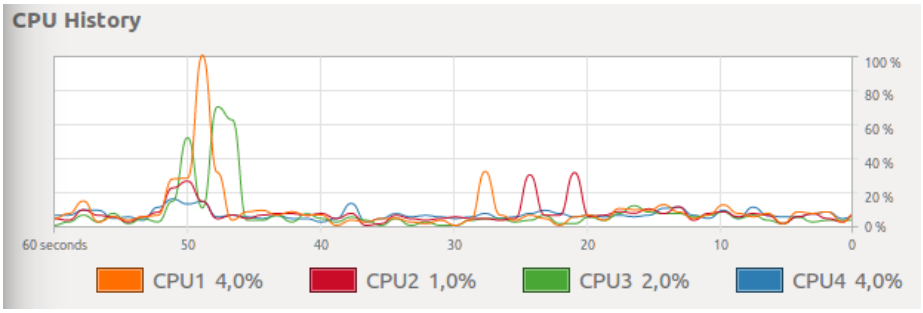

Figure 18.3: Web application CPU usage test

18.3.2 Server tier

The first step in the test is to start the server application. The second step is to access the IDE, run a "Hello world", and close the IDE by browsing to another website. This last step is repeated three times during the course of three minutes. Finally, the server application is killed.

The IDE was accessed and operated from the development machine. CPU-usage data was collected using a custom `bash` script, and the plot was generated with *gnuplot* [68]. Figure 18.4 diagrams the test.

Peaks during C++ program executions are not relevant. In the sandboxes, CPU-usage is limited and will never go above a certain threshold. It is then clear that the most CPU-consuming task is starting the sandboxes — it triggers the eight-second CPU peaks in the figure above. Starting a large amount of sandboxes in a short period of time can cause the system to become unresponsive; therefore, a series of stress tests are necessary to pinpoint the maximum number of sandboxes *ide.jutge* can start at a time.

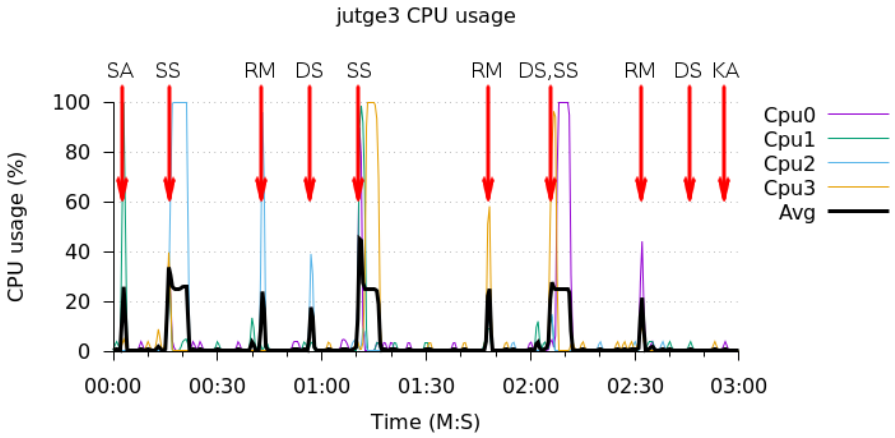

Figure 18.4: CPU usage test
SA: Start server application
KA: Kill server application
SS: Start sandbox (access *ide.jutge*)
RM: Run "Hello world" program
DS: Destroy sandbox (leave *ide.jutge*)

18.4 CPU stress tests

The objective of the stress tests is to determine the maximum rate at which `jutge3` can start new sandboxes while *ide.jutge* remains responsive to the users. To reach this objective, five tests were conducted; in each test, an increasing number of LXC containers (sandboxes) are started within the course of 1 minute. Test 1 starts 10 containers, test 2 starts 20 containers, etc. Container startup times and CPU-usage data will be analyzed to provide the results.

18.4.1 Tools

As in the previous test, CPU-usage data is collected using a `bash` script, and plots are generated with *gnuplot*. However, instead of starting the sandboxes by accessing the IDE, a Python script is created to automate this task. This script generates a Poisson distribution, and the samples are used to determine when to start new sandboxes. The Python script will start the sandboxes within the course of one minute, via another custom `bash` script, which also measures their startup times.

18.4.2 Results

Results indicate that instantiation of more than one LXC container every two seconds will increasingly deteriorate container startup times. As figure 18.5 diagrams, instantiation of 40 containers or more raises sandbox startup times beyond the 10-second usability threshold in under 1 minute.

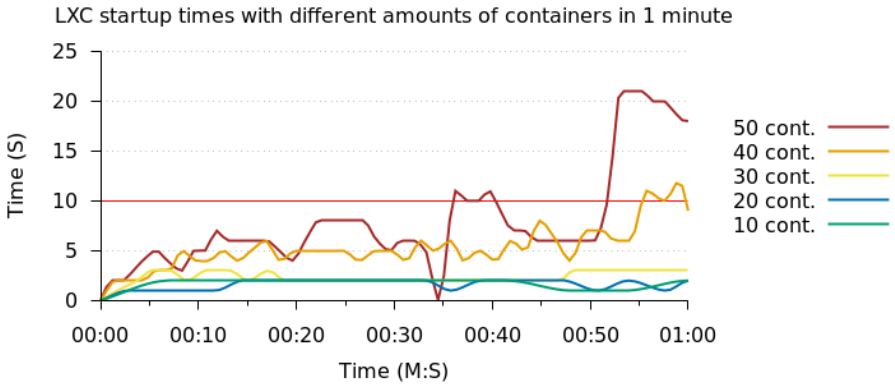

Figure 18.5: Container startup times for tests 1-5

Figure 18.6 shows the results for test 2, in which sandboxes can be started at a sustainable rate. Figure 18.7, in contrast, displays how test 4 puts an extremely heavy load on the system and container startup times surpass the 10-second limit.

ide.jutge can deal with such situations because it is a horizontally scalable solution. The way to maintain availability under excessive workloads is to use more *Slave Servers*. Cloud-computing providers such as *Amazon Web Services* [3] could provide a stable infrastructure for *ide.jutge*, if its use were to grow beyond the operating capabilities of the *Jutge.org* servers.

18.5 Summary

This chapter details how the *ide.jutge* performs in terms of memory, network and CPU-usage, within the *Jutge.org* server `jutge3`. Potential performance problems were identified, and the ways to deal with them were described. It was also confirmed that CPU-usage is *ide.jutge*'s major performance bottleneck. The consequent stress tests provided an upper bound for the rate at which new sandboxes can be served.

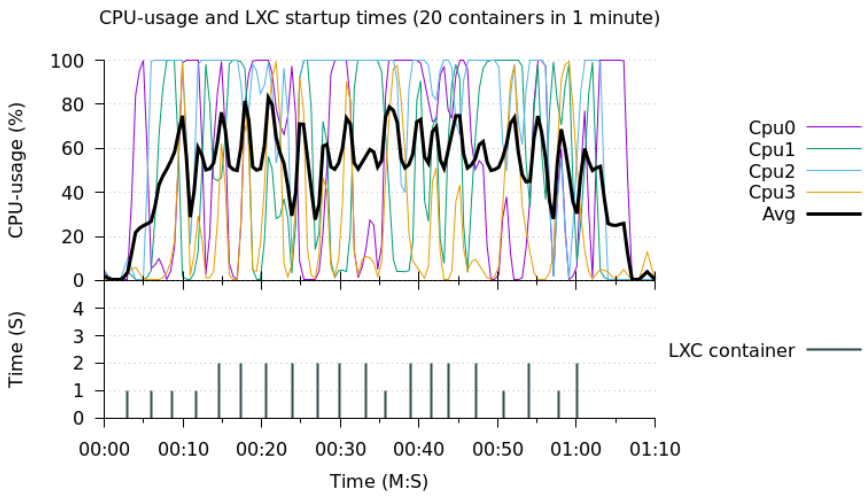

Figure 18.6: Stress test 2, a sustainable workload

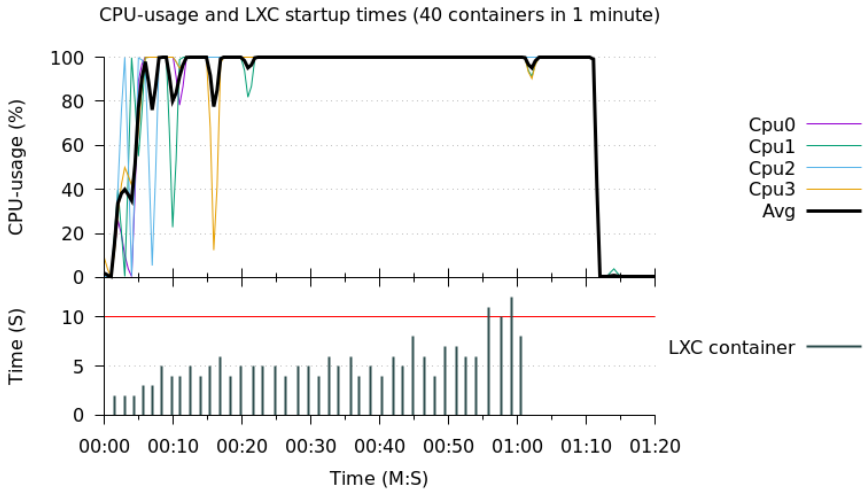

Figure 18.7: Stress test 4 overloads the CPU

Chapter 19

Validation

This chapter provides a validation summary of both main and secondary objectives, and explains deviations from the initial planning during the implementation phase in terms of time and cost.

19.1 Validation summary

This section confirms, through objective evidence, that *ide.jutge* will perform its intended functions.

19.1.1 Main objectives

As stated in section 3.1:

> *The main objective of this project is to build a Web IDE for students to create and inspect their C++ programs*

The *ide.jutge* client application allows editing, compiling, executing, and debugging C++ code, thus providing the Web IDE. Its effective operation is guaranteed by the backing server-side infrastructure. The sandboxes in the *Slave Servers* provide the isolated environment where programs are compiled, executed and debugged; the communication channel between the IDE and the sandboxes is supplied by the *Master Server* application. Performance analysis and usability tests ensure that the system operates as intended. Hence, *ide.jutge* accomplishes the main objective.

19.1.2 Secondary objectives

As stated in section 3.2, secondary objectives are:

> *Create an extensible plugin system for customizing the environment*

Such system is created in the Plugins Prototype (ch. 14). Testing of said proto-type ensures this secondary objective is achieved.

> *Make the infrastructure scalable, stable and secure*

Chapter 4 describes the design of an architecture that possesses said characteristics. Indeed, special attention has been paid throughout the development of the project to implement such infrastructure.

Security mechanisms for *ide.jutge* are implemented in the Express Prototype (ch. 11) and the Sandbox Prototype (ch. 12). Horizontal and vertical scalability is added in the Sandbox Prototype and the Plugins Prototype, respectively. Usability testing and performance analysis results prove that the infrastructure is in effect stable.

> *Create a step-by-step debug mode, detect memory errors in the student's programs, import and export code from GitHub.com*

The remaining three secondary objectives are achieved in the ide.jutge Prototype (ch. 16).

19.2 Planning review

This section evaluates how closely the implementation phase of the project followed the initial plan regarding time and costs.

19.2.1 Time management

The original time plan underwent severe changes during the implementation phase. The fact that the project started almost a year prior to its final presentation allowed the author to spend more time in the development and polishing of *ide.jutge*. In hindsight, the availability of an additional 6 months proved

to be key in the completion of the project —*ide.jutge* is a highly-sophisticated and complex software solution with a myriad of interacting components. It is estimated that 300 extra hours were spent in the development of *ide.jutge*.

Also, task scheduling was changed to embrace the RAD methodology. This methodology consists in short development cycles where prototypes are built and integrated. Therefore, analysis and design of the prototypes is required throughout development, and so is testing and polishing. Table 19.1 diagrams the final amount of work done on each of the individual tasks.

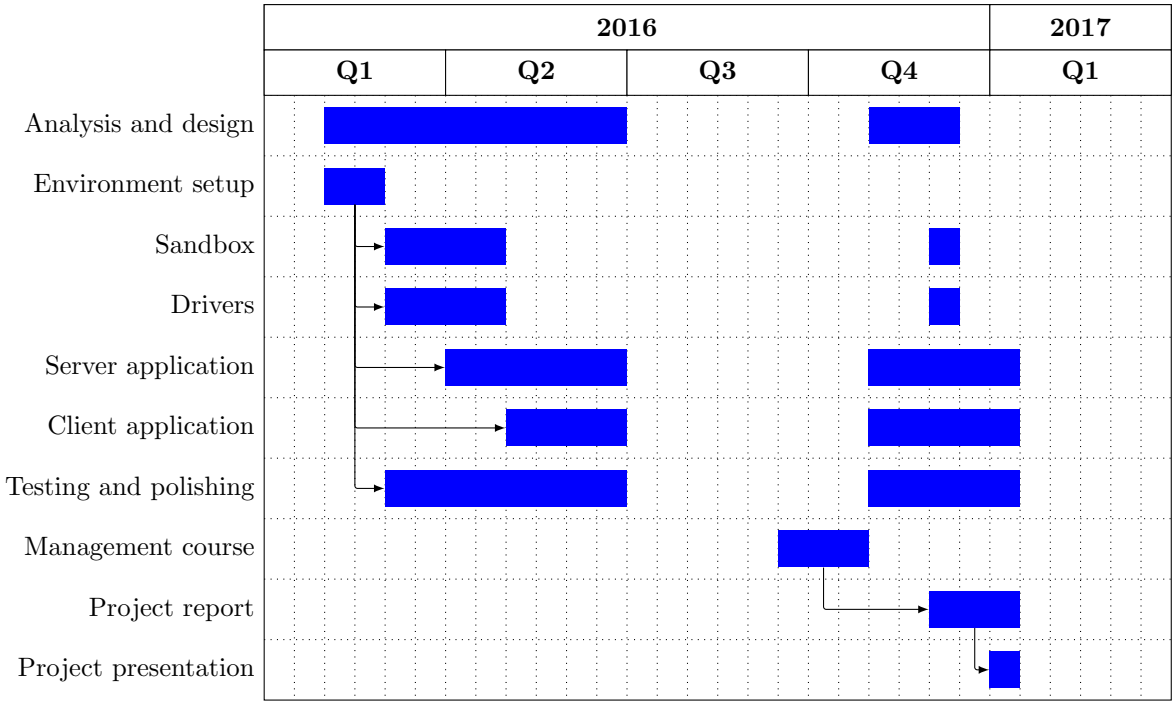

Table 19.1: Final Gantt chart

19.2.2 Economic cost

The acquisition of *jQuery EasyUI* was not figured into in the initial budget. The licence costs $449, works indefinitely, and allows use of *EasyUI* in any number of projects [71]. Since this JavaScript framework will also be used in another Degree Final Project by a fellow student, its amortized cost is $224.5. Said extra cost would be paid using the 10% contingency deposit.

The extra time spent in the development of the project would have

incurred an estimated 10,000€ additional fee for human resources (see table 19.2). In this case, the 10% contingency budget would have been insufficient.

Role	**Time (h)**	**Cost(€)**
Software engineer	100	4000.00
Software developer	200	6000.00
Total	300	10000.00

Table 19.2: Extra cost in project staff

Chapter 20

Legal Aspects

No special laws apply to this project. The software is available under the MIT licence:

Chapter 21

Conclusions

The final chapter of the project presents a summary of this document, and the possible next steps of the development of *ide.jutge* are reviewed. Finally, the author closes the report with his personal thoughts.

21.1 Summary

This document detailed the building process of a Web IDE with C++ debugging capabilities.

Part I: The project requirements were established, and it was found that there were no existing solutions that satisfied those requirements. Therefore, a new solution was devised: *ide.jutge*.

Part II: Development of the solution was planned. Initial estimations for the time and economic costs were produced, along with a sustainability analysis.

Part III: Implementation of the solution was detailed. Using the RAD methodology, prototypes were incrementally built —each one contributing a key set of features to finally compose the *ide.jutge* solution.

Part IV: The implemented solution was evaluated with usability and performance tests. The project's objectives were validated, and time and cost divergences from the original planning were outlined.

21.2 Future work

The future of this *ide.jutge* is a bright one. Given that it will be used as an educational tool for the UPC, many enhancements will be made. For one, student feedback will help further refine the IDE's usability.

The next steps towards scaling the solution will be:

A complete desktop: *ide.jutge* will allow students to use the computational power of the *Jutge.org* servers. Granting persistent storage as well would be a major improvement. After all, each students gets a certain disk-space quota in the UPC servers, which could be made available through the IDE. This would also facilitate managing not just one file but complete projects with *ide.jutge*.

Plugins: With the creation of plugins, additional functionality can be made available in the IDE. For example, a new plugin could automate the task of executing the public test cases for *Jutge.org* problems. Another plugin could provide a debugger for Java or Python code. Instructions on how to create a new plugin can be found in section 14.5.

***ide.jutge* in the cloud:** Cloud-computing providers offer very reliable infrastructures on which to run the *ide.jutge* solution. This way, the number of sandbox containers would auto-scale dynamically with computing load, which would eliminate the problem of running out of resources in the *Jutge.org* servers.

21.3 Personal thoughts

Thinking about the experiences of this project, I ended up reflecting on my personal journey through the world of computer programming. That journey began with a younger me playing lot of video-games. The joy I felt playing motivated the most basic of questions: "How do they do it!?!". Ergo, I set off on a learning path.

This project is the latest milestone in my journey; I have gained a great deal of knowledge in the making of *ide.jutge*. The sheer breadth of the solution attests to this. I used JavaScript as the runtime for a server application with *Node.js*. I learned about real-time server-client communication using WebSockets. I experienced the power of operative system virtualization with LinuxContainers. I studied GDB and its machine-oriented input/output syntax. I built a plugins system on top of the *AngularJS* framework. The list is just too long to be included in this section!

I greatly enjoyed making this project, and I regard it as an extremely positive experience. Needles to say, it has been a huge challenge —then again, perhaps that is the reason it was so fun. In the future, I hope to continue working with the development of Web-based tools for programmers.

My best efforts were put into *ide.jutge*, and I am happy to think that the work done for this project will help students who are on a journey similar to the one I started years ago.

21.4 Acknowledgements

This work would have not been possible without the continuous support of the project advisor, Jordi Petit. I would also like to thank all the collaborators in the usability test, especially Pere-Pau Vázquez. A special thanks goes to my girlfriend, Emily Shirk, for providing moral support and help in the polishing of this document.

Bibliography

[1] ajax.org cloud 9. *Ace - The High Performance Code Editor for the Web*. Jan. 2, 2017. URL: https://ace.c9.io/.

[2] Central Computer & Telecommunications Agency. *SSADM Foundation (Business Systems Development with SSADM)*. Business Systems Development with SSADM. Stationery Office Books, 2000. ISBN: 978-0113308705.

[3] Inc. Amazon Web Services. *Amazon Web Services (AWS) - Cloud Computing Services*. Jan. 2, 2017. URL: https://aws.amazon.com/?nc2=h_lg.

[4] Pierre Carbonnelle. *Top IDE Index*. Dec. 29, 2016. URL: http://pypl.github.io/IDE.html.

[5] Pierre Carbonnelle. *Top ODE Index*. Dec. 29, 2016. URL: http://pypl.github.io/ODE.html.

[6] Inc Cloud9 IDE. *Cloud9 - Your development environment, in the cloud*. Dec. 29, 2016. URL: https://c9.io/.

[7] European comission. *Analyse one indicator and compare countries*. Jan. 2, 2017. URL: http://digital-agenda-data.eu/charts/analyse-one-indicator-and-compare-countries#chart=%7B%22indicator-group%22:%22any%22,%22indicator%22:%22Price_Internet_Fixed_Tel%22%7D.

[8] Creative Commons. *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national*. Dec. 29, 2016. URL: https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.txt.

[9] Virtuozzo company. *OpenVZ Virtuozzo containers wiki*. Jan. 2, 2017. URL: https://openvz.org/Main_Page.

[10] Derbyjs contributors. *DerbyJS is a full-stack framework for writing modern web applications*. Jan. 2, 2017. URL: http://derbyjs.com/.

[11] Hapi contributors. *Hapi.js - A rich framework for building applications and services*. Jan. 2, 2017. URL: https://hapijs.com/.

[12] Socket.io contributors. *Socket.io*. Jan. 2, 2017. URL: http://socket.io/.

[13] Ryan Dahl. *Ryan Dahl: Original Node.js presentation*. Jan. 2, 2017. URL: https://www.youtube.com/watch?v=ztspvPYybIY.

[14] GDB developers. *GDB: The GNU Project Debugger*. Jan. 2, 2017. URL: https://www.gnu.org/software/gdb/.

[15] Linux-VServer developers. *Linux-VServer provides virtualization for GNU/Linux systems*. Jan. 2, 2017. URL: http://linux-vserver.org/Welcome_to_Linux-VServer.org.

[16] Valgrind developers. *Valgrind is an instrumentation framework for building dynamic analysis tools*. Jan. 2, 2017. URL: http://valgrind.org/.

[17] Fabrizio Balliano and Kevin Dalman. *UI Layout - The Ultimate Page Layout Manager*. Jan. 2, 2017. URL: http://layout.jquery-dev.com/.

[18] Michal Forišek. "Security of programming contest systems". In: *Information technologies at school: Selected papers of the 2nd international conference "Informatics in Secondary Schools: Evolution and Perspectives" (Valentina Dagiene, Roland Mittermeir, ed.), Institute of Mathematics and Informatics, Vilnius* (2006), pp. 553–563. DOI: https://pdfs.semanticscholar.org/891c/c6a05cf24e17bf9226a8eb523fa4756cd265.pdf.

[19] Free Software Foundation. *About the GNU project*. Jan. 2, 2017. URL: https://www.gnu.org/gnu/thegnuproject.en.html.

[20] Free Software Foundation. *GDB/MI - Debugging with GDB*. Jan. 2, 2017. URL: https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.

[21] Free Software Foundation. *GNU Emacs: An extensible, customizable, free/libre text editor — and more.* Dec. 29, 2016. URL: https://www.gnu.org/software/emacs/.

[22] Free Software Foundation. *What is free software?* Jan. 2, 2017. URL: https://www.gnu.org/philosophy/free-sw.en.html.

[23] Node.js Foundation. *A JavaScript runtime built on Chrome's V8 JavaScript engine*. Jan. 2, 2017. URL: https://nodejs.org/.

[24] Node.js Foundation. *Express - Node.js web application framework*. Jan. 2, 2017. URL: http://expressjs.com/.

[25] Node.js Foundation. *Express Routing*. Jan. 2, 2017. URL: https://expressjs.com/en/guide/routing.html.

[26] Node.js Foundation. *Using Express middleware*. Jan. 2, 2017. URL: http://expressjs.com/en/guide/using-middleware.html.

[27] OpenSSL Software Foundation. *OpenSSL - Cryptography and SSL/TLS Toolkit*. Jan. 2, 2017. URL: https://www.openssl.org/.

[28] The Eclipse Foundation. *Eclipse - The Eclipse Foundation open source community website*. Dec. 29, 2016. URL: https://eclipse.org/.

[29] The jQuery Foundation. *jQuery user interface*. Jan. 2, 2017. URL: https://jqueryui.com/.

[30] The jQuery Foundation. *jQuery - write less, do more.* Jan. 2, 2017. URL: https://jquery.com/.

[31] Google. *AngularJS: Developer Guide: Dependency Injection.* Jan. 2, 2017. URL: https://docs.angularjs.org/guide/di.

[32] Google. *AngularJS: Developer Guide: HTML compiler.* Jan. 2, 2017. URL: https://docs.angularjs.org/guide/compiler.

[33] Google. *AngularJS - Superheroic JavaScript MVW Framework.* Jan. 2, 2017. URL: https://angularjs.org/.

[34] Google.com. *Chrome DevTools Overview.* Jan. 2, 2017. URL: https://developer.chrome.com/devtools.

[35] Docker Inc. *Build, ship and run any app, anywhere.* Jan. 2, 2017. URL: https://www.docker.com/.

[36] Facebook Inc. *A JavaScript library for building user interfaces - React.* Jan. 2, 2017. URL: https://facebook.github.io/react/.

[37] GitHub Inc. *About gists - User documentation.* Dec. 29, 2016. URL: https://help.github.com/articles/about-gists/.

[38] GitHub Inc. *OAuth - GitHub Developer Guide.* Jan. 2, 2017. URL: https://developer.github.com/v3/oauth/.

[39] Meteor Development Group Inc. *Meteor - The fastest way to build javascript apps.* Jan. 2, 2017. URL: https://www.meteor.com/.

[40] npm Inc. *npm.* Jan. 2, 2017. URL: https://www.npmjs.com/.

[41] Sencha Inc. *Ext JS - JavaScript framework for cross-platform web apps.* Jan. 2, 2017. URL: https://www.sencha.com/products/extjs/.

[42] Statista Inc. *Electricity prices by country.* Jan. 2, 2017. URL: https://www.statista.com/statistics/263492/electricity-prices-in-selected-countries/.

[43] Iyad Zayour, Hassan Hajjdiab. "How Much Integrated Development Environments (IDEs) Improve Productivity?" In: *Journal of Software* 8.10 (2013), pp. 2425–2431. DOI: https://pdfs.semanticscholar.org/019e/36673a0821b1864b9d62b527813e900d13ba.pdf.

[44] Jordi Petit, Omer Giménez, Salvador Roura. "Jutge.org: an educational programming judge". In: *SIGCSE '12 Proceedings of the 43rd ACM technical symposium on Computer Science Education* (2012), pp. 445–450. DOI: http://dl.acm.org/citation.cfm?doid=2157136.2157267.

[45] Jordi Petit, Salvador Roura. *Jutge.org - The Virtual Learning Environment for Computer Programming.* Dec. 29, 2016. URL: https://jutge.org/.

[46] JSFiddle. *jsfiddle - Create a new fiddle.* Dec. 29, 2016. URL: https://jsfiddle.net/.

[47] json.org. *Introducing JSON.* Jan. 2, 2017. URL: http://www.json.org/.

[48] Meggin Kearney. *Measure Performance with the RAIL Model*. Jan. 2, 2017. URL: https://developers.google.com/web/fundamentals/performance/rail?hl%3Den%23response_respond_in_under_100ms.

[49] knockoutjs.com. *Knockout - Simplify dynamic JavaScript UIs with the Model-View-View Model (MVVM) pattern*. Jan. 2, 2017. URL: http://knockoutjs.com/.

[50] Steve Krug. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability (3rd Edition)*. Voices That Matter. New Riders, 2014. ISBN: 978-0321965516.

[51] Sphere Research Labs. *ideone.com - Online Compiler and IDE*. Dec. 29, 2016. URL: http://ideone.com/.

[52] Canonical Ltd. *LinuxContainers.org - Infrastructure for container projects*. Jan. 2, 2017. URL: https://linuxcontainers.org/.

[53] James Martin. *Rapid Application Development*. Macmillan Coll Div, 1991. ISBN: 978-0023767753.

[54] Microsoft. *Visual Studio — Developer Tools and Services — Microsoft IDE*. Dec. 29, 2016. URL: https://www.visualstudio.com/.

[55] Mik Kersten, Gail C. Murphy. "Using Task Context to Improve Programmer Productivity". In: *SIGSOFT '06/FSE-14 Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), pp. 1–11. DOI: https://www.tasktop.com/sites/default/files/2006-11-task-context-fse.pdf.

[56] Yahoo! developer network. *Mojito - A JavaScript MVC framework for mobile applications*. Jan. 2, 2017. URL: https://developer.yahoo.com/cocktails/mojito/.

[57] Nicolas Petton, Esteban Lorenzano, Damien Cassou. *Pharo: The immersive programming experience*. Dec. 29, 2016. URL: http://pharo.org/.

[58] Jakob Nielsen. *Powers of 10: Time Scales in User Experience*. Jan. 2, 2017. URL: https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/.

[59] John Papa. *Angular 1 Style Guide*. Jan. 2, 2017. URL: https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md.

[60] John Papa. *Bootstrap - The world's most popular mobile-first and responsive front-end framework*. Jan. 2, 2017. URL: http://getbootstrap.com/.

[61] Hunter Perrin. *PNotify - Beautiful JavaScript notifications*. Jan. 2, 2017. URL: https://sciactive.github.io/pnotify/.

[62] *Smalltalk.org^{TM} — versions — ANSIStandardSmalltalk.html*. Dec. 29, 2016. URL: http://web.archive.org/web/20060216073334/http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html.

[63] SourceLair. *Xterm.js is a terminal front-end component written in JavaScript that works in the browser*. Jan. 2, 2017. URL: http://xtermjs.org/.

[64] stackoverflow.com. *'node.js' tag wiki*. Jan. 2, 2017. URL: http://stackoverflow.com/tags/node.js/info.

[65] Bower team. *Bower - A package manager for the web*. Jan. 2, 2017. URL: https://bower.io/.

[66] Grunt Development Team. *Grunt: The JavaScript Task Runner*. Jan. 2, 2017. URL: http://gruntjs.com/.

[67] Massachusetts Institute of Technology. *The MIT License*. Dec. 29, 2016. URL: https://opensource.org/licenses/MIT/.

[68] Thomas Williams, Colin Kelley. *Gnuplot is a portable command-line driven graphing utility*. Jan. 2, 2017. URL: http://www.gnuplot.info/.

[69] TutorialsPoint. *Free online IDE and terminal*. Dec. 29, 2016. URL: https://www.tutorialspoint.com/codingground.htm.

[70] Webismymind. *EditableGrid, build powerful editable tables*. Jan. 2, 2017. URL: http://www.editablegrid.net/en/.

[71] www.jeasyui.com. *EasyUI - Purchase and Contact*. Jan. 2, 2017. URL: http://www.jeasyui.com/contact.php.

[72] www.jeasyui.com. *jQuery EasyUI - helps you build your web pages easily*. Jan. 2, 2017. URL: http://www.jeasyui.com/.

[73] Bennet Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*. 2009. DOI: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34913.pdf.