LabBack: an extendible platform for secure and robust in-the-cloud automatic assessment of student programs

Master’s Thesis

Vlad A. Vergu
LabBack: an extendible platform for secure and robust in-the-cloud automatic assessment of student programs

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Vlad A. Vergu
born in Bucharest, Romania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
© 2012 Vlad A. Vergu.

Cover picture: The last straw, drawing by Maria I. Ionescu, 2012
LabBack: an extendible platform for secure and robust in-the-cloud automatic assessment of student programs

Author: Vlad A. Vergu
Student id: 1195549
Email: v.a.vergu@student.tudelft.nl

Abstract

In software engineering education manual assessment of students’ programs is a time consuming endeavour that does not scale to high numbers of pupils or to large amounts of practice material. Numerous automatic grading programs, aiming to alleviate this problem, have been developed since 1960. These, however, only support fixed programming languages and fixed assessment methods, thus prohibiting their reuse throughout different programming curricula. Educators investigating new grading methods either have to accept the engineering burden of creating a complete grading system or revert to manual grading.

This thesis presents LabBack - a reusable automatic grading platform that is extendible with language- and assessment-specific functionality by means of plugins. LabBack provides the necessary infrastructure for building and hosting automatic grading functionality, eliminating the need to consider the issues of scalability and security against malicious programs. LabBack can be hosted in the cloud and provides immediate student feedback. Plugins providing automatic assessment for Scala, JavaScript and C have been implemented and LabBack has been validated in a university-level course with over 100 students.

Thesis Committee:

Chair: Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor: Dr. E. Visser, Faculty EEMCS, TU Delft
Committee member: Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee member: Dr. ir. J.F.M. Tonino, Faculty EEMCS, TU Delft
Preface

About this thesis  This thesis essentially consists of two parts. The first part (Chapter 1 to Chapter 5) presents the architecture of LabBack’s proof of concept - LabBack 2.0 - and its validation in a case study, serving as motivation for the second part. The second part (Chapter 6 to Chapter 8) describes the architecture, implementation and evaluation of LabBack 2.0 - an extendible and reusable platform for automatic grading that builds on LabBack 2.0’s main security concepts.

Acknowledgements  I am grateful to my supervisor Eelco Visser for the inspirational discussions and feedback sessions and for his development of WebLab, without whom this project would not have been possible. I would like to thank Karl Trygve Kalleberg for our interesting discussions during his visit to The Netherlands and Danny Groenewegen for his support with using WebDSL. Last, but certainly not least, I am grateful to my family for supporting me during my time at the university.

Vlad A. Vergu
Delft, The Netherlands
October 30, 2012
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Research Questions</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Outline of this Thesis</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Massive Open Online Course Platforms</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Case study: COPL</td>
<td>9</td>
</tr>
<tr>
<td>2.3 WebLab</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Summary</td>
<td>19</td>
</tr>
<tr>
<td><strong>3 Analysis of LabBack Requirements</strong></td>
<td>21</td>
</tr>
<tr>
<td>3.1 Functional Requirements</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Non-functional Requirements</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>32</td>
</tr>
<tr>
<td><strong>4 Satisfying the Non-Functional Requirements of LabBack</strong></td>
<td>33</td>
</tr>
<tr>
<td>4.1 Native Tooling</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Virtualised Solutions</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Build Tools</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Continuous Integration Systems</td>
<td>39</td>
</tr>
<tr>
<td>4.5 Custom Solution</td>
<td>40</td>
</tr>
<tr>
<td>4.6 Conclusion and Recommendation</td>
<td>41</td>
</tr>
<tr>
<td><strong>5 Proof of Concept: LabBack 1.0</strong></td>
<td>43</td>
</tr>
<tr>
<td>5.1 Architecture</td>
<td>43</td>
</tr>
<tr>
<td>5.2 Implementation Details</td>
<td>56</td>
</tr>
<tr>
<td>5.3 Evaluation</td>
<td>64</td>
</tr>
</tbody>
</table>
# Contents

5.4 Conclusion ................................................. 67

6 LabBack 2.0 Architecture 69
6.1 Java Plugin Frameworks ................................. 69
6.2 LabBack 2.0 Context .................................... 71
6.3 System Decomposition ................................... 72
6.4 Information Architecture ............................... 77
6.5 Concurrency ............................................... 82
6.6 Business processes ....................................... 85
6.7 Services .................................................. 94
6.8 Security .................................................. 103
6.9 Deployment Requirements .............................. 106
6.10 Conclusion ............................................... 107

7 LabBack 2.0 Implementation and Integration Details 109
7.1 WebDSL and Java code ................................. 109
7.2 Plugin System ............................................. 111
7.3 WebLab as Client: Contributions to WebLab .......... 117
7.4 Summary ................................................ 119

8 Evaluation 121
8.1 Goals and Structure ..................................... 121
8.2 Evaluation of Correctness, Robustness and Security 122
8.3 Evaluation of Scalability and Load Distribution ..... 127

9 Related work 141
9.1 Automatic Grading ..................................... 141
9.2 Web IDEs and Containment of Third-party Programs 142
9.3 Load Balancing and Distribution ...................... 144

10 Conclusion 145
10.1 Summary ............................................... 145
10.2 Conclusions ........................................... 147
10.3 Future Work ........................................... 150

Bibliography 153

A LabBack 1.0 Scheduler Implementation 159

B LabBack 1.0 Classloader Implementation 163

C CounterLogic Execution Implementation 165

D Plugin Loader Implementation of LabBack 2.0 167
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Course edition dashboard in WebLab as seen by an instructor</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>WebLab course information page for the COPL 2012 course.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>WebLab course information page showing correlation plots.</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>WebLab page for editing a programming question.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Specification test template for Javascript question.</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>WebLab page for answering a programming question.</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Editing of question’s grading options in WebLab.</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>Page for submission grading in WebLab.</td>
<td>17</td>
</tr>
<tr>
<td>2.9</td>
<td>WebLab architectural components.</td>
<td>18</td>
</tr>
<tr>
<td>2.10</td>
<td>Example unit tests using the Scalatest framework.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Summarised classification of LabBack requirements.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Classification of potential LabBack security attack vectors and their inten-</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>tended purposes.</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Example of JVM memory starvation implemented in Java.</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Example fork-bomb in C.</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Example fork-bomb in Java.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of ecosystem alternatives</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>LabBack 1.0 context diagram. Arrows indicate usage.</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>LabBack 1.0 system decomposition.</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Job data model.</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Example XML representation of completed job.</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>LabBack 1.0 processes and threads.</td>
<td>51</td>
</tr>
<tr>
<td>5.6</td>
<td>Job state diagram.</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>BPMN blueprint of LabBack’s main process.</td>
<td>53</td>
</tr>
<tr>
<td>5.8</td>
<td>LabBack 1.0 security policy.</td>
<td>54</td>
</tr>
<tr>
<td>5.9</td>
<td>LabBack 1.0’s scheduler maintenance loop algorithm.</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>Custom class loader algorithm.</td>
<td>61</td>
</tr>
<tr>
<td>5.11</td>
<td>Usage example of LabBack’s custom class loader.</td>
<td>62</td>
</tr>
<tr>
<td>5.12</td>
<td>Example stoppable thread in Java.</td>
<td>64</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Comparison of Java plugin frameworks. Highlighted cells indicate critical</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>limitations.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>LabBack 2.0 multi-instance deployment tree.</td>
<td>71</td>
</tr>
<tr>
<td>6.3</td>
<td>LabBack 2.0 system decomposition.</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Plugin manifest for the JavaScript plugin.</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Class diagram of LabBack 2.0’s core data model.</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Data model of LabBack 2.0’s job.</td>
<td>78</td>
</tr>
<tr>
<td>6.7</td>
<td>Data model of LabBack 2.0’s main plugin entity.</td>
<td>81</td>
</tr>
<tr>
<td>6.8</td>
<td>Process and thread model of LabBack 2.0</td>
<td>82</td>
</tr>
<tr>
<td>6.9</td>
<td>State model of a single job or a chains of jobs in LabBack 2.0.</td>
<td>84</td>
</tr>
<tr>
<td>6.10</td>
<td>Transition function of job chain state machine.</td>
<td>85</td>
</tr>
<tr>
<td>6.11</td>
<td>Blueprint of the Plugin Deployment (PD) process.</td>
<td>86</td>
</tr>
<tr>
<td>6.12</td>
<td>Blueprint of the Execution Loading (EL) process.</td>
<td>87</td>
</tr>
<tr>
<td>6.13</td>
<td>Partial blueprint of the Local Job Scheduling (LJS) process.</td>
<td>89</td>
</tr>
<tr>
<td>6.14</td>
<td>Schematic blueprint of Remote Job Scheduling (RJS) process.</td>
<td>91</td>
</tr>
<tr>
<td>6.15</td>
<td>Summary of job-related services offered by LabBack 2.0.</td>
<td>95</td>
</tr>
<tr>
<td>6.16</td>
<td>Meaning of flag combinations in response of queryJobStatus service.</td>
<td>98</td>
</tr>
<tr>
<td>6.17</td>
<td>Summary of plugin-related services offered by LabBack 2.0.</td>
<td>100</td>
</tr>
<tr>
<td>6.18</td>
<td>Comparison of LabBack 1.0 and LabBack 2.0 functional groups.</td>
<td>104</td>
</tr>
<tr>
<td>6.19</td>
<td>Software and hardware requirements of LabBack 2.0.</td>
<td>106</td>
</tr>
<tr>
<td>7.1</td>
<td>Definition of a WebDSL page with inter-thread message passing.</td>
<td>110</td>
</tr>
<tr>
<td>7.2</td>
<td>Processing steps of plug.sc.counter plugin.</td>
<td>111</td>
</tr>
<tr>
<td>7.3</td>
<td>Manifest of plug.sc.counter plugin.</td>
<td>112</td>
</tr>
<tr>
<td>7.4</td>
<td>Compulsory interface of all plugin entry points in LabBack 2.0.</td>
<td>112</td>
</tr>
<tr>
<td>7.5</td>
<td>Execution logic in CounterLogic.</td>
<td>114</td>
</tr>
<tr>
<td>7.6</td>
<td>Pseudocode specification of plugin loading mechanism.</td>
<td>115</td>
</tr>
<tr>
<td>7.7</td>
<td>Illustration of plugin detachment upon upgrade.</td>
<td>116</td>
</tr>
<tr>
<td>7.8</td>
<td>Summary UI contributions to WebLab.</td>
<td>117</td>
</tr>
<tr>
<td>7.9</td>
<td>Monitoring and configuration UI for LabBack 2.0 instances in WebLab.</td>
<td>118</td>
</tr>
<tr>
<td>7.10</td>
<td>WebLab’s Language configuration UI.</td>
<td>119</td>
</tr>
<tr>
<td>8.1</td>
<td>Distribution of job states and languages in evaluation dataset</td>
<td>123</td>
</tr>
<tr>
<td>8.2</td>
<td>Differences between original and new grades in the failure cases.</td>
<td>124</td>
</tr>
<tr>
<td>8.3</td>
<td>Clustering of grade validation failures into questions.</td>
<td>125</td>
</tr>
<tr>
<td>8.4</td>
<td>Summary of variables for simulation control.</td>
<td>129</td>
</tr>
<tr>
<td>8.5</td>
<td>User interface of LabBack’s deployment simulator.</td>
<td>130</td>
</tr>
<tr>
<td>8.6</td>
<td>Simulation results for control experiment.</td>
<td>132</td>
</tr>
<tr>
<td>8.7</td>
<td>Simulation results for 10 node star topology.</td>
<td>133</td>
</tr>
<tr>
<td>8.8</td>
<td>Simulation results for 10 node tree of depth 6.</td>
<td>134</td>
</tr>
<tr>
<td>8.9</td>
<td>Simulation results for 10 node chain topology.</td>
<td>135</td>
</tr>
<tr>
<td>8.10</td>
<td>Simulation results for 30 node tree of degree 3.</td>
<td>136</td>
</tr>
<tr>
<td>8.11</td>
<td>Simulation results for 30 node tree of degree 5.</td>
<td>137</td>
</tr>
<tr>
<td>8.12</td>
<td>Simulation results for 100 node tree of degree 8.</td>
<td>138</td>
</tr>
<tr>
<td>8.13</td>
<td>Simulation results for 100 node tree of degree 8 and varying job durations.</td>
<td>139</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Covering all areas from smartphones, portable music players, cooking robots to Internet core-infrastructure, certificate registrars and nuclear power plants software continues to increase its infiltration in all aspects of our modern societies. Software has become such a fundamental day-to-day presence that on the one hand the software engineering field appears largely unaffected by the current global recession. On the other hand software is so widespread that governments have finally become aware of the opportunities it creates but also of the potentially dangerous implications it brings.

Dangers posed by software typically trace back to its poor quality which in turn stems from the inadequate application of software standards and the intrinsic difficulty of building high quality software. At the very core of the problem lies a shortage of good software engineers in the marketplace, caused both by a general lack of engineers and an abundance of insufficiently educated ones.

It is primarily the abundance of insufficiency. Many software engineering vacancies state “a minimum of 5 years enterprise-level experience with technology X is a pre”.

One can replace X with any industry-grade technology and still face a shortage of experts. Industry players continue to succeed in recruiting university students long before they graduate: unbound riches and better and quicker training, they promise. Ideally universities should produce such great engineers that businesses on the one hand line up to be themselves recruited and on the other hand do not dare to interfere with the educational process from fear of damaging it and thus their assets.

But engineers become good engineers through lots and lots of deliberate practice. Malcolm Gladwell [27] claims that 10,000 hours of practice seems to be the magic number for producing brilliant experts, citing such gurus as Bill Gates of Microsoft, Bill Joy of Sun Microsystems, Steve Jobs of Apple, Eric Schmidt of Google and The Beatles. They all had two things in common: (1) a favourable context for quickly becoming domain experts and (2) they all fought their way to privileged unlimited access to practice material.

There are three ironical discrepancies between Gladwell’s experts and today’s software engineers. Firstly, 10,000 hours is likely insufficient for today’s engineers that do not benefit from the software greenfield of the 70s. Secondly, in today’s dependence on software engineers the industry cannot idly wait for the one in a million student who deliberately works his way to becoming a brilliant engineer. And finally, today’s stu-
1. INTRODUCTION

dents have less access to practice materials relative to the complexity of the software domain as compared to the 70s.

This thesis reports on our engineering attempt to open up the field of software engineering education for massive availability of practice materials; to facilitate a paradigm shift from:

engineers become good engineers through practice

to

students become good engineers through practice

In fact students do not necessarily suffer from insufficient availability of practice material. Instead they suffer from the inability of their universities to provide this material and to train and evaluate their skills at solving engineering tasks. This is why the industry promise of quicker and better training is probably true: they have a seemingly endless supply of practice material and large teams provide immediate feedback and a continuous learning context.

This is currently unattainable in universities because grading of and giving feedback to students’ solutions already requires many man-hours. Significantly increasing the amount of practice material leads to implausible grading workloads. Even a modest semester-class of 100 students and two tiny weekly assignments generates a workload of at least 30 man-hours per week for grading alone and does not even get close to providing sufficient training.

This thesis proposes that part of the solution is to provide a language-generic platform that automatically verifies, executes and evaluates students’ solutions. In support of this proposition this thesis reports on the design and development of such a platform - called LabBack. Teachers and students access LabBack’s services indirectly through classroom management applications. In this thesis WebLab is used as LabBack’s client. Educators need only focus on creating assignments and the corresponding evaluation criteria and the framework takes care of grading 1,000 solutions with same ease as grading only one. Designing this framework is not a trivial task to undertake. Such a framework has to be openminded enough to support heterogenous programming languages and at the same time obtuse enough to be secure against malicious or generally badly written software. By employing this framework educators can publish an unlimited amount of practice material and give digital exams knowing that grading is objective and free.

This thesis presents the architecture of LabBack, a framework as-a-service that can objectively, securely and robustly execute and evaluate guest code written in a variety of languages. LabBack is designed to be scalable, dynamically extendable and usable from a variety of web-based and desktop-based clients. LabBack’s architecture is validated by means of two implementation iterations.

The first iteration - LabBack 1.0 is validated in conjunction with WebLab as its client in a real-life case study: a semester-long programming course in Scala, JavaScript and C at the Technical University of Delft. WebLab and LabBack 1.0 were used as the sole mechanisms for development, submission and automatic evaluation of students’ solutions.

http://eelcovisser.org/post/213/
The second iteration - LabBack 2.0 - builds on the successful security and scheduling models of the prototype to produce a RESTful web application that can be scaled to multiple cloud-based servers and is dynamically extendable with language-specific and experiment-specific support by means of plugins. The second iteration is validated by re-execution of the dataset generated in the case study and by simulation.

This thesis addresses the previously described problem of insufficient quantity of practice material. Given that LabBack currently only evaluates programs by means of unit tests, there remains a need to investigate more declarative testing avenues to capture criteria beyond functional correctness of programs. A major challenge and goal in the design of LabBack is to ensure its flexibility for use in new types of experiments, thereby seeding the greenfield for further investigations into pedagogical improvements. For now LabBack can objectively evaluate programs automatically using handwritten unit tests and is extendable and reusable, which is claimed to be a significant improvement over existing automatic grading systems.

1.1 Research Questions

The explorations in this thesis are driven by a fundamental engineering challenge:

| Research Question A: | Is it possible to design and build a software system that uses mainstream technologies to host automatic execution and evaluation of guest programs in an immediate, flexible, scalable, secure and robust way? |

This question encompasses the entire LabBack challenge and raises a number of subquestions. Firstly, receiving third party programs and running them on behalf of the students is a form of code injection. In contrast to the industry-wide effort to prevent code injection attacks LabBack intentionally makes itself available to code injections. The problem that arises is securing the host system from malicious software. The potential effect of security attacks is not limited to availability attacks but reaches as far as information theft, plagiarism and fraud. This problem raises the second research question:

| Research Question B: | How can a system both execute untrusted programs and be secure against them without severely restricting their functionality? |

Secondly, a factor to LabBack’s usability hinges on its ability to provide timely execution feedback to its users. This is necessary to ensure interactivity of the programming environment that the students use. Simply trying to execute all arriving programs at once will lead to servers quickly becoming overloaded. Additionally badly written programs may starve the system of resources and never terminate. A solution to these difficulties lies in scheduling. This gives rise to a third research question:
1. INTRODUCTION

**Research Question C:** How can a system schedule execution of guest programs to ensure their timely execution while ensuring its robustness against resource-starving programs?

A third factor to LabBack’s success hinges on its ability to execute programs written in a wide range of programming languages and to perform various experiments with them. Since LabBack deployments may achieve large scales it is unfeasible to hardwire language and experiment support into the application. Instead LabBack needs to be dynamically extendable with new features and maintain a loose coupling between itself and the type of programs it can execute. This raises a fourth research question:

**Research Question D:** What is a lightweight architecture that supports the runtime evolution of language and experiment-specific functionality in an application that executes third-party software?

1.2 Contributions

The main contribution of this thesis is providing a solution to the problem of scalability in programming education through the design of a framework - LabBack - that automatically compiles, executes and tests untrusted programs and its validation through two implementations. The main features of the resulting framework are:

1. Hosts the compilation, execution and testing of untrusted computer programs
2. Does not require clients to possess any language-specific functionality
3. Provides reports of program compilation, execution and test success to its clients
4. Is secure against malfunctioning or ill intended programs
5. Supports runtime evolution of language-specific and experiment-specific support by means of plugins, by providing its own secure and lightweight plugin framework
6. Schedules program executions to promote timely feedback and to remain robust against malfunctioning guest programs
7. Is web-based and provides RESTful services to its clients
8. Can be replicated on multiple instances and can be deployed on cloud-based servers
9. It is built with mainstream technologies (Java) and has lightweight platform requirements
10. Ships with a deployment simulator to assist deployment engineers in designing multi-instance topologies

In addition to contributing this platform this thesis makes a set of contributions to the more general field of software engineering:
1.3 Outline of this Thesis

This thesis is organised as follows. The context of this research is described in Chapter 2 where the evaluation case study and WebLab are presented. The requirements that must be met by a reusable and extendible platform for automatic grading are discussed in Chapter 3, followed, in Chapter 4, by a search for existing technologies that could be reused to satisfy the non-functional requirements of security, robustness and scalability, among others. This sets the scene for the description and evaluation of a proof of concept product - LabBack 1.0 - in Chapter 5. Chapter 6 and Chapter 7 describe the architecture and provide implementation and integration details for the second iteration - LabBack 2.0 - which builds on LabBack 1.0’s proven secure and concurrent execution model and Chapter 8 evaluates the correctness and scalability of the resulting platform. This thesis is concluded with an overview of related work in Chapter 9 and a conclusion and discussion of future work in Chapter 10.

---

1 Without requiring modification of the Java Virtual Machine
Chapter 2

Background

Many online learning platforms and even some online programming education platforms exist to date. None allows automatic execution and evaluation of unrestricted programs and above all none is available as a service or product.

This chapter contributes an overview of existing online platforms for programming education and the description of a case study that provides motivation and proving grounds for the development of LabBack. The case study presents the use of WebLab + LabBack in a real course at Delft University of Technology. A separate section of this chapter outlines WebLab’s functionality and its use of LabBack for automatic grading.

2.1 Massive Open Online Course Platforms

A generally agreed upon problem is that there is a world-wide shortage of computer scientists and highly-skilled engineers, a problem likely to continue well into the current decade [7]. In part this problem is caused by the relatively unpopular nature of computer science education and in part by the shortcomings of the computer science education itself.

Quality versus Quantity Tradeoff

Increasing the number of students gives rise to a new problem: computer science education does not scale well because manual grading of lab work and exams does not scale at all. This problem is in fact already present and manifests itself as absolutely minimal amounts of practical work for students and few and sometimes subjectively graded exams. Teachers are not to be blamed however. There appears to be a compulsory tradeoff between quality of education and its accessibility to the masses; a problem that cannot be solved by continuously increasing teacher ranks.

MOOC for programming education

A significant development in the direction of mass education appeared only recently, under massive open online course (MOOC) name. MOOC platforms (MOOC-ware) offer freely available online courses from reputable universities. Although MOOC platforms existed since 2008 [48] their application to programming education is only a
very recent development. MOOC programming courses have weekly practical assignments which students solve without leaving their web browsers and their submissions are automatically evaluated for functional correctness. Automated grading relieves the educators from this tedious task allowing them to focus on better education.

MOOC-ware overview

One of the first successful companies to offer such a platform is Codecademy\(^1\), funded in 2011. Codecademy offers JavaScript courses. Students program directly in the browser and their code is automatically executed and evaluated. By September 2011 Codecademy had over 550,000 users having completed more than 6 million exercises\(^2\).

Funded in 2012 by a former Carnegie Mellon University professor\(^3\) Udacity offers a more formal course model. Courses consist of video lectures, quizzes and homework assignments. All homework assignments are programmed using the Python programming language and are automatically graded.

A much larger effort is directed by Coursera\(^4\), a private education company funded in 2012, offering online courses provided by 16 universities across 5 countries. Courses involve video lectures, quizzes and homework assignments. Different courses have different assessments methods, some automatic and some manual but students typically write programs offline and are required to install software locally on their computers.

Khan Academy\(^5\) is another notable example of MOOC-ware. Funded in 2006, Khan Academy offers free micro-lectures in various fields. Its computer science course portfolio includes a number of very basic computer science and programming lectures. Courses are structured on a self-study principle without schedules and without assessments. Programming assignments and examples use a tiny subset of JavaScript and are neither manually nor automatically evaluated.

Khan Academy’s integration of audio lectures and programming playground is unique across MOOCs. Lectures take the form of interactive programming tutorials with an audio track and program code appearing in an editor. The audio track can be paused at any point and the code present at stage in the editor can be edited and executed. The lecture can then be resumed and changes to the editor are overwritten.

Shortcomings

A pattern can be detected in the above examples: MOOC-ware that supports online programming only supports a limited set of languages (JavaScript or Python) and MOOC-ware that does not support any online programming does not have any unified automatic solution grading. A plausible reason is the lack of frameworks to allow safe evaluation of student code written in various languages. A definitive answer cannot be given since all the mentioned platforms are proprietary and no open platform could be found.

---

Secondary to their inflexibility the MOOC-ware platforms are not available for reuse, neither as services nor as products. Codecademy allows teachers to host courses, but they are restricted to tutorial-style JavaScript courses, must be publicly available, exam environments are not supported and there is no classroom management. A teacher wanting to teach programming courses with online programming and automatic solution evaluation is offered zero choices.

2.2 Case study: COPL

In the second semester of 2012 Dr. Eelco Visser taught the Concepts of Programming Languages (COPL) course at Delft University of Technology for the second time. The COPL course is a semester-long first year Bachelor of Science course with the goal of teaching students fundamental programming idioms such as functional programming, memory management, pointers and prototype inheritance using a variety of programming languages (Scala, C and JavaScript). WebLab + LabBack began from the need for supporting tools for programming education.

2.2.1 Motivation

In its 2011 edition the lab consisted of 6 programming assignments using the same three languages. The lab assignments had been simple and small, and in the educators’ opinion only barely verified students’ knowledge. At the same time there was a consensus that the assignments had been too difficult for the little amount of practice that students had. In other words the students were insufficiently prepared to tackle practical applications of the concepts they had learnt.

The need to drastically increase the number of assignments available to students in 2012 was clear. At the same time it was predicted that more than 150 students would take the course making grading very difficult even with just 6 assignments.

Example 2.1

A modest semester-long course with 100 students and two weekly exercises generates 3200 submissions that require grading, excluding exams. At a minimum cost of 10 minutes per submission grading requires more than 30 man-hours per week. Doubling the number of weekly exercises to 4 requires 60 hours for grading, i.e. two almost full-time employees just for grading alone.

The only solution to both teach the larger number of students and improve the quality of the course was to have fully automated grading for submission correctness. None of the MOOC-ware platforms discussed could meet the requirements for three programming languages, automatic grading, classroom management and exam environments. Above all, none of the MOOC-ware is available as a service or as product and can hence not be reused. The solution was to investigate, design and build a system to meet these criteria.
2. BACKGROUND

2.2.2 Case study description

To support such a large amount of assignments and students Dr. Eelco Visser designed and built an web-based application to support programming labs - WebLab - where students can program directly in the browser. This allows teachers to create and publish assignments, manage classrooms and allows students to program directly in the web browser.

The code that students submitted needed to be automatically graded for functional correctness. For this purpose a prototype of a secure and robust program execution engine - LabBack - was designed and built. WebLab passes the students’ source code together with specification unit tests to LabBack which compiles and runs the program and returns tests reports. WebLab + LabBack therefore allow students to independently submit solutions and have them automatically and immediately evaluated.

2.2.3 Case study results

The COPL course used WebLab + LabBack for all lab assignments and three actual programming exams. In total students answered 45 multiple choice questions and solved 275 programming assignments. In total 49,415 submissions to programming assignments have been automatically graded. Without automatic and immediate grading these submissions require at least 20 weeks of full time manual grading.

Students were given an unprecedented amount of practice material, yet at the end of the course the consensus was that the material was still insufficient to allow students to fully master the targeted skills.

WebLab together with the proof of concept LabBack performed well in the case study. Shortcomings of the prototype and a desire for more flexibility have motivated the design of a second iteration of LabBack, which is the focus of this thesis.

2.3 WebLab

WebLab\footnote{WebLab is the work of Dr. Eelco Visser. The author of this thesis has contributed integration with LabBack’s services} aims to resolve the lack of a reusable platform for online programming courses and their exams by generically supporting programming labs using a variety of languages in an all-in-one web-based solution. WebLab provides a unified front-end to assignments, exams, grading and course administration. LabBack - the subject of this thesis - is responsible for automatically compiling, executing and testing students’ solutions. All compilations and executions are performed on LabBack-side as opposed to client-side (WebLab or student-side).

This section first describes WebLab from a user’s perspective and subsequently gives some implementation highlights.

2.3.1 Management of Courses and Assignments

WebLab provides a unified user interface for course and assignment management. Instructors can create courses each with multiple editions allowing already existent ma-
2.3. WebLab

Figure 2.1: Course edition dashboard in WebLab as seen by an instructor.

terial to be remixed. Figure 2.1 gives an impression of the course edition dashboard as seen by an instructor.

Course contents

Each course edition in WebLab consists of assignments and collections of assignments. For example, the COPL course pictured in Figure 2.1 has three collections of assignments: tutorials, graded assignments and exams. Each of these collections contains other collections or individual assignments.

WebLab’s individual assignments can be of one of three types: programming assignments, multiple choice questions with an arbitrary number of choices or open answer questions. Teacher flexibility is supported by not making assignments visible to students automatically and not requiring that assignments are configured. In fact the instructor is free to create an unrestricted amount of assignments stubs and only publish them when ready. This allows the instructor to develop parts of the curriculum in advance while always being able to return to current matters.

Each assignment can set its own soft and hard deadlines or inherit these from its parent collection. Student submissions are permitted past the soft deadline but only before the hard deadline and a lateness penalty is automatically calculated and applied based on predetermined criteria. WebLab educators can manually override grades, deadlines and penalties on a per student basis.

WebLab is designed to work in support of physical lectures, as such publishing of lecture materials such as slides and recordings falls outside of its scope.

Course monitoring and statistics

For an instructor, being able to monitor courses’ progress and students’ performance should be as important as creating assignments and having students’ submissions automatically graded. WebLab allows instructors to monitor progress in near real-time.
Monitoring is possible ranging from the entire class to an individual student and from the entire course-load to individual assignments.

As an example, Figure 2.1 illustrates the information page of the COPL course showing per-course statistics and per-student results. A similar information and statistics page is associated with every assignment, assignment collection and course edition.

Course-wide student performance is normally a factor of the quality of the course and its suitability to the audience. Being able to take a classroom-wide pulse is paramount for continuous improvement of the education process. Educators often develop intuition for student performance but these are typically difficult to demonstrate in part due to a lack of software tools. In WebLab instructors can add correlation plots to information pages to assist in the diagnosis of students performance. For example Figure 2.3 shows students’ lab assignment grades plotted against their performance in exams. WebLab automatically updates statistics and correlation plots to show latest situation as courses evolve. Instructors can even monitor near-live progress of a lab session or exam.

Access control

A WebLab course may have more staff besides the course manager. Access to course management interfaces is controlled through roles assigned to users allowing course administrators to efficiently control the access of their staff. Roles can be attributed for each course edition, assignment collection or individual assignment per user. This allows complete access control of variable granularity.
2.3. WebLab

Figure 2.3: WebLab course information page showing correlation plots.

Figure 2.4: WebLab page for editing a programming question.
2. BACKGROUND

```javascript
test("Adding numbers works", function() {
  expect(3);
  ok(newAddition, "function exists");
  equal(4, newAddition(2, 2), "2 + 2 = 4");
  equal(100, newAddition(100, 0), "zero is zero");
});
```

Figure 2.5: Specification test template for Javascript question

Creation and editing of programming assignments

Figure 2.4 shows a typical WebLab page for editing a programming question. A programming question consists of the aggregation of eight attributes:

- Title
- Question
- Language
- Solution template
- Test templates
- Library code
- Reference solution
- Specification tests

The title and question are self-explanatory. It is noteworthy that the lightweight Markdown [32] markup language can be used to format the description of the question.

A programming assignment is written in a single language which must be indicated in the question’s attributes. Three languages are currently supported: Scala, C and JavaScript. Based on the language attribute WebLab automatically provides an editor with syntax highlighting and stubs for the solution and test templates, library code, solution and test code. For example, the code stub from Figure 2.5 is provided as a basis for the specification unit tests for a JavaScript assignment. Prior to the second LabBack iteration and associated modification of WebLab the set of supported languages and their associated code templates were hardcoded into WebLab.

Each programming assignment contains a reference implementation - the solution - and specification unit tests. These or details thereof (compilation or execution messages) are never visible to students. The specification unit tests define the criteria by which (part of) a student’s submission is automatically graded. The reference solution is used to verify the correctness of the specification tests and to serve as reference base point. If the students’ solution is expected to require certain shared code this can be provided in the library code assignment attribute. This code can be used but never seen by the students.

2.3.2 Question answering

WebLab allows students to master programming and concepts in various languages without requiring them to ever leave the web browser. Figure 2.6 shows a submission view as seen by the student when answering a programming assignment. The student
is shown a description of the assignment and two editors: one for editing the solution code and the other for writing custom unit tests.

A student resolves an assignment by implementing his solution directly in the browser using an editor with syntax highlighting for the language used. To test the behaviour of their solutions students can write additional unit tests than provided in the test editor. Student's solutions may make use of instructor-provided but invisible library code.

At any time a student may request a compilation of the code in the editors by pressing the *Save & Compile* button. This triggers WebLab to request the compilation of the program by LabBack. Compilation messages are presented below the editor area.

If the compilation is successful the student may request the execution of the self-defined unit tests or the specification unit tests by pressing the *Run Your Test* or *Run Specification Test* buttons, respectively. The output produced during execution and the test report are presented below the editor area. When executing the specification tests no output other than the test success ratio is shown. This measure is taken to prevent accidentally revealing information about the specification tests.

When finished but only if the code compiles the student may finalise the submission by pressing the *Submit* button. Students may resubmit solutions until the hard deadline has passed. After the soft deadline has passed WebLab will automatically apply lateness penalties. Only the final submission is used in the grade calculation.

At any time a student can monitor his progress within the current assignment collection by means of a navigable progress bar as shown at the top-right corner of Figure 2.6.

The correctness of a student solution and therefore its associated grade is determined as the ratio between the succeeding specification unit tests and the total number
of test in the specifications. A solution that works only partially (e.g. incorrect for some corner cases) receives a subset of the total points. The ratio of passed to total unit tests is shown to students almost immediately after clicking the Run Specification Test.

One drawback of handwritten programming exams is that the code written is rarely correct, i.e. it would rarely be accepted by a compiler. Handwritten exams therefore do not manage to verify a student’s skill in the respective programming language. Strictly contrasting with handwritten exams where a student’s solution may obtain full points even if in reality it would not even pass the compilation checks, WebLab does not allow students to submit code that does not compile. In other words successful compilation is the minimum requirement for a submission.

### 2.3.3 Grading

WebLab’s goal is to make grading automatic and objective, however automatic grading by means of unit tests can only check the functional correctness of submissions. Assignments may need to be graded on non-functional criteria and WebLab supports this by giving teachers the possibility of defining additional checklists for grading. These checklist weigh a certain ratio of the total assignments points. If checklists are defined, graders are required to fill them in for every submission they grade. Since evaluation of programming style by visual inspection is not time consuming a large volume of assignments can still be graded quickly. Checklist grading can be enabled on a per-question basis as seen in Figure 2.7. Figure 2.8 shows an example view for an instructor during grading.

There is unfortunately always a certain amount of instructor bias when grading assignments, voluntary or otherwise. It may occur for example when consecutively...
grading multiple submissions by the same student leading to subjective grading.

To alleviate the risk of bias WebLab randomly assigns submissions to graders. This allows graders to evaluate submissions anonymously and adopt a per-question workflow instead of per-student. To detect criteria inconsistencies between graders a certain percentage of cross-checks can be defined as shown in Figure 2.7 as the Double check percentage. A randomly selected set of submissions is then automatically assigned to two different graders and the discrepancies between the two evaluations are measured.

### 2.3.4 Exams

WebLab allows actual programming exams to be taken. Exams are normal assignment collections with a few extra security measures.

Firstly, exams can only be accessed by registered students after entering a personal One Time Password (OTP) in addition to the normal credentials. An OTP is generated for every student and is valid for a single attempt on a single exam. In addition to the single use aspect the key can also be set to expire at a particular time. For example one may require that students begin the exam within 30 minutes of the official starting time by settings the keys to expire at that time. These security measures prevent malicious students from committing fraud by having third-parties take the exam on their behalf, or by otherwise abusing the system.

Secondly, while a student is answering an exam, no other parts of WebLab are accessible to that student unless he intentionally exits the exam environment. Reentering the exam is then no longer possible because the OTP has already been used.
In addition to the security features built in WebLab students’ access to third-party websites must be restricted. This can be achieved with special locked-down versions of operating systems and browsers. Since WebLab is a web application it places no constraints on the choice of operating system, only requiring a modern web browser.

### 2.3.5 Implementation Highlights

This section briefly introduces various technologies remixed in WebLab.

#### System Level

WebLab is a web application written using the WebDSL [73] language, a domain-specific language for developing dynamic web applications with rich data models. Applications written in WebDSL benefit from automatic persistence for data entities, template-based weaving of pages and weaving of different aspects such as access control.

Figure 2.9 illustrates WebLab’s operating context. WebLab interacts with a separate application - LabBack - for the compilation and execution of code. Besides the fundamental difference in concerns and architecture the separation between WebLab and LabBack is in place to ensure that even in the event of a security breach through execution of students’ code the attack cannot be escalated to the entire WebLab application. The problem of security and scalability of guest code execution is non-trivial and is a concern of LabBack - the primary focus of this thesis.

The WebLab application runs in an application container that manages its lifecycle and provides an abstraction over client requests protocols - Apache Tomcat\(^1\) in this case. Data persistence is provided by the MySQL\(^2\) database engine.

---

2.4. Summary

User interface

The WebLab user interface is built using the Bootstrap framework developed at Twitter. Building user interfaces with Bootstrap simplifies the development of rich and responsive web applications that retain compatibility with a multitude of platforms.

Rich programming editor

Providing an in-browser programming editor is key to WebLab’s usability. WebLab embeds the Ace Editor developed at Cloud9 IDE for this purpose. Ace is written in JavaScript, runs entirely on client side and provides students and instructors with editor features such as syntax highlighting, automatic indentation and supports marking errors and warnings directly in the code body.

Statistical plots

Correlation plots in WebLab are built using the Flot JavaScript library. Flot uses of jQuery, CSS and HTML5 to display visually appealing and clear graphs which can be readily constructed from WebDSL application code.

Test frameworks

WebLab + LabBack utilise unit testing frameworks to simplify and automate evaluation of functional correctness of students’ solutions. Three languages are currently support: Scala, C and JavaScript. For code written in Scala the unit tests are specified using the ScalaTest framework. Figure 2.10 shows an example of such a unit test. For testing code written in C and JavaScript the CuTest and QUnit testing frameworks are used, respectively.

Authentication

WebLab maintains its own user and privilege store and has its own authentication mechanism. In the current production environment authentication of users is however offloaded to the Single Sign On (SSO) environment of Delft University of Technology. This ensures that any user in possession of valid university-wide credentials can use WebLab and at the same time reduces the security hazard of maintaining a separate credential store. The information required to personally identify students (such as student identification number) is provided by the SSO environment in a client-side cookie thereby defining the relation between a WebLab user and a student.

2.4 Summary

This chapter has presented a summary of the latest developments in a new trend of programming education - Massive Open Online Course-ware. MOOC-ware is typically
2. Background

```scala
import org.scalatest.Suite
class StackonTest extends Suite {
  def interp(instr: Instruction*) = 
    StackonInterpreter(instr.toList).interpret()

  def testLabelStore {
    val program = List(Push(1), Push(2), Push(99), Label("here"), Push(5))
    val interpreter = StackonInterpreter(program)
    expect(3) {
      interpreter.labelStore()
      interpreter.memory("here")
    }
  }
}
```

Figure 2.10: Example unit tests using the Scalatest framework.

very good at providing varying degrees of free programming education to masses of students. Unfortunately they all have two serious shortcomings:

1. They are not reusable by third parties, neither as a service nor as a product
2. They do not support web-based programming and automatic grading in multiple languages

Driven by a real need for high quality and scalable education WebLab + LabBack have been developed. WebLab is a web application allowing teachers to manage programming labs, classrooms and take exams. It allows students to program directly in the web browser and have their solutions automatically and immediately graded by LabBack which compiles, executes and tests their solutions.

WebLab + LabBack were evaluated during the case study of a real semester-long programming course at Delft University of Technology in 2012. WebLab + LabBack allowed a very large amount of submissions to be managed and automatically graded which in turn allowed educators to make a large number of programming assignments available. In total WebLab + LabBack is estimated to have performed a grading equivalent of 20 weeks of full-time work.
Chapter 3

Analysis of LabBack Requirements

This chapter presents a dissection of the functional (Section 3.1) and non-functional requirements (Section 3.2) that must be met by LabBack. These requirements directly drive LabBack’s design and implementation and are used as a constant reference point.

The primary stakeholders in LabBack are its clients to whom LabBack’s purpose is to perform automatic compilation, execution and evaluation of students’ programs. This primary purpose and its application domain define a set of functional and non-functional requirements. An overview of these requirements is outlined in Figure 3.1.

3.1 Functional Requirements

LabBack is a single-purposed application: it concurrently and automatically compiles, tests and reports on multiple student-written programs, which are referred to as jobs. Its client is specifically WebLab but could be any application requiring remote evaluation of programs. Clients do not manipulate the programs themselves outsourcing instead the entire process to LabBack.

3.1.1 Execute programs

LabBack must process job execution requests from its clients. Arriving jobs contain a student’s solution code for a particular assignment and a set of unit tests to measure the correctness of the solution. LabBack has to take the necessary steps to evaluate the runtime correctness of the program with respect to the unit test. LabBack must initially support the Scala, JavaScript and C programming languages and be extendible with further languages.

3.1.2 Record and report results

LabBack must record the output of and report on the results of the job evaluation. Jobs have either a successful outcome or a failure outcome. A failure outcome typically corresponds to a program’s failure to compile. A successful outcome corresponds to a program that compiles successfully and whose runtime evaluation was successfully started.
## 3. Analysis of LabBack Requirements

<table>
<thead>
<tr>
<th>Category</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional</strong></td>
<td>Compile and test programs</td>
</tr>
<tr>
<td></td>
<td>Record and report test results and program output</td>
</tr>
<tr>
<td></td>
<td>Support Scala, C, JavaScript languages</td>
</tr>
<tr>
<td></td>
<td>Allow experiment-specific processing</td>
</tr>
<tr>
<td></td>
<td>Network-able communication protocols</td>
</tr>
<tr>
<td></td>
<td>Asynchronous client-server communication</td>
</tr>
<tr>
<td></td>
<td>First-come, first-served scheduling of jobs</td>
</tr>
<tr>
<td></td>
<td>Web based</td>
</tr>
<tr>
<td></td>
<td>UI for monitoring job queue</td>
</tr>
<tr>
<td></td>
<td>UI for configuring language support</td>
</tr>
<tr>
<td><strong>Security and Robustness</strong></td>
<td>Authenticated communication</td>
</tr>
<tr>
<td></td>
<td>Defend against malicious guest code</td>
</tr>
<tr>
<td></td>
<td>Protect against availability attacks</td>
</tr>
<tr>
<td></td>
<td>Protect against information theft</td>
</tr>
<tr>
<td></td>
<td>Protect against fraud</td>
</tr>
<tr>
<td><strong>Deployment</strong></td>
<td>Platform independent</td>
</tr>
<tr>
<td></td>
<td>Light-weight network requirements</td>
</tr>
<tr>
<td></td>
<td>Short-lived connections</td>
</tr>
<tr>
<td></td>
<td>Common standards for communication protocols</td>
</tr>
<tr>
<td></td>
<td>Deployable to cloud-based machines</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
<td>Scala, C, JavaScript support</td>
</tr>
<tr>
<td></td>
<td>Extendable with new languages</td>
</tr>
<tr>
<td></td>
<td>Support other experiment setups (e.g. code coverage)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Support various job-timing characteristics</td>
</tr>
<tr>
<td></td>
<td>Minimal overhead to language tooling</td>
</tr>
<tr>
<td></td>
<td>Attempt/permit performance tuning of language tooling</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Resource-efficient job processing</td>
</tr>
<tr>
<td></td>
<td>Job distribution among multiple instances</td>
</tr>
<tr>
<td></td>
<td>Load balancing mechanisms</td>
</tr>
</tbody>
</table>

Figure 3.1: Summarised classification of LabBack requirements.
3.1. Functional Requirements

**Successful execution**  In the successful completion case LabBack must report two types of information. Firstly, the report contains the ratio of passed to total number of unit tests in the job. The report also contains the test names and failure messages for each of the failed test cases. Secondly, LabBack must capture and record the output produced by the tested program. The program output is invaluable to a student’s successful debugging of the solution. This recorded output must be included in the job report.

**Failed execution**  In the failure case LabBack must report the cause of failure. This typically arises from a program’s failure to pass checks performed by the compiler. The output produced during compilation is essential to debugging of the code. LabBack must capture and record the compilation errors and warnings that occurred and include them in the job report.

Modern compilers report structured (error and warning) messages which can be directly used for error marking in the programming editor. This increases usability of the programming environment - in this case LabBack’s client - by allowing problematic code fragments and a description of the problem to be highlighted directly. Classic compilers typically provide unstructured errors which cannot easily be integrated in the program editor. LabBack must support both reporting modes and preferably report structured errors where possible.

**Flexible reporting**  In addition to contents of the standard report, LabBack must be able to provide additional reports depending on the type of experiment performed. LabBack must therefore be non-restrictive in its reporting functionality.

**Example 3.1**
For example, to measure the quality and correctness of unit tests one needs to both run the tests and measure their coverage of the program tested. Reports of coverage analyses are intrinsically different from those of unit test reports.

3.1.3  Scheduling of jobs

LabBack must be able to execute multiple jobs concurrently. Attempting to execute too many jobs at once leads to overloading of the server. To avoid this LabBack must actively schedule jobs on the available processing capacity. Job scheduling must ensure maximal utilisation of the processing capacity and prevent overloading of the server. Clients expect that LabBack executes jobs fairly in a first-come, first-served order.

3.1.4  Dynamic execution process

LabBack must support runtime evaluations in programs written in different programming languages. Since universal compilers and universal runtime environments do not exist LabBack must use language-specific tooling for each language. Each language requires not only the use of its own tooling but also of particular compilation, execution and perhaps instrumentation sequences. LabBack must therefore retain the flexibility
of executing jobs in different languages, each with its own specific preparation process. Universality of the execution pipeline\(^\text{1}\) is also required by experiments other than unit tests.

**Example 3.2**

For example, programs written in Java require compilation prior to execution, whereas programs written in JavaScript can be directly interpreted. Other languages require multiple compilation steps. Domain Specific Languages (DSLs)\(^\text{70}\) often fall in this category where the DSL code is first compiled to a general purpose language then compiled to machine code and only thereafter executed.

It is therefore a LabBack requirement that job execution pipelines are dynamically determined based on each job’s characteristics such as program language and type of evaluation required.

### 3.1.5 Networked services

Due to various differences in non-functional requirements (performance, scalability, timings) of LabBack and its client, LabBack must not rely on its deployment on the same physical machine as or in geographic proximity to its clients. This gives rise to the functional requirement that LabBack must provide its services to its clients through a networked communication protocol. This protocol must handle efficient transport of job and report requests between LabBack and its clients.

**Asynchronous request handling** LabBack clients require their connections to LabBack’s services to be short lived and lightweight. A client first places a request for evaluation of a program (job) and then polls LabBack for the job’s status updates until it determines that the job was completed. Finally the client requests the results of the job. The client uses a separate connection for each request and all connections consists of the initial client request and LabBack’s confirmation. This mode of communication determines the functional requirement that LabBack’s services must communicate asynchronously.

### 3.1.6 User interfaces

Because LabBack needs to provide its service to clients which are web applications LabBack should be a web application as well. Its services must be accessible through web protocols and LabBack must provide user interface for monitoring and configuration.

**Configuration interfaces**

Support for a variety of languages implies the requirement of configuration for this support. A management interface must be available where authorised users can add and configure language and experiment-specific LabBack support.

\(^\text{1}\)The term execution pipeline is used when referring to the entire chain of processing steps that are required by a job.
3.2 Non-functional Requirements

The functional requirements and the execution context of LabBack define the set of its non-functional requirements. LabBack’s meeting of these requirements is of high importance as they directly influence its functioning and its effectiveness.

This section contributes an analysis and discussion of the non-functional requirements that LabBack is subject to. Fundamental issues such as security and robustness against code injections as well as scalability are carefully dissected.

3.2.1 Security and Robustness requirements

Attacks against web services originating from a wide range of sources continue to be very prevalent. Most attacks abuse vulnerabilities introduced by either software engineers or system engineers [51]. A 2008 study [31] puts IT services hosted by the education industry at the top as the most vulnerable systems. This puts LabBack in the category of applications at very high risk of security attacks and breaches.

The cost of breaches to LabBack’s security varies depending on their eventual impact on the system and its context. For example an attack causing all instances’ services to become temporarily unavailable has a lesser financial and legal impact than an attack that succeeds in altering test results thereby defrauding the grading process and the university. LabBack therefore needs to protect itself and be protected against a wide variety of attacks.

**Attack classification**   An unprotected LabBack is vulnerable to attacks through two pathways. Firstly, *job external* attacks originate on the outside of LabBack’s ecosystem and exploit vulnerabilities in LabBack’s communication protocols and web interfaces. Secondly, *job internal* attacks are delivered directly through the code execution functionality of LabBack and aim at compromising LabBack by having it execute malicious code.

The following analysis addresses external and internal attacks separately. Attacks are also differentiated by their intended purpose: compromising service availability, stealing information or defrauding the educational process. A classification of attack pathways is shown in Figure 3.2.

**Job-external attack vectors**

Attacks external to the execution job use means of attack that are not part of the to-be-executed program but instead abuse communication protocols, user interfaces or authentication and authorisation mechanisms.

In their simplest form, external attacks come as unauthenticated or unauthorised access allowing attackers to access parts of the system and perform action without be-
3. Analysis of LabBack Requirements

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Pathway w.r.t. input program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>External</td>
</tr>
<tr>
<td>Availability</td>
<td></td>
</tr>
<tr>
<td>Denial-of-Service</td>
<td>Internal</td>
</tr>
<tr>
<td>Unauthorized access</td>
<td>Internal</td>
</tr>
<tr>
<td>Message replay</td>
<td>Internal</td>
</tr>
<tr>
<td>Infinte loop</td>
<td>Internal</td>
</tr>
<tr>
<td>Cross-site scripting</td>
<td>Internal</td>
</tr>
<tr>
<td>Fork bombs</td>
<td>Internal</td>
</tr>
<tr>
<td>CPU hogging</td>
<td>Internal</td>
</tr>
<tr>
<td>Memory starvation</td>
<td>Internal</td>
</tr>
<tr>
<td>Reflection</td>
<td>Internal</td>
</tr>
<tr>
<td>Information theft</td>
<td>SQL injections</td>
</tr>
<tr>
<td>Authorization/authentication</td>
<td>Internal</td>
</tr>
<tr>
<td>Eavesdropping</td>
<td>Internal</td>
</tr>
<tr>
<td>Reflection</td>
<td>Internal</td>
</tr>
<tr>
<td>Fraud</td>
<td>Job hijacking</td>
</tr>
<tr>
<td>Eavesdropping</td>
<td>Test faking</td>
</tr>
<tr>
<td>Man in the middle</td>
<td>Test faking</td>
</tr>
</tbody>
</table>

Figure 3.2: Classification of potential LabBack security attack vectors and their intended purposes.

...ing asked for credentials. The effect of these simple attacks is exacerbated if combined with privilege elevation attacks, if successful attackers obtain managerial privileges on the target system. In the case of LabBack such attacks can be used to negatively affect service availability by deactivating the compromised instance or for theft of sensitive information such as students’ solutions or specification test suites. To prevent this type of attacks LabBack must require authentication and authorisation of users on all management interfaces as well as authentication of service interfaces.

A class of attacks that affect both LabBack and its clients are session hijacking attacks. These attacks occur by stealing or intercepting an authorised user’s session cookie and using it to masquerade as those users. This method allows attackers to act on behalf of the compromised users and perform any tasks which that user is permitted. LabBack must either not use session cookies or use secure cookies. By encrypting all communication LabBack can effectively prevent interception of the cookie and eavesdropping on the client-server communications.

By intercepting and altering communications between LabBack and its clients or between different instances of LabBack attackers can perform man-in-the-middle attacks. These attacks aim at information theft (solutions or specification tests) and fraud (altering test results in-transit). To mitigate the risk of this type of attack LabBack needs to prove its identity to its clients by using, for example, SSL certificates. The client needs to prove its identity by means of either a valid session identifier or...
3.2. Non-functional Requirements

credentials, depending on the implementation.

The combination of SSL certificates, encrypted communication and secure session identifiers also protects LabBack against job hijacking which targets theft of job parameters and its results with the purpose of defrauding the evaluation process. This type of attack is a type of information theft eventually elevated to fraud.

To further prevent information theft and availability attacks LabBack must take appropriate measures to prevent against code injection attacks such as SQL injection as part of the job request parameters (excluding input program) or HTML input fields. Code injection attacks through abuse of web page or request fields can be avoided by employing appropriate security models \[36\] and careful sanitisation of input values.

**Denial-of-Service** (DoS) attacks target service availability and are performed by overloading servers with requests. In the context of LabBack an effective way to launch a DoS attack is to apply message replay tactics to job requests. Without any protection against message replays an attacker can capture and repeatedly resend a client’s job request causing LabBack to queue a new execution job for every such request. This attack is particularly trivial since the attacker does not need to actually decrypt the contents of the message. To reduce its risk to DoS and message replay attacks LabBack must recognise duplicate messages and reject them. Simple mechanisms for this are for example sequentially numbering client’s request or signing of client requests with one time keys.

**Job-internal attack vectors**

Attacks internal to the execution job request LabBack to execute malicious programs. LabBack’s execution of uncontrolled guest programs constitutes a code injection vulnerability. The main threat to LabBack comes from malicious code targeted at availability, information theft or fraud. Attacks via this pathways are typically perpetrated by students or other course participants.

**Example 3.3**

The security risk associated with code injections is tremendous and well understood in the industry. A notable example is the Apple App Store\[1\] whose review process rejects applications that interpret on-demand downloaded code or compile code on-demand. Another notable example is the intrinsic lack of safety of the JavaScript eval function \[12, 11\] in face of received data such as JSON objects \[15\]. The general advice in this last case is to never directly evaluate JSON objects.

Received code need not have a malicious intent to be harmful as well-intentioned code may still inadvertently cause information leaks or affect service availability. Resilience to well-intentioned but badly written code is a robustness concern rather than a security one. From the perspectives of stability and availability there is however no distinction between security and robustness.
3. Analysis of LabBack Requirements

Applicability of common mitigation techniques

The common approach taken by systems to mitigate the code injection risks is to completely prevent them from occurring. In LabBack’s case this is however not possible since execution of injected code is LabBack’s main purpose. Typical input sanitisation techniques are also unsuitable for LabBack as they are designed to render injected code syntactically and semantically incorrect thereby preventing its evaluation. Code review tactics employed in third-party software distribution centres are not applicable to LabBack because of the large volume of programs and the requirement for realtime program execution. Whereas the typical solution is the prevention code injections LabBack must instead control them.

Information security threats

Any database access from the guest code to the LabBack database can be used to read and modify job execution results. Similarly, memory access to data owned by other jobs, either directly or by means of reflective calls, can be used to commit fraud by altering job parameters or stealing job inputs or outputs. File I/O and socket I/O can be used to affect the service availability. Preventing file operations is insufficient if the guest program can execute available native processes on the host system and by privilege elevation gain access to other parts of the system. LabBack must prevent similar attack patterns by denying access to the database, forbidding calls to native processes and by preventing a job’s access to other jobs’ data.

Availability threats

Systems can become fully occupied or overloaded by programs stuck in infinite loops or simply CPU hogging programs. Whereas kernel schedulers of most host platforms (e.g.: Unix based) are sufficiently preemptive to allow control processes to rescue the system in case of CPU hogging programs, the same is not true for cases of memory starvation. Memory starvation causes the system to run out RAM and begin swapping to permanent storage or causing large amounts of I/O operations and eventually leading to an operating system crash. Systems rarely manage to recover from such situations in part due to the lack of interactive preemption in I/O schedulers. Even if recovery eventually occurs this comes at a high cost to availability: numerous processes are forcibly terminated and services are unavailable during the memory starvation phase, typically lasting tens of minutes. Ironically the more installed memory a system has the longer it takes for the operating system to recover

Figure 3.3 shows a simple example of intentional memory starvation of the JVM. Although the JVM contains the memory leak and the host system is unaffected the JVM will still crash with an OutOfMemoryError stopping all other programs running in that JVM.

---

1 This is due to typical symmetry between the amount of RAM and amount of swap space installed. Recovery only commences when the swap space is full, therefore the bigger the swap space the longer it takes for it to fill up and the later the recovery takes place.

2 Java Virtual Machine.
3.2. Non-functional Requirements

```java
import java.util.LinkedList;

public class MemStarve {
    public static void main(String[] args) {
        LinkedList<Long> l = new LinkedList<Long>();
        while (true) {
            l.add(42L);
        }
    }
}
```

Figure 3.3: Example of JVM memory starvation implemented in Java.

```c
#include <unistd.h>

int main() {
    while (1) {
        fork();
    }
    return 0;
}
```

Figure 3.4: Example fork-bomb in C.

```java
public class Bomb {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            public void run() {
                new Thread(this).start();
            }
        };
        new Thread(r).start();
    }
}
```

Figure 3.5: Example fork-bomb in Java.

To protect itself and its host system from resource exhaustion and to ensure timely execution of programs LabBack must prevent guest programs from occupying the CPU for too long and from using too much memory.

While operating systems may eventually recover from memory starvation requirements, there exist common attacks that operating systems cannot recover from. A notable example is the fork bomb [6] which spawns new threads with the sole purpose of having them spawn other threads. Optionally each spawned thread may also wait for its children threads to complete execution (join them). An example fork bomb in C is shown in Figure 3.4. Both the simplicity and effectiveness of the code in bringing down any Unix-type system without countermeasures are astonishing. A similar example in Java is shown in Figure 3.5. Although this does not have the effectiveness of the C example it is sufficient to crash the JVM that is running it.

Fork bombs cause an exponentially fast growing number of processes to be started and occupy both memory and CPU and exhaust the ability of CPU scheduler to maintain system interactivity. The loss of interactivity occurs as most CPU cycles are
wasted by the scheduler itself performing maintenance. Additionally the large number of running threads causes the pool of available PIDs to be exhausted. As a consequence no new process can be started which implies that for example the command `kill` cannot be used. Any PIDs becoming available are immediately consumed by the fork bomb again.

Being able to kill the entire process tree of the fork bomb is the only recovery solution but it is unlikely to succeed for two reasons: (1) it has to happen before the pool of PIDs becomes exhausted and (2) certain programming languages (such as C) allow the creation of new processes that are not part of the their creator’s process tree.

Countermeasures against fork bombs are typically implemented as a limit on the number of processes that a system user may own. Although this countermeasure prevents fork-bombs from crashing the entire system they do not prevent fork bombs from incapacitating the user account that started them.

LabBack must ensure that fork bombs cannot be launched from guest code. This can be either by completely denying process forking or by throttling or limiting the number of children processes of running guest programs.

### 3.2.2 Traceability of actions

An existential truth about software is that it can never have impenetrable security or be entirely bug-free. This is especially true for software at the beginning of its lifecycle or under rapid evolution, such as LabBack. Acknowledging that security breaches and crashes will happen software engineers must design systems such that their actions are retraceable. Traceability of the system significantly facilitates post-mortem examination and subsequent vulnerability patching.

In the case of LabBack the impact of security breaches and system misbehaviour can have potentially serious legal implications for the education institution. LabBack must produce tracing information by producing structured and fine grained system activity logs. LabBack must store these logs separate from the main data store to avoid concurrent corruption of both data store and diagnostic information.

### 3.2.3 Deterministic job outcomes

LabBack must ensure that job results are correct. Incorrect job results cause not only usability problems but can also give rise to financial, legal and public image costs for LabBack’s clients and their client alike.

For example, students taking part in an exam may contest their results if the grades obtained do not reflect their submissions. Resolution in such situations depends on the traceability of the system’s behaviour. Successful diagnostics depend on the system’s deterministic behaviour. In other words LabBack must produce identical results for identical jobs. Jobs must not be permitted to affect other jobs’ results.

### 3.2.4 Platform independence

All clients will be accessing LabBack’s services remotely. To increase compatibility with a variety of client platforms LabBack should be a web application, providing

---

1. Process IDentifier
3.2. Non-functional Requirements

web pages for the required monitoring and configuration features and providing its job scheduling services via common standardised transport protocols. The transport protocol should be readily accessible from mainstream programming languages.

In the future LabBack may be accessed directly from client applications on mobile platforms with varying network connectivity. The duration of each connection should be minimal therefore requiring asynchronous processing of requests.

To minimise restrictions on the development and deployment platforms LabBack has to be compatible with common operating systems such as Microsoft Windows and Unix-based systems (Linux, Apple OS X). LabBack must be equally well suited to run on domestic servers as well as on cloud-based servers.

3.2.5 Extendability

The goals of LabBack are both to support current experiments such as unit testing as well as serve as an experimentation platform for research into automatic evaluation and grading techniques. Although initially LabBack need only support the Scala, C and JavaScript programming languages addition of more programming languages is expected.

LabBack must therefore be readily extendable with new language and experiment support without requiring significant development efforts.

3.2.6 Performance

Performance of job execution is a LabBack concern that directly affects usability. LabBack must not add significant overhead to the language tooling it uses. At the same time LabBack should improve the performance of the used tooling by, for example, reducing initialisation times for recurring job types.

Different languages have different timing characteristics both during compilation and at runtime. Some compilers are very fast while others require tens of seconds even for small programs. Runtime timing characteristics are a factor of both the runtime environment initialisation and by the actual code or test to be executed. LabBack must therefore support jobs with varying execution durations.

3.2.7 Scalability

Large exams and imminent lab deadlines can cause peak job loads for LabBack. In the average scenario half of the exam population requests jobs at the same time. In the worst case scenario all students could place jobs nearly simultaneously. The job loads increase linearly with the number of concurrent exams. Additionally, an IDE-like development environment (WebLab included) amplifies the number of generated jobs since jobs may be submitted in quick succession with only minor source code modifications.

On the one hand LabBack needs to be sufficiently efficient to handle large amounts of jobs. This can be achieved by minimising its memory and CPU footprint and by careful balancing of aggressive memory management with performance. On the other hand the peak load is expected to surpass the capabilities of a single machine. LabBack must provide functionality to support deployments consisting of multiple LabBack instances and balance the job load across them.
3. Analysis of LabBack Requirements

3.3 Summary

This chapter has presented a thorough classification and analysis of LabBack’s functional and non-functional requirements which are summarised in Figure 3.1.

The primary task of LabBack is to compile and execute student guest code as it receives it from its clients. LabBack is a web-application which exposes services to its clients via common networked protocols and provides monitoring and configuration web interfaces.

As LabBack executes uncontrolled programs originating from students this places it at risk of security breaches through remote code injection. LabBack protects itself against these without severely restricting the functionality of the guest programs. In addition to code injection requirements other security threats with respect to system availability, security of information and fraud are summarised in Figure 3.2.

In addition to the very important security and robustness considerations LabBack must meet determinism, scalability and performance requirements.
Chapter 4

Satisfying the Non-Functional Requirements of LabBack

Many frameworks exist that allow both security and transparency in program execution. Before deciding to embark on greenfield development of LabBack 1.0, a number of alternatives were investigated. These range from restricted user accounts to continuous integration solutions and virtual machine pools.

This chapter presents a survey and comparison of avenues for technological reuse in building LabBack. Alternatives are discussed in terms of LabBack’s main non-functional requirements under the assumption that its functional requirements can be met by adaptation of, or integration with, the reused technologies. The chapter concludes that due to shortcomings of alternative technologies their reuse would compromise LabBack’s ability to meet its requirements thus recommending that LabBack should be designed and developed as a fresh product.

4.1 Native Tooling

The simplest solution for LabBack would be to reuse the native tooling delivered with various languages. This permits LabBack to reuse all existent language-specific tools as well as use the framework provided by the operating system for task management and security. Three solutions are discussed: users with restricted privileges, security with SeLinux [47] and execution jails with BSD-jails [38].

4.1.1 Restricted system users

In the simplest case LabBack would utilise the operating system’s access control mechanism to run and contain the execution jobs. Executed guest programs can be run under an unprivileged user account, for example nobody on Linux. This approach has the major advantage of being very easy to setup and use. Guest applications that run under restricted user accounts have direct access to the required language tooling and are able to perform all their required tasks.

The ease of use comes at a great cost to security, however, as the granularity of access control specification is insufficient and control is limited to file access permissions (including hardware devices and sockets). Any user on a Linux system is able to
4. Satisfying the Non-Functional Requirements of LabBack

at least read files that are world-readable. These typically include most system commands (e.g.: applications in /usr/bin) and hardware devices (/dev). Additionally all Linux users need to have access to /etc/password where major group memberships and home directory paths are stored. Read access to this file alone gives attackers sufficient information to aim attacks at higher privileged users and escape the restriction of their own user account. Programmers executing code in this setup are in essence given console access to the host machine. With minimal effort they could execute a fork bomb as exemplified in Figure 3.4 and compromise the availability of the system.

Temporarily foregoing security considerations causes another set of limitations of this method. Firstly, LabBack cannot guarantee consistency of execution results. The easiest way to compromise this consistency is for software to leave traces of execution on the file system. As previously mentioned this is possible due to the lack of granularity of the access control mechanism which leaves at least a temporary directory (e.g. /tmp) writable to all users.

Secondly, LabBack must meet its scalability requirement for coping with a large influx of execution jobs. In the current model the entire system needs to be replicated on another machine which is time consuming and resource expensive.

Thirdly, the system’s access control mechanism does not have provisions for enforcing CPU or memory utilisation quotas. A seriously misbehaving process could therefore rob the rest of the system of resources. BSD based operating systems, such as FreeBSD, do have finer grained provisions for enforcing memory and CPU quotas, but they have other shortcomings as later shown in this discussion.

4.1.2 Security-Enhanced Linux

Security-Enhanced Linux (SeLinux) [47] was originally developed by the United States Department of Defense and subsequently released as open-source in 2000. SeLinux provides support for mandatory access control of applications running on Linux platforms. It consists of user-land tools for specification of fine-grained security policies and kernel-level patches to enforce them and is well integrated in many mainstream enterprise-level Linux distributions such as Red Hat Enterprise Linux. Using SeLinux policies, applications can be confined to certain security domains and their access to resources such as files can be controlled.

Strengths SeLinux has provisions for most of the security considerations mentioned in Chapter 3 regarding access control. SeLinux enforces security policies at the kernel-level thereby giving strong guarantees of them being obeyed.

Its primary use in industry is to restrict the access of server applications (e.g. database servers) to only those files that are needed during correct execution. This limits the impact of a security breach to the data that the compromised application is allowed to access. Because SeLinux is implemented as native tooling it is almost seamlessly integrated in the host operating system. Through this integration, SeLinux is transparent to the applications running on the system.

---

Limitations  SeLinux is poorly suited to support LabBack deployments for a number of reasons. Firstly, the subjects of SeLinux security policies are specific applications that are present on the system for longer periods of time. This is contradictory with LabBack’s jobs which are applications that exist on the system only for very short timespans and change every time. SeLinux does have provisions for the specification of generic policies that place applications not covered by other policies in a generic security domain. However this policy is designed to give a minimum amount of confinement to applications that do not ship with a factor-defined policy. As such these generic policies cannot both sufficiently limit LabBack jobs and be permissive enough to run the other applications of the host system that do not have specific security policies.

Secondly, the security offered by SeLinux is as good as the policy is. On the one hand incorrectly tagged files are still readable/writable by applications. On the other hand to prevent cross-job access each different job run by LabBack needs to create its own security domain and have its own security policy. This quickly generates a very large number of security domains, something SeLinux is not designed for. To prevent the explosion of security domains a system daemon is required to cleanup the leftover security domains. To be permitted to alter security domains and policies this daemon needs to have root privileges and be run outside of SeLinux enforcement, thereby constituting an attack pathway in itself. Even if this limitation is somehow overcame SeLinux can still not protect LabBack against DoS attacks such as fork bombs.

Thirdly, LabBack deployments relying on SeLinux for security are limited in portability since SeLinux only runs on Linux distributions that have support for it. Adding SeLinux support to non-supporting distributions requires at the very least compiling a custom kernel thereby breaking support for automatic upstream kernel updates, including critical security updates. Various other platforms have mandatory access control mechanisms similar to SeLinux but security policies are not readily reusable across these mechanisms.

4.1.3 BSD jails and chroot

BSD based operating systems such as FreeBSD offer support for jails. BSD jails are similar to chroot environments in Linux in the sense that both have the goal of isolating one or more applications from the rest of the system. While chroot environments isolate the guest application at the filesystem level, many system resources such as user accounts are shared with the chroot environment. As a result chroot environments have been shown to be insufficient for securing applications and many ways to escape the limited environment exist. BSD jails not only virtualise the filesystem but

---

1At least 170,000/month based on current usage statistics. Will increase drastically with the number of students and/or courses.

2Mandatory Integrity Control in Microsoft Windows, historically SEBSD in TrustedBSD and SEDarwin in Apple Mac OS X.

3The FreeBSD Project. [http://www.freebsd.org](http://www.freebsd.org)


5Most have been fixed since their discovery, but the security of chroot is generally not trusted for mission-critical systems.
also the networking subsystem and the system’s user base. Additional finer grained tuning can enforce further restrictions on the guest system.

**Strengths** In essence BSD jails provide a lightweight virtual machine hypervisor mechanism. Its lightness stems from the guest machine being only partially virtualised and the hypervisor framework being provided directly by the host operating system. In combination with a union filesystem solution, such as UnionFS [76], BSD jails can execute guest applications safely and return to a reference filesystem state after execution is finished. This guarantees a consistently clean execution environment and helps to meet the job determinism requirement.

**Limitations** BSD jails have a number of drawbacks and limitations. Firstly, jails are built by compilation of special binaries for applications that are installed on the host system. This implies on one hand that jails have to be run on the hosts that actually built them (sacrificing migration and cluster support) and on the other hand that maintenance of the jails is cumbersome. Upgrading jails or making new applications available inside requires root access and the jail must be shut down during the process.

Secondly, without extra protective actions the jailed applications are not limited in CPU and memory usage leaving the door open for DoS attacks via guest applications. BSD-based systems do offer quota mechanisms for both CPU time and memory usage to alleviate this issue.

As a third limitation it is unknown whether most languages are supported on BSD systems questioning the platform dependency introduced. By depending on BSD-specific functionality LabBack would partially sacrifice on the platform independence requirement, although both FreeBSD and NetBSD AMIs for Amazon’s EC2 are available for example. Management of jails may be overly complicated in the case of EC2 deployments as the BSD images are modified to be able to run on EC2.

### 4.2 Virtualised Solutions

Virtual machines in production environments have gained increased popularity recently with cloud computing momentum. Virtual machines are used by both small and large companies. For the former virtual machines permit outsourcing of hardware purchase and maintenance allowing companies to focus instead on the software issues. Cloud hosting companies range from relatively small to very large deployments such as Amazon’s Elastic Compute Cloud. For large organisations that own their own servers (whether on-site or collocated in data centres) virtualisation allows deployment engineers to encapsulate parts of the software infrastructure in self-sufficient machines that can be started, migrated and replicated on demand.

---

1. This can be achieved by union mounting the jail filesystem as read-only together with a volatile read-write filesystem. When the jail completes execution the volatile filesystem is replaced with a fresh one.
2. Amazon Machine Image
3. Amazon Elastic Compute Cloud
4.2. Virtualised Solutions

As there are many ways virtualisation can be achieved [66] this analysis limits itself to virtualisation of entire machines, i.e. virtual machines, due to their popularity and versatility.

There are two distinct deployment approaches: (1) each LabBack job is processed in its own virtual machine, the machine being reset at the end of the execution or (2) each virtual machine runs multiple jobs and one of the previous approaches is employed for job containment. In the latter case it is irrelevant to LabBack whether it is run on real or virtualised hardware and design concerns are those created by the security and containment solution chosen. The former approach is further analysed.

**Strengths** Setting up virtualisation in both utilisation cases (outsourced or self-provided hosting) is relatively simple for LabBack applications and containing each job in its own virtual machine presents a number of strengths.

Firstly, it is relatively easy to setup a pool of hypervisors\(^1\) on own hardware. Well established hypervisors such as Citrix Xen [4] and VMware ESX [74] have well documented configuration procedures. Configurations can range from single hypervisors running multiple guest machines to multiple pools of hypervisors with dynamic capacity scaling and automatic migration of live machines. At the same time hypervisors can control and limit the amount of resources that are available to individual virtual machines. In a setup where each job is executed in its own virtual machine this capability is essential for limiting network access, maximum CPU time, memory available and for forcing shutdowns of machines that run misbehaving programs.

Secondly, the strong encapsulation allows the virtual machines to be based on the same snapshot and be reset to that initial state after every job is executed. This guarantees that it is impossible to obtain sensitive data from other jobs as well as enforcing that all jobs are executed in fresh environments thereby satisfying deterministic execution.

Thirdly, virtual machines used in this way can be both very lightweight and customised to the task they perform. For example different virtual machine images can be prepared for running C and Scala programs.

**Limitations** Virtualisation of LabBack instances presents a number of technical difficulties and a high cost of operation.

Firstly, most hypervisors are designed to host machines that have a relatively long life cycles. In the case of LabBack the executed jobs are very numerous and very short lived. Typical jobs require less than 10 seconds from start to finish on mediocre hardware. In comparison to this, the startup times of virtual machines are between 100% and 200%. If the hardware permits, a potential solution is to create large pools of machines that are started but paused while pending for new job requests. However due to the long time required to recycle machines the pools will quickly be exhausted during sustained peak loads.

Secondly, as mentioned previously it is possible to construct lightweight machines specialised to a certain type of job. While this keeps the machines slim it introduces a problem of cooperation between virtual machines in the case of more complex jobs.

\(^1\)The term hypervisor is used here to refer to all virtual machine management technologies, not limited to a single technology.
For example a machine can be specialised in executing C programs by compiling them to LLVM bitcode \[42\] and then compiling that to JavaScript for interpretation using Emscripten \[78\]. A machine wanting to execute programs written in some language whose compiler generates C code should be able to outsource execution of the resulting C code to another machine. However, orchestrating this cooperation at the level of virtual machines is not trivial. Additionally the cost price of executing such a job increases with the number of machines required.

Thirdly, hardware capable of running many short lived virtual machines is very expensive to purchase. The only solution is outsourcing this service, which is also very expensive. For example Amazon’s EC2 instances are priced on machine uptime in increments of one hour. Their smallest price is $0.02/hour and therefore every job executed costs $0.02. The cost of a medium size exam would be approximately $200 and the estimated cost of running the entire 2012 COPL course would be $20,000. This makes the option of outsourcing hosting of virtual machines unfeasibly expensive.

### 4.3 Build Tools

Build tools such as Apache Ant\[2\] and Apache Maven\[3\] are designed to automate the software build process serving a similar purpose to GNU Make \[68\]. The build process is fully described in build files allowing it to be automatically repeated. Although most build tools offer a fairly primitive mechanism for the specification and composition of build steps, these mechanisms are sufficiently expressive such that even complex build sequences with deep task dependencies can be described. Since the only task of build tools is to orchestrate language tooling, they only introduce minimal overhead to it. This overhead is compensated by the existence of automated software building.

**Strengths**  Because of their simplicity in use and their apparent compatibility of functionality with LabBack, build tools appear to be good base solutions for a LabBack implementation.

**Limitations**  Build tools meet almost none of LabBack’s non-functional requirements. A central such requirement for LabBack is security of the system against attacks through guest code, for which tools such Apache Ant have no provisions. An intrinsic assumption in build automation tools is that developers whose programs are built have good intentions, i.e. they have a desire to write harmless code. As such, build tools have no mechanisms to limit the actions that the invoked code may take. Although Apache Ant has a mechanism to limit the permissions of invoked code by integration with Java’s Security Manager this is only applicable to the <java> task. Not specifying any custom permissions defaults to full privileges.

Apache Ant and Apache Maven provide powerful tooling for Java or JVM-based languages but have very limited or no support for other languages.

In addition to the security and language diversity limitations, build tools lack the robustness and determinism required for LabBack. Firstly, although Apache Ant is

---

1. 100 students \times 10 questions \times 10 runs/question/student \times $0.02 = $200  
very robust in itself, it is very hard to protect the top-level build process from misbehaving children. For example there is no readily available method to specify the maximum time that a task may take and tasks that take too long cannot be forcibly stopped without destroying the top-level process as well.

Secondly, guest code can leave traces on the system. This can lead to leaks of sensitive information and can break deterministic execution if fresh execution environments cannot be guaranteed.

Thirdly, the build tools only support build automation on a single machine. They have no support for the distribution of tasks on remote machines. For example, Apache Ant does provide the sshexec task\(^1\) which can be used to invoke commands on a remote machine. However this mechanism is low-level and requires login access to remote machines, which is a security vulnerability.

### 4.4 Continuous Integration Systems

Build farms and continuous integration (CI) solutions are designed to offer a functionality very similar to LabBack - to perform compilation and execution of programs and to make their results available. As in the case of build tools, build farms and continuous integration systems have a high degree of compatibility with LabBack’s core functionality. Two technologies are simultaneously evaluated: (1) the Hudson/Jenkins\(^2\) CI systems and (2) Hydra \(^3\) build farm tooling.

Continuous integration (CI) systems such as Hudson/Jenkins\(^2\) and Hydra \(^3\) aim to increase software quality by orchestrating and automatically triggering building and testing processes for software. Although the examples given have fundamentally different functional models, they share the common goals of automated testing and publishing of software deliverables.

**Strengths** CI systems eliminate some of the limitations of build tools. For example they provide (web-based) user interfaces for configuration and monitoring, can automatically retrieve programs to build or test and notify individuals of project status changes. As CI systems are designed to handle multiple software projects, they are designed with scalability in mind and multiple instances can be orchestrated by a single CI instance. This allows computing power to be incrementally added or removed to handle varying loads.

**Limitations** While CI systems could be well suited as basis for LabBack, a number of incompatibilities limit their applicability. Firstly, CI systems do not offer more security control than the build tools they orchestrate and therefore inherit much of their limitations. The inability to stop tasks that exceed a certain execution time is a noteworthy example thereof\(^5\).

---

\(^1\)Apache Ant sshexec Task, [http://ant.apache.org/manual/Tasks/sshexec.html](http://ant.apache.org/manual/Tasks/sshexec.html)

\(^2\)Hudson, Extensible continuous integration server, [https://hudson-ci.org](https://hudson-ci.org)

\(^3\)Jenkins, An extendable open source continuous integration server, [http://jenkins-ci.org](http://jenkins-ci.org)

\(^4\)Hudson and Jenkins had a common code base. Part of the Hudson community split in 2011 continuing under the Jenkins name.

\(^5\)This is particularly a problem for Hudson and Jenkins
Secondly, the CI systems are designed to track the evolution of software organised in projects, which is incompatible with LabBack’s model where submissions are distinct and self-sufficient entities. Organising LabBack submissions as projects is cumbersome and increases the risk of information leakage across submissions.

Thirdly, CI system activity is usually triggered by an event in a version control system. LabBack has no use for a version control system and implementing custom triggers for CI systems is complicated. Hudson provides a way to manually trigger build processes by visiting a URL but this currently only works on a project basis. Additionally clients need to know a build ID number to retrieve job results. Because build processes are not requested by clients but instead they are automatically triggered by LabBack, clients have no knowledge of these IDs. This unnecessarily complicates the process by which clients can submit jobs and retrieve their results.

4.5 Custom Solution

The last resort, if no alternatives are powerful enough or extensible enough to support LabBack’s goals, is a self-developed solution that provides sufficient functionality and retains flexibility to satisfy the non-functional requirements. There are many options for platforms to base a custom LabBack solution on, but the requirements of platform independence and robustness limits this to only a handful: Python, JavaScript or a Java Virtual Machine (JVM) based solution. Of the three mentioned the JVM-based solutions are the most industry proven.

Strengths Developing a custom solution on top of the JVM has a number of advantages over the alternatives discussed previously. The most important of them is the flexibility to design and build an application that fits the requirements perfectly. Most of the solutions previously mentioned are very strong in one respect but lack severely in others. The JVM is mature and powerful enough to allow software that is both secure and flexible to be developed.

Being an industry accepted technology for enterprise quality applications has the advantage that much existing code can be reused allowing LabBack to draw functionality from other projects. In particular all compilers created and maintained in the research department hosting LabBack’s development currently target JVM languages which promises many future integration opportunities. Especially hosting compilation and execution of DSLs on LabBack is an opportunity for experimentation.

With custom solutions LabBack can be engineered to be as secure and scalable as needed. In case security vulnerabilities are discovered they are likely to be resolved faster than if existent projects are reused and modified to suit LabBack’s needs. If the JVM platform or the host language have limitations that impede development, alternatives are much more likely to be found quickly than if an existent project is repurposed.

Limitations Developing a custom solution also carries a number of disadvantages. Firstly, the development effort of a custom solution is higher than of reusing a well-chosen existing product.

\[^1\]JavaScript application running server-side in Node.js \[^2\] or similar
4.6 Conclusion and Recommendation

This section has presented an exploration of avenues for reuse of software products in developing LabBack. Most of the analysed alternatives have shown severe limitations. For example, while BSD-jails have the potential to be very secure, they are notoriously hard to maintain and to scale across multiple machines. Another example is the case of continuous integration tools which seem to be very well suited for LabBack applications but their assumptions about the existence of software projects renders them very hard to repurpose for the task at hand. Figure 4.1 summarises the alternatives discussed, their strengths and weaknesses.

The recommendation resulting from this analysis is that developing a custom solution is likely to yield better results despite its inherent high development and validation costs. Secondly, most of the features that can be taken for granted when reusing software do not exist in greenfield engineering and will require research and most likely manual implementation.

Thirdly, security of LabBack’s custom implementation is likely to be difficult to validate. Where popular open source software has been tried and tested by large communities, the security of the LabBack implementation requires validation from scratch. This difficulty can be partially offset by specifically designing LabBack for security, containment and failure.
4. **Satisfying the Non-Functional Requirements of LabBack**

cost. The development of a custom solution is more likely to provide the invaluable security requirement while allowing LabBack to retain sufficient flexibility to facilitate further research into alternative automatic evaluation and grading of students’ programs with the ultimate goal of improving computer science education.
Chapter 5

Proof of Concept: LabBack 1.0

This chapter follows up on the analysis of reuse avenues of Chapter 4 and the recommendation to develop a custom application by reporting on the design and implementation of a prototype LabBack - LabBack 1.0.

The primary objective of designing and implementing LabBack 1.0 is to provide a proof of concept that meets LabBack’s fundamental requirement of automatically and securely executing and evaluating students’ programs.

LabBack 1.0’s presentation consists of a description of its architecture followed by implementation details from the domains of security, robustness and scheduling. This chapter concludes with the evaluation of LabBack 1.0 within the case study described in Section 2.2.

5.1 Architecture

LabBack 1.0’s design and implementation form an experiment aimed at demonstrating the feasibility of automatically evaluating runtime correctness of guest programs in a robust and timely manner.

This section presents the architecture of LabBack 1.0 from a variety of viewpoints, thereby giving a complete overview of how this novel application functions. The viewpoints addressed are context, system decomposition, information architecture, process architecture and security.

It is worth noting that LabBack 1.0 is designed as an application running on top of the Java Virtual Machine (JVM) and is implemented in a mix of Java and Scala.

5.1.1 Context

LabBack 1.0’s ecosystem, as outlined in Figure 5.1, is intentionally restricted to facilitate its rapid prototyping. The position of each of the entities in its ecosystem are discussed below. It is noteworthy that although all entities are actually within the organisational boundary of the Delft University of Technology, in fact they are regarded as external entities since no control can be exerted over them.

LabBack Is the first iteration - LabBack 1.0 - of the system that is described in this thesis.
5. PROOF OF CONCEPT: LABBACK 1.0

WebLab  WebLab is the only client of LabBack 1.0, to whom it outsources part of its services. It requests that LabBack - given a program in source code and a set of specification tests also in source code - compiles and executes the program and reports errors, warnings and test reports. WebLab sends program execution requests to LabBack 1.0 on behalf of itself and of its users.

Production engineer  Although not a direct client of either LabBack or WebLab, the production engineer is responsible for the deployment behaviour of both systems. LabBack provides status information to this user through a logging facility. This log contains sufficient information to help the production engineer detect and diagnose malfunctions.

Educator  Services required by the educator such as execution of reference solutions, compilation of source code and automatic grading of student submissions are outsourced by WebLab to LabBack. These services are indispensable to the educator when preparing assignments. During the case study of the prototype the educator is internal to the organisation.

Student  When working on practical assignments or exams requests by the student to compile or execute, his solution are passed by WebLab to LabBack. The service is essential to the student when programming.

Single Sign On (SSO)  SSO is an authentication system used inside the network of Delft University of Technology. It provides authentication for users based on their credentials. WebLab outsources the authentication of educators and students to this system. The SSO is an entity in the LabBack ecosystem because WebLab - the primary LabBack client - uses it to authenticate users on whose behalf LabBack executes jobs.

5.1.2 System decomposition

The LabBack 1.0 architecture consist of nine components that each contribute towards the global functionality of the system. This section describes the responsibilities and operations of each of these nine components. The decomposition of LabBack 1.0 into components is illustrated in Figure 5.2.
5.1. Architecture

Job Runner

The job runner component provides generic control interfaces for the execution of jobs. It abstracts over the language specific processors (Scala, JavaScript and C) and delegates execution tasks to them. The job runner is responsible for maintenance of the processors’ lifecycle during normal operation. The job runner has the responsibility of isolating each running processor from the remainder of the system in order to protect the remainder of the system from abnormally behaving processors. The job runner guarantees clean job execution environments by running each job in a fresh execution container.

Language-specific processors

The processors provide actual support for language specific processing and expose a uniform control interface abstracting over the specifics of the language tooling employed. Each of their executions is initialised from job parameters. When started, each processor first compiles the code delivered with job data, then executes the program and saves the results (or errors) in the job description.

Scala processor  As Scala programs run on the JVM and since the Scala compiler is itself implemented in Scala, support in LabBack is provided by a wrapper over the compiler API. Once the program is compiled it is dynamically loaded and invoked.
and the output streams are captured and recorded. The Scala compiler and the target program are loaded in the same JVM as LabBack but separated from the rest of the system using a custom class loading implementation.

**JavaScript processor** JavaScript support is provided by using Mozilla Rhino JavaScript which compiles and interprets JavaScript programs in the JVM. JavaScript code is first compiled to Java and the resulting Java bytecode is dynamically loaded and invoked. By first compiling JavaScript code the syntactic correctness of the program can be verified prior to attempting to run it.

**C processor** C programs typically use many native libraries and running them inside the JVM is challenging. Programs are first compiled to LLVM [42] bytecode using Clang [2]. The obtained bytecode is subsequently compiled to JavaScript using Emscripten [78] and interpreted with Mozilla Rhino. The resulting JavaScript code emulates a real machine and provides JavaScript implementations of functions in the C standard library. Simulation of these functions in JavaScript renders most malicious programs harmless.

**Scheduler**

The scheduler is a singleton component that provides first-come, first-served scheduling of execution jobs sent by WebLab which the scheduler receives from the Directory Monitor. The scheduler is responsible for jobs from their arrival, through their entire lifecycle until their completion.

The scheduler is configurable with the maximum number of parallel jobs it is allowed to execute. It manages the queues of pending and running jobs ensuring that capacity is maximally utilised and jobs incur only minimal waiting times. Jobs in LabBack have an upper bound on their execution duration. The scheduler aborts jobs that exceed this bound. It regularly performs maintenance of its queues to cleanup after jobs that have finished or have been aborted.

**Scheduler fault cases** The scheduler - although unlikely - may encounter crashes caused by badly misbehaving job processors that could not be contained. For example in the event that it detects a security attack that it cannot contain the scheduler may choose to immediately abort the job and stop its own execution to protect the integrity of the system.

**Directory Monitor**

Jobs and job requests in LabBack 1.0 are stored on the filesystem by means of a shared folder between LabBack 1.0 and WebLab. The directory monitor component provides the functionality of monitoring this folder (e.g. /tmp/labback). As new jobs are created, the directory monitor verifies their integrity and informs the Scheduler of the newly arrived jobs which in turn enqueues them for execution.

---

1 Mozilla Rhino, JavaScript for Java. [http://www.mozilla.org/rhino](http://www.mozilla.org/rhino)
2 Clang. [http://clang.llvm.org](http://clang.llvm.org)
An instance of the directory monitor monitors a single folder. Therefore, monitoring of multiple folders requires starting multiple directory monitors.

**Directory Monitor fault cases** The directory monitor is particularly at risk of failure caused by unhandled events in the underlying filesystem it monitors. For example a temporary failure to list the contents of the monitored directory could cause the Directory Monitor to crash and stop monitoring for new jobs.

**Watchdog**

The watchdog component is responsible for monitoring the health of the Scheduler and Directory Monitor. In the event of their crashing or exiting, the watchdog gracefully restarts them, allowing the system to continue processing jobs. Jobs that were already running or queued for execution will not be rescheduled, thereby allowing LabBack to skip over jobs that have previously caused failures. The watchdog only exits when a LabBack shutdown signal is detected.

**Job Data Provider**

Job request data and results are exchanged between LabBack and its client as actual files saved on the filesystem. The job data provider component is a shared library between the client and LabBack that transparently handles marshalling, un-marshalling and persistence of jobs to the underlying exchanged files. The component manages the persistence of the jobs and transparency guarantees that operations on the persisted data behave atomically. The data provider handles automatic persistence using a similar mechanism to the one presented Loh et al. The encoding of job requests is discussed in Section 5.1.3.

**Client Helper**

The client helper is a thin abstraction layer over LabBack’s jobs that resides with the client. It provides convenience methods for submitting jobs of different types, retrieving their status and cancelling them if required. Typically, the client helper together with the shared Job Data Provider are statically linked in the client application.

**5.1.3 Information Architecture**

LabBack processes requests for compilation, execution and evaluation of programs which are encapsulated in jobs. For each program to be evaluated clients create a job containing all information necessary for its execution by LabBack. As jobs advance in the execution pipeline, the progress made by individual jobs is reflected as updates on their data. Clients monitor a job’s progress and retrieve its results by observing the job itself directly. Encapsulation of the necessary execution information and of the progress updates in a single entity keeps the client-server communication mechanism simple.
5. **Proof of Concept: LabBack 1.0**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>A unique id identifying this specific job.</td>
</tr>
<tr>
<td>status</td>
<td>State</td>
<td>A representation of the job state. See Section 5.1.4</td>
</tr>
<tr>
<td>codeName</td>
<td>String</td>
<td>The name of the test suite to be ran or empty if not applicable</td>
</tr>
<tr>
<td>dataDir</td>
<td>File</td>
<td>The directory containing the source code to be executed</td>
</tr>
<tr>
<td>language</td>
<td>String</td>
<td>Indicates the language of the program. One of: Scala, Javascript or C</td>
</tr>
<tr>
<td>doExecute</td>
<td>Boolean</td>
<td>If false the program will only be compiled. If true the program will also be tested</td>
</tr>
<tr>
<td>doRecord</td>
<td>Boolean</td>
<td>Indicates whether the output produced by the compiler and program should be recorded or not</td>
</tr>
<tr>
<td>output</td>
<td>String</td>
<td>Holds the recorded program output if any and if recording is enabled otherwise empty</td>
</tr>
<tr>
<td>compileOut</td>
<td>String</td>
<td>Holds the output produced by the compiler after obfuscation if recording was enabled otherwise empty</td>
</tr>
<tr>
<td>privCompileOut</td>
<td>String</td>
<td>Holds the original output produced by the compiler if recording was enabled otherwise empty</td>
</tr>
<tr>
<td>numTests</td>
<td>Integer</td>
<td>The total number of test cases that were executed or -1</td>
</tr>
<tr>
<td>numFails</td>
<td>Integer</td>
<td>The number of tests that were executed but failed or -1</td>
</tr>
<tr>
<td>failDetails</td>
<td>String</td>
<td>A list of strings containing details about the failed tests</td>
</tr>
</tbody>
</table>

Figure 5.3: Job data model.

**Job data model**

Information pertaining to a job is stored as structural properties on the job itself, as shown in Figure 5.3. The paragraphs below give an explanation of the most important job properties.

**id and status** Each job has an id property which clients used to uniquely identify jobs. This property is not used by LabBack and it never changes during the lifecycle of a job. In addition to the id property each job has an associated status which indicates to
5.1. Architecture

both LabBack and its client the state of the job in its execution pipeline. Clients use this property to determine the progress, the failure cause or the success status of each job. Complete details about the various states jobs can assume are given in Section 5.1.4.

**Processing information** As LabBack supports processing of programs of various programming languages, clients have to indicate for each program the language it is written in. This information is stored in the `language` property and is used by LabBack to dynamically determine which language-specific processor to use for the compilation and execution of the program. For example, “js” indicates that the JavaScript processor should be used, and “scala” indicates that the Scala processor should be used. Clients and WebLab are expected to previously agree on the names used for every processor.

LabBack always executes programs delivered by the client as standalone test suites. In most cases LabBack therefore needs to be informed of the program’s entry point - such as a test suite name or a function name. Clients specify this by setting the `codeName` property to the corresponding entry point. For example, for Scala programs this corresponds to the class name of the test suite to be run.

**Compilation output** Parts of the code included with the job must be kept secret from the client’s users (e.g.: students of WebLab). For example, revealing an assignment’s specification tests to students may allow them to circumvent the tests and must therefore be avoided. Since specification tests typically invoke student code any static incompatibilities between them and the program tested will be revealed as compilation errors and these typically include code fragments from the test suites. To avoid accidentally revealing secret code in error messages, LabBack’s processors remove privileged messages from the recorded compiler output and store the sanitised result in the `compilerOut` property. The original compilation messages are stored in the `privCompileOut` property of the job. This allows the client to have access to both and determine itself which to show.

**Test results** Assuming a program can be compiled, the client is primarily interested in the ratio of succeeded to total unit tests. The test report consists of the `numTests` and `numFails` properties containing the total number of tests in the test suite and the number of tests that did not succeed, respectively. If the program could not be executed due to compilation errors both of these properties default to -1. The `failDetails` property is used to store failure details such as the names of failed tests and corresponding stack traces.

**Job persistence**

Job data is exchanged between LabBack and its client via the filesystem. Persistence of this data is handled transparently by the `Job Data Provider` which serialises and deserialises the jobs to and from a structured representation in XML. Figure 5.4 shows an example of a JavaScript job after it has been successfully executed. Because both LabBack and the client use the same shared library they can effectively and transparently communicate job requests and at the same remain completely independent of each other.
5. **Proof of Concept: LabBack 1.0**

Figure 5.4: Example XML representation of completed job.

```xml
<task>
  <id>7c89de92-70b4-8</id>
  <data-dir>/tmp/webapp/7c89de92-70b4-8</data-dir>
  <run/>
  <code-name/>
  <status>
    <done></done>
  </status>
  <compile-result/>
  <priv-compile-result/>
  <exec-result>
    <num-cases>3</num-cases>
    <num-failures>1</num-failures>
  </exec-result>
  <recorded-out enabled="true">
    FAIL: zero is zero (expected 100; got 101)
  </recorded-out>
  <lang>JavaScript</lang>
</task>
```

5.1.4 Concurrency

Figure 5.5 illustrates how LabBack’s ecosystem breaks down into three processes: user, WebLab (client) and LabBack. The WebLab and LabBack processes communicate with each other by means of jobs.

The LabBack process consists of three control threads corresponding to the Directory Monitor, Scheduler and Watchdog components. LabBack processes jobs and executes guest programs within the same JVM as itself. To permit multiple jobs to be run concurrently and to allow recovery in case of job failures or program misbehaviour each job is run in its own thread. The risk of non-deterministic behaviour in the resulting multi-threaded environment is mitigated by restricting threads’ access to shared data [25, 55].

The threads used for running jobs are taken by the scheduler from a self-maintained thread pool of fixed size. Upon starting a job it is assigned by the scheduler to a free thread in this thread pool. When the job is finished the thread is discarded and a fresh thread is added in its place in the pool. This ensures that the scheduler has sufficient threads to start a new job as long as there are free threads in the pool.

Since all job executions are contained within LabBack’s JVM, the language processors are also contained therein. A notable exception is the C processor which invokes the Clang and Emscripten compilers in a native process. After invoking this native process the C processor pauses to wait for it to complete before continuing to process the job.

**Job state model**

As a job progresses through its execution it transitions through a number of states which are stored in the job’s status property. Figure 5.6 illustrates the states that a job may assume and the triggers for state transition. A job’s status and properties are continuously updated by the language processor executing it such that they reflect
the job’s actual state. This allows the job state to serve the double function of (1) controlling execution transitions in LabBack and (2) function as signalling mechanism between itself and the client.

A job always starts in the Constructing state indicating that the client is in the process of building it. Once the client has set all its required properties the job is transitioned to the Pending state indicating that the job is ready to be scheduled but
still waiting for LabBack to process it. Once the scheduler has available capacity to process the job it is transitioned to the Starting state indicating that the job is being started and is waiting for its execution processor to initialise. The time spent by the job in this state depends on the time required to initialise the execution containment and the language processor.

Upon initialisation of the processor the job enters the Compiling state. If the compilation fails the job enters the final Compile Error state. If the compilation is successful actual execution of the program follows and is indicated by the job assuming the Running state. Execution of the program can end either successfully or in failure indicated by the Done or Crashed states, respectively.

The scheduler continuously monitors each started job until its completion. If a job exceeds its maximum permitted duration it is aborted by the scheduler. This is reflected by the Killed state.

At any given moment the client which requested execution of a job can signal LabBack that it is no longer interested in its outcome by setting the job state to Cancelled. This is used by the scheduler as a hint to abort processing the job and free the resources it uses.

5.1.5 Business Process

LabBack’s business process, shown in Figure 5.7, primarily consists of scheduling and executing client requests for program compilation and runtime evaluation. In broad lines the client process creates a job and awaits its completion by LabBack, then collects its results upon completion and disposes of the job. During its waiting period it can cancel the job and continue with other tasks.

Maintenance of running jobs  LabBack’s process is a daemon process which regularly performs job scheduling and job control tasks. In the absence of pending jobs the Scheduler components maintains running jobs at a configurable interval. In each maintenance pass the scheduler verifies whether previously running jobs have completed or have exceeded a predefined maximum allowed execution time. Jobs that have exceeded this limit are aborted. All completed and aborted jobs are cleaned up and their previously used resources (memory) are recovered. The scheduler does not use any resources while it waits between maintenance passes.

Starting new jobs  After performing job maintenance the Scheduler starts any pending jobs up to available capacity and then reenters a sleeping phase which is preempted by the arrival of new jobs. This ensures that the scheduler can immediately start new jobs upon their arrival if capacity is available.

Upon starting a new job the Scheduler creates a processing thread for the job and then either starts more jobs or enters its waiting phase. The processing thread either undergoes the normal compilation and execution stages or terminates abnormally in

---

1 If the job is running or compiling it will be stopped. If it is pending in the queue it will be dequeued and never started.
2 Defaults to 1 second.
3 As notified by the Directory Monitor
5.1.6 Security

Chapter 3 identified security as the major non-functional requirement for LabBack. By definition the attacks identified as part of the analysis of Chapter 3 cannot be complete and LabBack must defend itself against a wider array of possible attacks. LabBack 1.0’s security model consists of multiple countermeasures, together forming a four-tiered security model, which is presented in this section.

Figure 5.7: BPMN blueprint of LabBack’s main process. Certain details are omitted for increased readability.

one of the two following cases: (1) the job is aborted by the scheduler or (2) the program commits a security violation at runtime.

The scheduler process ends when shutdown of LabBack is detected and all already running jobs finish or are aborted.
keystore "userkeystore.store", "jks";

// Grants for our core application
grant signedBy "WebLabSrvDev" {
  permission java.security.AllPermission;
};

// Grants for guest code
grant codeBase "file:/memory/-" {
  permission java.lang.RuntimePermission "getProtectionDomain"
};

Figure 5.8: LabBack 1.0 security policy.

Simplifying factors  In the scope of this proof of concept three simplifying assumptions are made. Firstly, LabBack 1.0 and WebLab are assumed to run on the same machine. This allows us to ignore the communication between them as a possible attack vector by working on the assumption that the operating system hosting the two services is trusted. Secondly, LabBack 1.0 does not expose any user interfaces or services that would require authentication and authorization of users. Finally, LabBack 1.0 does not use any database that can potentially be compromised by an attacker.

Restricted users

As a last resort in case of successful attacks, LabBack 1.0 relies on the underlying operating system to limit the effect of an attack. This is achieved by running LabBack 1.0 itself as an unprivileged user on the system whose actions are restricted. This does not protect LabBack against attacks but offers some degree of protection to its host system.

Code-origin distinction and security policy

The second layer of defence against attacks builds on Java’s Security Manager to limit the actions that guest code can take. Java programs by default run outside of the security manager and can perform any tasks that the underlying user in the host system is allowed to. In the case of LabBack 1.0 both internal and guest code are executed in the same virtual machine as a single process and the operating system cannot distinguish between them. Distinguishing between the two however is fundamental. Internal code is intrinsically trusted whereas guest code is intrinsically untrusted. To make this distinction LabBack 1.0 uses a custom security policy and a custom class loading mechanism. These allow it to differentiate between the two code origins and to enforce different policies on them. LabBack 1.0’s custom class loading mechanism is discussed in Section 5.2.2. Figure 5.8 shows the security policy used by LabBack 1.0. The security system uses language constructs to restrict the access and effect of guest code, in a similar way to language-based security in operating systems [65].

The security policy grants internal LabBack code unrestricted privileges. The internal code is identified by the certificate that was used to sign it, in this case a certificate owned by WebLabSrvDev. The Java security mechanism is transparent to the
code that is secured. When classes are loaded by the JVM their security domains are determined based on either author or their location.

Guest code in LabBack is not located on a real filesystem however classes automatically loaded in Java typically must have a physical location on disk within a directory or a Jar file. The class loader typically associates this location with classes that it loads. LabBack 1.0’s custom class loader automatically associates guest code with the fictitious file:/memory/-location allowing the security policy of Figure 5.8 to define a security domain for it and restrict all privileges in that domain.

Enforcement of the security policy is transparent to the code secured unless it attempts to perform actions that are denied in its security domain, in which case an AccessControlException is thrown. The guest code executed in LabBack therefore need not be aware of running in a restricted security domain. The guest code policy prevents obvious attack pathways such as file I/O, networking and JVM exit commands, but also less obvious ones such as reflection, thread modifications and most importantly explicit class loading. By using a custom class loader not only can guest code be classified as such, but its ability to escape the security domain can also be disabled. Additionally, the combination of the custom class loader with the prevention of reflection forbids guest code from accessing LabBack 1.0’s or other guest code’s data.

Class blacklisting The Java Security Manager is designed to control programs without limiting their functionality. Its typical use case is to provide different security domains for different components of larger applications. This enforces some limitations on the granularity of the security policy. To offer fine grained control over the actions of guest code LabBack 1.0’s custom class loader uses a blacklist approach to prevent loading certain classes. The guest code is unable to circumvent this restriction because the custom class loader is disconnected from the system class loaders and the guest code is not allowed to explicitly retrieve other class loaders. This imposes only minor restrictions on the guest programs but provides an invaluable complementary security layer. While a class whitelist approach is more secure it is much more functionality restricting than the blacklist approach.

Example 5.1
For example, the security policy provides no mechanism for preventing guest code from forking execution by creating and starting new threads. As such, malicious guest code could start infinitely many threads that harm the JVM’s performance. To prevent this type of attacks the ‘java.lang.Thread’ class is blacklisted in the class loader. Upon trying to load a blacklisted class the class loaders rejects this by throwing an ‘AccessControlException’.

The security model of a LabBack 1.0 is therefore a four-tiered security, each with different granularities and purposes. Together they allow internal code to execute unconstrained and at the same time keep the system secure and separate from the guest code being executed. Whereas sandboxing approaches to security provide emulation of potentially dangerous features, LabBack 1.0’s four-tier security model simply re-

\[1\] Such as system calls
stricts the actions that guest programs can take.

5.2 Implementation Details

LabBack 1.0 is implemented using Scala and Java and runs as an application on the Java Virtual Machine. This section discusses three fundamental challenges encountered during the implementation of LabBack 1.0 and presents the solutions employed to overcome them. The three challenges presented are:

1. Scheduling and concurrent execution of programs within a single Java Virtual Machine
2. Management of class and application memory
3. Forced termination of unresponsive guest programs

5.2.1 Scheduling and concurrent execution of programs

LabBack executes all guest programs it receives within the same JVM as itself. Executing all programs immediately upon their arrival leads to the host machine becoming overloaded when the number of concurrently executing programs exceeds the number of CPU cores of the machine. Even at modest loads LabBack receives a new program every few seconds. To prevent overloading from occurring, while at the same time fully utilising all available CPU cores of the machine, execution of programs needs to happen concurrently, but limited to processing capacity available. LabBack 1.0 achieves this by maintaining queues of programs pending execution and by scheduling their execution on the available processing capacity. This functionality is provided by the Scheduler component which lies at the core of LabBack 1.0.

The process of the Scheduler component follows the blueprint of Figure 5.7. This section describes the algorithm shown in Figure 5.9 that drives the scheduler. Its complete implementation in Scala is included in Appendix A.

The algorithm progresses in five steps that are repeatedly executed until a signal that LabBack is shutting down is detected.

1. Each maintenance pass begins with a cleanup of finished jobs. These are simply removed from tasks pool.
2. Jobs that are still running but have been cancelled by the client or have exceeded the maximum execution time (maxJobDuration) are requested gracefully to stop. If such jobs exist the scheduler pauses for a previously determined grace period (gracefulTimeout) to give the job processors a chance to stop execution and save the obtained results.
3. Jobs requested to stop that have not gracefully exited are forcibly stopped and cleaned up.
4. Pending jobs are dequeued and started until either no more pending jobs remain or processing capacity is fully occupied (maxJobs). Queued jobs that have been cancelled while waiting are dequeued but not started.
5. The scheduler enters an interruptible sleep phase before its next pass.

The algorithm is configurable with four parameters:
5.2. Implementation Details

**Constant:** maxJobs ▷ Predefined maximum number of concurrent jobs

**Constant:** maxJobDuration ▷ Predefined maximum job duration

**Constant:** gracefulTimeout ▷ Predefined stop request grace time

**Constant:** maintenanceInterval ▷ Predefined maintenance interval

```
function RUN(pending) ▷ pending is a reference, the queue is maintained elsewhere
    runningTasks ← Ø

    while ¬shutdown do
        entryTime ← NOW

        if ¬runningTasks.isEmpty then
            doneTasks ← Ø
            cancelTasks ← Ø
            expireTasks ← Ø

            for all t ∈ runningTasks ∧ ¬t.isRunning do
                doneTasks ← doneTasks + t
            end for

            for all t ∈ runningTasks ∧ t.isCancelled do
                REQUESTSTOP(t) ▷ Request graceful stop of t
                cancelTasks ← cancelTasks + t
            end for

            runningTasks ← runningTasks \ (doneTasks ∪ cancelTasks)

            for all t ∈ runningTasks ∧ t.getStartTime + maxJobDuration < entryTime do
                REQUESTSTOP(t) ▷ Request graceful stop of t
                expireTasks ← expireTasks + t
            end for

            runningTasks ← runningTasks \ expireTasks

            if cancelTasks.length + expireTasks.length > 0 then
                SLEEP(gracefulTimeout)
            end if

            for all t ∈ cancelTasks ∪ expireTasks ∧ t.isRunning do
                FORCESTOP(t) ▷ Abort task t
            end for
        end if

        while runningTasks.length < maxJobs ∧ ¬pending.isEmpty do
            t ← DEQUEUE(pending) ▷ Retrieve and remove the head of pending

            if ¬t.isCancelled then
                runningTasks ← runningTasks + t
                START(t) ▷ Execute task t
            end if
        end while

        NOTIFYWATCHDOG ▷ Announce liveness to watchdog

        sleepTime ← entryTime + maintenanceInterval − NOW

        if sleepTime > 0 then
            SLEEP(sleepTime)
        end if
    end while

end function
```

Figure 5.9: LabBack 1.0’s scheduler maintenance loop algorithm.
1. *maxJobs* defines the maximum number of jobs that the scheduler is allowed to run concurrently. A typical value is \( N - 1 \) where \( N \) is the number of physical CPUs of the system running LabBack.

2. *maxJobDuration* defines the maximum time that a job is allowed to execute for.

3. *gracefulTimeout* defines the grace time given to jobs that have been asked to stop before they are forcibly terminated.

4. *maintenanceInterval* defines the time interval between two consecutive maintenance passes of the algorithm.

The waiting period between maintenance passes is dynamic and is varies depending on the time the maintenance pass has taken. This allows the scheduler to maintain as regular as possible a maintenance interval, even under heavy loads.

It is noteworthy that the final call to *SLEEP* before the algorithm loops is interruptible by the arrival of new jobs. This allows the scheduler to expedite its maintenance passes if jobs are waiting.

In effect the amount of CPU time required by the scheduler is very small. Its execution can be effectively weaved with that of the *Directory Monitor* on a single processing unit since they never need to execute at the same time. LabBack instances therefore only require one CPU core for controlling execution allowing the remainder to be used for job processing.

### 5.2.2 Management of class and application memory

LabBack 1.0 executes all guest programs within the *same* JVM as itself. This section discusses the difficulties that arise from this mode of execution and introduces the method used by LabBack to overcome them.

Running all programs within the same process has the advantage of reducing initialisation time and the memory footprint of each execution, but introduces four problems of class and memory management:

1. The large number of executed jobs requires thorough management of Heap Space
2. Java’s lazy and caching class loading mechanism prevents different classes with identical names from being loaded
3. Loaded classes have to be unloaded after use to prevent exhaustion of Permanent Generation Heap which the JVM does not garbage collect
4. For security purposes certain classes have to be blacklisted and classes pertaining to guest programs must be identified as such

#### Problem 1: Heap Space management

LabBack is expected to run a very large number of jobs and programs without needing to be restarted. Since programs are executed in the same JVM, recovery of the memory used by guest programs relies on the JVM’s garbage collection mechanism. Due to the large number of programs, not recovering all memory used by them quickly leads to exhaustion of the Heap Space and to LabBack crashing.

Java’s garbage collector can only recover memory used by objects that are not referred to by alive objects. Not relinquishing references to unused objects therefore
causes memory leaks. In the case of LabBack, the most significant sources of memory leaks can be (1) language processors or (2) guest programs.

In the first instance, language processors whose underlying language-specific tooling has memory leaks will cause memory leaks in LabBack. LabBack keeps language processors loaded for longer periods of time to avoid lengthy initialisation routines, but this contrasts with the typical short-lived life cycles of language tooling, for example compilers. This different utilisation pattern of language tools may lead to memory leaks within the language tools themselves being uncovered.

In the second instance a guest program can cause memory leaks, for example, by accessing long-lived internal LabBack objects and causing them to hold references to its own data structures. In this case the garbage collector regards the program’s data structures as alive since LabBack’s objects are referencing them, preventing it from cleaning them up and thereby causing memory leaks.

To guarantee service availability LabBack must therefore ensure that (1) memory leaks in language tooling do not affect it and that (2) guest programs are entirely garbage collectible after their execution.

**Problem 2: Loading different classes with the same name**

Class loading in the JVM is performed lazily at runtime by a tree of registered class loaders. In Java every class has an associated class loader which is the class loader that has loaded it. When a class is required, the class requiring the load asks its own class loader to load the class. If this class loader can find the class itself it loads it and returns it, otherwise it delegates the loading to its parent loader.

Once loaded, classes and their metadata are cached and subsequent requests for them are resolved from the cache. Classes are identified in the JVM by their qualified names, e.g. `java.lang.String`. The caching mechanism guarantees that every request to load the `java.lang.String` produces a reference to the same unique instance of the `Class` class representing the `java.lang.String` class. This is typically required to prevent ambiguities when resolving a class name to its implementation.

**Example 5.2**

For example, imagine a loaded class `AClass` which was loaded by its class loader `ALoader` which in turn was loaded by a top-level class loader `TopLoader`. When `AClass` requires loading of a new class, say `BClass`, it requests this from `ALoader`. If `ALoader` can find `BClass` in the segment of the class-path that it handles then it loads it and returns it. If on the other hand `ALoader` cannot find `BClass` it delegates loading to its parent `TopLoader`. If `TopLoader` cannot find the class it raises a `ClassNotFoundException` exception, otherwise it loads and returns the `BClass` class.

Classes that have already been loaded are cached and in subsequent requests for `BClass`, either `ALoader` or `TopLoader` will return a cached version of it. This is done by either `ALoader` or `TopLoader`, depending on whether or not `ALoader`
caches classes that are not on its class path. The class caching prevents other implementations of BClass from being loaded.

The JVM’s class caching mechanism breaks LabBack’s functionality unless special precautions are taken. In LabBack the class paths of two programs differ in the location where provided classes must be loaded from. Caching of classes across multiple guest programs prevents each program from loading its own class implementations. This causes the programs to share each other’s implementations. This in turn causes programs to behave unpredictably at runtime. The risk of class-name collisions and implementation sharing significantly increases with the uptime of LabBack and due to the fact that LabBack executes many different implementations of the same problem.

To fulfil its function LabBack must circumvent the class caching mechanism for classes contributed by guest programs. It must do this without disabling class caching altogether, thus avoiding expensive reloading of classes that may be safely shared.

Problem 3: Unloading no longer needed classes

Strongly related to the problem of class caching is the problem of unloading unneeded classes to recover memory. Not unloading classes contributed by a guest program when it finishes leads to memory leaks which in time prevent new programs from being loaded.

Because of Java’s class loader hierarchy and its class caching mechanism all Java applications have in fact memory leaks. These are caused by instances of the Class class not being garbage collected. The JVM keeps all loaded classes and their metadata in a reserved part of the memory called the Permanent Generation (PermGen) Heap. This part of the memory is not garbage collected, leading to memory leaks. Instances of the Class class are always alive because their class loaders keep references to them in their caches. In turn instances of the root class loaders are always kept alive by the JVM itself. This renders the entire tree of class loaders and loaded classes always live and therefore not garbage collectible.

Example 5.3

Many Java applications that support deployment and loading of classes at runtime suffer from this problem. Their typical solution to this problem is to restart the application prior to loading of new classes or when the PermGen space is full.

An example is the Eclipse IDE and plugin framework. Although it supports deployment and loading of new plugins at runtime the results of doing so are unpredictable. Eclipse itself recommends that it be restarted before loading new plugins.

Another example is the web-application container Apache Tomcat, which supports (re)deployment of applications at runtime. However, after a number of deployments the JVM itself will crash with an ‘OutOfMemoryError: PermGen

1LabBack’s typical use case is execution of students’ solutions to a predefined set of programming assignments.
5.2. Implementation Details

Constant: blacklist ▷ Predefined list of forbidden classes

function LOADCLASS(name)
    if name ∈ blacklist then
        throw AccessControlException ▷ Class is blacklisted
    end if
    if c ← FINDLOADED(name) then
        return c ▷ Class is cached
    end if
    if c ← FINDSYSTEMCLASS(name) then
        return c ▷ Class is system class
    end if
    data ← READFILE(name + ".class")
    c ← DEFINECLASS(name, data, "file : /memory") ▷ Load and defined class
    return c
end function

Figure 5.10: Custom class loader algorithm.

space’ when the PermGen Heap is full and the new classes cannot be loaded. The only solution is to restart Apache Tomcat.

To ensure that it can keep executing programs without needing to be restarted LabBack must allow the garbage collection of classes loaded by guest programs that have finished execution.

Problem 4: Blacklisting of classes and recognition of guest classes

LabBack has to prevent certain types of attacks against its security and robustness by disallowing certain classes from being loaded. One example is the java.lang.Thread class which can be used by malicious guest code to launch thread bombs.

Also, as part of its security model, LabBack must take appropriate measures to distinguish its own code from the code of guest programs. This is required by the Java Security Model in order to enforce different security constraints on them.

Solution

LabBack 1.0’s solution to the four problems presented above lies with the implementation of a custom class loader that is used for loading guest programs. Each guest program is loaded with its own instance of the custom class loader and class loaders are not reused for multiple programs. This class loader allows LabBack to run guest programs in isolation. Programs are not permitted to circumvent the class loader and the class loader itself and the classes it loads are garbage collectible. The implementation is based on the algorithm of Figure 5.10. The complete implementation in Java is given in Appendix B.

The class loader has access to a predefined blacklist of classes that are forbidden and loading of classes consists of four phases:
5. Proof of Concept: LabBack 1.0

```scala
def test(suiteName: String) = {
  val loader = new InsulatingClassLoader()
  val clazz = loader.loadClass(suiteName)
  val suite: Suite = clazz.newInstance().asInstanceOf[Suite]
  val reporter = new SimpleSuiteReporter
  suite.run // ...
  // ...
}
```

Figure 5.11: Usage example of LabBack’s custom class loader.

1. The algorithm begins by checking whether the class is blacklisted and raises a security exception if that is the case.
2. If the class is not blacklisted the loader attempts to retrieve it from cache. The cache only contains classes loaded by the guest program the loader serves.
3. If the class is not cached the loader checks whether the class is a system class\(^1\) and returns it if so.
4. If no system or cached class could be found the loader defines and loads the class from the guest program’s repository of classes. The loaded class is annotated with the file:/memory origin which indicates to the Security Manager that it belongs to a guest program.

This algorithm overcomes all of the four problems outlined above. Figure 5.11 shows an usage example of this class loader - InsulatingClassLoader - where it is explicitly used to load a test suite. Firstly, because the class loader identifies the class as originating in a guest program the security manager can enforce its security policy. This policy explicitly forbids access of guest programs to the JVM’s class loading infrastructure thereby preventing the guest code from escaping the insulating class loader. Secondly, because there exists no reference to the InsulatingClassLoader instance outside of the test method, this instance is garbage collectible as soon as the method returns. This allows the class loader and all the classes contributed by the guest program to be unloaded, thus freeing PermGen Heap space.

It is important to note that as a security measure the custom class loader violates Java’s class loading order by not allowing guest program to override system classes.

---

**Example 5.4**

Java class loaders always lookup the required class in their own segment of the class path before passing the request to their parent loader. This allows applications to override implementations that are part of the Java runtime or libraries it uses. For example, a program can contribute its own implementation of `java.lang.Thread` by placing it at the end of the class path. When requested, the class loader will provide the custom implementation instead of the one included in the Java runtime.

\(^1\)A class part of the Java runtime. For example `java.lang.String`
Allowing guest programs to override classes provided by the runtime, LabBack itself or language processors may allow them to circumvent security countermeasures or the class blacklist. To avoid this LabBack’s custom class loader prioritises system classes over classes provided by guest programs.

5.2.3 Forced termination of unresponsive guest programs

LabBack executes each guest program in a separate thread within the same JVM as itself. This has a clear advantage over running each job in a separate process: thread creation is significantly less expensive than creation and initialisation of new JVMs. The latter is resource intensive and the time required for JVM initialisation depends on the complexity of each language’s runtime environment\textsuperscript{1} thus rendering this approach unfeasible.

Problem

Guest programs executed by LabBack may have bugs causing them to get caught in infinite loops and therefore necessitate their forceful termination by the scheduler. The difficulty associated with forceful termination of threads is that the \texttt{stop} method in Java’s \texttt{Thread} class has been deprecated because it is unsafe. This leaves the JVM without a safe way to abort a thread. The safety concern arises from the possibility of data structures in inconsistent states being exposed to the rest of the JVM as the thread is abruptly terminated. Upon termination of a thread all its locks are automatically released rendering even objects in inconsistent states visible.

The recommended way to stop a thread is to signal to it that it must stop. The thread, out of its own goodwill, is expected to observe this signal and terminate its own execution. This solution is, however, not applicable to LabBack where no assumptions can be made regarding the guest programs’ implementation or good intentions. This is illustrated in Figure 5.12 which shows a thread that runs a job by calling its \texttt{perform} method. A separate thread (e.g. the scheduler) may call its \texttt{pleaseStop} method to request a stop but the thread itself may be forever caught in the \texttt{perform} call and therefore never see the stop request.

Solution

LabBack’s solution to the problem of aborting thread execution is to actually use the deprecated \texttt{stop} method in \texttt{java.lang.Thread} and to show that it is safe to do so for stopping guest programs.

With the exception of actual Job entities, guest programs are not allowed to and cannot access or alter objects in the rest of the JVM. In the event of abrupt termination, with the exception of Job entities, all locks held by the guest program apply only to its own data structures which are not shared with the rest of LabBack. Thus the risk of inconsistencies is only limited to Job entities.

Since the scheduler is the only LabBack component that has references to Job instances while their programs are running, it is only the scheduler that must be protected.

\textsuperscript{1}For example, set-up times for some JVM-languages such as Scala is higher than five seconds
5. **Proof of Concept: LabBack 1.0**

```java
public class Stoppable extends Thread {
    private boolean doStop;

    public void pleaseStop() {
        doStop = true;
        this.interrupt();
    }

    @Override
    public void run() {
        // set-up ...
        if (doStop)
            return;
        job.perform();
        if (doStop)
            return;
        // tear-down ...
    }
}
```

Figure 5.12: Example stoppable thread in Java.

from the possibly inconsistent Job instances. But the scheduler is also the one that maintains the guest programs’ threads and aborts them when needed. Thus the scheduler can assume that a Job is inconsistent immediately after aborting its corresponding thread. The object can then be either repaired or directly discarded. If possible, the scheduler marks the job as having been killed to inform LabBack’s client. In both cases LabBack is able to safely terminate the runaway guest program and recover all the resources it used.

5.3 **Evaluation**

The proof of concept LabBack - LabBack 1.0, whose design and implementation aspects have been discussed in the previous sections, has been evaluated in a real life case study. As introduced in Section 2.2, the case study consisted of running WebLab + LabBack to fully support the lab work and digital exams of the 2012 Concepts of Programming Languages (COPL) course at Delft University of Technology. This section presents the results of this case study and evaluates LabBack 1.0’s effectiveness.

5.3.1 **General evaluation**

Over the six month duration of the COPL course, LabBack 1.0 has been used to compile, execute and test students’ solutions for 275 programming assignments divided over a number of tutorials, graded assignments and three programming exams. At the end of the case study LabBack had provided 49,415 test reports which were actively used in students’ grade calculations. Overall LabBack 1.0 is estimated to have executed and evaluated over 2,000,000 programs.
5.3. Evaluation

Functionality  Throughout the case study LabBack 1.0 has provided adequate functionality in compiling and evaluating programs written in Scala, JavaScript and C. During the deployment no serious bugs were uncovered and no crashes of LabBack 1.0 occurred. Additionally, LabBack 1.0’s scheduler has shown to be successful in providing maximally concurrent job executions without overloading the host system.

Security and robustness  Before making LabBack 1.0 publicly available its security was tested with a trial group. Five experienced programmers were tasked with breaking into or simply crashing LabBack 1.0 by writing malicious programs. Over the course of one day no security incidents and no crashes had occurred.

The robustness and security of LabBack 1.0 has been proven to be sufficient throughout the case study. No security incidents and no crashes were caused by malicious or malfunctioning guest programs.

Determinism of test results  Throughout the case study no complaints regarding unexpected test results were filed by any of the approximately 150 students nor by the other 75 enrolled participants. It can therefore be concluded, with a high degree of confidence, that LabBack 1.0 has executed programs and evaluated them in a deterministic fashion, thus generating predictable and reproducible results.

5.3.2 Shortcomings

Throughout the case study a set of shortcomings of LabBack 1.0 have been uncovered:

1. Communication between LabBack and WebLab by means of files on the filesystem has proven fragile and has caused a high degree of evolution coupling between the two
2. The job state model is unnecessarily complex
3. LabBack 1.0’s architecture makes extensions for different languages and experiments difficult
4. During peak loads hosting all job executions on a single machine is insufficient

This section discusses these shortcomings and makes recommendations towards improvement. These recommendations have already been included in the requirement analysis of Chapter 3.

Integration layer of WebLab + LabBack 1.0

The communication between WebLab and LabBack 1.0 consists of XML and payload files temporarily stored and accessed by both sides, a mechanism which was chosen for its simplicity. Communication by file access introduces the problem of mutual exclusion and file locking mechanisms. The universally accepted POSIX alternative to file locking is a copy-write-move pattern. This however does not resolve the issue of reading outdated data or overwriting outdated data. The resulting logic for controlling file access has thus become complex and fragile.

The advantages of having a structured data representation (XML) for information exchange are outweighed by the need for perfect version synchronisation between
WebLab and LabBack. Both communication endpoints need to read and write XML based on the same structure. This forces both sides to have a tightly coupled evolution requiring the front-end to be recompiled and redeployed with every change in the structured XML representation of the jobs. Additionally limitations of the filesystem were encountered when the default limit of 32,000 sub-directories of the ext3 filesystem was reached. But the anecdote, however, distances the subject from the object.

Other shortcomings of this pattern include the requirement that both parts either run on the same physical machine or share a filesystem; both parts requiring read-write access to shared files implies that both programs run under the same user account thereby exposing a potential security risk.

In conclusion, communication through file exchange has proven to be fragile and complex to manage. The representation of job data as XML has led to coupled evolution of LabBack and WebLab. The recommendation is for LabBack to employ a more flexible solution for client-server communication that allows decoupled evolution and permits client and server to be hosted on different servers.

Job state model

As Section 5.1.4 presented, the job state model consists of ten states. During the design stage of LabBack 1.0 the belief was that all states would be required for both internal transitions and signalling with WebLab. In practice it was observed that two of the job states are only very briefly maintained, and are thus unobservable to the client most of the time. Because the client sometimes observes these states it needs to encode support for them. This unnecessarily complicates the client application and forces LabBack itself to make the transitions through the short-lived states explicit.

A second issue regarding the job state model comes from the exposure of the Compiling and Running states to the client. This is required as WebLab does not allow students to submit a solution unless it compiles successfully. While this is reasonable there is no fundamental difference between a solution that does not compile and a solution that passes none of the specification tests. WebLab therefore need not make a distinction between these cases for anything other than reporting purposes and LabBack need not report the two states.

An additional argument towards elimination of the distinction between Compiling and Running states is that interpreted languages (e.g. JavaScript) do not have a compilation stage. Because it has to report transitions into and out of the Compiling state LabBack has to either simulate a compilation stage or completely fake this by always making these transitions. Both situations may cause WebLab to take an incorrect decision towards accepting or rejecting a submission. This can occur, for example, if the solution program has syntactic errors but LabBack reports that the compilation was successful.

The recommendation is for LabBack to employ a much simpler state model for the job and to replace progress reporting by means of state transitions with messages in the job’s metadata.

\[1\text{Namely the Constructing and Starting states.}\]
5.4. Conclusion

Evolution and extendibility

LabBack 1.0 was initially designed to support the Scala programming language and re-engineering was expected for JavaScript and C. Implementation of support for C and JavaScript highlighted the fact that execution containment interfaces are too restrictive, thus requiring a significant development effort to support new languages. The recommendation is thus that the execution containment interface be redesigned to be more flexible and reduce the coupling between the container and contained code.

A second shortcoming of LabBack 1.0 with respect to supporting new experiments was uncovered. LabBack 1.0 only evaluates programs by means of unit tests and is not flexible enough to support additional experiments. For example, measuring the coverage of code by a suite of tests is a useful evaluation of the quality of the test suite. This, however, requires instrumentation of the compiled program before its execution. The only way to support this in LabBack 1.0 is to implement a new language processor to support three-phase job processing. This new language processor cannot reuse any of the existing language processors and thus must duplicate their functionality. The recommendation is to design LabBack’s job execution pipeline to be dynamically determined by the type of job to be processed in order to allow fine-grained weaving of language processors’ functionality.

Scalability

The requirement that WebLab and LabBack 1.0 have to be co-located on the same server imposes the limitation that only a single LabBack server can be used.

Assuming that one computation thread is required for the client and one for the LabBack scheduler, LabBack can only execute $N - 2$ concurrent jobs, where $N$ is the number of CPUs of the host machine. On the deployment server used in the case study this corresponds to 14 concurrent jobs. The compilation steps for C programs are very CPU intensive, regularly taking more than 10 seconds. This generated very large queues for the scheduler. A simple replication mechanism can use a filesystem shared among different machines each running their own LabBack instance, and WebLab performing round-robin scheduling between them. This has the shortcomings of being fragile and constituting a violation of concerns for WebLab. In conclusion the recommendation is that LabBack should lend itself to being replicated on multiple servers thereby being able to concurrently evaluate more jobs.

5.4 Conclusion

This chapter has presented the design and implementation of LabBack’s proof of concept - LabBack 1.0. This prototype was designed and implemented with the purpose of evaluating the feasibility of automatically executing and grading students’ solutions to programming assignments in a secure, robust and timely manner.

The architecture of LabBack 1.0 - presented in Section 5.1 - characterises it as a JVM application that concurrently runs guest programs as threads within its own process. Security against malicious guest programs is obtained by employing the Java Security Manager and a custom class loading mechanism. Robustness of LabBack 1.0 against malfunctioning guest programs is obtained by combining the custom
5. **Proof of Concept: LabBack 1.0**

class loading mechanism with a preemptive scheduling algorithm, together allowing
guest programs to be aborted without compromise to LabBack’s availability and data
integrity.

LabBack 1.0 has been evaluated in conjunction with WebLab as its client during a six month long case study. During this period LabBack 1.0 has executed over 2,000,000 programs. The case study has revealed that LabBack 1.0’s security and robustness mechanisms are effective. A number of shortcomings were identified resulting in the following five recommendations for improvement:

1. LabBack should employ a more flexible client-server communication mechanism that allows their decoupled evolution and permits them to be hosted on different servers.
2. The job’s state model should be simplified to avoid unnecessary complexity in both LabBack and its clients.
3. LabBack’s execution container interface should be redesigned to be more flexible, thus facilitating the addition of new language processors.
4. To support a wider variety of grading and evaluation methods the job execution pipeline should be dynamically determined by the type of job to be processed. This is expected to allow the fine-grained weaving of functionality from multiple language processors.
5. LabBack should lend itself to being replicated on multiple servers, thereby increasing its scalability to higher job loads.

The conclusion, following the development and evaluation of this proof of concept, is that secure, robust and automatic evaluation of uncontrolled programs is attainable.
Building on the successful security and robustness model of the empirically validated proof of concept - LabBack 1.0 - an improved system - LabBack 2.0 - is designed to overcome the prototype’s shortcomings and implement the five recommendations of Chapter 5. The resulting application is a hybrid between a traditional middleware web application and a plugin framework. LabBack 2.0 combines LabBack 1.0’s secure runtime model with dynamic composition of execution pipelines to form an extendible platform for hosting secure evaluation of students’ programs. LabBack 2.0 is extendible with new language-specific and experiment-specific program evaluation techniques by means of cooperating plugins. Its design allows it to scale across multi-instance deployments and employs load distribution techniques that balance execution time with cost.

After presenting the results of a survey of reusable Java plugin frameworks, this chapter presents the architecture of LabBack 2.0 from a variety of viewpoints: context, components, information, concurrency, business processes, services, security and deployment.

6.1 Java Plugin Frameworks

Three Java plugin frameworks are evaluated and compared in order to determine their usability for LabBack’s plugin needs. These frameworks are: OSGi, the Java Plugin Framework and the Java Simple Plugin Framework. Results of this evaluation are summarised in Figure 6.1.

Open Services Gateway initiative (OSGi)

In lack of built-in plugin support for Java, the OSGi framework is the de facto standard module system for Java. Its flexibility and rich feature set have made it a popular choice for large enterprise systems and highly modular applications such as the Eclipse IDE. While it supports a wide variety of requirements, ranging from hot deployments to customisable plugin security, our experience with the use of OSGi in Eclipse IDE and Eclipse RCP suggests that OSGi is much too powerful and too heavyweight for LabBack’s needs. Additionally, although OSGi is security aware, it suffers from a number of vulnerabilities which make it unable to fulfil LabBack’s
6. LabBack 2.0 Architecture

<table>
<thead>
<tr>
<th></th>
<th>Hot Deployments</th>
<th>Plugin Dependencies</th>
<th>Lightweight</th>
<th>Documentation</th>
<th>Ease of use</th>
<th>Security</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Plugin Framework</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Java Simple Plugin Frame</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OSGi</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Custom solution</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Figure 6.1: Comparison of Java plugin frameworks. Highlighted cells indicate critical limitations.

requirements. The development and learning efforts required to use OSGi would likely outweigh its benefits to LabBack.

**Java Plugin Framework (JPF)**

Inspired by Eclipse’s architecture before its migration to OSGi, the Java Plugin Framework (JPF) is plugin framework that appears as a good candidate. Unfortunately JPF has two severe shortcomings: (1) it lacks any support for security against malicious code and (2) it appears to have been abandoned since 2007. Depending on legacy systems is very detrimental to software evolution and clearly excludes JPF as candidate plugin framework for LabBack.

**Java Simple Plugin Framework (JSPF)**

At the opposite spectrum of plugin frameworks from OSGi lies the Java Simple Plugin Framework (JSPF). It aims to be a very lightweight module system for Java. While there are many advocates for the use of JSPF, it has three serious shortcomings that make it unsuitable for LabBack.

Firstly, JSPF has very good support for linking plugins at compile time by means of annotations and injections, but support for adding and updating plugins at runtime is very limited. Although, for example, downloading of plugins from a URL is partly supported, JSPF itself offers no means for a developer to redeploy a plugin on a running system.

Secondly, JSPF is very open about *not* supporting any security model thereby explicitly requiring application developers to fully trust plugin developers. In the case of LabBack this is a major limitation as guest programs running are intrinsically untrusted.

Thirdly, the documentation available for JSPF is limited. In fact the documentation is sufficiently limited to not permit an informed decision to be taken about the suitability of JSPF for any application.

6.2 LabBack 2.0 Context

In a deployment consisting of a single LabBack instance its ecosystem is identical to that of LabBack 1.0, as described in Section 5.1.1. However, if a single machine is insufficient to handle the processing requests and multiple LabBack instances are deployed the situation will differ. This section briefly presents the context of such a larger deployment.

In a larger deployment multiple instances of LabBack are connected to form a tree of unrestricted degree and height as shown in Figure 6.2. In this deployment configuration clients connect to the root node which is responsible for balancing the load across its branches. Each non-root node only sees its parent as a client.

The height and degree of the tree are unrestricted. A very shallow tree of high degree has the advantage of shorter paths in communications. Having a deeper tree

---

Project Jigsaw. [http://openjdk.java.net/projects/jigsaw](http://openjdk.java.net/projects/jigsaw)
of lower average node degrees has the disadvantage of longer communication paths but the grouping of nodes in subtrees can be used to dynamically increase or decrease a deployment by more than one instance at a time. This “grouped scaling” is harder to achieve in a flatter structure which does not have any natural grouping logic. A disadvantage of deeper tree shapes is that failures of a single instance may actually cause the failure of an entire subtree.

The preference towards a flat or a deep deployment tree depends on the dynamics of the jobs that have to be run and the stability/predictability of loads. In effect this decision is left up to the production engineer, depending on the specific requirements of a deployment. LabBack is designed and built to allow this customisation. The topics of deployment topology and load distribution are further discussed in Section 6.6.4 and Chapter 8.

### 6.3 System Decomposition

LabBack 2.0’s architecture consists of ten components which together orchestrate its functionality. This section introduces these components, as shown in Figure 6.3, and discusses their roles in LabBack 2.0. While some components such as the job scheduler are familiar from the proof of concept design, other components such as the plugin manager, service providers and monitoring views are new in LabBack 2.0.
6.3. System Decomposition

<?xml version="1.0" encoding="UTF-8"?>
<plugin id="lang.js" version="0.03">
  <provides>
    <jobtype id="lang.js.compile" executor="lang.js.JSCompileLogic" />
    <jobtype id="lang.js.execute" executor="lang.js.JSEexecuteLogic" />
  </provides>
  <depends />
</depends>
</plugin>

Figure 6.4: Plugin manifest for the JavaScript plugin.

6.3.1 Plugin

LabBack is designed to permit the runtime extendibility of language- and experiment-specific support. To support this, each unit of language-specific functionality is encapsulated in a plugin. Typically each plugin implements a number of compulsory interfaces that permit the rest of the system to use it. Most importantly, each plugin declares a set of job types that it can process and implements processing code for each of them. The processing code is implemented according to a predefined system-wide interface, thus permitting the rest of the system to use the provided functionality. This interface exposes callbacks to the execution controller for tasks, such as beginning of processing and handling exceptional situations such as uncaught exceptions during processing or cancellations.

While it is common that plugins are wrappers for language tools, such as compilers, this does not have to be the case in LabBack 2.0. A plugin can serve the generic purpose of doing any (pre-)processing phase on a job. For example, a plugin could provide functionality for changing the character encoding of a specific job input prior to its execution by the intended engine.

Because plugins are triggered by job types all plugins must contribute mappings between the job types supported and the provided execution logic. These mappings are announced to LabBack by means of a mandatory manifest file specified in XML. The plugin can use this same manifest file to specify dependencies on other job types. Figure 6.4 exemplifies a plugin’s manifest. The manifest presented corresponds to the Javascript plugin which can process lang.js.compile and lang.js.execute jobs with the JSCompileLogic and the JSEexecuteLogic implementations, respectively. In addition to the compulsory manifest, a plugin contributes one or more Jar files that contain the plugin logic.

Plugins are managed by the plugin manager which, together with the execution controller, dynamically dispatch job processing to the indented plugin by matching on the job type. The anatomy of a LabBack plugin is presented in detail in Section 7.2.1. The exact execution dispatch process is explained in Section 6.6 and the LabBack-internal representation of a plugin is discussed in Section 6.4.2.
6.3.2 Plugin Manager

The plugins that encapsulate language-specific and experiment-specific functionality have to be managed, verified, loaded and unloaded. This role of orchestrating plugins is assumed by the plugin manager.

**Plugin lifecycle management** The lifecycle of a plugin begins upon its deployment when its manifest, libraries and dependencies are verified and it is stored in a database. On demand, the plugin manager can retrieve and load plugins from the database. Plugins can be unloaded to free up resources when they are temporarily unnecessary or can be completely uninstalled when they are no longer needed. The plugin manager supports deployment of plugin upgrades at runtime, without requiring the entire LabBack instance to be restarted. The functionality for deploying plugins or their updates is provided both through a web interface and through a communication protocol.

**On-demand plugin loading** Plugins installed on LabBack always contribute behaviour towards handling a specific job type. As such plugins need to be loaded on demand and called upon when a job they handled needs to be executed. The plugin manager thus maintains an aggregated mapping between job types and the plugins that provide them. When the scheduler starts a job it requests an executor for the job from the plugin manager. The plugin manager is responsible for resolving the job to a plugin that provides the required behaviour and returning this to the scheduler who can trigger the execution to start. This, in essence, is a form of dynamic dispatch on the job type.

**Plugin insulation** The plugin manager loads each plugin using an insulating class loader that prevents loaded plugins from interacting directly with each other or other parts of the system. This insulation adds an extra layer of security and at the same time allows the JVM to recover a unloaded plugin’s memory.

**Plugin verification** The plugin manager is also responsible for plugin verification. Upon deployment of a plugin it first determines whether all explicit dependencies in the plugin’s manifest are satisfied. If this is the case it then loads the plugin and attempts to simulate the execution of a job by the plugin. While this is not a foolproof mechanism of validating plugins it is sufficient to catch obvious cases where the plugin would fail to execute. The health status of plugins can be monitored on a dedicated web interface where authorised users can also perform removal, upgrading, configuration and manual activation of plugins.

**Plugin downloading** The plugin manager provides a service for retrieving a packaged version of any plugin. This is primarily used to distribute plugins to LabBack instances in a multi-instance LabBack deployment. Using this service an instance requiring a specific plugin for execution can retrieve it itself from another instance in the deployment tree. This gives LabBack a mechanism for plugin dissemination without requiring manual intervention on all instances.

\[[1]

---

74
Internal processes of the plugin manager and the interfaces of the services it offers are discussed in detail in Sections 6.6 and 6.7, respectively. Implementation details regarding loading and unloading of plugins are discussed in Chapter 7.

6.3.3 Scheduler

LabBack’s primary functionality is to schedule and concurrently execute guest programs on the available processing capacity. Most of this functionality is provided by the scheduler component, which is responsible for a job during the entire time it is present at the LabBack instance.

**Job scheduling**  The scheduler exposes service interfaces which clients use to submit jobs. Upon submission of a job the scheduler determines whether the job will be executed locally or outsourced to another instance, depending on the available local capacity. This in essence applies load balancing logic for jobs across multiple execution instances.

Jobs that are to be executed locally are placed in special queues where they are maintained by the scheduler until they are started. When a job is started the scheduler delegates a request for execution to an execution controller which obtains the correct handler for the specific job type from the plugin manager. While implementation details vary from those of the prototype, its algorithmic basis remains that shown in Figure 5.9.

**Remote job scheduling**  The scheduler of a LabBack instance can outsource execution of jobs to another instance of LabBack. In this case the scheduler in the former instance acts as a client to the scheduler in the latter instance. This mechanism allows a single implementation of the services to be used for both client-server and server-server communication. After a job is outsourced the outsourcer transparently proxies requests from the client regarding that job until its is completed and discarded. This transparent proxy mechanism allows the client to remain oblivious to LabBack’s deployment architecture and its load balancing schemes.

**Job cancellations**  In contrast to the scheduler implementation in LabBack 1.0 the new scheduler no longer handles cancellations and forced terminations of executions itself but instead delegates these to subcomponents. This has the advantage of keeping the scheduler as simple as possible and concerned with as few different issues as possible.

**Services**  In addition to job scheduling, the scheduler offers services for retrieving job statuses and their results. It also provides a web interface where authorised users can monitor the queue of executing and pending jobs.

As with other components the internal processes of the scheduler with respect to load balancing and scheduling are discussed in Section 6.6 while the interfaces for client-server and server-server communication are presented in Section 6.7.
6.3.4 Messaging

Control and signalling between plugin-provided job execution and the scheduler is completely decoupled in LabBack 2.0. Instead of direct method calls which make plugins dependent on a specific scheduler implementation, a message passing approach is used. This functionality is provided by the messaging component. Message listeners and message providers register themselves with the messaging component and can exchange triggers and payloads. Different implementations of the messaging service can employ either the send-and-wait or the send-and-forget principles for message delivery. The latter is the default behaviour in LabBack where message sending causes a message delivery thread, from a pool of available such threads, to be used for the asynchronous delivery of the message to the receiver.

The messaging service is used sparingly in LabBack. Specifically, it is used for signalling between the scheduler and the executing plugins and for signalling between the scheduler and its complementary subcomponent that provides job cancellation and termination support. Messaging is also used between the service exposed to the client for scheduling jobs and the scheduler itself. Upon completing the scheduling request from the client, the service dispatches a message for the scheduler instead of directly calling it. This allows the service to quickly terminate the connection to the client and make the socket available for another request. In addition to offering a better utilisation of network resources, this separation decreases the coupling between the service (view) and the scheduler itself (controller).

The architecture of the messaging service allows multiple message dispatchers to be hierarchically composed, thereby creating a unicast communication tree. This allows LabBack instances to be connected to an enterprise-wide AMQP\(^1\) provider or ESB\(^2\) in the future.

6.3.5 Lifecycle, Configuration and Watchdog

While not strictly speaking a single component, the lifecycle management, configuration management and watchdog are complementary in functionality. The configuration management subcomponent is responsible for allowing users to edit LabBack configuration parameters at runtime using a web interface. Configuration parameters range from the maximum number of CPUs to use for guest program execution to the hierarchy of LabBack instances in the deployment.

At application startup the lifecycle management component is responsible for reading the configuration and initialising other components such as the messaging service and the scheduler. It is also the responsibility of this component to monitor the state of the JVM and in the event of a shutdown to deactivate components, allowing for a graceful and controlled shutdown.

Similarly to LabBack 1.0, a watchdog is charged with constantly monitoring the behaviour and availability of the scheduler and the messaging service. In the event of a failure of one of these components the watchdog restarts the faulty component and allows LabBack to continue execution. The logic which the watchdog uses for re-initialisation of the faulty components is provided by the lifecycle manager.

\(^1\)Advanced Message Queuing Protocol
\(^2\)Enterprise Service Bus
6.3.6 User Management and Authentication

LabBack 2.0 has both human and machine users. The latter represent clients that require access to provided services as well as LabBack instances themselves that proxy requests on behalf of their clients. Human users have the role of setting instance configuration parameters, managing plugins and monitoring the state of the job queues. The user management component provides the logic and web interfaces for management of users, with their credentials and roles. The same component is also responsible for performing authentication and authorisation tasks on behalf of other components. This latter functionality is a crosscutting concern that, as Chapter 7 explains, is readily provided by the application framework used.

6.3.7 Logging and Persistence

Logging and persistence are typical crosscutting concerns in any application. Configuration, jobs and plugins are persisted automatically to a relational database. This allows deployment instances to behave statefully with respect to setup parameters and job results. Persistence in LabBack 2.0 is automatic and provided transparently by the underlying application framework.

Maintaining a log of events is essential to meeting LabBack’s traceability requirement. Each component and subcomponent in LabBack 2.0 therefore contributes entries directly and independently to a centralised logbook, thus permitting system behaviour to be analysed and reconstructed in the event of a failure. LabBack uses the underlying log keeping component of the application framework, which outsources actual log keeping to deployment-specific providers such as Apache log4j. To ensure that event logs are always available for analysis, LabBack does not limit itself to storing logs on the filesystem. Instead the persistence provider automatically stores individual log entries in the database and forwards the entries to the central JVM logging mechanism.

6.4 Information Architecture

LabBack’s processing is job-centric, where each job represents a unit of processing required for the compilation, execution or evaluation of students’ program. In LabBack 2.0 each job is executed using logic provided by plugins. This section presents the entities shown in Figure 6.5 and their properties.

6.4.1 Job entity

The Job is the central unit of information in LabBack. Each job is initially created as the result of a client’s request for processing of a program. After its creation it undergoes transformations during the execution pipeline until it is completed, the client retrieves its results and the job is subsequently discarded. All jobs have a number of associated properties which are summarised in Figure 6.6 and presented below.

\footnote{Apache log4j, \url{http://logging.apache.org/log4j/}}
6. **LabBack 2.0 Architecture**

![Class diagram of LabBack 2.0’s core data model.](image)

Figure 6.5: Class diagram of LabBack 2.0’s core data model.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>A unique id identifying this specific job</td>
</tr>
<tr>
<td>pre</td>
<td>Job</td>
<td>A reference to the prerequisite job if applicable</td>
</tr>
<tr>
<td>post</td>
<td>Job</td>
<td>A reference to the successor job if applicable</td>
</tr>
<tr>
<td>created</td>
<td>Date</td>
<td>The time and date when the job was created</td>
</tr>
<tr>
<td>started</td>
<td>Date</td>
<td>The time and date when the job was started</td>
</tr>
<tr>
<td>finished</td>
<td>Date</td>
<td>The time and date when the job was completed</td>
</tr>
<tr>
<td>success</td>
<td>Boolean</td>
<td>True if the job completed successfully, False otherwise</td>
</tr>
<tr>
<td>cancelled</td>
<td>Boolean</td>
<td>True if the job finished by cancellation, False otherwise</td>
</tr>
<tr>
<td>jobType</td>
<td>PluginDependency</td>
<td>Reference to soft dependency on a plugin by means of a job type</td>
</tr>
<tr>
<td>parameters</td>
<td>JobParameters</td>
<td>A reference to the key-value repository storing input and output parameters</td>
</tr>
<tr>
<td>remoteInfo</td>
<td>RemoteInfo</td>
<td>A reference to an entity holding client-specific information identifying this job</td>
</tr>
</tbody>
</table>

Figure 6.6: Data model of LabBack 2.0’s job.
6.4. Information Architecture

**Job properties**

Each job in LabBack has an associated unique identification number (UUID) which allows the job to be identified in messages passed between components. In addition to the unique identification number each job is associated with an instance of the `JobRemoteInfo` entity holding searchable information to help identification of the job by the client. This information contains a unique identification number meaningful only to the client, and a remote user property uniquely identifying the client. Because the combination of these two properties is very likely to be globally unique, the client can use this information to identify the intended job in its communications.

**Key-value store**

To reduce the coupling between LabBack itself and the processing support provided by plugin, jobs no longer encapsulate the inputs and outputs, but instead reference an instance of the `JobParameters` entity. The `JobParameters` are a key-value store for job inputs and results. This store allows both standardised keys, such as `program`, and language-specific parameters to be stored in a single place. This resolves the coupled evolution problem encountered in LabBack 1.0 and permits plugins to evolve their data models without affecting LabBack’s internal data model or the communication protocol.

**Job type**

All valid jobs are associated with precisely one job type. This job type is initially specified by the client in its request and is used by scheduler and plugin manager to determine and initialise the plugin required to execute the job. The job stores this type as a reference to an instance of the `PluginDependency` entity. The `PluginDependency` acts as a traversable soft dependency on functionality provided by a plugin. This dependency is traversable because the `PluginDependency` itself can resolve its job type to a plugin that supports its. The dependency is soft because an instance of `PluginDependency` can exist even if a plugin for its job type does not exist. The softness of the dependency allows a LabBack instance to manipulate jobs that it cannot itself execute. This is helpful, for example, when job execution is outsourced during load balancing.

**Job chains**

Jobs in LabBack 1.0 were single units of computation meaning that a single language processor was expected to execute an entire job at once. With extendability in mind, this model may prove limiting when implementing support for new languages or new experiments. This suggests that encapsulating each of the measurement functionalities in its own plugin may be a good idea. In strict contrast to LabBack 1.0, jobs in LabBack 2.0 are no longer solitary units of computation, but are instead entries in a non-cyclical doubly-linked list of jobs. In this chain every job is a separate execution phase with its own job type. Jobs can therefore consist of many execution stages with functionality provided from various plugins.

**Example 6.1**

For example, compilation of certain languages may require many sequential steps to be performed using different tooling. Significant parts of these sequences could be identical across multiple languages, lending themselves to encapsulation in...
their own plugins to reduce duplication of functionality. Another example is support for multiple experiments within the same evaluation of a guest program, such as test coverage metrics and performance metrics. Both of the metrics can be provided by instrumentation of the guest program, but neither of the experiments requires the other.

Given a chain of jobs, the scheduler guarantees the transfer of intermediate results from one responsible plugin to the next. This allows plugins to generate and reuse results that cannot be easily mapped to a key-value store and hence cannot be kept in the JobParameters store. Many of the static properties listed in Figure 6.6 have dynamically determined global variants that determine a global state over the entire chain. For example the local success property has the equivalent globalSuccess that is dynamically computed as a conjunction over individual success properties in the entire job chain.

**Dynamic chain evolution** Essential to the flexibility of the chained jobs is the ability to dynamically create chains depending on the job at hand. To this end, each plugin executing a job may add to it prerequisite and post-processing jobs. Post-processing jobs are added immediately after the current job, whereas prerequisite jobs are placed immediately before the current job. This allows the chain of jobs to evolve dynamically, as dictated by the plugins themselves, without any concern from LabBack’s internal circuitry.

**Example 6.2** The use case for adding prerequisite jobs to the chain is particularly interesting. For example, the evaluation of Java programs requires at least two stages: compilation of the source and execution of the resulting bytecode. It is desirable to keep these two execution stages separate. This is, for example, useful if bytecode instrumentation phases need to be performed after the compilation but before the execution of the program. A client that needs some Java code, for example, to be performance benchmarked can simply create a job for direct execution of the program without mentioning the separate compilation and instrumentation stages. When the corresponding execution plugin is invoked it can observe that the bytecode is not available and add the compilation and instrumentation stages as prerequisite jobs and simply exit. The scheduler is then responsible for first scheduling the prerequisite jobs and then invoking the execution plugin again.

Currently, the services exposed by the scheduler to the client give it no way of specifying a chain of jobs, instead allowing only single jobs to be scheduled. Equivalent functionality can be obtained by deploying plugins for specific job types whose sole computational contribution is to dynamically explode the single jobs into full chains.
6.4. Information Architecture

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>A unique id identifying this specific plugin</td>
</tr>
<tr>
<td>version</td>
<td>Float</td>
<td>Version number of this plugin</td>
</tr>
<tr>
<td>configured</td>
<td>Boolean</td>
<td>True if the plugin is enabled, false otherwise</td>
</tr>
<tr>
<td>validated</td>
<td>Boolean</td>
<td>True if the plugin is enabled and was successfully loaded, false otherwise</td>
</tr>
<tr>
<td>jobTypes</td>
<td>JobType+</td>
<td>The set of job types contributed by the plugin</td>
</tr>
<tr>
<td>dependencies</td>
<td>PluginDependency*</td>
<td>Job types contributed by other plugins that</td>
</tr>
<tr>
<td>mainLibrary</td>
<td>PluginData</td>
<td>Reference to the library containing core logic for the plugin</td>
</tr>
<tr>
<td>libraries</td>
<td>PluginData*</td>
<td>Libraries required by the plugin that are not in the core</td>
</tr>
</tbody>
</table>

Figure 6.7: Data model of LabBack 2.0’s main plugin entity.

6.4.2 Plugin entity

The plugin framework of LabBack serves as an extension point for functionality. Plugins encapsulate logic specific to particular job types and can be deployed, loaded and unloaded dynamically at runtime. Each plugin is represented as an instance of the Plugin entity with a number of properties, which are summarised in Figure 6.7.

Each plugin is identified by a unique identifier thereby preventing the coexistence of multiple versions of the same plugin.

A plugin consists of at least one archive containing the encapsulated logic and optionally many libraries which are used for that plugin alone. To guarantee that job chains can actually be executed each plugin specifies a set of dependencies. These dependencies are expressed solely on the basis of job types, without reference to a specific plugin. This loose coupling is established using instances of the PluginDependency entity in an identical way to jobs’ dependencies. A plugin’s dependencies are satisfied if plugins exist that provide the required job types and those plugins’ dependencies are satisfied.

Because the configuration of plugins can change at runtime the validity of a plugin’s dependencies cannot be cached and is therefore determined dynamically on demand. To simplify this process each plugin maintains the configured and validated internal indicators to signal whether the plugin is enabled and whether it can be loaded, respectively.

The loose coupling between plugins is further increased by not permitting plugins to share each others libraries directly. While this approach is clearly memory inefficient it simplifies plugin dependency management, decouples plugin evolution and strengthens security and robustness by isolating plugins.

All LabBack plugins contribute processing for at least one job type. The provided
Figure 6.8: Process and thread model of LabBack 2.0. Continuous lines indicate direct calls. Dotted lines indicate communication through messages, provided by the Messaging component.

job types are stored as references to instances of the JobType entity with unique id's. Each JobType is a mapping between the unique id, the providing plugin and the qualified name of the providing class in the plugin. The uniqueness of job types ensures that at any given time only a single plugin can contribute a specific job type.

6.5 Concurrency

This section describes the various processes and threads that are present in a LabBack deployment and how they interact to achieve LabBack's functionality. LabBack executes as many jobs as possible, as quickly as possible, by running jobs concurrently, up to the capacity of the host system. The execution pipelines are dynamically set-up by LabBack according to the jobs’ parameters and their states. This chapter also discusses the state model of the job entity.

6.5.1 Processes and Threads

At the core of LabBack lies a set of processes and threads that collaborate to realise the system behaviour. Figure 6.8 gives an overview of these processes and threads. In the figure the actor symbol is used to refer to LabBack’s clients - either human or machines.
6.5. Concurrency

**Single-instance deployment** Every instance of LabBack runs within the native process of the web application container it resides in. In the case of a single-instance LabBack deployment there are two processes that interplay: the clients processes and the LabBack process. Clients communicate with the server by means of request-response services.

**Multi-instance deployment** Adding additional LabBack instances to the deployment adds correspondingly large number of LabBack processes. Instances in different processes communicate with each other using the same services as the client-server communication. In essence, in the communication between multiple LabBack instances one of the parties masquerades as a client.

**Inter-thread communication** The majority of inter-thread communication within each instance is achieved by means of message passing, a sub-service provided by the messaging component. In response to direct calls from message senders the messaging service assigns the task of delivering the message to one of the message delivery threads in a pool of fixed size. The delivery thread then calls the message handler of the receiver to process the message.

A clear advantage of message based inter-thread communication is that strict startup and shutdown sequences are not required. In a send-and-forget message passing scheme each component has to work under the assumption that its messages may not be received by their intended recipient. Under this assumption components must implement logic for coping with (temporary) failure of neighbouring components. Uncoordinated startups and shutdowns have the same apparent effect on the system: components appear to be randomly failing. A system which is designed to cope with temporary component failures can therefore also easily handle uncontrolled startups and shutdowns.

**Client Request and Job Listener threads** To illustrate the inter-thread communication mechanisms and the function of different threads, the case of a client submitting a job for execution is used. Each client executes in its own independent process and issues requests to LabBack. Each client connection is transparently assigned by the web application container to a handling thread within LabBack. This thread, one in a pool of such threads, handles the request by receiving the client information and creating a LabBack job for it. Upon creation of the job, it sends a message to the job listener thread within the scheduler component to inform it that a job may arrive in near future. The job listener attempts to batch multiple new jobs before asking the scheduler to enqueue the jobs. This communication is in fact a message sent from the job listener to the scheduler mentioning the unique IDs of the jobs. The job listener, which runs in its own thread, therefore serves as an aggregator for multiple jobs.

**Scheduler, Cancellation and Execution threads** The scheduler thread is responsible for maintaining the pending and running jobs. It runs in its own thread because its timing characteristics are different from both the jobs it executes and the job listener or the client requests. Similarly to LabBack 1.0 each job is executed independently in its own thread. This guarantees that at least a minimum amount of processing time is available to the scheduler for maintenance of the pending and running jobs.
Contrary to LabBack 1.0, job cancellation is no longer handled in the scheduler. Instead, the scheduler sends cancellation request messages to a dedicated thread. The separation of these concerns guarantees that the scheduler thread takes regular maintenance passes which are not influenced by user-defined job cancellation timers.

6.5.2 Job state model

Evaluation of LabBack 1.0 identified the job state model as overly complex and a potential cause for engineering difficulties. In particular, many job states and transitions were unnecessarily exposed to the client, forcing it to implement dedicated handling code. In response to the corresponding recommendation of Chapter 5, the job state model of LabBack 2.0 is simple and only consists of four states: pending, running, success and failed, as shown in Figure 6.9. All additional state information that is not absolutely needed by the client or LabBack itself is stored in the job parameters and returned to the client upon job finalisation. This approach keeps the interfaces and the state model clean, provides sufficient progress information and allows for future extensions.

Global state of chained jobs

An interesting aspect to the job state machine is the state model assumed by a chain of jobs. The state model illustrated in Figure 6.9 is applicable to a chain of jobs as well. Whereas each job maintains its own states, the global state of a job chain is not stored anywhere, but is instead derived from the individual states of the job. A chain of jobs is a virtual entity which assumes the same states as the individual jobs with the exception that transitions are expressed as comprehensions over the entire chain. Figure 6.10 summarises the transition function for the global state machine.

The global state of a chain of jobs is essential to the scheduler in deciding what needs to be done with it, if anything. The scheduler and the jobs themselves guarantee that the individual job states are consistent with the global state and that job prerequisites are met before a job executes. This can be summarised in the following invariant:

If \( j \) is a job let \( \text{pre}(j) \) access the prerequisite job of \( j \), then:

\[
\forall j \in \text{JOBS}, \ j.\text{state} \in \{\text{Failure}, \text{Success}\} \Rightarrow \text{pre}(j).\text{state} = \text{Success}
\]
6.6. Business processes

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td>$\exists j \in \text{JOBS}, j.state = \text{Running}$</td>
<td>Running</td>
</tr>
<tr>
<td>Running</td>
<td>$\exists j \in \text{JOBS}, j.state = \text{Failure}$</td>
<td>Failure</td>
</tr>
<tr>
<td>Running</td>
<td>$\forall j \in \text{JOBS}, j.state = \text{Success}$</td>
<td>Success</td>
</tr>
</tbody>
</table>

Figure 6.10: Transition function of job chain state machine.

It is noteworthy that the inverse of this relationship does not always hold because a job may be started at a later time than its predecessor job has completed. It is also noteworthy that if a job is in the running state its predecessor need not have been executed, or more formally:

$\exists j \in \text{JOBS}, j.state = \text{Running}, \text{pre}(j).state = \text{Pending}$

In fact this is caused by the ability of a plugin to insert prerequisites to the job it is executing. Before it is started, a job must have all its prerequisites satisfied, but during the execution itself this relationship can be temporarily invalidated.

Transactional persistence

Due to the multithreaded nature of LabBack and its inter-thread communication by means of message passing\(^1\), components may observe jobs in inconsistent states leading to unpredictable system behaviour. To counteract this risk components must be reentrant from the point of view of the job. This is achieved by using a transaction-oriented job persistence layer\(^2\).

Whenever components need to access jobs they begin a transaction, make changes if needed and commit it when they are ready. Given that transactions are atomic, other components cannot see partially updated job information, i.e. components cannot see jobs in inconsistent states. At the same time transactions allow components to be reentrant. Although no guarantees can be made that transactions can eventually be successfully committed\(^3\), components can transparently resume computation. The case of transaction failure after reentrancy is no different than a simple case of temporary failure in the persistence component, thus all components already have error handling code for this case.

6.6 Business processes

A number of business processes are essential to the functionality offered by LabBack. These processes are triggered by incoming client requests or by internal triggers and typically span multiple components.

This section describes the architecturally relevant processes incorporated in LabBack. It begins with a description of the plugin deployment process, followed by an

---

\(^1\) As opposed to direct calls and object locks.

\(^2\) Which is automatically provided by the underlying framework

\(^3\) Other transactions may have updated structures that would cause a conflict.
LabBack 2.0 Architecture

6.6.1 Plugin Deployment (PD)

LabBack provides two mechanisms by which plugins can be deployed: (1) using a web interface or (2) calling a dedicated service. Regardless of which one of the two is used, the internal process triggered for the deployment process is identical. The Plugin Deployment (PD) process is responsible for installing or upgrading a LabBack plugin during LabBack runtime. Figure 6.11 shows the process blueprint for PD.

In brief, the process begins in the plugin management component after the entire plugin payload has been retrieved from the client’s request. If a plugin with the same ID already exists the plugin manager will perform an upgrade at runtime or reject the deployment action depending on the version number in the plugin manifest. If the deployment action is not rejected, a new instance of the plugin will be created. Subsequently, if the plugin passes configuration and validation tests it is persisted and becomes active. An older version of the plugin, if it exists, is deleted. If the new plugin

Figure 6.11: Blueprint of the Plugin Deployment (PD) process.
fails either the configuration or the validation test the plugin is rejected and the client informed.

### 6.6.2 Plugin dissemination and loading

LabBack 2.0’s language- and experiment-specific functionality is provided by plugins which have to be installed on the executing instance before a job can be started. LabBack avoids the difficulty of manually installing plugins in a multiple instance deployment by automatically disseminating plugins. This section presents the process by which LabBack instances disseminate plugins throughout the deployment tree. The presentation begins with the simple case of deployment of new plugins and ends with the case of dissemination of plugin updates.
Execution Loading (EL)

Prior to a job being processed the intended handling plugin and the specific entry point has to be determined and specialised with the job. This is the responsibility of the Execution Loading (EL) process, whose blueprint is shown in Figure 6.12. The EL process is responsible with resolving a job type to an actual instance of a plugin-provided processor.

For every job it must start, the scheduler creates a unique containment which it parameterises with the job to be contained. Upon creation, this containment invokes the EL process to obtain an instance of the actual executor class for the specific job. EL looks up the required job type in the plugin manager’s registry of job types contributed by plugins and, in the case of normal termination, it returns an instance of the specific processor as declared by the plugin.

There are two ways the EL process can terminate normally: (1) the plugin is available locally or (2) the plugin is not available locally.

It is possible that the job arrives at an instance of LabBack which does not have the required plugins installed. In this case, the plugin manager contacts the plugin manager running in its parent instance in the deployment tree. This parent is then responsible for searching for a plugin that provides the required job type, packaging it and sending it to the initial LabBack instance. Upon receiving the plugin payload the local EL process invokes the PD process to deploy the plugin locally and then continues as in the straightforward case.

The parent instance may itself not have the required plugin either, but it may have a parent itself. In this case the process is repeated to the parent’s parent, and if the plugin is eventually found it is passed back to the parent who deploys it locally (caches it) and then passes it on to its requesting child.

The EL process can terminate abnormally if the required plugin cannot be found locally or anywhere on its path to the root node of the deployment tree. In this case the job cannot be executed. It will be dequeued, unloaded and the client informed of the failure.

The purpose of the automatic dependency resolution across multiple instances is to reduce the effort required for actively pushing plugin configurations to all instances in a deployment. By using this method, which is informally called plugin gossiping, instances can be added and removed to the deployments at runtime knowing that each new instance will automatically synchronise to the required configuration as it finds necessary. The core advantage of this automatic configuration is that pre-configured deployment instances are not required and adding extra instances remains lightweight.

Plugin updates

Handling updates to already disseminated plugins is unfortunately not as straightforward. We discuss this as a side-note to the EL process. Two approaches are possible, assuming that the update is applied at the root of the LabBack deployment tree.

The first approach is to have each instance check with its master for an updated version of a plugin every time that plugin is required. While relatively simple to implement this approach has the disadvantage of instances performing many unnecessary checks to their parents. This problem is acute since large deployments tend to have
very stable plugin versions and conversely, unstable plugins tend to be encountered only in very small (testing) deployments.

The second approach to update dissemination, which is the best choice for LabBack, is to have the root node notify its immediate children when a plugin is updated. Upon receiving an update notification, children record that an update is available and forward the notification to their children. Whenever a node requires a plugin that has an update available it requests the updated plugin from its parent.

### 6.6.3 Local Job Scheduling (LJS)

As presented in Chapter 5, LabBack must balance concurrent execution with the available capacity of the host system. Jobs therefore need to be scheduled (throttled) on the available capacity. The fundamentals of the scheduling process presented in Section 5.1.5 remain unchanged in LabBack 2.0. This section presents fragments of the Local Job Scheduling (LJS) process corresponding to the invocation of the EL process and to the cancellation of expired jobs. The sketch blueprint corresponding to these fragments is shown in Figure 6.13; the parts omitted are similar to Figure 5.7.
The scheduler process is invoked similarly to the prototype. To ensure that the regularity of the maintenance passed is preserved, responsibility for actual job cancellation and abortion is outsourced to a separate daemon thread. This thread receives job cancellation requests from the scheduler via the messaging service, first requests the execution container to gracefully stop and eventually aborts the execution if the container is non-responsive.

When starting a new job the scheduler needs to first obtain an instance of an execution container that can execute the specific job type. It obtains this by invoking the EL process described previously. The scheduler does this for every job it needs to starts. Upon having scheduled all pending jobs or having reached full capacity utilisation, the scheduler pauses temporarily after which is begins a new maintenance pass.

### 6.6.4 Remote Job Scheduling (RJS)

Under high job loads, a single host machine, no matter how powerful, is not sufficient to run all jobs in a timely manner. To achieve proper scalability, LabBack needs to be scalable across the boundaries of a single machine. In this section we describe the two mechanisms used to achieve scheduling of jobs balanced on multiple machines: (1) a mechanism to decide whether a job should be scheduled locally or remotely and (2) a mechanism to perform the actual remote scheduling and retrieval of the result.

The performance bottleneck in LabBack is the time required for each job to execute. It is claimed that client-server communication and server-server communication is not a scalability concern, yet. As such the multi-instance deployment of LabBack has a tree-shaped topology where clients only communicate to the root LabBack instance which relays communication down in the tree. Given that communication is not a bottleneck, the capacity increase (scaling factor) of adding an extra LabBack instance is dependent on average number of jobs that a single machine can run concurrently.

#### Loads and schedule target determination

The process that LabBack uses to determine whether a job should be scheduled remotely, instead of being executed locally, and, in the former case, how the remote execution target is selected is discussed. The process described bases its estimates and decision on a novel measurement of branch **badness**.

**Distribution goal** As introduced in Section 6.2 the shape of a multi-instance LabBack deployment is a tree of unrestricted degree and height whose precise parameters are customisable by a production engineer. This customisability on one hand reduces the concerns of LabBack itself, but at the same time requires that implemented load distribution processes are flexible enough for varying deployments.

In large deployments, of web applications for example, load balancers are typically used to ensure an even distribution of the work across all available machines [8]. A motivation for this is an insurance of the lowest possible waiting times for requests. Another reason is to attempt to even-out the wear and tear of hardware. Evenly distribution work across all available machines has some drawbacks. For example, if the hardware used is not utilised to full capacity energy is wasted and wear and tear incurred without a real need. This problem is exacerbated in cloud deployments, on
Amazon’s EC2 for example, where usage is billed per hour of virtual machine uptime. If a machine is not used to 100% capacity it simply wastes money. Traditional load balancing techniques such as static, round-robin or shortest-queue do not solve this problem.

The solution is to employ dynamically scaling load balancing pools where machines can be added and removed from the operating pools based on loads. The approach taken in LabBack resembles a shortest-queue load balancing scheme but inverted. To ensure that the maximum amount of machines can be paused, LabBack’s load distribution mechanism attempts to keep machine utilisation close to 100%, even if at times this causes a slight increase in the lengths of job queues. Instead of striving to distribute load uniformly, LabBack strives to use as few instances as possible. Only when queues substantially increase in length does LabBack actually schedule jobs remotely.

**Remote Job Scheduling process**

This section presents the actual process used for balancing local and remote execution. Figure 6.14 shows the blueprint for the Remote Job Scheduling (RJS) process.
6. LabBack 2.0 Architecture

It is noteworthy that jobs that are part of a chain cannot be scheduled on different LabBack instances. This is, on the one hand, because jobs may share intermediate results that are not easily serialisable for networked communication. On the other hand, it is questionable whether it would be wise to split chains across multiple machines because the jobs in the chains are intrinsically sequential. Not only are gains likely to be zero, but additional synchronisation between LabBack instances would be necessary. Job chains are therefore the smallest unit that is scheduled.

**RJS Triggering**  Upon receiving a job request the RJS process is triggered to decide whether the job execution will be local or remote. To make this decision RJS compares the calculated waiting times for its own instance and for each of its subtrees. This is where LabBack distances itself from traditional load balancing techniques.

**Branch load detection**  There are no reliable or accurate means of predicting how long a processing task is going to take [67]. As such a heuristic based on queue length and queue age is used for estimating the loads of subtrees. This load heuristic is explained below.

Recall that, in LabBack, a multi-instance deployment is shaped as a tree of processing instances (nodes). Every node on the path between the root node and a job’s processing node is responsible for relaying the communication between the job’s client and that processing node. This relaying of communication allows each node on the path to include tree metadata in the job’s result package as it propagates upwards in the tree. The idea of including metadata with the communication payload is not novel as it is commonly used in all networking devices that operate at OSI Layer 2 or 3. The novel idea employed by LabBack is that, instead of including information just about the individual nodes on the relay path, one should include information about the entire subtrees that are rooted at the nodes on the relay path. This has the advantage that a single job result can provide information about a large branch of the deployment tree. For example, if the deployment tree is balanced, has a total of \(T\) nodes and the root node has a degree of \(D\), then a single job result that propagates upwards provides the root node with metadata about \(T - 1\) nodes.

The tree metadata included consists of only two values: (1) tree capacity and (2) a heuristic for the load called tree badness. Firstly, the tree capacity is given by the total number of jobs that be executed concurrently in that tree and typically corresponds to the total number of CPU units used for job execution. As a package propagates upwards, each node on the relay path adds the capacity of its branches, excluding the incoming branch, and its own capacity to the capacity entry in the metadata. This allows all nodes to maintain updated views of the processing capacities in all their branches. In the paragraphs to come, \(Q_{T_n}\) refers to the capacity of the tree \(T\) rooted at node \(n\) and \(Q_n\) refers to the capacity of node \(n\) alone.

Secondly, the tree badness embeds information about the current load and overbooking ratios in a tree. The tree badness takes into account both the badness of the subtrees and the local badness of a node. If \(W\) is the queue of waiting jobs and \(A_W\) is the set of job ages in \(W\), then the local badness \(LB_n\) of node \(n\) is computed as follows:

\[
LB_n = \frac{|W| \times \max A_W}{Q_n}
\]
The local badness is in fact a rough upper estimate of the time it is going to take a node to work through its queue. Calculation of max$A_W$ is inexpensive since $W$ is an first-in first-out queue and its two endpoints are readily accessible. It is noteworthy that if $|W|$ was increasing when the calculations was made then $LB_n$ is a low estimate because the max$A_W$ was possibly going to increase and conversely, it will be high if $|W|$ was decreasing.

Furthermore, a heuristic measure $A_{T_n}$ for the capability of a subtree to absorb new jobs is needed, which is called the absorption factor. Using $\text{deg}(n)$ to denote the degree of node $n$, the absorption factor is defined as

$$A_{T_n} = \frac{Q_{T_n}}{\text{deg}(n)}$$

and $A_{T_n} = Q_{T_n}$ if $\text{deg}(n) = 0$ (which only happens when the tree consists of a single node). Note that $A_{T_n}$ increases when $T_n$ is deeper or when its nodes have a higher capacity, thus signifying an increased capability to absorb the burden of new jobs.

The local badness and the absorption factor are taken into account when calculating the badness for the entire tree rooted at that node. If $C_n$ denotes the immediate children of node $n$, then the tree badness $TB_{T_n}$ of the tree rooted at node $n$ is computed as follows:

$$TB_{T_n} = LB_n + \sum_{c \in C_n} TB_{T_c}$$

The tree badness thus expresses the overbooking of the individual trees in terms of time as well as the lengths of waiting queues and it is adjusted with the absorption factor: a tree becomes less bad if it can absorb new jobs more easily. The waiting times themselves represent information about the processing capabilities of the nodes with respect to job difficulty and the nodes’ capacities. Thus, the badness gives a numeric value to the loads experienced in an entire subtree, where higher numbers indicate higher loads and a reduced capability to take on additional jobs.

**Example 6.3**

The calculation can be illustrated with a fictitious example. Suppose node $N_1$ has a local badness of 25 and a capacity of 2 and that $N_2$ and $N_3$ are its leaves with capacities of 10 and 2, respectively. $N_2$ initially has a tree badness of 0. If $N_3$ has a queue length of 10 with the maximum job age 5 then it has a local badness of 25. Then the entire tree $T_{N_1}$ has a local badness of $\frac{25+25+0}{14/3} \approx 10.71$. If however $N_2$ has a badness of 25 as well, then $T_{N_1}$ becomes $\frac{25+25+25}{14/3} \approx 16.07$.

The tree badness of a random node has no relevance on the scale of the entire tree and the badness values are significantly lower closer to the root node (as the subtrees become deeper). Failing subtrees are detected as being unreachable for communication. Upon detection of a failed subtree its capacity and badness are excluded from the calculations. Subtrees are re-included in the calculations if and when they recover.

---

1A subtree is failed if its root or one of its ancestors fails
As will be confirmed in Chapter 8, the badness values are suited for comparing the loads of neighbouring branches.

**Target selection** Typical (dynamic) load balancing algorithms will schedule a job on the lowest loaded branch of the deployment subtree. This however contradicts LabBack’s goal of minimising the number of machines that are actively used.

RJS selects the subtree (or itself) with the **highest** tree badness that is lower than a predefined threshold. By packing loads on already loaded subtrees, but not yet overloaded, the algorithm avoids scheduling loads to subtrees which are either (1) close to being idle or (2) significantly deeper than their neighbours. If all subtrees are above the threshold badness, then RJS reverts to classic load balancing by choosing the subtree (or itself) having the lowest badness. Note however that in this later case, RJS will favour deep trees over shallow trees as their absorption factors are higher. In the calculation of the tree badness a tree’s depth is balanced with its total capacity, causing smaller values to correspond to trees that have a higher likelihood of executing jobs quickly.

It is also worth noting that RJS attempts to schedule as many jobs as possible at once. This is similar to the local scheduling case were jobs are scheduled in batches if possible, while allowing RJS to pack as much load as possible on underloaded trees.

**Scheduling and routing** If RJS decides to schedule the job locally, it is simply enqueued by invoking the LJS process. If on the other hand the job is to be outsourced, RJS connects to the root of the chosen subtree and emulates a client’s request. After outsourcing the job, RJS stores a mapping from the unique job ID to the subtree to which the job was sent. This mapping is later used to determine the path to query for job results. In other words LabBack instances route queries to the intended destination using a content-based routing mechanism [9].

The load estimation mechanism and the RJS process are evaluated by simulation under various deployment topologies, loads and threshold values. The results are presented and deployment recommendations are formulated in Chapter 8.

### 6.7 Services

LabBack provides its functionality by means of services with predefined APIs. Services exposed by an instance of LabBack are used both by clients and by other instances in the deployment. This section begins with a brief presentation of the service transport layer, followed by detailed presentation of services offered by LabBack. Services are grouped in two categories: (1) services for job scheduling and signalling and (2) services for plugin deployment and dissemination. The goals of this section are both to present these services and to serve as an implementation guide for client developers.

#### 6.7.1 Service and communication structure

Client-server and server-server communication in LabBack uses HTTP. This is applied on top of the SSL/TLS protocol to obtain secure end-to-end communication using
6.7. Services

<table>
<thead>
<tr>
<th>Service name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>echoTest</code></td>
<td>Simple echo-reply test for instance availability</td>
</tr>
<tr>
<td><code>getJobTypes</code></td>
<td>Obtain the job types available on the queried instance</td>
</tr>
<tr>
<td><code>scheduleJob</code></td>
<td>Submit a guest program to be scheduled for execution</td>
</tr>
<tr>
<td><code>queryJobStatus</code></td>
<td>Retrieve status information for an already scheduled job</td>
</tr>
<tr>
<td><code>queryJobResult</code></td>
<td>Retrieve detailed results for an already finished job</td>
</tr>
</tbody>
</table>

Figure 6.15: Summary of job-related services offered by LabBack 2.0.

HTTPS. The HTTPS protocol is used to transport data within an HTTP request in the form of JSON objects or arrays. All offered services can be considered RESTful as they meet the relevant RESTfullness criteria.

Both the request and the response are encoded as key-value pairs inside a serialised JSON object. The request JSON objects contain authentication information in addition to request-specific data. This is used to identify and authorise the client as well as offer protection against replay attacks. Security considerations are further discussed in Section 6.8. The basic structure of a request is shown below:

```json
{
    "user" : String,
    "secret" : String,
    "otp" : Int,
    ...
}
```

The `user` value maps a string username identifying the user on whose behalf the request is made. The `secret` field maps the password set for the user. The `otp` field is a time-based password. LabBack protects itself against replay attacks by rejecting requests with expired or incorrect `otp` values.

Services are addressed by the base URL of the LabBack instance and the service name. For example, the URL `https://localhost/labback/scheduleJob` addresses the `scheduleJob` service running on the LabBack instance deployed at localhost.

### 6.7.2 Job services

LabBack provides five services for job processing. These are summarised in Figure 6.15. This section presents these services and gives extended overviews of the requests and response structures for each of them.

In the structure of the requests and responses presented the symbol `?` is used next to a field type - e.g. `String?` - to indicate an optional field. The `user`, `secret` and `otp` fields are omitted from the overview of request structures for clarity. Their presence in all requests is mandatory.
echoTest

echoTest is the simplest service offered. It can be used by a client to check the availability of the targeted LabBack instance.

Request A JSON object with a single echo entry with an associated string message:

```json
{
    "echo" : String
}
```

Response If the targeted instance is available it will respond with an identical message mapped to the reply field:

```json
{
    "reply" : String
}
```

getJobTypes

The plugin architecture allows LabBack instances to dynamically change the job types that can be supported. Clients therefore need to query the target instance for an up-to-date list of supported job types. This is offered by the getJobTypes service.

Request Aside from the mandatory authentication the client need not provide any parameters.

Response A successful request results in a JSON array of objects mapping job type parameters:

```json
[ { ... }, ... ]
```

Each entry in the array corresponds to one job type and takes the following form:

```json
{
    "jobtype" : String,
    "idemode" : Boolean,
    "version" : Float
}
```

The jobtype field maps to a unique string identifying the job type on the LabBack instance. The idemode is a flag indicating to the client whether jobs with this type result in structured messages that can be marked directly in the editor (the true case) or simply unstructured messages (the false case). The version field corresponds to the version of the plugin that provides support for this computation type and is offered purely for the convenience of the client.

scheduleJob

The scheduleJob service is used by clients to request a job to be executed on the target LabBack instance. In a deployment consisting of multiple instances, parent instances themselves access the scheduleJob service of their children. This is the mechanism by
which the RJS process (Section 6.6.4) outsources jobs to instances in subtrees. Requests made to this service do not block the client until execution is complete, instead a response is given immediately, the connection terminated and the job executed asynchronously.

**Request** A client may request scheduling of multiple jobs in a single request. The request consists of a JSON object mapping the compulsory fields and the *jobs* field mapping to an array of job description objects:

```
{
    "jobs" : [ {...}, ... ]
}
```

The job description object takes the following form:

```
{
    "jobtype" : String,
    "remoteid" : String,
    "PROGRAM" : String,
    "LIBRARY" : String,
    "TEST" : String
}
```

The *jobtype* field identifies the type of the job. This is used by LabBack to dynamically dispatch execution to the correct plugin. Clients can provide a unique identification string for the job which will be used to refer to this job in further communications. The remaining fields provide the actual source code and tests needed for execution.

**Response** The response to scheduling requests consists only of a flag, indicating whether the request was successful or not. In the latter case a *message* field is also provided with an indication of the failure cause.

```
{
    "ok" : Boolean,
    "message" : String?
}
```

**queryJobStatus**

After a job is scheduled the client can use the *queryJobStatus* service to check on the progress of the job. This service is available to the client for retrieving updates on the job progress. If the identified job is part of a chain of jobs the result is an aggregated view over the entire chain.

Successive calls to this service return different results depending on the job’s progress. Calls to this service do not alter the state of the job. Responses of this service may return slightly outdated results as a consequence of the transaction-oriented persistence mechanism.

The LabBack instance whose service is accessed either locates the intended job locally or transparently retrieves its details from another LabBack instance. Requestors to the *queryJobStatus* service can therefore be both LabBack clients (i.e. WebLab) or a LabBack instance.
6. LabBack 2.0 Architecture

<table>
<thead>
<tr>
<th>Started</th>
<th>Finished</th>
<th>Cancelled</th>
<th>Success</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>∄</td>
<td>∄</td>
<td>false</td>
<td>false</td>
<td>The job is pending</td>
</tr>
<tr>
<td>∞</td>
<td>∄</td>
<td>false</td>
<td>false</td>
<td>The job is currently running</td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>false</td>
<td>true</td>
<td>The job has completed normally</td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>false</td>
<td>false</td>
<td>The job has ended in a crash of the guest program</td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>true</td>
<td>false</td>
<td>The job has ended because it exceeded the allowed time</td>
</tr>
</tbody>
</table>

Figure 6.16: Meaning of flag combinations in response of `queryJobStatus` service. The symbol ∄ indicates that the field is missing; the symbol ∞ indicates a present field mapping any value. Unlisted combinations are unsupported.

**Request** The request made by the client must contain the `remoteid` of the job used when the `scheduleJob` service was called.

```json
{
    "remoteid" : String
}
```

**Response** The queried instance locates the intended job by matching on the mapped `remoteid` value and responds with a subset of the following:

```json
{
    "ok" : Boolean,
    "started" : DateTime?,
    "finished" : DateTime?,
    "cancelled" : Boolean?,
    "success" : Boolean?
}
```

The `ok` field indicates whether the request was successful or not. A `false` value indicates that the job could not be found. Causes are either a `remoteid` that never existed or one referring to a job that has already been garbage collected. If the `ok` field is `false` none of the other fields will be present. A value of `true` for `ok` indicates that the request was successful, the job was found and meaningful results are returned in the response.

The `cancelled` and `success` fields indicate whether the intended job has been cancelled or has completed successfully, respectively. Both fields are only present if the `finished` field is present, which indicates the time and date when the job completed. In turn the `finished` field is only present if the `started` field is present. The latter indicates the time and date when the job began execution.

Figure 6.16 summarises the meanings of the various possible combinations of flags.

---

1 As previously stated, LabBack automatically cleans up completed jobs after a predetermined period.
6.7. Services

queryJobResult

The `queryJobResult` service is used by clients and other LabBack instances to obtain detailed results for a completed job. If the job is part of a chain of jobs the response will be an aggregated view over all the jobs in the chain. A job that is not completed does not yet have detailed results. The completion status of the job can be determined using the `queryJobStatus` service.

**Request** The request made by the client must contain the `remoteid` of the job used when the `scheduleJob` service was called.

```
{
    "remoteid" : String
}
```

**Response** The queried instance attempts to locate the intended job by matching on the `remoteid` from the request and returns a subset of the following structure:

```
{
    "ok" : Boolean,
    "created" : String?,
    "started" : String?,
    "finished" : String?,
    "climsg" : String?,
    "idemsg" : [ [{...},... ]?,
    "stdout" : String?,
    "stderr" : String?,
    "tests" : Int?,
    "fails" : Int?,
    "passes" : Int?
}
```

A `false` value for `ok` indicates that either the job could not be found or the job is not yet finished. In this case no other fields will be present. The `true` case indicates that the request was successful.

If `ok` is `true` all fields can be present in the response. The `created`, `started` and `finished` fields represent the creation date, start date and finish date of the job, respectively. Dates are encoded as “yyyy-MM-dd H:mm:ss”.

The `stdout` and `stderr` fields map the recorded output produced by the executed guest program and split according to its standard and error output streams, respectively. If the program produced no output the two fields will be present and contain empty strings.

The `tests`, `fails` and `passes` fields map the total number of tests executed, the number of failed tests and the number of passed tests, respectively. The values are always positive integers or zero and the equation `tests − fails = passes` is an invariant property.

The `climsg` field contains unstructured messages produced by language-specific tooling. The `idemsg` field maps an array of structured messages according to the following structure:

```
{
    "beginrow" : Int,
    "begincolumn" : Int,
    "endrow" : Int,
}
```
### 6. LabBack 2.0 Architecture

<table>
<thead>
<tr>
<th>Service name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getPlugins</code></td>
<td>Retrieve all deployed plugins in short representation</td>
</tr>
<tr>
<td><code>getPlugin</code></td>
<td>Retrieve a specific deployed plugin in long representation</td>
</tr>
<tr>
<td><code>deployPlugin</code></td>
<td>Deploy a new/updated plugin</td>
</tr>
<tr>
<td><code>notifyPluginUpdate</code></td>
<td>Notify a subtree of an updated plugin</td>
</tr>
</tbody>
</table>

Figure 6.17: Summary of plugin-related services offered by LabBack 2.0.

```json
{
  "beginrow": Int,
  "begincolumn": Int,
  "endrow": Int,
  "endcolumn": Int,
  "level": Int,
  "message": String
}
```

The structured messages can be directly applied as markings in a text editor. The `beginrow`, `begincolumn`, `endrow` and `endcolumn` field are positive integers or zero indicating a beginning and an ending position of the marker. The `level` field can take the values 1, 2 or 3 corresponding to the note, warning and error message levels, respectively. The `message` field carries the string message to be marked at the given position with the given level.

#### 6.7.3 Plugin services

A plugin framework must provide mechanisms to manage deployed plugins. In the case of LabBack, these services are accessed not only by plugin developers and production engineers, but also by LabBack instances themselves for automatic dissemination of plugins as described in Section 6.6.2. These services are summarised in Figure 6.17. This section provides detailed explanations of these services and their request and response structures. Again, for clarity, the authentication information from the example structures is omitted and the `?` symbol is used to indicate optional fields.

### Plugin serialisation

All LabBack plugins have a serialised JSON representation that can be transported over the communication protocol. The structured representation of a serialised plugin is presented below as it is used by many of the plugin-related services.

Plugins have a `short` and a `long` depending on whether the codebase of the plugin is included or not, respectively. These representations take the following form:

```json
{
  "pluginid": String,
  "version": Float,
  "specfile": JSONFile,
  "mainlibrary": JSONFile?,
  "libraries": [JSONFile, ...]?
}
```

The `pluginid` field maps the unique ID of the plugin and the `version` field maps the plugin’s current version number. The `specfile` field contains a serialised representation
of the plugin’s manifest (recall Figure 6.4). Fields mainlibrary and libraries are only present in the long plugin serialisation. The former contains the main Jar file of the plugin. The latter contains an array of additional library Jar files required by the plugin. For convenience, the text refers to this serialised representation of the plugin as the JSONPlugin type.

JSON frameworks do not provide any built-in support for transporting files. To this end LabBack provides its own method corresponding to the JSONFile type. The JSONFile type always takes the following form:

```json
{
    "filename" : String,
    "mime-type" : String,
    "data" : String
}
```

The filename always maps to the name of the original file, including its extension. The mime-type field specifies the MIME Type of the original file. This is used by the receiver for decoding the file. The actual file data is mapped to the data encoded using Base 64 encoding. LabBack uses this encoding because of its ASCII compliance, popularity and widespread availability in programming languages.

**getPlugins**

The getPlugins service is useful for obtaining the current plugin configuration of a LabBack instance. It is used by LabBack for plugin dissemination and is also useful to deployment/production engineers for consulting deployment configurations.

**Request**  
Aside from the compulsory authentication fields, there are no parameters required for requests to the getPlugins service.

**Response**  
Responses to getPlugins requests follow the following structure:

```json
{
    "ok" : Boolean,
    "message" : String?,
    "plugins" : [ JSONPlugin, ... ]?
}
```

The response contains the ok field indicating the success of the request. In the false case the failure cause is provided in the message field which is otherwise absent in successful cases.

The list of plugins is provided in the plugin field as an array of JSONPlugin.

**getPlugin**

The getPlugin service is used to retrieve all plugin information data for a specific plugin. The service is typically used by LabBack instances to obtain a required plugin from its parent in a multi-instance deployment. The service provider returns a locally installed plugin, if it exists, or transparently retrieves it from its parent instance.
6. **LabBack 2.0 Architecture**

**Request**  A request to the *getPlugin* service must provide the unique ID of the plugin it requests:

```json
{
    "pluginid" : String
}
```

**Response**  Responses to client requests take the following form:

```json
{
    "ok" : Boolean,
    "plugin" : JSONPlugin?
}
```

The *ok* field indicates the success or the failure of the request. In the successful case the *plugin* field maps the *long* serialised representation of the plugin - JSONPlugin.

**deployPlugin**

The *deployPlugin* is the counterpart to the *getPlugin* service. It allows developers to *push* a plugin to an instance of LabBack for deployment. Upon receiving, decoding and validating the plugin the target instance deploys the plugin locally.

**Request**  Requests to *deployPlugin* take the following form:

```json
{
    "plugin" : JSONPlugin
}
```

The targeted LabBack instance enforces two security constraints. Firstly, the user authenticated in the request must have the necessary privileges for plugin deployment. Secondly, the service provider expects plugin data to be signed with an X.509 certificate [54]. The signing certificate is used to identify the plugin developer as a trustworthy source. Details regarding plugin signing and certificate management are discussed in Section 6.8.

**Response**  The typical response to requests has the following structure:

```json
{
    "ok" : Boolean,
    "message" : String?
}
```

Requests can fail - corresponding to a *false* *ok* field - in a number of situations. Firstly, a failure to authenticate or authorise the user will cause a failure. Secondly, the request will fail if the developer’s signing certificate is not known/authorised, the plugin cannot be decoded or, most importantly, if a plugin with the same ID already exists and the received plugin is not an *update*. In the failure case a field named *message* provides a textual failure cause.

The *true* value for the *ok* field indicates that the plugin payload was decoded successfully, the plugin was accepted and deployed locally. At the point when the service provider issued the response the new/upgraded plugin was deployed and became active.
notifyPluginUpdate

The notifyPluginUpdate service is used for the dissemination of plugin updates in multi-instance deployments. Recall the detailed process description of Section 6.6.2. Upon deployment of updated plugins at the tree root this service is accessed downwards in the tree to notify subtrees of the update. Service providers take no immediate local action in response to this service call, but immediately relay the update notification to their direct subtrees. Downward relaying of notification is asynchronous, i.e. callers are unblocked immediately after the response is sent.

Request  The request must contain the unique ID and the new version of the update plugin:

```json
{
    "pluginid" : String,
    "version"  : Float
}
```

Response  The response only contains a success indicator:

```json
{
    "ok" : Boolean
}
```

A true value indicates that the user credentials used were known and authorised and that the request was decoded successfully.

6.8 Security

LabBack must be secure and robust both against malicious guest code and against the more common attacks via the communication pathway. This section presents the security measures in place in LabBack 2.0.

Security in LabBack needs to ensure both the availability and integrity of the system itself and the secrecy and determinism of the guest programs. To this end LabBack 2.0 builds on the empirically validated security measures employed in LabBack 1.0 and provides additional tactics for securing new components that were not present in the prototype. Figure 6.18 gives an overview of the functional groups in LabBack 2.0 versus LabBack 1.0. In the remainder of the section only tactics employed to secure the new functional groups are discussed.

LabBack 2.0 employs security countermeasures that can be grouped in five categories: user authentication, user authorisation, plugin authorisation, guest code restriction and communication protection. Security against malicious guest code remains similar to that presented in Section 5.1.6 and is not further discussed here.

6.8.1 User Authentication and Authorisation

In contrast to the prototype, LabBack 2.0 provides services and user interfaces which require user identification prior to access being granted. A user in LabBack can be either a human user or a machine user. Administrators and production engineers are examples of the former, while clients - WebLab - and other LabBack instances are
6. LabBack 2.0 Architecture

<table>
<thead>
<tr>
<th>LabBack 2.0</th>
<th>LabBack 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest code</td>
<td>•</td>
</tr>
<tr>
<td>Language tooling</td>
<td>•</td>
</tr>
<tr>
<td>Plugins</td>
<td>•</td>
</tr>
<tr>
<td>Monitoring UI</td>
<td>•</td>
</tr>
<tr>
<td>Configuration UI</td>
<td>•</td>
</tr>
<tr>
<td>Services</td>
<td>•</td>
</tr>
</tbody>
</table>

Figure 6.18: Comparison of LabBack 1.0 and LabBack 2.0 functional groups.

examples of the latter group. Users are identified by means of username and password combinations which are centrally validated by an authentication component.

Access of a user to resources such as services and user interfaces is only granted if the user has the required privileges. Privileges in LabBack are grouped into and granted as roles associated to user accounts. We currently distinguish 5 roles:

**Client** The client is the role associated to users that can submit jobs to the LabBack instance for processing. All job related services require the requesting user to have either the client role or the LabBack role. The client role is typically associated to machine users rather than human ones.

**LabBack** All LabBack instances of a deployment tree have an associated machine-user account. Plugin related services require users to be associated with the LabBack role.

**Monitor** LabBack offers user interfaces for monitoring the health and status of the system. These interfaces expose sensitive information about the executed jobs and the deployment system and therefore require authenticated access. The monitor role is required to access these interfaces.

**Configuration** Configuration interfaces allow users to modify parameters such as job timeouts, maximum number of concurrent jobs and the structure of the deployed system. These interfaces provide access to crucial LabBack runtime parameters and therefore require authorised access. Only users having the configuration role are authorised. In addition to configuration of the deployment structure the configuration role is also required to install, remove, enable or disable plugins using the plugin management interface or the deployPlugin service.

**Administrator** The administrator privilege is required for user management tasks such as creating and deleting users and associating roles to them. A user with the administrator privilege can perform any action in LabBack.

Section 3.2 identified the requirement that LabBack behaviour must be traceable. To meet this requirement LabBack logs authentication and authorisation events - both successes and failures.

1 Either a single user per deployment tree or unique users per instance.
6.8. Security

6.8.2 Communication protection

Security of client-server and server-server communication of LabBack is paramount to meeting the security requirements identified in Section [3.2]. This section describes the measures taken in LabBack to secure the communication channels against eavesdropping and replay attacks.

**Eavesdropping** Eavesdropping on LabBack communication creates significant vulnerabilities. Firstly, guest source code and specification tests are passed in service requests. Unauthorised viewing of these may lead to privacy violation and may facilitate plagiarism. In serious cases, man-in-the-middle attack can compromise the integrity of programs received or of the specification tests to fraudulently improve results. Secondly, authentication information is provided by the client with every request made as required by the targeted services. Theft of authentication data opens the door to attacks compromising the integrity and availability of the LabBack system. To this end all client-server and server-server connections use HTTP with SSL/TLS encryption which mitigates the risk of eavesdropping.

**Replay attacks** All LabBack services are protected against request replay attacks. Such an attack can allow otherwise unskilled attackers to severely compromise the availability of the system without the need to decrypt messages. More skilled attackers could mount distributed denial-of-service (DDoS) attacks with the captured packets. To protect themselves against replay attacks all services perform three-factor authentication of the user. In addition to the user’s name and password the user provides a time-synchronised password which the server-side validates. The time-synchronised is only valid during a narrow time window and may be used multiple times during this window. At the expiration of the validity window the password is no longer accepted and the requests carrying an expired key are rejected by the service provider.

The time synchronised password mechanism is based on the Time-based One-time Password (TOTP) algorithm [53]; implementations of which exist in the public domain, e.g. Google Authenticator [30]. LabBack assigns to every user an automatically generated unique key used for seeding the TOTPs. The main advantage of TOTPs is that they prevent replay attacks at the same time as strengthening the authentication system against brute-force attacks. The only disadvantage of this mechanism is that it requires communication parties to have synchronised system clocks.

6.8.3 Plugin authorisation

LabBack 2.0 only accepts for deployment or upgrades plugins that are signed using known X.509 certificates. The mechanism employed is similar to that of LabBack 1.0 where security policy enforcement is dependent on the developer certificate used to seal the codebase. Plugins libraries that are not sealed with a known certificate, or not sealed at all, are rejected upon deployment. This mechanism protects LabBack from deployment of potentially malicious plugins.

---

1Time desynchronisation is tolerated up to the validity time of generated TOTPs and is adjustable server-side
Developer certificates are stored in a certificate store which is accessed by the security manager. The security manager is provided by the hosting web application container and is compliant with the Java Security Manager.

LabBack currently has two limitations with respect to developer certificates. Firstly, the plugins whose signing certificate expires or is revoked are only disabled automatically when the plugin reloads. Secondly, LabBack does not provide any services or user interfaces for managing certificates at runtime. Instead the certificates can be added directly in the certificate store. Possible enhancements for these limitations are discussed in Section 10.3.

## 6.9 Deployment Requirements

This section briefly discusses the software and hardware deployment requirements of LabBack 2.0. As will be shown, LabBack has no esoteric software or hardware requirements. Figure 6.19 shows a summary of the minimum requirements.

### Software requirements
LabBack can be hosted on a variety of mainstream operating systems: various Linux distributions, Apple OS X, FreeBSD and NetBSD. A web application container for Java application is required to host LabBack. Apache Tomcat 6.x or higher is known to perform well. Because LabBack and Apache Tomcat are written in Java a version of the Java Runtime Environment needs to be available. Both require Java 5.0 Standard or Enterprise Edition or higher. All instances of LabBack need to have access to a relational database; Oracle MySQL is currently supported. It is preferred that the database is hosted on the same machine as the LabBack instance itself.

---

1. In cases such as plugin upgrade, plugin disable or LabBack instance restart.

---

### Table: Minimum requirement

<table>
<thead>
<tr>
<th></th>
<th>Minimum requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating System</strong></td>
<td>Linux, Apple OS X or FreeBSD/NetBSD</td>
</tr>
<tr>
<td><strong>Java</strong></td>
<td>Java 5.0 or higher</td>
</tr>
<tr>
<td><strong>Application Container</strong></td>
<td>Apache Tomcat 6.x or higher</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>Oracle MySQL</td>
</tr>
<tr>
<td><strong>General</strong></td>
<td>Either physical and virtual machines</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>2 CPU cores</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>2GB available</td>
</tr>
<tr>
<td><strong>Disk space</strong></td>
<td>3GB free</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Available, at least one TCP port open</td>
</tr>
</tbody>
</table>

Figure 6.19: Software and hardware requirements of LabBack 2.0.
6.10. Conclusion

**Hardware requirements**  LabBack instances have non-specific hardware requirements. A machine with a minimum of 2 processing cores and 2 GB of RAM is required. As the number and size of plugins installed on each instance increases, so does the minimum amount of RAM required. The machines hosting LabBack instances do not have to be physical machines; LabBack can run on virtual machines such as Oracle Virtualbox, VmWare ESX, Citrix Xen and Amazon EC2. As job loads increase, the hardware capability requirements increase as well, to ensure short job waiting times. Each machine hosting a LabBack instance must have network connectivity so that its services are available to clients and other LabBack instances. LabBack has no specific requirements towards available disk space although a minimum of 3 GB is recommended. Configuring LabBack to cleanup completed jobs earlier will lower disk usage and vice versa.

**Scaling and variability**  Noteworthy aspects for multi-instance deployments of LabBack are briefly discussed. Firstly, software requirements are the same for all deployment instances. However there is no requirement that all instances run on the same operating system; i.e. different instances can run on different operating systems.

Secondly, the hardware capabilities need not be uniform over all instances. In fact it is recommended that instances close to the root instance\[1\] have higher, thus sufficient, network bandwidth to handle a number of concurrent requests that is twice the number of nodes in the subtrees they root. Following this recommendation ensures that limitations of the communication media will not interfere with the normal task of scheduling jobs.

Thirdly, LabBack attempts to condense as much of the workload in as few tree branches as possible. The load distribution mechanism will propagate jobs depth-first in the tree. It is therefore recommended that the deployment trees are kept balanced to keep communication overhead in the upper-level instances at a minimum.

Finally, machines whose corresponding LabBack instances have been idle for longer periods of time can be paused to save costs. If paused machines can be resumed by network access\[2\] LabBack’s load distribution mechanism will schedule jobs to them in a depth-first fashion as loads in the deployment tree increase, thereby resuming previously paused machines.

### 6.10 Conclusion

Building on the empirically validated architecture of LabBack 1.0, this chapter has presented a new and improved architecture for LabBack. This new design allows compilation and runtime evaluation to be scalable and deployable in the cloud. Reusing LabBack 1.0’s proven security design, LabBack 2.0 extends it to provide a service-oriented web application that aims to be scalable, flexible, dynamically extendable by means of plugins, and above all robust and secure against common attack pathways.

After positioning LabBack 2.0 in its ecosystem in comparison with LabBack 1.0, its component-level architecture was discussed. The architecture of LabBack 2.0 was

---

1 Root node of the deployment tree.
2 For example using Wake-on-LAN (WoL).
further presented by assuming fundamental viewpoints such as components, information, processes and security. Section 6.7 gave an in-depth presentation of the services offered by single or multi-instance deployments with the purpose of offering a complete overview of the architecture of LabBack and serving as a development guide its future clients. Section 6.9 gave an overview of the minimum deployment requirements of LabBack 2.0.

Implementation details covering the anatomy of plugins and the use of LabBack 2.0 services in WebLab are presented in Chapter 7. An evaluation of LabBack 2.0’s design by means of re-execution of a large data set and by simulation is presented in Chapter 8.
Chapter 7

LabBack 2.0 Implementation and Integration Details

This chapter presents implementation details of the (important) features of LabBack 2.0. This outline of LabBack implementation targets potential LabBack integrators as well as architects designing lightweight Java plugin frameworks and developers aiming to integrate WebDSL use in their workflows.

This chapter begins with a presentation of LabBack 2.0’s use of the WebDSL programming language in conjunction with Java and it’s effects. Section 7.2 presents the algorithm behind the dynamic plugin loading mechanism and its implementation and presents the anatomy of a plugin by means of a simple example. Section 7.3 presents the way WebLab accesses services provided by LabBack 2.0. Through the WebLab example it is shown that LabBack 2.0 induces and facilitates support for dynamic configuration of experiments in its clients.

7.1 WebDSL and Java code

LabBack 2.0 runs on the Java Virtual Machine and its implementation is written partially in Java and partially in WebDSL [73]. WebDSL was chosen as a second programming language because of its powerful abstractions for specification of dynamic web applications and possibility of extending WebDSL applications with native code. WebDSL is a domain specific language that abstracts over common boilerplate and crosscutting concerns such as entity models, persistence, access control and web pages to provide a consistent and easy to use language for web application development. The WebDSL compiler generates HTML, Javascript and Java, thus permitting WebDSL applications to integrate with code written in pure Java.

WebDSL orchestration Through its abstractions over traditional web application artefacts such as sessions, servlets and transactions, WebDSL simplifies application development. It is for this simplification that WebDSL is used in LabBack. In general, LabBack 2.0 uses WebDSL code to provide data persistence and to specify the application’s entry and exit points. Background processing functionality, such as job scheduling, is implemented directly in Java. Referring to Figure 6.3, the Plugin Manager component, for example, administers plugin entities whose data model is specified
Transactions and explicit concurrency Developers of Java code that integrates with WebDSL code must pay special attention to database transactions which are hidden away by WebDSL. For example, the code in Figure 7.1 shows the declaration for a `hello` page which simply increments a visitor count and notifies another thread of the modification by passing a message to it. When a client visits the page a new transaction is started just before the page is loaded and is committed only after the call is finished. Updates to the `Visitor` entity are thus only visible to code outside of the page call hierarchy after the call has finished and the transaction has been committed. In fact, committing the transaction may actually fail\(^1\).

This delay introduces special considerations for inter-thread communication across transactions. In the example of Figure 7.1 there is no guarantee that the receiver of the message will be able to observe the update to the `Visitor` entity immediately. In fact, the update will never take place at all if the transaction fails to be committed. Message receivers must therefore take special precautions when dealing with possible cached transactions.

The solution in Java code consists of two parts. Firstly, inter-thread communication should only be used as a hint to the receiver of a potential data modification. Secondly, background threads should themselves make use of transactions to access and update data.

This mechanism has three fundamental benefits over other solutions. Firstly, it simplifies inter-thread synchronisation by using transaction starting and ending points as synchronisation points. The synchronisation points are then automatically handled by the underlying persistence framework.

Secondly, working only on the suggestion instead of the guarantee of data availability forces developers to write more defensive code and leads to applications that are naturally more resilient to faults.

Finally, because they are already decoupled from possibly failing components, the background threads that are implemented following this pattern are easier to relocate in another process (or a different machine). While these measures only slightly in-

---

\(^1\) Due to, for example, conflicts with an earlier transaction.
crease difficulty of programming, threads implemented this way are far more robust in the face of communication or persistence failures and are believed to require less reengineering during the lifecycle of the encompassing system.

7.2 Plugin System

At the core of LabBack 2.0’s runtime-extendible execution support lies a custom-developed plugin framework. This framework allows LabBack to manage, load and unload self-contained language-specific functionality encapsulated as plugins. It consists of (1) a thin orchestration layer, (2) a mechanism for dynamically loading and unloading plugins and (3) a set of services and management interfaces.

This section presents the anatomy of LabBack 2.0’s plugins by means of an example, and describes the dynamic plugin loading mechanism.

7.2.1 Plugin anatomy

A LabBack plugin is in principle a self-sustained collection of language-specific functionality. The anatomy of a typical plugin is illustrated by means of a naive example plugin called `plug.sc.counter`.

This plugin provides the functionality of counting the number of methods in a compiled Scala class and comparing that with an expected count. The plugin obtains the bytecode of the compiled class, loads it locally and reflects over it to count its methods. After obtaining this count it compares it with an expected count that is included in the job’s parameters and stores the result back in the job parameters. Figure 7.2 summarises the steps that the plugin needs to take to complete a job execution.

Components and languages All LabBack plugins consist of a manifest, a sealed Jar archive containing at least one execution entry point and optionally one or more library Jar archives.

Plugins themselves need not be written in Java; any language that runs on the standard JVM and interoperates with Java code is equally well suited. For example, a plugin written in Scala, packaged in a Jar archive and deployed together with Scala library is suited for running on LabBack. The only pitfall is that all Jar archives deployed with the plugin must be sealed, therefore the Scala library would have to be manually

---

Figure 7.2: Processing steps of `plug.sc.counter` plugin.
7. LabBack 2.0 Implementation and Integration Details

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="plug.sc.counter" version="0.1">
  <provides>
    <jobtype id="proc.sc.counter" executor="lang.sc.counter.proc.CounterLogic"/>
  </provides>
  <depends>
    <jobtype id="lang.sc.compile"/>
  </depends>
</plugin>
```

Figure 7.3: Manifest of plug.sc.counter plugin.

```java
package weblab.plugins.control;
import webdsl.generated.domain.Job;
public interface IExecutionLogic {  
  public Object execute(Job job, Object preResults);
  public Object handleFailure(Throwable t, Job job);
  public Object handleFailure(CancellationException cex, Job job);
  public void stop();
}
```

Figure 7.4: Compulsory interface of all plugin entry points in LabBack 2.0.

sealed with a developer certificate prior to deployment. For simplicity we implement the plug.sc.counter consisting of single entry point and a single Jar archive in Java.

**Manifest**   The plugin’s manifest file informs LabBack’s plugin manager of essential details: unique identification name, version, job contributions and job dependencies. Figure 7.3 shows the manifest for the example plug.sc.counter plugin. In addition to specifying the plugin’s ID and version, the manifest is required to make the dependencies to job types explicit. Contributions of the plugin are specified as mappings between job types and the executor classes. In a similar way to Eclipse’s Extension Points [17], LabBack stores these mappings and uses them to dispatch jobs to their intended processors.

**Entry points**   Each plugin must contribute one or more job types that it can execute. These are used by LabBack as entry points to execution. Each entry point is required to implement the IExecutionLogic interface shown in Figure 7.4 consisting of four methods.

When LabBack begins execution of a job it automatically instantiates the intended implementation of IExecutionLogic. Its execute method is subsequently called to actually begin execution. Along with the call, the job itself and the results of the prerequisite job are passed as arguments. When the execution of a job completes its executor can save results within the job’s parameters and return a generic result for the successor job. LabBack ensures that this intermediate result is preserved between successive jobs in a chain.
Job executions may fail due to misbehaving or incorrect guest code or because they exceed the time limit. In LabBack, the execution containment which encompasses the running implementation of `IExecuteLogic` is responsible for monitoring execution failures and handling any uncaught exceptions. Upon detection of a failure or cancellation, the corresponding `handleFailure` method of the execution logic is called. This notifies the executing plugin of the failure cause and it is given a chance to save intermediate results and messages for the user.

Executors also implement a fourth method - `stop` - which the scheduler will call if it requires the job to stop. Upon received this call, implementors should attempt to gracefully terminate execution or otherwise risk being aborted.

**Example entry point** The implementation of `IExecuteLogic` is exemplified by means of a schematic implementation for `CounterLogic - the entry point for the proc.sc. counter job type. The core logic for the measurement is contained in the `execute` method as shown in Figure 7.5. The complete implementation is included in Appendix C.

Because the method for the counting operation requires that the guest program’s source code already be compiled, the execution of the job begins by verifying that results from the predecessor job are present. If this is not the case, a predecessor job of type `lang.sc.compile` is inserted as a prerequisite job and the execution is stopped. The scheduler will automatically start the prerequisite job first and subsequently invoke the counting job again. This is an example of how plugins can dynamically alter the composition of a job chain as introduced in Section 6.4.1.

If the bytecode of the class under analysis is present, the class is automatically loaded, its method count is obtained by reflection and the analysis result is stored on the job.

**Automatic security** It is important to note that simply loading the analysed class may trigger code specified in its static blocks to be executed, which may be malicious. This indicates that security attacks do not necessarily require their classes to be instantiated. Instead, simply loading them may expose the system to attacks. LabBack automatically protects itself against this attack pathway by identifying the loaded class as foreign code and placing it in a restricted security domain prior to its effective loading. In the event of a security violation the execution is automatically stopped and a corresponding exception is presented to the `handleFailure` method.

**Plugin building and testing** LabBack plugins can be built with any build tool that is capable of producing Jar files. To build plugins a Jar file containing the LabBack data model needs to be available on the class path. After Jar files are generated they should be sealed with a known certificate using Java’s `keytool` command.

Plugins can be tested by deploying them either in a testing or in a production instance of LabBack. While functional testing is possible and easy, it is currently not easy to run unit tests against execution logic implementations. This is due to the underlying persistence layer which is absent in unit testing environments.

---

1LabBack is designed to defend itself against misbehaving code
7. **LabBack 2.0 Implementation and Integration Details**

```java
public Object execute(Job j, Object pres) {
    if (pres == null || !(pres instanceof HashMap<?, ?>)) {
        Job pre = Job.buildFrom(scala_compile, j,
                                new String[] { Constants.PROGRAM });
        j.addPreJobJob_(pre);
        j.setNeedsReset(true);
        j.setSuccess(false);
        return null;
    }

    final InsulatingClassLoader loader = new InsulatingClassLoader(
        (HashMap<String, byte[]>) pres);
    final Class<?> clazz = loader.loadClass(j.getParameters().
                                            getString_(
                                            Constants.PROGRAM_NAME));

    if (stop) {
        j.setCancelled(true);
        j.setSuccess(false);
        return pres;
    }

    final int expectMethods = Integer.parseInt(j.getParameters().
                                             getString_(Constants.TEST));
    final int numMethods = clazz.getMethod().length;

    if (numMethods == expectMethods) {
        final String msg = "Failed. Expected " + expectMethods + "] but got [" + numMethods + "]";
        j.getParameters().putString__Text_(Constants.JOB_MSG, msg);
    } else {
        j.setSuccess(true);
    }

    return pres;
}
```

---

Figure 7.5: Execution logic in CounterLogic.

### 7.2.2 Plugin loading/unloading

Loading, unloading and instantiating correct execution logic implementations falls under the responsibility of the plugin loader. With the dual purpose of highlighting the mechanism and providing reference for developers of similar plugin frameworks, this plugin loading mechanism is described in the following section. Recall that the main reason for LabBack’s plugin architecture is to support runtime evolution of languagespecific and experiment-specific processors, which previously required a redeployment of the entire LabBack 1.0 instance.

**Plugins as disjoint entities** Runtime evolution of functionality implies at least the ability to install, update or uninstall plugins at runtime. Java plugin frameworks such as OSGi do not or only partially support the latter.

The difficulty associated with runtime evolution is that plugins, in typical systems,
7.2. Plugin System

```plaintext
loaders ← ∅  
versions ← ∅  

function LOAD(plugin)
    if ISLOADED(plugin) ∧ ¬ISUPDATE(plugin) then
        return ▶ 1. Plugin is already loaded
    end if
    if ¬ISSATISFIED(plugin) then
        throw InitializationException ▶ 2. Plugin has broken dependencies
    end if
    jarData ← ∅  
    for all jar ∈ plugin.jarFiles do ▶ 3. Retrieve plugin data from database
        jarData ← jarData + GETDATA(jar)
    end for
    data ← ∅  
    ▶ Class name to data key-value store
    for all jar ∈ jarData do
        for all entry ∈ jar do
            if ∄ data(entry.name) then
                data(entry.name) ← entry.data
            end if
        end for
    end for
    loader ← new PluginClassLoader(plugin.ID)  ▶ Add read classes to class loader
    loader.addData(data)
    loaders(plugin.ID) ← loader
    versions(plugin.ID) ← plugin.vesion
end function
```

Figure 7.6: Pseudocode specification of plugin loading mechanism.

directly interact with each other. LabBack alleviates this difficulty by not permitting direct interaction of plugins, i.e. the only interaction mechanism provided is through the processing of jobs. LabBack takes this separation further by keeping plugins’ functionality and code bases (including libraries) completely disjoint. If, for example, two plugins require the Apache Commons Java library they each have to provide this library and the library itself will be loaded separately for each of the plugins. While this has the disadvantage of increasing memory usage it has the strength of allowing a completely decoupled evolution of plugins.

Loading mechanism  Lying at the core of the plugin loading mechanism is a key-value store between plugin IDs and the plugin’s unique class loader. This maintains a mapping at runtime between each (actively) loaded plugin and the class loader that is responsible for loading its classes. Additional to this mapping, another key-value store is maintained for keeping track of the actual version of each plugin that is loaded. This latter store is used to identify incoming plugins as updates such that older versions can be evicted from memory.

Loading takes place in four steps, as shown in Figure 7.6. Firstly, the loader detects whether the specific version of the plugin is already loaded or not and stops further processing if this is the case.
Secondly, if it must proceed with loading, the dependencies of the plugin are deeply verified and loading is aborted if unsatisfied dependencies are detected. Although a plugin’s dependencies have to be satisfied prior to its loading they are not immediately loaded. Instead their loading is deferred until their functionality is actually required. This allows plugins to be loaded faster and to temporarily curb memory usage.

In the third stage the loader retrieves the plugin’s main code archive and libraries from the database to prepare them for loading.

Finally, the loader reads each of the code archives class by class and assigns them to a newly created plugin-specific class loader which it stores in the key-value store. From this moment on the plugin can be used and its required classes are lazily loaded into the virtual machine on demand. The complete implementation of the plugin loader is provided in Appendix D.

Unloading and upgrading The plugin loader allows plugins to be completely unloaded or upgraded at runtime. A plugin upgrade process begins by first unloading the previously loaded version of the plugin it is replacing, subsequently followed by loading of the new plugin. LabBack takes advantage of the JVM’s class loading and garbage collection mechanisms to allow plugins that are actively executing to continue undisturbed even during an upgrade.

To unload a plugin, the loader simply removes its reference to the class loader for the previous version, making it (and thereby the entire plugin) garbage collectible from its perspective. If however an instance of the execution login that is actively running exists, the scheduler maintains a reference to it, thereby making it uncollectible by the garbage collector. As the job completes the scheduler relinquishes its reference to the execution logic instance and the entire tree of instances and classes is trashed. We illustrate this case in Figure 7.7 where black lines indicate current plugins and references and grey lines indicate replaced but still active plugin data.

It is worth noting that it is possible to momentarily have multiple distinct versions of the same plugin active at the same time without disturbing the execution of active jobs.

Execution instantiation Instantiation of plugin-provided execution logic implementations falls under the responsibility of the plugin loader. As the scheduler activates a

---

1 Depth-first
7.3. WebLab as Client: Contributions to WebLab

WebLab was modified to take advantage of the services offered by LabBack 2.0. This section introduces the modifications made and the new features contributed to WebLab from a user’s perspective.

From a WebLab’s administrator’s point of view, two configuration pages, as highlighted in Figure 7.8, have been contributed: Backends and Languages. These are discussed in the following paragraphs.

7.3.1 Backends

LabBack 2.0 eliminates the need for WebLab and LabBack to be hosted on the same server and introduces the possibility for WebLab to utilise multiple LabBack 2.0 deployment trees. This adds another dimension to the dynamism of WebLab with respect to the types of programs and evaluations it can run.

As part of the integration of LabBack 2.0 as a service provider in WebLab, an user interface to manage and configure used LabBack 2.0 instances has been implemented. Figure 7.9 shows this user interface where a single LabBack deployment instance is
already set up. The interface allows administrators to configure and monitor multiple LabBack deployments. Each deployment is identified by the unique URL of its root LabBack instance and provides a set of job types. The job types are not statically configured in WebLab but instead are regularly updated from this root LabBack instance. When needing to execute and evaluate programs, WebLab dynamically determines a LabBack instance that provides the necessary job types and issues a scheduling request to that instance.

WebLab continuously monitors the **health** of LabBack instances that it knows and automatically disables and reenables them in its configuration based on their availability. This allows it to transparently switch over execution to available LabBack deployments and ensure service availability even in the case of failures. WebLab records availability and events for each instance and shows logs and availability summaries to administrators.

### 7.3.2 Languages

Prior to LabBack 2.0 language configurations in WebLab were hardcoded. Configurations consisted of hardcoded language names and solution and unit test templates. This was acceptable when LabBack 1.0 was used, as any change in the processing capabilities of LabBack required a recompilation and redeployment of WebLab anyway.

With LabBack’s new support for dynamic feature evolution, WebLab can make language configurations dynamic as well. Administrators can now use the interface shown in Figure 7.10 to add, configure and monitor the health of configurations.

Each configuration entry serves as a soft-link between the language-specific processors located on LabBack, syntax highlighting modes for the Ace editor and code and test templates. Upon the creation of a new assignment WebLab offers the educators a number of assignments types as alternatives. These consist of statically de-
7.4 Summary

This chapter has presented important highlights of LabBack 2.0’s implementation and the improvements it induced in WebLab upon integration.

This chapter begun with an introduction to the application-wide benefits of implementing web services using WebDSL. Section 7.2 presented the anatomy of LabBack 2.0’s plugins by means of an example, serving the dual purpose of explain-
ing plugin structure and guiding future LabBack plugin developers. Subsequently, Section 7.2.2 presented the algorithmic implementation of a dynamic plugin loading mechanism which allows runtime deployment, upgrading and unloading of plugins. A complete implementation in Java is included in Appendix D.

Section 7.3 presented the improvements to the usability and sustainability of Web-Lab resulting from the implementation of support for LabBack 2.0. This illustrated how LabBack dynamic feature-set facilitates and motivates clients to relinquish static configurations and allow the dynamic evolution of programming assignment types and their evaluation strategies.
Chapter 8

Evaluation

8.1 Goals and Structure

The goal of this evaluation is to assess the extent to which LabBack’s design and implementation meet the goals of securely, robustly and scalably hosting automatic compilation, execution and evaluation of guest programs. LabBack’s correctness, robustness and security are evaluated separately from its scalability.

The methodology used for the first part of the evaluation is execution of a previously acquired large dataset of real student-written programs. The large dataset is the result of the case study of WebLab and LabBack 1.0 during an actual programming course at the Delft University of Technology, as presented in Section 2.2. The evaluation consists of comparing results produced by LabBack 2.0 against those produced by LabBack 1.0 when executing the programs in the dataset. The correctness of the results produced by LabBack 1.0 was previously verified by the students taking part in the course who used WebLab and LabBack 1.0 for a period of 6 months. The aim of the evaluation is threefold:

1. Evaluate the ability of LabBack 2.0’s core components to contain and control misbehaving programs without suffering crashes
2. Evaluate the correctness of LabBack 2.0 and of the Scala, JavaScript and C processors as plugins
3. Verify the sanity of the services provided by LabBack 2.0 to its clients and thus the correctness of modifications made to WebLab

The scalability goal is evaluated by performing multiple simulations of the load distribution algorithm under different deployment structures. The simulations are driven by virtual workloads with similar timing characteristics to those in the dataset. The aim of this evaluation by simulation is threefold:

1. Evaluate the effectiveness of the load distribution algorithm at balancing job waiting times with the number of deployment instances active simultaneously
2. Prescribe general advice for LabBack’s production engineers with respect to the structure of deployment trees
3. Provide production engineers with a simulation tool to aid the decision of deployment structures
8.2 Evaluation of Correctness, Robustness and Security

LabBack’s and its plugins’ ability to correctly execute and grade students’ programs is evaluated against a large data set. The correctness of the results is determined by comparison against the results produced by LabBack 1.0 against the same dataset and conclusions are drawn.

8.2.1 Dataset description

The large dataset used for this evaluation consists of program source code collected during the 2012 class of the Concepts of Programming Languages course at Delft University of Technology. The dataset consists of programs written using the Scala, JavaScript and C programming languages and are believed to fully exercise LabBack’s functionality with respect to program evaluation and plugin contributions. One could claim that this dataset is not representative as it consists of many different solutions to a fixed set of questions. Increasing the variety of input programs would, however, not alter the confidence in LabBack’s execution correctness but would instead target the correctness and effectiveness of the evaluative method used to grade the programs.

The dataset is believed to be correct, except for a number of executions of C programs that received higher grades than desired due to a bug in the unit testing mechanism that was later remedied. These C programs correspond to solutions to tutorial assignments whose grades do not determine the final course grade and for which students did not complain about the abnormally high grades.

The programs primarily consist of students’ answers to tutorial assignments, graded assignment and assignments from three digital exams. Programs are written in either Scala, JavaScript or C. The programs also include answers to the same assignments created by teachers (1 teacher) and teaching assistants (6 assistants). An unknown number of programs consist of solutions to the same assignments created by students not enrolled in the course and by third-party individuals at the TU Delft exploring WebLab capabilities.

In total there are 98,834 program executions in one of 9 different states, as supported by LabBack 1.0. Approximately two thirds consist of programs that have never been executed. This is caused, in part by intentional duplication of assignment collections in WebLab, and in part by WebLab preparing execution entries immediately upon a students’ opening of an assignment. Some of the jobs are left in non-final states. This is caused by WebLab’s old behaviour of only updating executing results from LabBack as long as the requesting students’s webpage is active. A client closing a page during a job execution results in a job whose state is not updated.

Of interest for verification of execution correctness are the jobs (executions) that have been started and are in a final state. Additionally, to keep the size of the experiment tractable, only executions corresponding to evaluations against specification...
8.2. Evaluation of Correctness, Robustness and Security

Figure 8.1: Distribution of job states and languages in evaluation dataset. Values in \textit{Executed} column are divided by 100.

tests\footnote{Specification unit tests are used to evaluate the correctness of students’ programs, as described in Chapter \ref{ch:specification}.} are included in the test. This has the advantage of limiting the dataset to unit tests that are created by the instructors, and thus are not malicious. The data set is migrated from the old (LabBack 1.0) state model to the new state model (LabBack 2.0) and is filtered to contain just the executions that meet the above criteria. Executions corresponding to C programs whose execution results are known to be flawed are also eliminated.

This results in the dataset used for this evaluation containing 14,183 jobs. Figure 8.1 summarises the execution states of jobs in the resulting dataset, plotted per each of the three programming languages. Prevalences of compilation failures, timeouts, crashes and successful executions are 1.25\%, 0.22\%, 0.87\% and 97.67\%, respectively. The overall prevalences of programs written in Scala, JavaScript and C are 72.12\% (10,229), 19.65\% (2,787), 8.23\% (1,167), respectively.

8.2.2 LabBack 2.0 experiment setup

Recall the evaluation goals stated in the first section of this chapter. Since the experiment is not relevant for scalability concerns a simple setup of one machine running LabBack 2.0 and one machine running WebLab is used.

A command was added to WebLab which places requests for job executions to LabBack 2.0 in a mechanism identical to the action triggered when users press the \textit{Run Spec Test} button. This command causes WebLab to schedule 100 jobs at a time from the dataset until all of the 14,183 jobs are executed again.

After scheduling the jobs WebLab repeatedly queries the LabBack 2.0 instance for job results until all jobs are completed. Results from the executions are stored by WebLab in its database and are later retrieved for analysis. To preserve both existent and new results, a separate database table is used to store both. This ensures that results can be compared and are in the same place.
Figure 8.2: Differences between original and new grades in the failure cases.

The analysis part of the evaluation consists of measuring the job deviations in the final states they achieve and in the ratio of failed to total test cases by comparing each individual job’s properties in the reference and resulting datasets.

### 8.2.3 Results

All 14,138 jobs were executed on LabBack 2.0 and the results are compared with those produced by LabBack 1.0. Of all the executions 97.32% produced the exact same execution state and student grade. Conversely, only 2.98%, corresponding to 424 executions, did not. No LabBack crashes or hangs were encountered during the execution and the mean service time per job was approximately 1.5 seconds.

**Validation per language** Of the total programs executed for the Scala, JavaScript and C languages validation failures had the 2.03% (208), 5.56% (155) and 5.22% (61) occurrences, respectively. Programs from each group were manually examined and revealed no anomalies suggestive of the cause of the validation failures. Results of a further investigation into the cause of these failures are presented below.

**Failures with different grades** From the 424 validation failures cases only in 22 of those did LabBack produce different grades for the students. This corresponds to a difference in grade in 0.16% of the total programs evaluated. Figure 8.2 summarises, per language, the differences between the original and new grades. Of particular interest is that in the Scala and JavaScript cases the grade validation failures are packed in particular questions, as summarised in Figure 8.3. This correlation is weaker but also present in the C cases. These correlations are suggestive of two facts:

1. LabBack appears to be deterministic in producing non-deterministic results
2. Some questions and, in particular, their specification tests appear to be defective and certain programs trigger these defects
8.2. Evaluation of Correctness, Robustness and Security

<table>
<thead>
<tr>
<th>Language</th>
<th>Question</th>
<th>Failures</th>
<th>Grade difference</th>
<th>µ</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>Word Frequency</td>
<td>2</td>
<td></td>
<td>0.19</td>
<td>0.032</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Inheritance</td>
<td>4</td>
<td></td>
<td>-0.20</td>
<td>0.063</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Classical Inheritance</td>
<td>1</td>
<td></td>
<td>-0.42</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>Algebraic Data Types</td>
<td>5</td>
<td></td>
<td>-0.10</td>
<td>0.088</td>
</tr>
<tr>
<td>C</td>
<td>Simple arrays</td>
<td>3</td>
<td></td>
<td>-0.47</td>
<td>0.377</td>
</tr>
<tr>
<td>C</td>
<td>Logical proposition evaluation</td>
<td>2</td>
<td></td>
<td>0.14</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>Hello world on heap</td>
<td>2</td>
<td></td>
<td>-0.50</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>Fortytwo</td>
<td>1</td>
<td></td>
<td>-1.00</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>Factorial with pointer</td>
<td>1</td>
<td></td>
<td>-0.75</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>Linked List insert</td>
<td>1</td>
<td></td>
<td>-0.67</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Figure 8.3: Clustering of grade validation failures into questions. Mean and standard deviations of grade differences are included.

The latter appears to be most prevalent in the case of C programs and has a straightforward explanation. The unit tests in question rely on comparing the reference value (or structure) with the value obtained after dereferencing a pointer returned by the function being tested. Values referred to by the pointer may be unpredictable if the pointer references stack-memory instead of heap-memory. At the same time, because the values stored in memory are not typed, allocated and unallocated memory may be coerced by the test suite into expected structs producing completely unpredictable data structures. Why this problem affects only those questions mentioned before is not certain. Intuitively one may postulate that some of these questions are simple enough that unit testing does not receive sufficient attention and the remaining ones are so difficult that the unit tests contain bugs. This problem is thought to decrease as programs increase in size, and thus in the amount of allocated and deallocated memory, and test results are expected to converge towards a high failure ratio. In other words, as programs increase in size, the likelihood of students getting lucky with bad memory management decreases.

Particularly in the case of C programs, the suggestion is that unit tests are not a good grading method and that alternative testing avenues must be investigated.

Failures with same grade Of the total 424 experiment validation failures, 402 executions resulted in the same grade but different execution final states. The three measurements corresponding to the final states are considered: (1) deviations in compilation failures, (2) deviations in timeouts and (3) deviations in runtime crashes.

Firstly, executions that deviate in the compilation status may indicate defects in LabBack’s compilers. There are 177 compilation failures and 194 compilation successes in the 402 validation failures excluding timeouts. No deviations were found in the outcome of the compilation, indicating that failures are not caused by new defects
in LabBack’s compilers.

Secondly, deviations in timeouts may indicate variations in LabBack’s ability to deal with high loads or in its exigence towards misbehaving programs. This however is a weak benchmark because recreating the original execution contexts (loads, servers, delays) is impossible. Of the failures, 31 input executions are marked as timed out and 26 of them preserved this status. All of 5 remaining executions resulted in crashes. Manual examination of the stack traces of these five executions reveals that they crashed due to stack overflow errors. This is compatible with the facts that (1) in LabBack 2.0’s experiment environment jobs are allowed to execute for longer before cancellation and (2) that the JVM’s stack size setting is lower than in LabBack 1.0’s production environment.

Of the set that failed validation, 49 are not marked as timeouts on input but on output result in timeouts. Of these, 40 were executed successfully in the input dataset but were subsequently cancelled in the experiment. All have zero successful tests in the input. Manual examination of a subset of these revealed inefficiently incorrect programs. No correlation exists between the occurrence of timeouts and specific language processors. LabBack’s end result of executing these is consistent in the resulting grade, however the variations in runtime success states indicate that LabBack’s experienced load plays a role in its success of executing very inefficient programs. The validation failure of these programs is thus attributed to the experiment system being momentarily overloaded. LabBack was configured to run two programs concurrently but the machine only had two CPU cores, thus LabBack’s control threads had to be weaved with those of the programs, resulting in less CPU time for the guest programs. This result validates the advice of Chapter[5] that LabBack should be configured to execute \( N - 1 \) concurrent programs where \( N \) is the number of CPU cores on the host machine. This also suggests that LabBack should assume that this is the default setting; a trivial change to the scheduler.

Thirdly, deviations in runtime crashes of programs can be indicative of variations in LabBack’s exigence and robustness against guest programs. All of the 17 programs that are crashed on input are crashed on output, and none of the 177 successful executions on input crashed on output. Out of the entire dataset LabBack dealt with 123 programs that are marked as crashed on input. These results confirm that LabBack is constant in its robustness against malfunctioning guest programs.

Security evaluation

It can be argued that the dataset does not constitute a relevant sample to evaluate LabBack’s security against attacks via guest programs. This claim is fundamentally correct because the dataset is unlikely to include code that attempts to break-in since it only consists of the final programs submitted by students. However, since LabBack 2.0’s security model is largely based on that of LabBack 1.0, and since LabBack 1.0 has executed all students’ intermediate programs without suffering security violations, it can be assumed that LabBack 2.0 is in fact secure against guest code attacks. The counter-claim to this assumption is that the authors of those intermediate programs are mostly first year computer science students and do not possess sufficient skill to carry out sophisticated attacks. This may be true, but one may recall that LabBack 1.0’s security was tested by experienced programmers prior to its deployment in the case
8.3. Evaluation of Scalability and Load Distribution

study. There is thus reason to believe that LabBack 2.0 is reasonably secure against attacks via guest programs.

LabBack’s security from attacks on the communication pathway is not evaluated and therefore the effectiveness of the protection mechanisms in place is not tested. Although there is a definite need for this evaluation, these mechanisms are all industry-accepted and are expected to provide sufficient protection. A complete evaluation of these is impossible as even an evaluation employing the skills of security experts will miss certain attack vectors. In fact LabBack’s security from communication attacks is certain to be incomplete as no completely secure software systems can exist. However, it is believed that LabBack is as secure as can be reasonably expected.

8.2.4 Conclusion

In order to evaluate LabBack 2.0’s correctness, robustness and security, 14,138 previously executed jobs were rerun and produced identical results in 97.32% of the cases. Primarily the experiment revealed that LabBack produces different grades for students in only 0.16% of executions. The analysis of these cases revealed that LabBack’s execution is in fact correct and grade differences had two fundamental causes:

1. Specification unit tests are sometimes defective
2. Unit testing is not a good method of evaluation for C

With the highest degree of confidence this evaluation leads to the following conclusions:

1. LabBack 2.0 is correct in its executions
2. LabBack cannot guarantee deterministic results for programs that themselves behave non-deterministically
3. LabBack should assume a default configuration of \( N - 1 \) concurrent executions where \( N \) is the number of CPU units on the hosting system
4. Correctness of resulting grades is highly dependent on the grading method that is used

With an acceptably high confidence this evaluation suggests that LabBack is very robust against malfunctioning guest programs. Most importantly, although the final state of these programs may vary, LabBack ensures the correct grade is produced.

With confidence the experiments confirmed that LabBack is secure against malicious programs.

8.3 Evaluation of Scalability and Load Distribution

Chapter 6 described LabBack’s mechanism for distribution of jobs in a multi-instance deployment. This section presents the evaluation of this mechanism in a variety of deployment scenarios. Recall from Section 6.6.4 that the goal of load distribution in the case of LabBack is to balance job queuing times while minimising the number of instances in a deployment tree that are actually used. In effect this section evaluates the ability of the load distribution mechanism to balance usability with cost.
The evaluation is by performed by means of simulations. Simulation was chosen as the evaluative method instead of real experiments for the following reasons:

1. The load distribution mechanism is not fully implemented in LabBack 2.0 yet. The capacity of a single server has so far been sufficient.
2. A deployment simulator is invaluable to a deployment engineer when designing LabBack deployments.
3. Experimental real deployments require many unused machines to be harnessed which are not available.
4. The deployment and configuration time required to evaluate load distribution under many different contexts is very high.
5. The dataset available is heavily biased towards the timing characteristics of programs written in Scala, JavaScript and C and thus has insufficient variability.
6. Simulation allows for careful control of the execution environment and proper observation of the behaviour of every processing instance on an individual basis.
7. Simulators have the capability of simulating various job arrival rates, which is not possible in physical experiments
8. Formulating general advice for LabBack production engineers requires experimentation/simulation of the load distribution mechanism under varying deployment trees and varying job characteristics.

The purpose of this section is to evaluate the effects of LabBack’s load distribution on the following four parameters:

1. Job queue lengths and queueing times
2. Maximum job capacity
3. Network convergence speed
4. Number of machines required

This evaluation begins with a description of the simulation setup, the simulator and experiment methodology, followed by a presentation of the simulation results. The evaluation concludes with a set of recommendations for deployment topologies.

8.3.1 Simulation setup

For this evaluation the load distribution mechanism described in Section 6.6.4 was implemented in NetLogo [69]. NetLogo allows simulation of agent-based systems using a flexible programming language and a rich UI.

The simulation interface built is shown in Figure 8.5. It allows operators to alter simulation parameters, run the simulation and simultaneously monitor the effects of loads on the deployment tree. The interface shows LabBack instances as round nodes in the deployment tree and connections between them as edges in the tree. The pentagon-shaped node corresponds to the root LabBack instance that communicates directly with the clients. LabBack instances that are active are represented as black nodes and instances that are available but paused are represented as grey nodes. The length of edges in the tree have no attached meaning.

The simulation interface allows the operator to control a variety of simulation parameters, as Figure 8.4 summarises. Most importantly it allows tuning parameters of
### Variable name | Description
--- | ---
jobs-per-second | Job arrival rate in jobs/sec
avg-duration-second | Mean duration in seconds for jobs with normally distributed duration
dev-duration-second | Standard deviation of normally distributed job duration
arrivals-enabled | Enable or disable job arrivals
num-nodes | Number of nodes of the constructed processing network
build-by-height | Enable network arrangement with the goal of reaching a maximum path length, or disable to build a network with a normally distributed node degree
tree-height | Target path length to reach when constructing networks by height
mean-degree | Mean of normally distributed tree degree when constructing networks by degree
dev-degree | Standard deviation of normally distributed tree degree when constructing network by degree
avg-capacity | Mean of normally distributed processing capacity per node
dev-capacity | Standard deviation of normally distributed processing capacity per node
thresh-badness | Tree badness threshold used to determine execution nodes
sleep-delay-second | Idle time in seconds after which nodes can be paused
remote-scheduling | Enable or disable distribution of jobs in the network

Figure 8.4: Summary of variables for simulation control.

the deployment tree, such as node degree or tree height, and of parameters regarding dynamics of the jobs, such as their arrival rate and their duration. Parameters pertaining to job dynamics can be altered while the simulation is active. The simulation interface can construct trees using one of two algorithms which together cover a wide variety of topologies. Firstly, trees can be constructed with the goal of reaching a maximum tree height. This allows experiments with trees that are very deep and out of balance. Secondly, trees can be constructed to have a certain mean and standard deviation of node degree. This allows more balanced (star-shaped) trees to be simulated. Each node in the tree is allocated a certain capacity according to a mean and standard deviation. This capacity is equivalent to the number of concurrent jobs it can execute.
Figure 8.5: User interface of LabBack’s deployment simulator.
8.3.2 Measurement method

It is of particular interest how the above vary with the following parameters:

1. Deployment topologies of varying tree depths and node degrees
2. Deployment size
3. Job arrival frequency
4. Job duration

A number of experiments are performed under controlled variation of the above parameters and results are discussed. All simulations are run at the speed of 100 simulation ticks per second of time and all nodes have 16 CPU cores of capacity. All simulations use constant job arrival rates and vary this rate twice during the simulation, the final job rate being more than the network is expected to handle. Experiments are run for 1000, 3000 or 5000 ticks corresponding to 10, 30 and 50 seconds of real network time.

Experiments begin with the network at rest, i.e. no jobs at all. The arrival rate is subsequently increased in three steps to:

1. near maximum expected capacity
2. maximum expected capacity
3. above maximum expected capacity

Throughout all simulations the following parameters are measured:

1. Job arrival rate
2. Number of running and pending jobs
3. Awake capacity measured in CPU cores
4. Mean and standard deviation of queue lengths normalised per awake capacity
5. Mean and standard deviation of waiting times incurred by jobs recorded at their completion

The number of running and pending jobs (2) indicate whether the amount of work in the system is increasing or is being kept constant. The ideal result is that the number of pending jobs stay constant or decrease when the arrival rate is stable at full capacity of the network.

The mean and standard deviation of queue lengths (4) indicate how well work is divided over the awake capacity of the network. Upon convergence, and at constant job arrival rates less than the network capacity, the ideal result of a load balancing algorithm is a small mean queue length and near-zero standard deviation. In the case of LabBack’s greedy work-packing mechanism it is acceptable for the standard deviation to be within the mean queue length. This is acceptable and expected because the mechanism greedily assigns all non-scheduled jobs to the node with the least tree badness, under a threshold, thus potentially overloading that node.

The waiting times incurred by jobs (5) upon their exit measures the time completed jobs have spent waiting before beginning execution. Because the measurement is performed upon completion of the nodes, the values are delayed precisely by the amount of time spent waiting. The ideal result for a load balancing mechanism under
8. Evaluation

![Network topology](image)

(a) Network topology

![Queue length normalised by capacity](image)

(b) Queue length normalised by capacity

![Jobs in system and awake capacity](image)

(c) Jobs in system and awake capacity

![Waiting times (on exit)](image)

(d) Waiting times (on exit)

Figure 8.6: Simulation results for control experiment.

Constant load below maximum capacity is close to zero. However, for a load packing algorithm such as LabBack’s, which attempts to balance waiting times with the number of utilised machines, the acceptable mean waiting time lies within the duration of the job. In other words it is deemed acceptable for jobs to incur a waiting time equal to their duration. The ideal standard deviation of the waiting times is around the mean waiting times, thus corresponding to some jobs that incur zero waiting time and some that incur double waiting time.

The awake capacity (3) of the network relates how many CPU cores of the network are awake and is measured as the sum of all capacities of all nodes that are awake. The ideal result is a low value such that the queue lengths and thus the job waiting times are acceptable.

One control experiment and seven other experiments are performed. The experiments correspond to deployment consisting of 1, 10, 30 and 100 nodes.

### 8.3.3 Control experiment

A control experiment is performed to validate that the simulation engine correctly simulates job executions. The deployment tree consists of the single centrum node with a capacity of 16 concurrent jobs. At this processing capacity and a job duration of two seconds it is expected that an influx of 8 jobs/sec is the maximum capacity. The network is started and immediately 8 jobs/sec begin to arrive. After five seconds the load is increased to 10 jobs/sec. Figure 8.6 shows the results. It can readily be seen that the network stabilises as soon as sufficient jobs arrive in order to fill the capacity. As soon as the arrival rate is increased beyond capacity the lengths of queues increases.
8.3. Evaluation of Scalability and Load Distribution

Figure 8.7: Simulation results for 10 node star topology.

unboundedly and consequently the waiting times increases unboundedly. The single node simulation behaves according to expectations.

8.3.4 10 nodes

Three experiments are performed with three networks of size 10 of varying topologies.

Star topology

A network with a star topology consisting of 10 nodes directly linked to the centrum is constructed and simulated. The network is expected to handle a maximum job arrival rate of 80 jobs/sec. The simulation begins with the network at rest, the capacity is immediately increased to 20 jobs/sec. After 10 and 20 seconds the influx of jobs is increased to 80 jobs/sec and 150 jobs/sec, respectively. Figure 8.7 shows the results of the simulation. Corresponding to the three arrival rates three observations are made:

1. The network stabilises quickly (approximately 2 seconds) after an increase in work from 0 to 20 jobs/sec. The awake capacity corresponds to the number of running jobs. The queue lengths are short (under 5 jobs) and equally distributed. The mean waiting times are very low (under 0.2 seconds per job) but are spread out with a standard deviation of twice this mean. Some jobs therefore incur waiting times of one second.
2. The network copes very well with an increase in load up to its maximum capacity of 80 jobs/sec and queue lengths are only increased for 5 seconds after which they return to lower values. During the adjustment period the waiting
8. Evaluation

![Figure 8.8: Simulation results for 10 node tree of depth 6.](image)

Times increase to a mean of 0.5 seconds and quickly decrease to 0.3 seconds subsequently. Of fundamental importance is that at maximum load the awake capacity is 120, well under the total of 160, without the waiting times and queues increasing. The mechanism thus proves to balance acceptable job service times on three quarters of the available capacity.

3. At overloading arrival rates of 150 jobs/sec all of the available capacity is awakened within two seconds and the queue lengths have a low standard deviation compared to their mean values. When overloaded the scheduler can therefore fully utilise all available capacity equally.

Tree topology

A tree of 10 nodes is constructed with the goal of it reaching a maximum height of 6 nodes, as shown in Figure 8.8a. The arrival rate of jobs is varied identically to the simulation of the star topology. Result are shown in Figure 8.8 and two observations can be made:

1. At the expected maximum load, after the network stabilises (within 5 seconds) the queues are short and well packed. The exit waiting times return to approximately 0.5 seconds within 10 seconds of the increase in capacity. Only three quarters of the full capacity is utilised.

2. At overloading arrival rates all available capacity is utilised and queue lengths increase unboundedly. The queue lengths are not equally distributed and thus the work load is not evenly distributed. This is caused by the greediness of
8.3. Evaluation of Scalability and Load Distribution

Figure 8.9: Simulation results for 10 node chain topology.

the algorithm which schedules all incoming jobs on a single node, thus further overloading that node. In effect the algorithm overloads nodes in a round-robin fashion.

Linear topology

Figure 8.9 shows the simulation results for a linear network in identical situations as those of the star and tree topologies. On normal and full loads the algorithm performs well. At overloading arrival rates the same pattern as previously repeats where queue lengths are not equally distributed to the available capacity. This is caused by a combination of the algorithm’s greediness and the fact that tree badness values decrease higher up in the tree. The conjecture is that normalising the tree badness to the height of the tree would partially resolve this issue.

8.3.5 Topologies of size 30

Two topologies of size 30 are separately simulated. At this size and 16 simulated CPU cores per node, the expected capacity of the network is of 240 jobs/second. The simulations begin with the network at rest, subsequently increasing the load to 150, 240 and 300 jobs/sec at simulation times of 0, 20 and 30 seconds. The simulation is run for a duration of 50 seconds.
8. Evaluation

(a) Network topology

(b) Queue length normalised by capacity

(c) Jobs in system and awake capacity

(d) Waiting times (on exit)

Figure 8.10: Simulation results for 30 node tree of degree 3.

Tree of degree three

A tree of 30 nodes is constructed such that node degrees are normally distributed with an expectation of three and a standard deviation of 0.5, as shown in Figure 8.10a. Three observations are made with respect to the arrival rates of jobs:

1. Upon increase of arrival rate to 150 jobs/sec the network converges very quickly (approximately 5 seconds) utilising approximately half of the available capacity. The queue lengths are low (approximately 2 jobs) and well distributed. The waiting times upon exit return to low values (0.2 seconds mean) after approximately 10 seconds, however their variance is relatively high. Jobs routinely experience waiting times of up to 0.7 seconds.

2. Upon increase of arrival rate to full expected capacity the network raises the awake capacity and converges immediately. Job waiting times and queues do not change. As the network stabilises the awake capacity decreases to two thirds of full capacity.

3. Upon increase to an expected overloading job arrival rate of 300 jobs/second the mechanism increases the awake capacity in the network and within 10 seconds the mean queue length decreases. The standard deviation of queue lengths is not a relevant measure here since it is skewed by the scheduler’s greediness in batching jobs at individual nodes. At this arrival rates the mean waiting time incurred by jobs is still at 0.4 seconds and peaks up to 1.2 seconds are common. The scheduler mechanism therefore shows that it can handle overloads without having to immediately wake up all computing nodes.
8.3. Evaluation of Scalability and Load Distribution

(a) Network topology

(b) Queue length normalised by capacity

(c) Jobs in system and awake capacity

(d) Waiting times (on exit)

Figure 8.11: Simulation results for 30 node tree of degree 5.

Tree of degree five

A tree of 30 nodes constructed such that all nodes have a degree of 5 was simulated, as shown in Figure 8.11a. Results are similar to those of the previous simulation, shown in Figure 8.11. The primary observation is that the scheduling mechanism is able to cope better with maximum and overloading loads in the taller tree of the previous simulation.

8.3.6 Topology of size 100

A tree consisting of 100 nodes is constructed with a normally distributed node degree with expectation 8 and standard deviation of 4, as shown in Figure 8.12a. Its performance is simulated in two separate experiments varying both the job arrival rates and the job duration.

Fixed job duration

Firstly, the simulation is started with the network at rest and job arrival rates are increased to 700, 900 and 1,100 jobs/second at simulation times 0, 20 and 30 seconds, respectively. All arriving jobs have a fixed execution duration of two seconds. At this network size and job durations the expected maximum processing capacity of the network is 800 jobs/second. Corresponding to the increases in arrival rates, three observations are made:

1. Upon increase of job rate to 700 jobs/sec, the network converges quickly and
queue lengths begin to decrease immediately. Within 7 seconds the overall number of pending jobs begins to decrease as well, corresponding to awakening of processing nodes. Waiting times on exit stabilise to a mean of 1.5 seconds within 25 seconds.

2. Upon increase of arrival rates to the full capacity of 800 jobs/second the network takes no action. Neither the queue lengths nor the waiting times are changed and the network can handle the full load.

3. Upon increase of arrival rates to the overloading 1,100 jobs/second the network awakens the remainder of the sleeping nodes and converges quickly. After stabilisation of the network the mean job waiting times are approximately two seconds and a sudden decrease in deviation is recorded.

**Normally distributed job duration**

The same topology of 100 nodes is evaluated under varying loads and job execution durations. Jobs durations are modelled as being normally distributed with an expectation of 2 seconds and deviation of 4 seconds. Thus jobs routinely require 6 seconds of execution time. The simulation starts with the network at rest and the job arrival rate is increased and again decreased to 500, 800 and 500 jobs/second at simulation times of 0, 20 and 30 seconds, respectively. Results are shown in Figure 8.13 and three observations corresponding to the arrival rates are made:

1. Upon the initial increase of arrival rate to 500 jobs/second the network stabilises quickly. The initial peak in queue lengths is lower than previously and is caused
8.3. Evaluation of Scalability and Load Distribution

Figure 8.13: Simulation results for 100 node tree of degree 8 and varying job durations.

by the variation in job duration. The network does not have time to converge before the arrival rate is increased again, however it is observed that by 20 seconds the queue lengths had dropped to a mean of five. This strongly indicates that the mechanism was able to catch up with the pending jobs it had built up while it was awakening sleeping nodes.

2. The system reacts quickly to the increase in jobs to 800 jobs/second corresponding to an overloaded configuration. The effect of this is however masked by a slow adaptation to the initial job increase, as can be seen in Figure 8.13c.

3. Upon return of the arrival rate to normal values the system makes quick progress in decreasing the amount of queued work. By the end of the simulation the network has nearly converged on a mean waiting time of less than one second. As jobs with various durations are present in all the queues, it is normal that the standard deviation of the waiting times is relatively high.

8.3.7 Conclusion

The mechanism for load distribution based on the tree badness heuristic was evaluated using 8 simulations. The simulation results show that it performs well under a variety of conditions and that it is able to balance acceptable job waiting times with the amount of nodes that are used. Based on these results the following can be concluded about the performance of the load packing algorithm:

1. It is best suited for trees that are relatively balanced.
2. Jobs of varying durations can be scheduled efficiently.
3. In general the network converges quickly after changes in load arrival rates.
4. The network adapts quicker in the presence of higher amounts of work than in low load situations.
5. Even under full loads certain parts of the network can be asleep without increasing the waiting times to unacceptable levels.
6. The scheduling mechanism reacts slowly to very large increases in sustained job arrival rates.
7. The greediness of the scheduling leads to an increased standard deviation of job waiting times.
8. The scheduling mechanism utilises each subtree to full capacity before scheduling jobs on new subtrees.

Although these conclusions are based on simulations alone, there is an expectation that real experiments would lead to similar conclusions. In a real scenario, however, nodes have non-zero startup and teardown time and thus an increase in network convergence time is expected.

Based on the simulations and results the following three recommendations for deployment topologies can be formulated:

1. Low threshold values for the tree badness ensure low waiting times for jobs and conversely, high threshold values ensure distribution of jobs to asleep subtrees only during excessively high loads. Medium values thus balance waiting times with the number of awake processing nodes. The threshold value therefore tunes the balance between deployment cost and jobs’ waiting times.
2. The preferred deployment topology is a multi-level star topology.
3. As trees and subtrees increase in size the threshold value used for target selection should decrease if deeper subtrees are preferred as processing targets.
4. Shallow networks are better suited for jobs with low mean duration but high standard deviation.
Chapter 9

Related work

9.1 Automatic Grading

Chapter 2 gave an overview of existing Massive Online Open Course (MOOC) platforms. The various MOOC platforms discussed offer online tuition and some offer online programming and automated graded possibilities. None of the platforms, including popular ones such as Udacity, Coursera or Khan Academy are reusable either as products or as services. Even more so, all of the platforms that support online programming and automated grading of programs do so only for one or two programming languages. The platforms that do support grading are believed to evaluate software either by matching on a program’s output or by unit testing. A more comprehensive discussion of MOOC platforms is given in Section 2.1.

Automatic grading systems for students’ programs have been developed for decades, with the first in 1960 [19]. In 1965 an automatic grader for ALGOL programs was created at Stanford University [23]. This automatic grader required students’ code to call a specific procedure provided by the grading software. The grading procedure exercised the students’ implementations with randomly generated inputs, guaranteeing that a student’s implementation would never be exercised with the same inputs more than once.

A successful automated grading and courseware management system has been the Ceilidh System [5]. It could evaluate students’ programs either interactively or over night, and supported programming languages such as C, C++ and Pascal. Security of the system against students’ programs was a concern and this concern necessitated the careful maintenance of comprehensive audit trails. Ceilidh combined a variety of program evaluation techniques including correctness and source code metrics. Correctness evaluation was performed by comparing students’ programs output to that of an oracle solution, by means of regular expressions.

The Homework Generation and Grading project (HoGG) at Rutgers University [52] can automatically evaluate programs written in Java. It does so by first modifying the students’ source code to change private-defined fields to public, then compiles the program using javac and executes the result. The grade is determined after evaluation

---

9. Related Work

of the output produced by the student program. It is noteworthy that a human grader is involved in the grading process, as well as the fact that HoGG can successfully grade student programs that contain minor errors such as incorrect function names. As an advantage to its restriction to Java programs, HoGG can take advantage of polymorphism to replace defective student-provided functions, thus allowing grading of partial solutions. None of the security concerns associated with running uncontrolled guest programs are documented.

Nordquist [57] describes an automatic grading system for Java programs that runs client-side. It consists of a Java program which is downloaded by the students and is launched by Java Web Start. The program automatically compiles and executes the students’ code and applies a set of criteria to obtain a grade. In this case there are no robustness concerns because a student’s program runs on his own computer. Security concerns with respect to malicious guest code altering the runtime behaviour of the automatic grader are undocumented. Additionally, no precautions are taken to prevent the exposure of the testing suites to the students. While alleviating the robustness concerns, the limitation to Java and the requirement for the student to actually have a full Java Development Kit installed is severe. Nordquist acknowledges that unit testing is not a good method for grading students’ programs and instead relies on examination of the output produced by programs.

It is important to note that LabBack provides none of the grading techniques discussed above. Instead, LabBack is a platform that can support all of the above grading techniques by means of dynamically added and composed plugins. It alleviates the engineering effort of implementing automatic graders and enables them to be effective and secure. Web-CAT [22] in many ways provides similar functionality to LabBack; deployments are, however, limited to single servers which only execute programs one at a time, thus generating long waiting lines for students. A much more complete overview of automatic grading projects is given by Douce et al. [20].

9.2 Web IDEs and Containment of Third-party Programs

Execution of uncontrolled third-party programs raises security and robustness concerns. Chapter 4 presented a comparison of technologies for contained execution of third-party programs, among which SELinux [47] and BSD-jails [38]. Technologies discussed vary in the guarantees of security against malicious programs that they give.

In general, programs written in virtualised languages, i.e. languages executed in virtualised environments, cannot effectively be contained at the level of the operating system. This is due to the virtualisation environment taking actions seemingly unrelated to the executed program. Containment of such programs can thus only be effectively achieved by the virtualisation environment itself, with an approach similar to language-based security as initially proposed by Kozen [41].

Security of programs at the level of the Java Virtual Machine [45] is achieved by means of the Java Security Manager that audits actions performed by executed programs in real-time. Thus, the security manager is effective in limiting the actions taken by hosted software. When a JVM executes multiple seemingly unrelated programs, the security manager is, however, unable to provide robustness guarantees or prevent modification of one program’s data by another program. This can be achieved
with custom dynamic class loading mechanisms [43]. Dynamic class loading mechanisms can also provide support for the runtime evolution of classes. Class loaders can only replace entire classes at once and the problem of schema evolution [3] limits their effectiveness to this latter purpose. The unit of evolution is further refined by Würthinger et al. [77] with a mechanism for dynamic evolution of code in the JVM, that requires modification of the JVM’s online compiler - HotSpot. LabBack relies on custom class loader implementations to provide insulation of guest code and plugin support.

The problems of hosting multiple programs within a single JVM are inherent to all applications that execute third-party programs and to plugin frameworks. Java Community’s Java Specification Request 121 (JSR-121) provides a specification of Isolates allowing contained execution of multiple programs within a single JVM. Although not widely used, propositions based on this specification have been made [2, 16, 29] where communication between Isolates is performed via RPC. Geoffray et al. [26] introduce I-JVM which allows plugin isolation in OSGi, however their approach requires modification of the Java Virtual Machine. An implementation of Isolates compliant with JSR-121 is the Multi-tasking Virtual Machine [34]. Because LabBack does not require guest programs - Isolates - to communicate directly with each other, a mechanism based on dynamic class loading [43] is sufficient and has the advantage of not requiring a non-standard JVM.

Web Integrated Development Environments (IDEs) suffer from the problem of execution containment as well. IDEs that support programming languages that can be executed in the browser, such as JavaScript, alleviate this problem by executing all programs client-side. Recently, Cloud9 IDE [1] has added support for server-side plugins to support language-specific features for other languages. The security concerns regarding execution of guest-programs (the plugins and the IDE user’s programs) and the possibility of information theft (the plugin stealing the IDE user’s programs) are acknowledged but not fully resolved.

Codepad [2] provides compilation execution support for a variety of native languages, including C, C++ and Haskell. The security is strong and relies on ptrace and the Geordi compiler wrapper, thereby disabling or ignoring many system calls. The programs are executed in a chroot jail. The security measures are good for native programs but not viable for programs running in virtualised environment for the reasons previously discussed.

Ideone [3] supports evaluation of programs written in more than 40 programming languages. The security measures are weak and undisclosed. One can easily determine, using guest programs, that programs are executed in a chroot environment with a custom build of the Linux Kernel, that Java and Scala libraries can be overridden at runtime and that the chroot can be escaped. The Online Judge [4] is a plugin for the Moodle Course Management System [21] which uses Codepad for automated grading of students’ programs.

\[^{1}\text{Cloud9 IDE} \text{http://www.cloud9ide.com}\]
\[^{2}\text{Codepad} \text{http://codepad.org}\]
\[^{3}\text{Geordi} \text{http://www.eels.net/geordi}\]
\[^{4}\text{Ideone} \text{http://ideone.com}\]
\[^{5}\text{Online Judge} \text{https://github.com/hit-moodle/moodle-local_onlinejudge}\]
9. Related Work

9.3 Load Balancing and Distribution

Much research regarding load balancing exists. Load balancing algorithms are typically concerned with the even dispersal of workloads among all available processing nodes [61,44]. LabBack’s load distribution mechanism does not aim to evenly distribute work, but instead aims to schedule the work on as few processing instances as possible without causing large delays. This problem is thus a relaxation of the bin packing problem [13]. The problem is relaxed by eventually accepting that capacity of bins may be exceeded, thus leading to a stochastic bin packing problem. Attempts at solving this latter problem are presented, among others, by Kleiberg et al. [40] and Goel and Indyk [28]. However, these approaches require the load scheduler to possess information about the entire network.

In contrast, LabBack’s scheduler relies on the tree badness heuristic which is calculated at every processing node with local information only. The tree badness is not exchanged explicitly between nodes and is instead piggybacked on the result of each job as it propagates upwards towards the root node. This piggybacking on job results is inspired by ants in an ant colony [79]. The tree badness heuristic is inspired and resembles ideas of the cluster pressure heuristic introduced by Pruteanu et al. [60] and that of the cluster density [49]. In contrast to these approaches, the algorithm proposed cannot calculate a tree badness if the network contains cycles. The distribution of workloads based on the tree badness resembles the simple first-fit greedy algorithm for the bin packing problem. LabBack’s scheduler achieves a worse approximation than this algorithm because it greedily schedules all new jobs on the first available subtrees. LabBack’s algorithm is expected to function without this latter greediness equally well however.
Chapter 10

Conclusion

10.1 Summary

Background and Requirements

Chapter 2 presented the case of the 2012 edition of the Concepts of Programming Languages course at Delft University of Technology which motivated the research and engineering effort presented in this thesis. The large number of participating students, the need to alleviate the burden of manually grading programming assignments and exams, the desire to offer more practice material to students and the unavailability of any viable MOOCware platform drove to the creation of WebLab by Dr. Eelco Visser and of LabBack by the author of this thesis. WebLab provides online management of programming courses and LabBack hosts the automatic, secure and robust execution and grading of students’ solutions. By the end of the case study, LabBack’s proof of concept version had performed the equivalent of 20 weeks of manual grading.

The case study generated a large dataset of real-life student programs that is used to evaluate the grading correctness and robustness of LabBack 2.0 - a reengineering of the proof of concept version that can meet the functional and non-functional requirements identified in Chapter 3. LabBack’s functional requirements specify the ability to automatically apply various preprocessing transformations to programs in source code form, such as compilation, and to subsequently execute them and report runtime output and grading results. LabBack must be capable of providing this functionality for a variety of programming languages, such as Scala, JavaScript and C. Two major non-functional requirements arise from LabBack’s function of openly accepting code injections: security and robustness against malicious or malfunctioning guest code. Openly accepting code injections requires LabBack to be protected against malicious or malfunctioning guest code and contradicts the industry trend of closing down all avenues for injections.

Proof of Concept

Chapter 4 analysed the possibility of reusing existent technologies to satisfy LabBack’s non-functional requirements of security, flexibility, maintainability and scalability. A variety of tools and platforms were discussed resulting in the conclusion that greenfield
engineering of LabBack would ensure the highest chance of successfully meeting the non-functional requirements.

Chapter 5 introduced the architecture and implementation of LabBack 1.0 as a proof of concept for secure execution of students’ programs within a single instance of the Java Virtual Machine. LabBack’s solution to security relies on the joint use of the Java Security Manager with a custom class loading mechanism. Topics such as memory management, program isolation and concurrent and interruptible execution scheduling are discussed in the face of the challenges they pose and algorithmic and implementation solutions are presented. LabBack 1.0’s ability to concurrently and securely execute uncontrolled programs was evaluated in a large real-life case study.

**LabBack 2.0**

Results of the case study and recommendations resulting from LabBack 1.0’s evaluation are used in the design and implementation of LabBack 2.0. LabBack 2.0’s architecture, presented in Chapter 6, builds on LabBack 2.0’s validated security model to describe a web application as a platform for the processing and execution of students’ programs under various experiment setups. LabBack 2.0’s programming language and experiment support is extendible at runtime through plugins whose contributed functionalities are dynamically orchestrated to form program execution pipelines. Deployments of LabBack 2.0 can be scaled to multiple instances, possibly hosted in the cloud, that are organised as trees and cooperate to handle higher numbers of program executions than could be supported on single instances. Scheduling of executions across multiple instances utilises the tree badness metric (Section 6.6.4), a heuristic for estimation of loads in subtrees, which allows execution waiting times to be balanced against the number of processing nodes required.

Highlights of LabBack 2.0’s implementation were presented in Chapter 7. The mechanism that allows LabBack 2.0’s runtime evolution of execution pipelines was discussed, and an example illustrating the anatomy of LabBack plugins was presented. Support for LabBack 2.0 was implemented in WebLab, which showed how dynamic execution pipelines allow clients to avoid static configurations for programming assignment types.

Chapter 8 performed two evaluations of LabBack 2.0. In the first instance LabBack 2.0’s ability to correctly compile, execute and grade programs was evaluated. This evaluation was performed by executing all programs in the dataset obtained from the LabBack 1.0 case study. Execution results and grades between the two executions were compared and failures were discussed. LabBack 2.0 generated different grades in only 0.16% of the 14,138 executed programs. It was discovered that all of these failures were caused by defects in the unit tests used for grading, and thus did not indicate defects in LabBack 2.0 itself.

In the second instance LabBack 2.0’s algorithm for job distribution based on the tree badness heuristic for subtree load estimation was evaluated. This evaluation was performed by repeated simulations under various deployment topologies and job arrival rates. Results of the simulations indicate that the tree badness heuristic and its use in the selection of an executing instance are good methods for tuning the balance between the job waiting times and the running-cost of multi-instance deployments.
10.2 Conclusions

**Research Question A**: Is it possible to design and build a software system that uses mainstream technologies to host automatic execution and evaluation of guest programs in an immediate, flexible, scalable, secure and robust way?

This thesis presented the LabBack application which automatically executes and evaluates students’, thus guest, programs. It is implemented in Java and WebDSL (which generates Java source code) and is hosted by web application containers that run on unmodified standards-compliant versions of the Java Virtual Machine. LabBack therefore uses mainstream technologies without modifying them. Execution timeliness depends strictly on the experienced workload and the ability of the host application to manage this. In LabBack’s case, the execution scheduler is responsible for distributing and controlling workloads on the available capacity. LabBack makes only few assumptions about the programming languages that guest programs are written in and about the grading methods employed. By means of the plugin framework, LabBack’s language-specific and experiment-specific functionality is contributed by plugins as needed and is dynamically composed into execution pipelines. The scalability requirement is addressed by supporting multi-instance deployments whose instances may be hosted in the cloud. The execution scheduler can automatically distribute workloads across machines in the deployment tree. The security and robustness concerns in LabBack are addressed by isolating executions of guest programs using a custom class loading mechanism and the standard Java Security Manager. This security and robustness model has been validated during the case study of the Concepts of Programming Language course. This research question can thus be answered positively.

**Research Question B**: How can a system both execute untrusted programs and be secure against them without severely restricting their functionality?

The security threat posed by execution of guest programs has serious implications both to the availability of the hosting application and to the correctness and privacy of other guest programs. LabBack runs on the Java Virtual Machine and achieves its security by combining the use of the Java Security Manager with a custom class loading mechanism. Guest programs are thus contained in two ways. Firstly, the Security Manager prevents them for performing certain potentially malicious tasks such as file I/O. Secondly, the custom class loader prevents data sharing between guest programs themselves and between guest programs and the host system. This in effect only allows guest programs to access data in their own reserved part of the JVM’s heap space.

LabBack currently enforces a number of restrictions on the guest programs. For example, reflection, file I/O and thread manipulation are currently not permitted. With the exception of reflection, there is no strict need to continue forbidding these in the long run. If, for example, a more secure custom implementation of `java.lang.Thread` is used instead of the default one, then thread support can be provided. Relaxation of
these restrictions by providing custom implementations can in theory be provided by LabBack’s plugins without requiring modification of LabBack itself. It is noteworthy that plugins and guest code do not run in the same protection domain and thus plugins, and the language-tooling they wrap, are not restricted in their capabilities.

Of the alternative technologies discussed in Chapter 4, only BSD-jails are capable of the necessary security without introducing dependencies to a particular programming language or significantly increasing running costs. The drawbacks of BSD-jails are that they are notoriously hard to maintain and introduce the platform dependency to FreeBSD or OpenBSD. In general, programming languages that are compiled to native code are hard to secure.

The answer to this research question is therefore that to ensure largely unrestricted but secure execution guest programs should be compiled to virtualised languages such as Java, Scala and C#. Programs can either be directly written in virtualised languages or their compiled forms can be recompiled to a virtualised language, such as the case of C programs which LabBack recompiles to JavaScript and Java.

**Research Question C**: How can a system schedule execution of guest programs to ensure their timely execution while ensuring its robustness against resource-starving programs?

In the case of LabBack, execution timeliness is provided by the execution scheduler which balances the number of concurrent executions on the available capacity. The scheduler ensures that executions cannot exceed a certain maximum duration and aborts executions which attempt to do so. This limits the amount of CPU time that each guest program may take and ensures that system availability is unaffected by, for example, looping programs. LabBack does not currently limit the amount of memory that can be allocated by guest programs. During the case study, memory allocation restrictions proved both unnecessary and hard to achieve in Java. It is typical in the JVM that as heap memory is exhausted an `OutOfMemoryError` exception is raised to thread requesting memory. Because the execution duration of guest programs is limited, guest programs can only exhaust heap space by intensively allocating memory. In the event of an out of memory error it is very likely that they will receive this exception themselves.

The answer to this research question is thus twofold. On the one hand timeliness requires that executions are efficiently scheduled on available capacity without overloading the hosting system and without starving the execution controlling threads of CPU time. On the other hand programs must be limited in the amount of resources that they can use. In many cases limiting their CPU time is significantly easier than successfully limiting memory consumption. Limiting the memory utilisation of programs written in virtualised languages is much easier than limiting native processes. In general, it is easier and more effective to employ security features provided by the programming language than to rely on those provided by the operating system.
10.2. Conclusions

Research Question D: What is a lightweight architecture that supports the runtime evolution of language and experiment-specific functionality in an application that executes third-party software?

Evolution of software functionality at runtime implies the ability to dynamically load, link and utilise functionality provided by software that is not available at compile time. Dynamic languages make this easy and static languages make this difficult. At the same time, software written in dynamic languages is inherently more likely to be insecure especially due to the languages’ ability to easily alter the behaviour of software. LabBack is written in Java, a static language, which makes it difficult to extend functionality at runtime but offers a customisable degree of security when doing so. LabBack provides no language-specific or experiment-specific by itself, and thus for execution of programs relies entirely on functionality contributed by plugins that can be loaded and unloaded at runtime. Some Java plugin frameworks such as OSGi are heavyweight and difficult to use, while other platforms such as JSPF\footnote{Java Simple Plugin Framework} do not have any provisions for security. LabBack provides its own plugin framework which is both tailored to LabBack’s domain and is lightweight. This is required to ensure that execution of third-party software is both secured and that its performance is not harmed by the plugin framework.

The answer to this research question consists of three criteria that a runtime-extendible platform should meet. Firstly, the domain of execution of third-party software is sufficiently narrow that the use of generic plugin frameworks is unnecessary. Instead of allowing plugins to contribute any interfaces and functionality, the platform should require that each plugin implements an as small as possible set of predefined simple interfaces. This leads to less sophisticated execution entry points for the plugin and thus makes securing the execution contained by the plugin simpler.

Secondly, as plugins require functionality provided by other plugins, the expression of these dependencies should be specifiable only in terms of required transformations of the third-party software to be executed. This leads to the third criteria, that plugins should not be allowed to depend on or interact directly with specific implementations of plugins. While this may lead to an increased burden of self-sufficiency on plugins, it leads to an increase in securability and to the untangling of dependencies between plugins. The former is fundamental in the execution of third-party software, and the latter is fundamental in allowing plugins to be added, upgraded and removed at runtime.

Thirdly and finally, plugin-provided functionality should be orchestrated dynamically. This increases flexibility with respect to the processing phases required by various experiments and programming languages, and stimulates the reuse of plugin functionality. However, to maintain security, the orchestration of various phases should be triggered only by data specific to each program executed. For example, in LabBack’s case, the plugin that provides execution of C programs relies on the JavaScript plugin for the final compilation stage and for the actual execution phase. The C plugin achieves this by adding a JavaScript successor job in the job chain which is executed independently by the JavaScript plugin.
10. Conclusion

10.3 Future Work

Plugin testability

Testing of LabBack plugins is tedious and requires actual deployment of the plugin within a LabBack instance. This is currently the case due to two factors. Firstly, plugins need to manipulate Job entities which are wired by the WebDSL compiler to the automatic persistence framework. This means that unit testing of plugins without either tedious mocking\(^1\) of LabBack's data model or providing an actual database instance is impossible. Secondly, plugins that dynamically alter execution pipelines depend on functionality provided by other plugins. Currently there is no mechanism to simulate executions of pipelines outside of a running instance of LabBack.

A possible avenue for enabling unit testing of plugins is to ensure that LabBack's data model follows certain interfaces. The WebDSL compiler only generates full implementations of modelled entities and would need to be modified to also generate interfaces. This modification is a simple one and would allow distribution of dummy implementations for use in unit testing. Once interfaces for the data model become available, it is trivial to simulate LabBack's scheduler with a simple class that only composes pipelines based on mocked Job entities.

Distributed scaled deployments

LabBack currently mandates that multi-instance deployments are shaped as trees and that all communication between the clients and LabBack takes place through the root of the tree. This clearly renders the root node a central point of failure in a deployment.

In essence there is no real need for jobs to enter and leave LabBack only via the root node and it should be possible to have root nodes of subtrees act as entry points as well. As long as jobs are only allowed to be scheduled downwards in tree, each subtree could form its own deployment tree and at the same time be part of a bigger deployment. Intuitively the load estimation mechanism is expected to work effectively even in this scenario serving. The tree badness heuristic of a node uses only information available in its subtree and hence should not be influenced by this change. Simulations, as performed in Chapter\(^8\) are sufficient to verify this hypothesis, and the simulator built for LabBack can readily be adapted for this purpose.

A definite requirement is that the deployment topology remains a tree since the tree badness heuristic does not work in the presence of cycles.

Interactive programs

LabBack cannot currently execute interactive programs. Interactive programs need to block waiting for a user's input but they will be aborted by the scheduler if they exceed the maximum allowed duration. Thus LabBack cannot evaluate programs that expect user input, and more importantly does not currently support utilisation of interactive interpreters (REPLs).

The difficulty with achieving this is that the scheduler needs to be able to either save an execution's state or pause it. The first case is impossible without the modification of guest programs or REPLs to pass a continuation to the scheduler. This is

\(^1\)Mocking, as in providing mock implementations
10.3. Future Work

clearly infeasible. The second option is to pause the execution’s thread when the guest
program (or REPL) makes a blocking call on the input stream. Java Thread’s suspend
method is deprecated because it raises the risk of deadlocks but may be safe for use
in this case. This option for supporting interactive programs may be viable but clearly
further investigation is needed, particularly due to the security and robustness concerns
it raises.

Custom languages and web IDEs

LabBack’s extensibility with support for various programming languages has only
been used for C, Scala and JavaScript. This extensible approach to language support
may prove useful to provide server-side execution for web based Integrated Develop-
ment Environments [39] (IDE). While many web IDEs exist, most only provide IDE
features for JavaScript and limit themselves to text editor functionality for other lan-
guages. In this scenario LabBack could provide compiler features for these IDEs. This
may prove particularly interesting for Domain Specific Languages (DSL) such as those
built with the Spoofax Language Workbench [37] where a web based IDE is currently
not generated but the generated compiler runs on the Java Virtual Machine, and thus
could be hosted on LabBack. This can support the creation of web based IDEs that
provide syntactic and semantic error reporting directly in the browser without requir-
ing the compiler to first be ported to the web browser. This direction needs thorough
research before it can be realised and can lead to interesting developments.

Automatic grading mechanisms

LabBack was currently only used in real courses when it was in its proof of concept
phase. At that time it was identified that unit testing is not particularly well suited to
automatic grading, as Chapter 8 also revealed. An observation to the detriment of unit
testing is that it only works if the author of the software under test has the intention of
building working software. If, however, the author only intends to pass the unit tests
then they can be circumvented without actually implementing the tested functionality.
Stemming from this observation, LabBack 2.0 is not bound to grading unit testing and
can be extended with new automatic grading mechanisms by means of plugins.

Until LabBack, no platform existed that allowed new automatic grading mecha-
nisms to be prototyped and evaluated in large case studies. As such any intention by
educators to support automatic grading was faced with an engineering burden, rather
than a pedagogic one. As LabBack makes development and deployment of secure and
robust grading techniques easy, it provides the basis for lightweight greenfield research
into innovative evaluation methods for both code functionality and code style.

Pedagogical improvements

Chapter 1 claimed, as a motivating factor for this research, that one of the causes
for the shortage of good software engineers is that computer science students receive
insufficient practice material during their studies. This claim is not validated. Even
more so, the inverse claim that increasing the amounts of practice material leads to
better engineers is also not validated. In fact the pedagogical effects of increasing the
amount of practice material are unknown.
An interesting research direction is to evaluate the validity of these claims by means of a long-term social study to measure the pedagogical improvement of increasing practice material. This study must not limit itself to measuring the knowledge of fresh graduates, and should ideally extend to measuring their skill and career advancements long after graduation day. LabBack provides the necessary engineering advancement to make such quantitative pedagogical experiments possible.
Bibliography


Appendix A

LabBack 1.0 Scheduler Implementation

package back.engine

// imports ommitted...

class ExecutionEngine {
  watchdog: ApplicationWatchdog,
  maxThreads: Int,
  jobTimeOutMillis: Long,
  gracefulStopTimeOut: Long,
  maintenanceInterval: Long) extends Thread("ExecutionEngine") {

  private val log = Logger.getLogger(this.getClass.getPackage().getName())
  this.setName("ExecutionEngine")
  this.setDaemon(true)

  // Tasks currently running, should only be touched by the maintenance loop
  private var tasks: Set[ExecutionThread] = Set()

  /**
   * queue for new tasks, items get added by the scheduleTask method and removed by the maintenance loop
   */
  private val newTasks = new SynchronizedQueue[ExecutionTask]()

  private var stopRequested = false

  def enqueueTasks(t: ExecutionTask*) {
    log.info("Scheduling " + t.size + " new tasks.")
    newTasks.enqueue(t:_*)
    this.interrupt()
  }

  def requestStop() {
    stopRequested = true;
    this.interrupt();
  }
}
override def run() {
    while (!stopRequested) {
        val entryTime = System.currentTimeMillis()
        var exitTime = System.currentTimeMillis();
        if (!tasks.isEmpty || !newTasks.isEmpty) {
            maintainTasks()
            scheduleTasks()
        }
        watchdog.pulse(this)
        val sleepTime = entryTime + maintenanceInterval - System.currentTimeMillis()
        if (sleepTime > 0)
            try {
                Thread.sleep(sleepTime)
            } catch {
                case e: InterruptedException =>
            }
    }
}

private def maintainTasks() {
    val currentTime = System.currentTimeMillis()
    /* 1. remove any tasks that have already completed */
    val doneTasks = tasks filter {x =>
        x.getTask().isDone() && x.getTask().getStatus() != Cancelled()}
    tasks = tasks filterNot (doneTasks contains)
    log.finest("Cleaned up tasks [" + doneTasks.size + "]")
    /* 2. try to stop any cancelled tasks */
    val cancelledTasks = tasks filter {
        _.getTask().getStatus() == Cancelled()
    }
    tasks = tasks filterNot (cancelledTasks contains)
    cancelledTasks.map(_.requestStop())
    log.finest("Cancelled tasks [" + cancelledTasks.size + "]")
    /* 3. request graceful stop for all timed out tasks */
    val timeoutTasks = tasks filter {
        _.getStartTime() + jobTimeOutMillis < currentTime
    }
    tasks = tasks filterNot (timeoutTasks contains)
    timeoutTasks.map(_.requestStop())
    log.finest("Timedout tasks [" + timeoutTasks.size + "]")
    if (timeoutTasks.size > 0)
        try {
            Thread.sleep(gracefulStopTimeOut)
        } catch {
            case e: InterruptedException =>
        }
    /* 4. force stop all non-responding tasks */
    val toKill = cancelledTasks ++ timeoutTasks
    toKill.map(_.forceStop())
    log.finest("Killed zombies [" + toKill.size + "]")
}

private def scheduleTasks() {
/* If we have capacity and something to queue, do queue tasks */
while (!newTasks.isEmpty && tasks.size < maxThreads) {
    val newTask = newTasks.dequeue
    if (newTask.getStatus() == Pending()) {
        val newThread =
            newTask.getLanguage() match {
                case Scala() =>
                    new ScalaExecutionThread(newTask)
                case C() =>
                    new CExecutionThread(newTask)
                case JS() =>
                    new JSExecutionThread(newTask)
            }
        tasks = tasks + newThread
        newThread.start()
    }
}
}
public class InsulatingClassLoader extends SecureClassLoader {
    private static List<String> blacklist = new ArrayList<String>();

    public InsulatingClassLoader(VirtualDirectory path) {
        super();
        this.binDir = path;
    }

    @Override
    public synchronized Class<?> loadClass(final String name, final boolean resolve) throws ClassNotFoundException {
        // 0. check to see if is black-listed
        if (blacklist.contains(name)) {
            throw new AccessControlException("Access to blacklisted class: " + name);
        }
        // 1. check to see if it’s already loaded
        Class<?> c = findLoadedClass(name);
        if (c != null) {
            return c;
        }
        // 2. check whether it is a system class
        try {
            c = findSystemClass(name);
            if (c != null) {
B. LabBack 1.0 ClassLoader Implementation

```java
    return c;
}
} catch (Throwable t) {
}

// 3. load the class ourselves
byte[] data = null;
AbstractFile classFile = binDir.fileNamed(name + "\class")
;
try {
data = classFile.toByteArray();
} catch (IOException ioex) {
    throw new ClassNotFoundException(name, ioex);
}

URL u = null;

try {
    u = new URL("file:/memory");
} catch (MalformedURLException e) {
    throw new ClassNotFoundException(e);
}

/* 4. define class and place in specific CodeBase for
   security manager to recognize */
c = defineClass(name, data, 0, data.length, new CodeSource
    (u, (CodeSigner[]) null));

    return c;
}
```
package lang.sc.counter.proc;
import java.util.HashMap;
import java.util.concurrent.CancellationException;
import webdsl.generated.domain.Job;
import weblab.plugins.control.IExecutionLogic;

public class CounterLogic implements IExecutionLogic {

    private final static String scala_compile = "lang.sc.compile ";
    private boolean stop;

    @Override
    public Object execute(Job j, Object pres) {
        if (pres == null || !(pres instanceof HashMap<?, ?>)) {
            Job pre = Job.buildFrom(scala_compile, j, new String[] {
                Constants.PROGRAM });
            j.addPreJobJob_(pre);
            j.setNeedsReset(true);
            j.setSuccess(false);
            return null;
        }
        assert pres instanceof HashMap<?, ?>;
        final InsulatingClassLoader loader = new InsulatingClassLoader((HashMap<String, byte[]>) pres);
        final Class<?> clazz = loader.loadClass(j.getParameters().getString_(Constants.PROGRAM_NAME));

        if (stop) {
            j.setCancelled(true);
            j.setSuccess(false);
            return pres;
        }

        final int expectMethods = Integer.parseInt(j.getParameters().getString_(Constants.TEST));
        final int numMethods = clazz.getMethods().length;

        if (numMethods == expectMethods) {
            final String msg = "Failed. Expected [" + expectMethods

            return msg;
        }
    }
}
+ "] but got [" + numMethods + "]");
j.getParameters().putString__Text_(Constants.JOB_MSG, msg);
j.setSuccess(false);
} else {
    j.setSuccess(true);
}

return pres;

@Override
public Object handleFailure(Throwable t, Job j) {
    j.getParameters().putString__Text_(Constants.JOB_MSG, "Execution failed: " + t.getMessage());
    j.setSuccess(false);
    return null;
}

@Override
public Object handleFailure(CancellationException cex, Job j) {
    j.getParameters().putString__Text_(Constants.JOB_MSG, "Execution cancelled: " + cex.getMessage());
    j.setSuccess(false);
    j.setCancelled(true);
    return null;
}

@Override
public void stop() {
    stop = true;
}
Appendix D

Plugin Loader Implementation of LabBack 2.0

package labback.plugins.loading;
// imports omitted

public class PluginLoader extends APluginLoader {
    private final static Logger log = new Logger(PluginLoader.class);

    private final Map<String, Float> pluginVersions = new HashMap<String, Float>();
    private final Map<String, PluginClassloader> pluginLoaders = new HashMap<String, PluginClassloader>();

    public PluginLoader freshInstance() {
        return new PluginLoader();
    }

    @Override
    public void unload(Plugin plugin) {
        final String pluginName = plugin.getNaturalId();
        log.log(" Unloading plugin \[", pluginName + "]\]");
        pluginVersions.remove(pluginName);
        pluginLoaders.remove(pluginName);
    }

    @Override
    public void load(Plugin plugin) throws InitializationException {
        final String pluginName = plugin.getNaturalId();
        log.log("Loading plugin \[", pluginName + "]\]");

        // Step 1 -- Check that it isn’t loaded yet
        if (isLoaded(plugin) && !isUpdated(plugin)) {
            log.log("Already loaded and not updated \[", pluginName + "]\]");
            return;
        }

        // Step 2 -- Check that dependencies are satisfied
        if (!plugin.isSatisfied_()) {
            log.log("Plugin has broken dependencies \[", pluginName + "]\]");
            throw newInitializationException("Broken dependencies")
        }

    }
}
D. PLUGIN LOADER IMPLEMENTATION OF LabBack 2.0

// Step 3 -- Gather the plugin libraries
Collection<File> jarFiles = new ArrayList<File>();
for (PluginData lib : plugin.getLibraries()) {
    jarFiles.add(lib.getData());
}
if (plugin.getMainLibrary() != null) {
    jarFiles.add(plugin.getMainLibrary().getData());
}

log.log("Plugin [" + pluginName + "] loads [" + jarFiles.size() + "] jars");

// Step 4 -- Load the libraries
pluginVersions.put(plugin.getNaturalId(), plugin.getPluginVer());
final PluginClassloader loader = new PluginClassloader(pluginName);
pluginLoaders.put(plugin.getNaturalId(), loader);
final HashMap<String, byte[]> data = new HashMap<String, byte[]>();
int numclasses = 0;
for (File file : jarFiles) {
    try {
        final JarInputStream jis = new JarInputStream(file.
            getContentStream());
        JarEntry e = jis.getNextJarEntry();
        int libnumclasses = 0;
        while (e != null) {
            data.put(e.getName(), smallStepRead(jis));
            libnumclasses++;
            e = jis.getNextJarEntry();
        }
        log.log("Plugin [" + pluginName + "] library [" + file.
            getFileName() + "] loaded [" + numclasses + " entries]");
        numclasses += libnumclasses;
    } catch (IOException e) {
        log.log("Failed to load plugin [" + pluginName + "]
            library [" + file.getFileName() + "]");
        throw new InitializationException("Failed to load library", e);
    } catch (SQLException e) {
        log.log("Failed to load plugin [" + pluginName + "]
            library [" + file.getFileName() + "]");
        throw new InitializationException("Failed to load library", e);
    }
    loader.addData(data);
    log.log("Plugin [" + pluginName + "] loaded [" + numclasses + " entries]");
}

private byte[] smallStepRead(final JarInputStream jis) throws IOException {
    byte[] t = new byte[1024 * 1024];
    int length = 1;
    int offset = 0;
}
while (jis.read(t, offset, length) > 0) {
    offset += length;
}
byte[] r = new byte[offset];
System.arraycopy(t, 0, r, 0, offset);
return r;

@Override
public boolean isLoaded(Plugin plugin) {
    boolean isLoaded = pluginVersions.containsKey(plugin.getNaturalId());
    log.log(" Plugin "+ plugin.getNaturalId() + "] is loaded "+ isLoaded + "]");
    return isLoaded;
}

@Override
public boolean isUpdated(Plugin plugin) {
    boolean isUpdated = isLoaded(plugin)
        && pluginVersions.get(plugin.getNaturalId()) < plugin.getPluginVer();
    log.log(" Plugin "+ plugin.getNaturalId() + "] is updated "+ isUpdated + "]");
    return isUpdated;
}

@Override
public IExecutionLogic newExecutionLogic(JobType jobType) throws InitializationException {
    Plugin plugin = jobType.getPlugin();
    log.log(" Constructing execution logic for plugin "+ plugin.getNaturalId() + "] job-type "+ jobType.getNaturalId() + "] executor "+ jobType.getClass() + "]");
    load(plugin);
    IExecutionLogic logic = null;
    final PluginClassloader loader = pluginLoaders.get(plugin.getNaturalId());
    try {
        Class<?> extends IExecutionLogic > clazz = (Class<?> extends IExecutionLogic >) loader
        .loadClass(jobType.getClass());
        if (clazz != null) {
            Constructor<?> extends IExecutionLogic > constr = clazz;
            logic = constr.newInstance();
            log.log(" Execution logic for jobtype "+ jobType.getNaturalId() + "] loaded by "+ logic.getClass().getClassLoader() + "]");
        }
    } catch (Exception e) {
        log.log(" Plugin "+ plugin.getNaturalId() + "] executor "+ jobType.getClazz() + "] could not be instantiated", e);
        throw new InitializationException("Initialization failed ", e);
    }
    return logic;
}