Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

IDRA: AN OUT-OF-PLACE DEBUGGER FOR NON-STOPPABLE APPLICATIONS

Matteo Marra

June 2017

Promotors:

Prof. Dr. Elisa Gonzalez Boix (VUB)
Dr. Guillermo Polito (CNRS, Inria)

Advisor:
Carmen Torres Lopez

Faculty of Science and Bio-Engineering Sciences

“Limits, like fears, are often just an illusion”

Abstract

Big data processing applications have emerged in a diverse domains ranging from medicine to social networking. Those applications continuously analyze data coming into the system in a parallel way. In this context, when failures or bugs occur it is not always an option to stop the running application to investigate the cause because it can result in a loss of time, data and availability of the overall application. This is why we call them *non-stoppable applications*.

Non-stoppable applications are normally debugged post-mortem, by means of logs or, more recently, trace-based debuggers. This means that debugging operations happen after the system already failed and stopped executing. Alternatively, it is possible to debug these systems on-line, allowing developers to interact and inspect the state of the failing program. Traditional on-line debuggers, however, need to pause the execution and are not completely suited to such non-stoppable applications.

In this thesis we introduce *out-of-place debugging*, a novel debugging technique to remotely debug non-stoppable applications without affecting their execution. We propose IDRA, an out-of-place debugger that provides on-line debugging primitives while avoiding the suspension of the *overall* application's execution. IDRA can be deployed in a distributed system, allowing to remotely debug different nodes. It also includes a committing system for developer's changes.

We evaluate our solution on several real and synthetic applications which vary from a data intensive application consuming tweets from an infinite stream to an IoT device consuming hardware sensor data. Our results show that debugging operations are on average from a thousand to ten thousand times faster than a traditional remote online debugger. IDRA presents negligible performance overhead if no failures are generated. In the case of a failure IDRA shows an increase of the execution time. We also show that IDRA supports a load of ten thousand concurrent failures, given by the limits imposed by the operating system.

Acknowledgements

To start, I would like to thank my parents for the incredible and continuous support in this and all the other steps of my life and academic career.

Grazie mamma, grazie papà!

A big thank goes to my promotors and advisor: Elisa Gonzalez Boix, Gulliermo Polito and Carmen Torres Lopez. Elisa for all her support, ideas, critiques, suggestions and even papers to read throughout the whole thesis. Guille for the big help on the technical part of the thesis, for listening to my complaints about Pharo, motivating me to find always a better solution, and for always being there, even when I needed a place to stay in Lille. Carmen for all the feedback and support and for all the corrections to my (sometimes awful) English, together with the periods at the end of any caption. Thank you for also sharing your office with me.

Thanks to Steven Costiou for sharing ideas and work with me, and for helping me with the raspberry use case and its benchmarks. Thanks to Yesplan, in the person of Andy Kellens, for the idea of the use case and for letting me use their application model.

I would also like to thank the whole Software Languages Lab of the Vrije Universiteit Brussel for listening to my presentations and giving continuous and useful feedback, and professor Wolfgang De Meuter for helping me finding the first idea for this thesis.

Thanks to Stéphane Ducasse and all the RMOD research group in Lille for hosting me many times and giving me a big technical help.

A big thank you to all my family and friends back in Italy and here in Brussels, especially my kotmates for many good moments (and for listening to my complains everyday) and all the ESNers for letting me neglect them when I had to work on this thesis. Thanks also to my classmates, especially Bruno and Silke, for being with me from Day 1 in Brussels, as the unforgettable buddy group 3, which was with me almost every day during my first year. Also thanks to Elisa Gilormello for the logo, which I promise you will see one day.

Thanks to all the people of Elements, especially Federico, Marco and Michele, for all the work we did together in the last two years, and for never giving up on me, even in the most busy moments of this thesis.

I want to thank the jury for reading my thesis, and you, for at least reading this acknowledgements. I hope you will find the courage to read the rest of this thesis.

Finally, I would like to thank Marie-Louise for being with me and besides me every day that I worked on this thesis.

Contents

1 Introduction **1**
1.1 Research Context: Big Data Processing 1
1.1.1 Case Studies 2
1.2 Research Problem: Debugging non-stoppable applications 3
1.3 IDRA: an out-of-place debugger 4
1.4 Structure of this thesis 5

2 Motivation **7**
2.1 Data Intensive Applications 7
2.2 Long Running Systems 8
2.3 Non-stoppable applications 8
2.4 Motivating examples 8
2.4.1 Use Case 1: Twitter Analyzer 9
2.4.2 Use Case 2: Yesplan Testing 10
2.4.3 Use Case 3: Sensor Monitoring 12
2.4.4 Conclusion 12
2.5 State of the Art 12
2.5.1 Offline Debugging 13
2.5.2 Online Debugging 16
2.6 Remote debugging 17
2.6.1 Online remote debuggers 18
2.7 Debugging non-stoppable applications 20
2.7.1 Offline debugging 20
2.7.2 Online debugging 21
2.7.3 BigDebug 21
2.8 Problem Statement 23
2.9 Conclusion 26

3 A distributed programming model for Pharo **27**
3.1 Master/Worker Architecture 27

3.2 Design and Implementation of Master/Worker Framework in Pharo . . . 29
3.2.1 Communication between Master and Worker . . . 30
3.2.2 Scheduling tasks on the Worker . . . 31
3.3 Conclusion . . . 32

4 IDRA **33**
4.1 IDRA's Overview . . . 33
4.1.1 Debugger monitor and manager . . . 34
4.2 Handling exceptions or breakpoints . . . 37
4.2.1 IDRA Monitor . . . 37
4.2.2 IDRA Manager . . . 38
4.2.3 Breakpoints . . . 39
4.3 Reconstructing exceptions or breakpoints . . . 40
4.3.1 Handling the state . . . 41
4.4 Fixing and committing . . . 42
4.4.1 IDRA Changes Handler . . . 43
4.4.2 Applying changes . . . 45
4.4.3 Detected changes . . . 46
4.4.4 Restarting . . . 46
4.4.5 Atomicity of changes . . . 48
4.5 Overview of IDRA architecture . . . 49
4.6 IDRA front-end in Pharo . . . 50
4.6.1 Interacting with the exception queue of IDRA . . . 52
4.7 Conclusion . . . 54

5 Implementation **55**
5.1 Communication architecture . . . 55
5.1.1 Communication layer . . . 56
5.1.2 Communication protocol . . . 57
5.2 IDRA Debugger . . . 57
5.2.1 Breakpoints and exception handlers . . . 59
5.3 IDRA Changes Handler . . . 63
5.3.1 Changes detection tool . . . 64
5.4 Conclusion . . . 65

6 Evaluation **67**
6.1 Benchmarking scenarios . . . 67
6.1.1 Twitter analyzer . . . 67
6.1.2 Yesplan testing . . . 70
6.1.3 Sensor monitoring application . . . 71

6.1.4 Buggy observer 72
6.2 Evaluation overview and setup 72
6.2.1 Benchmark setup 73
6.2.2 Benchmark framework 74
6.2.3 SMark benchmarks 74
6.3 Micro-Benchmarks 75
6.3.1 Setup 75
6.3.2 Benchmarks 76
6.3.3 Results 77
6.4 Network overhead benchmarks 80
6.4.1 Setup 80
6.4.2 Benchmarks 81
6.4.3 Results 82
6.5 IDRA overhead benchmarks 85
6.5.1 Setup 85
6.5.2 Benchmarks 85
6.5.3 Results 86
6.6 IDRA scalability test 89
6.6.1 Setup 89
6.6.2 Benchmark 7 - IDRA scalability test 89
6.6.3 Results 89
6.7 Conclusion 90

7 Conclusion **93**
7.1 Problem statement revisited 93
7.2 IDRA: An Out-of-place Debugger 94
7.2.1 The IDRA debugger 94
7.2.2 IDRA changes handler 95
7.2.3 Evaluation 95
7.3 Contributions 96
7.4 Limitations and future work 97
7.4.1 Future work 98

List of Figures

2.1 A model for the Twitter analyzer. 9
2.2 Twitter analyzer deployed on a master and a worker. 10
2.3 Architectural overview for the Yesplan testing scenario. 11
2.4 Core model of Mercury (Extracted from [PBFD15]) 19
2.5 Data transformations of a word count (Extracted from [GIY+16]). 22
2.6 Simulated breakpoint in a stage (Extracted from [GIY+16]). . . 22

3.1 Overview of *master/worker* architecture. 27
3.2 Worker selection phase in *master/worker* architecture. 28
3.3 Class diagram of the *master/worker* Pharo API. 29
3.4 Communication of distributed *master* and *worker* on Pharo images. 31

4.1 Representation of IDRA instances, manager and monitor, in a distributed system of two machines. 35
4.2 Representation of IDRA instances in the master/worker architecture. 36
4.3 Handling of an exception in *IDRA Monitor* - Part 1. 37
4.4 Handling of an exception in *IDRA Monitor* - Part 2. 38
4.5 Overview of exception handling operations in *IDRA Manager*. . 39
4.6 Representation of a call stack and a context. 40
4.7 Representation of a call stack and an exception. 41
4.8 Representation of a call stack and a breakpoint-context. 42
4.9 Overview of IDRA architecture with an IDRA Changes Handler. 44
4.10 Recording changes of a debugging session 45
4.11 Applying changes in a remote machine. 45
4.12 Adding a restarting strategy to an exception. 47
4.13 Atomic apply of changes when using *master/worker* framework. 49
4.14 Overview of IDRA components in a simple *developer/worker* setup. 50
4.15 The Pharo Debugger. 51
4.16 The view on the *errors queue*. 52

4.17 The IDRA debugger developer interface. 53

5.1 An overview of the connection between *IDRA Manager* and *Monitor*, and two *IDRA Changes Handler* 56
5.2 Class diagram of IDRADebugger. 57
5.3 Code to setup IDRA as Manager 58
5.4 Code to setup IDRA as Monitor 58
5.5 Code of `IDRADebugger` >> #handleError:inContext :restartingStrategy: 59
5.6 Code of `IDRADebugger` >> `#serveQueueMonitorMode` 59
5.7 Class diagram corresponding to the implementation of exceptions 60
5.8 Class representation of the breakpoints. 61
5.9 Placing a breakpoint in the code. Code extracted from *Twitter analyzer*. 61
5.10 Implementation of `DefaultHandler` >> `#handleException:`. . 62
5.11 Handle an exception with an `ExceptionHandler`. 62
5.12 Setting a default strategy to an `IDRADebugger`. 62
5.13 Default implementation of `UnhandledError >> #defaultAction` 63
5.14 New implementation of `UnhandledError >> #defaultAction` . 63
5.15 Class diagram of a `ChangesHandler`. 64

6.1 UML class diagram of `TwitterMaster` and `TwitterWorker`. . . 68
6.2 Class diagram of `Tweet` and the related classes. 69
6.3 Class diagram of `RemoteTestRunner` and `ControlledTestRunner` 70
6.4 Architecture of the sensor monitoring. 71
6.5 Setup of idra on two machines for benchmarking. 76
6.6 Setup of the pharo remote debugger (PharmIDE) on two machines for benchmarking. 76
6.7 Boxplot of the session initialization time for an exception 78
6.8 Bar plot of the execution time of single debugging operations on the *buggy observer* 79
6.9 Bar plot of the execution time of single debugging operations on the *sensor monitoring* application 80
6.10 Plot of the number of bytes exchanged for an increasing number of exceptions. 83
6.11 Plot of the number of bytes exchanged for one exception. 83
6.12 Bar plot of the bytes exchanged to commit one change. 84
6.13 Overhead of the execution with active IDRA Monitor. 87
6.14 Overhead of the execution with active IDRA Manager. 87
6.15 Execution of `AST-core-tests`, increasing number of failures. . . 88

Chapter 1

Introduction

1.1 Research Context: Big Data Processing

In the latest years we saw the spreading of multi-core and parallel machines, from processors for personal computers to clusters for cloud computing. These machines can perform many parallel operations, and are used today to analyze big sets of data from any domain, from medicine to social networking, continuously flowing on the internet. This is commonly known as *big data processing*.

One of the most representative examples of big data applications are the so called *data intensive applications*. As described by Gorton et al. [GGSW08] data intensive applications manage and process a growing set of data, potentially coming from non-stoppable streams. The analysis on the data is time sensitive to avoid losing data and computational time. These applications are normally deployed on distributed clusters using engines such as Apache Spark [Apab], Hadoop MapReduce [DG08], Apache Giraph [Apaa] and many others. These engines allow developers to write applications in a functional style, enforcing parallel execution and scaling it to all the cores of the machine where they are executing.

Other applications, such as *long running systems* share the same characteristics. These are systems that run a continuous computation, in a classic or distributed way. Their interruption potentially leads to:

- A loss of partial results of computation.
- A loss of produced or processed data.

Examples of long running systems include web servers, which continuously listen for client requests, or Wireless Sensor Networks (WSN) and Cyber Physical Systems (CPS). In particular, we could have one or more microcontrollers

analyzing data coming from sensors and executing different operations on them. In contrast to data intensive applications, applications for these systems are built using object oriented languages, which involve a stack of execution and a complex variable state.

Both data intensive applications and long running systems are executed on remote machines, should not crash, nor be stopped and process data that should not be lost. We call these applications **non-stoppable**.

1.1.1 Case Studies

In this thesis we will employ three case studies that motivate the need for new debugging support. We will use those three cases also to evaluate our solution.

First the *Twitter analyzer*, a data intensive application which continuously analyzes tweets to provide different statistics about it. It is directly analyzing tweets coming from the Twitter stream, and should not stop if, for instance, there is an error in parsing one particular tweet.

Second, as long running system we consider the testing infrastructure of the Belgian company Yesplan [Yes], which provides a web application for planning of events. Before releasing, they execute a series of tests, which take considerable amount of time, on their application. These tests are non deterministic because are initialized with different variables to add entropy to the system, simulating the end-user use. As a result, if a test fails, it is very difficult to reproduce the exact conditions that make the test do not fail. They are running on a remote non accessible server which just provides a log of the failures. Since there is a lack of specialized debugging support, developers normally try to run again a failed test, which will pass or not pass again only depending on the non-determinism of the system. A representative of the company told us that is very rare that a new version of the product is released with all the tests correctly passing. We call this case study *yesplan testing.*

Third, example of *long running system* is in the context of cyber physical systems (CPS). Imagine a CPS monitoring the room temperature of a food storage room. If the sensor returns a wrong value, for example in text format, the CPS does not have to stop reading the next temperature from the sensor. In the described systems such failures are non-deterministic, and not predictable. Without proper debugging support, such unpredicted failure would stop the execution of the CPS. We call this case study *sensor monitoring.*

1.2 Research Problem: Debugging non-stoppable applications

We believe that, due to the analyzed constraints of *non-stoppable applications*, when a failure occurs the system should react in the following way:

- The failure should not stop the execution of computation on other nodes.
- The failed node should not crash.

The solutions commonly used to deploy *data intensive applications* do not react like specified when an failure occurs and are notoriously difficult to debug. In fact they are normally debugged post-mortem which analyze execution logs or, lately, using trace-based replay debuggers such as Graft [SSK+15] and Arthur [DZSS13]. However, replay debuggers add important overhead to the execution to record a trace, and in the context of *data intensive applications* cannot be deployed in production.

Few solutions, such as BigDebug [GIY+16], propose the use of online debugging primitives to debug such applications. When using BigDebug, a failure will not trigger the immediate end of execution of all nodes, but will instead allow the different nodes to finish their job correctly and then stop the execution of the overall program. It does not prevent the crash of the failed node but it allows the re-execution of the failed node and supports crash culprit determination. More concretely, it offers some online debugging primitives, allowing the insertion of *simulated breakpoints* and *watchpoints*. Breakpoints and watchpoints, opposite to a failure, do not stop the execution of the program and allow the user to analyze the state in a particular moment of the execution.

Long running systems normally execute remotely, and they are difficult to locally debug for lack of physical access, or even because some of these systems do not provide a graphical interface. In this context, a failure of a part of the system should not imply to stop the overall application to debug it. For example, in CPS a failure should not make a microcontroller stops its execution. Instead, it should be possible to debug the system while it keeps working.

Different solutions exist to debug remote processes. In fact implementations of remote debuggers are available for most of the mainstream languages, such as C, Java and .NET . Among the others, Mercury [PBFD15] is a debugger designed to remotely debug applications in Pharo Smalltalk [Pha]. This systems, however, are not fully suited to debug non-stoppable applications.

1.3 IDRA: an out-of-place debugger

In order to tackle the described debugging problems we propose to use an out-of-place debugging technique, which allows to locally debug remote exceptions coming from different remote machines. When an exception occurs on a remote machine which is continuously executing some tasks the debugger will:

1. Extract the debugging information, such as current state, and let the machine proceed to the next task.
2. Reproduce the error on a different machine which the developer can access easily.
3. Allow the developer to provide a fix to the exception and locally test it.
4. Detect the changes made by the developer during the debugging session.
5. At command of the developer, distribute the changes to the bugged machine(s).
6. Apply the changes on the bugged machine(s).
7. Re-execute failed tasks with the fixed code.

We propose IDRA, a debugger for non-stoppable applications such as *data intensive applications* and *long running systems* aforementioned.

IDRA is composed by:

- **IDRA Monitor**: executes on the remote machine, and can detect errors avoiding the machine to crash.
- **IDRA Manager**: executes on a local accessible machine, connected to the remote ones, and can reconstruct the errors detected by the *IDRA Monitor*.
- **IDRA Changes Handler**: detects the local changes to the code and provides a mechanism to propagate the changes to all the connected machines.

Both IDRA Monitor and Manager handle a queue of errors. When an error happens on a machine where IDRA Monitor is executing, it is stored in a queue. One by one the errors are sent to the connected session of IDRA Manager. When receiving an exception, the IDRA Manager will open one debugging session at a time. The developer can then interact with the debugging session, changing the code and restarting it from a chosen point. When he decides the

fix should be provided to the remote machine(s), it invokes the IDRA Changes Handler to send the changes to the connected IDRA Changes Handler in the system.

We developed our solution in Pharo Smalltalk [Pha], a live object-oriented programming language. It provides an easily extensible debugger [BNDP10] and allows to access execution and state structures. It also provides a remote debugger [PBFD15, Kud], to which we can compare our solution.

The IDRA out-of-place debugging technique and its implementation in Pharo form the main contributions of this work.

The master/worker architecture

The Pharo community does not provide a programming model for distributed computing. Normally developers program distributed application directly on top of networking technology or by means of proxies. Because of this, in order to implement a *data intensive applications*, we implemented a *master/worker* framework on which such applications can be built in Pharo. This is a common architecture adopted in distributed systems for big data processing like Apache Spark [Apab]. Actually, *Master/worker* is a classic parallelism model for execution of parallel tasks. In the model applications consist of a *master* and one or more *workers*. The *master* instructs the worker on which operations to execute, and the *workers* execute the instructed operations in an independent way between each other.

The implementation of the master/worker architecture in Pharo is a technical contribution of this thesis in the context of the Pharo community.

Evaluation

In order to evaluate IDRA we executed different benchmarks on different use cases, comparing it to the Pharo Remote Debugger and evaluating the overhead IDRA introduces to the system.

We executed micro-benchmarks on a simple application and on the sensor monitoring application to compare IDRA with the Pharo Remote Debugger. We then executed some overhead and scalability benchmarks on the other motivational examples.

1.4 Structure of this thesis

Chapter two analyzes our motivating examples and provides a literature review of the different debugging techniques. We conclude this chapter with the problem statement.

Chapter three describes the *master/worker* architecture and our implementation.
Chapter four introduces an overview on IDRA and its functionalities, analyzing its different components.
Chapter five analyzes the implementation details of IDRA.
Chapter six contains the results of our evaluation and teir analysis.
Chapter seven contains a final conclusion on the thesis, describing IDRA and possible future work.

Chapter 2

Motivation

Data intensive applications and long running systems nowadays are widely used to process and analyze data coming from different sources and in different scenarios. In this chapter, we characterize those applications and we offer a literature review of different debugging approaches. At the end of the chapter we present our problem statement.

2.1 Data Intensive Applications

As the computational power of processors grew up to the technological limit [SL05], the clock rate of processors stopped increasing as it used to. In this context we assisted the development of parallel systems, from multi-core processors for personal computers to large scale servers and cloud computing platforms. Internet became a big set of data constantly flowing giving rise to cloud computing platforms, which offer high parallelization of processes. Such volume of data is typically analyzed through *data intensive applications*.

A data intensive application is a program that [GGSW08]:

- manages and processes an exponentially growing set of data.
- potentially processes data coming from an unstoppable stream.
- applies an analysis on time sensitive data, which has to be processed in short time to avoid losing incoming information.

Examples of unstoppable streams are Internet streams and sensors. Data coming from these streams can be processed online, while the data is flowing. Alternatively, it can be stored into databases or distributed file systems and processed afterwards. In both cases it is crucial not to stop the flow of data analyzed by the system: in the first case, because the application should not

lose any data, in the second one because, even if the incoming data comes from a persistent file or database, one single machine crashing in a big cluster could generate a significant reduction on availability of the system. As a result, this reduces the processing capabilities and could lead to a sensitive time loss, so it is preferable not to stop the machines when an error happens.

2.2 Long Running Systems

Not only *data intensive applications* handle non-stoppable streams or non-stoppable computation. Classic distributed systems like web servers are constantly listening to client requests, and should not be stopped to be debugged. Another example is a repetitive long running system on a remote server, like unit testing of a complete distribution of a programming language or an end-user application. Developers might want to intercept a failed test immediately, while other tests are still running, examine its state to find out the reason why it failed and provide a fix.

Wireless Sensor Networks (WSN), or more recently, Cyber Physical Systems (CPS) also exhibit similar characteristics. As a CPS we can have a micro-controller executing continuous analysis on sensors. The execution should not be stopped after a single failure, but rather debugged while it keeps reading other sensors.

2.3 Non-stoppable applications

Data intensive applications and *long running systems* share different common characteristics:

- Processes should not be stopped.
- They are executed on remote machines.
- The data that provoked a failure should not be lost.

Hence, we refer to both of them as **non-stoppable applications**.

This thesis aims to provide debugging facilities for non-stoppable applications that take into account the aforementioned concerns.

2.4 Motivating examples

In this section we describe three use cases that are representative of non-stoppable applications studied in this thesis. We employ them to motivate

debugging techniques to be employed and in later chapters to apply our novel debugging support and showcase its functionalities.

We first introduce a Twitter analytics application. This is an artificial example that we created, which analyzes tweets, which are constantly read from the Twitter stream, running on a distributed system. We will refer to this case as the *"Twitter Analyzer"*.

We then describe the continuous test integration system of the Belgian company Yesplan [Yes]. This system is in charge of executing all the automated tests of the software in production. We will refer to this case as *"Yesplan Testing"*.

Finally, we describe a core subset of a cyber physical system that monitors temperature, which can present different errors when parsing the result of the reading.

2.4.1 Use Case 1: Twitter Analyzer

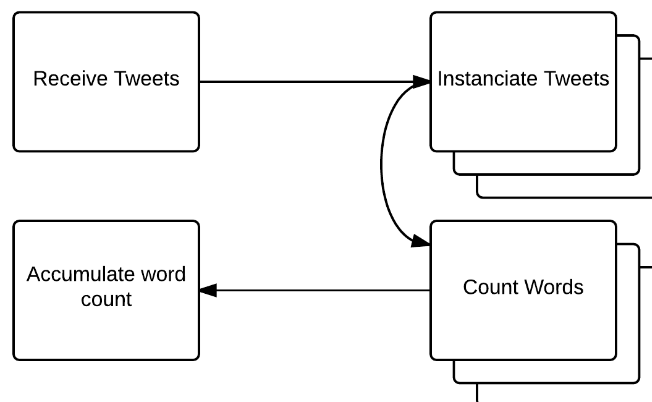

Figure 2.1: A model for the Twitter analyzer.

The Twitter use case represents an example of a data intensive application. An infinite random set of public tweets is read from the Twitter Streaming Public API [Twi], and then analyzed by reconstructing them from a JSON into a *Tweet* object. The application then outputs some statistics, e.g. a count of the words used in the tweets. An overview of the system is showed in Figure 2.1. Each node in the figure represents one operation, some operations, like *instanciate tweets* and *count words* can be executed in parallel.

This application can be modelled in a distributed architecture in which one node is the one connected, through HTTPS and SSO, to the API. We call this

node *master*. It parses the strings to recognize the different JSONs and then it sends those strings to other nodes, that will instantiate the *Tweet* object. The result is sent back to the master, that will instruct another node to analyze the tweet. The analysis of the *Tweet* produces a collection composed by words used and frequency, that will be returned to the master node. The master node will then merge the elements of the collection in an unique word count. An overview of the system is showed in Figure 6.1. Each square represents a function, and is placed either on the master on the other node, called *worker*.

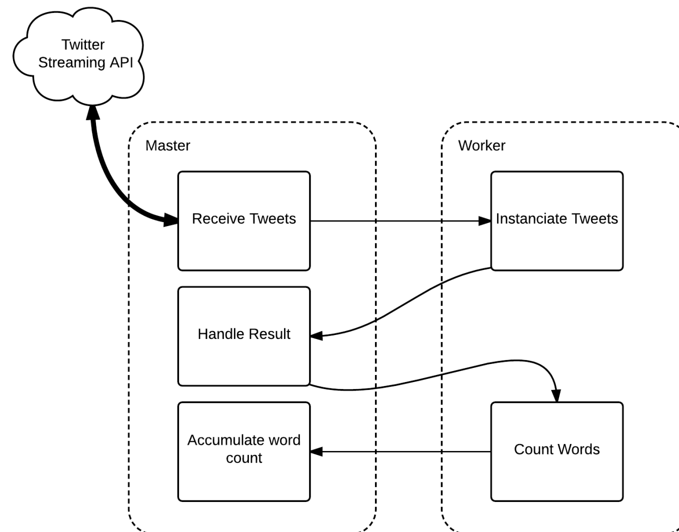

Figure 2.2: Twitter analyzer deployed on a master and a worker.

Note that the different operations executed in parallel do not have a shared state, so these operations do not contain any read nor write on a shared memory space. Only the single master node maintains a global state that contains all the words, and the associated count, summed for all the analyzed tweets.

2.4.2 Use Case 2: Yesplan Testing

Yesplan is a Belgian software development company whose main product is an event planning platform mostly developed in Pharo Smalltalk. More concretely, Yesplan uses a browser interface in Javascript which allows users to plan events (allocating time-slots, staff, resources, etc.). It has a Pharo Smalltalk back-end which provides all the logic of the application and the database communication.

In the development and deployment of Yesplan, many tests are deployed to check the correctness of the solution after applying changes and before releasing a new version to their clients. To test whether users actions are correctly executed, a browser environment is set up and the Selenium library [Pro] is used to simulate user actions on the web interface such as clicks, mouse movement, fill in of fields, etc. Different tests check that the right calls are made in the back-end. The rendered web-page is then checked to verify the correctness of the result shown to the user. Figure 2.3 shows a model of the testing setup.

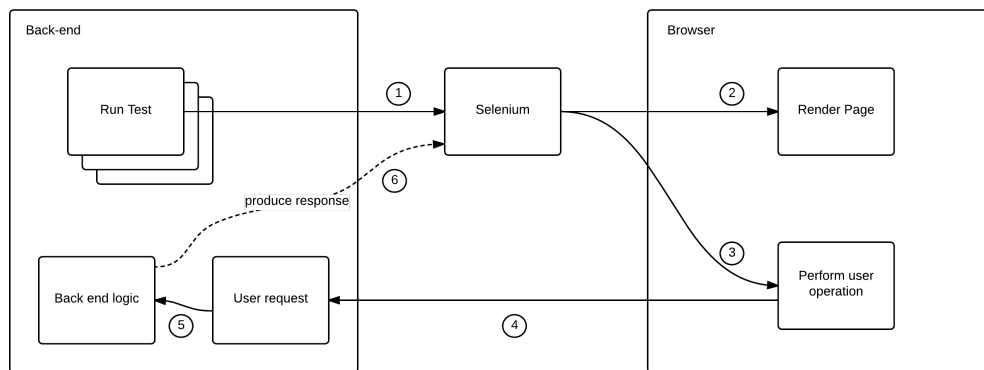

Figure 2.3: Architectural overview for the Yesplan testing scenario.

Every time a test is executed, a different setup is used:

- A different table and configuration of events is shown in the browser.
- Fields are updated with different values.
- The simulated click will not happen always in the same position.

This testing process is actually automatized. The tests are run in a different setup depending on the date, hour, and many other variables. This is done to add entropy to the executed tests, trying to cover all the possible cases that an end-user could encounter.

Due to all those variables, the setup on which the tests are run is non-deterministic. When a test fails a log is produced, but the non-determinism of the setup makes it impossible to properly reconstruct it. Many times it happens that, when re-executing the test, the Selenium engine simulates clicks on a slightly different position at the web front-end, and the test will not fail anymore. The company representative we talked to acknowledges that it is really rare that a version is released with all the test passed, just because of

this non-determinism. Furthermore, the Yesplan tests are executed remotely, which makes it really difficult to locally debug the tests on that machine. As a result, most of the times failing tests cannot be reproduced and the information gathered when a test fails is very minimal.

2.4.3 Use Case 3: Sensor Monitoring

The last motivating example is a core subset of a Cyber Physical System (CPS) for sensor monitoring. This monitoring system is made of a Raspberry-pi connected to a temperature sensor and an LCD screen. We deploy the device in the room that we are interested to track. The sensor probes the room's temperature and displays the result on the LCD screen. This device is connected to the internet via *wi-fi*. It can be configured to send alarm notifications to a remote user if the temperature of the room exceeds a given level (*e.g* in a food storage room). The communication is actually bidirectional: the device can also receive updates such as user application and firmware updates.

2.4.4 Conclusion

In this thesis we want to provide a debugging support for non-stoppable applications, such as our *Twitter analyzer*, the *YesPlan testing* scenario and the *sensor monitoring* scenario. They all share the main concern that their execution should not be stopped if an error occurs, and they need to be debugged remotely.

In the next section we analyze different current debugging solutions for such applications through a literature study.

2.5 State of the Art

In the development of modern applications developers spend most of their time debugging and verifying their code to find structural or behavioral errors [HS01]. There are different approaches to debug an application. Based on the paper from Pacheco [Pac11] and the survey from McDowell et al. [MH89] we categorize most of the debugging tools in two big families: *Offline Debugging* and *Online Debugging*.

Terminology

Before analyzing different debugging techniques present in literature, we discuss the terminology used in this thesis. We base our terminology on Avizienis

[ALRL04].

A **failure** is an event that occurs when a system provides different services from the one that it was designed for. An **error** is the difference between the service provided and the theoretically correct one, that might lead to a **failure**. The **fault** is an incorrect part of the program that might cause an **error**. If it does is called **active**. Otherwise if is present and not manifested is called **dormant**. A **crash** is a **failure** that leads to the end of the execution of the program.

In this thesis we use the term **bug** to identify a **fault**.We call **debugging** the process of analysis a **failure**, identification of a **fault** and correction of the **fault**.

2.5.1 Offline Debugging

Offline debugging refers to analysing programs after the execution of the program completed, failed or terminated with an unexpected result. In literature it is also referenced as **post-mortem debugging**, and is nowadays widely used in many systems, from cloud computing to operating systems and simple applications.

Logs and dumps

The simplest example of offline debugging, mostly used by inexperienced developers new to a particular platform, is the ***printf debugging***. This consists of inserting in the code some console prints that then it allows the developer, during and after the execution, to analyze if a state or if a particular branch is reached. Even though prints happen at runtime, the developer cannot do any immediate operation on the code, nor pause the execution to analyze the state. Although this system is pretty immediate and does not add considerable overhead, it is difficult to deploy in production. Many applications are not deterministic, so a simple re-execution of the code will not produce the same output as the one that produced a crash, making the use of `printf` not sufficient to understand what caused the crash.

When debugging such undeterministic systems, a complete view of the system is necessary to analyze the state that made the program crash. This was achieved introducing **core dumps**, a screenshot of the whole state of the program when a crash happens, and providing interpreters for these dumps [MM80]. Different well-known debuggers like the *GNU project debugger (GDB)* [GNU] offer primitives to navigate through core dumps and analyze the state of the execution by means of assembly commands and the state of the variables by means of pointers. Those dumps are normally generated by the Operative

System, but when running higher level languages, especially if they are executed on top of virtual machines, a dump generated by the operative system does not offer the required level of abstraction to analyze the state of the program at the moment of the crash. Furthermore, a virtual machine will probably not crash if the program crashes, avoiding the intervention of the operating system to generate a dump.

To conclude, a core dump does not provide enough information to debug high level programs, since it only provides the state of the memory and a call-stack, that in many occasions are not enough to totally understand the nature of a bug. For example the value of the function arguments are not provided, nor the values referenced in the different levels of the stack.

Trace-based debuggers

Debugging high level applications using dumps does not provide much information to reconstruct a proper debugging section. This happens because it only records the state at the moment of the crash and not the single steps and different values of the variables that would allow to totally reconstruct the environment of the crash. To debug high level applications those different steps and values are recorded in a **trace** . Debuggers can offer different ways to analyze this trace:

- Browsing: A tool is offered to analyze the events history and possibly to show the relations between events and state changes.

- Replay: A debugger analyzes the trace and re-executes the program following that trace. The debugger allows then to use online debugging primitives such as breakpoint and stepping to examine the state of the program without altering its behavior.

In both approaches the recording of the execution is crucial, since it can introduce high overhead in the execution. Furthermore, in concurrent and distributed systems, a partial order of variable access or events (such as message sending/receive in message passing systems) has to be stored in the trace. We describe now different solutions for replay debugging.

In concurrent systems, Leblanc [LMC87] supposes that, given some parallel processes, the overall output is non deterministic. But for each single process, given an input, the output of that process can be processed again in a deterministic way. There is no need to store all the output value of the processes but only the input values that then can produce the error again. Netzer [Net93] optimizes the tracing, including only the variables subject to race conditions, which improves the performance of the recording. DrDebug

[WPP+14] allows the developer to select the region to record, and uses cyclic debugging on *dynamic slices* of code that played a role in the computation of a value, and allowing to fast forward the part of code developers are not interested into.

In distributed systems, Ronsse [RK98] proposes to use Lamport clocks to trace order of access to variables in the distributed memory.

In Friday [GAM+07], the library *liblog* [GASS06] is used to record the side effects of non-deterministic calls also using Lamport clocks, then the execution is re-played with the possibility to add watchpoints, breakpoints and to query into the distributed state of the system.

Omniscent debuggers

Replay and dump debuggers offer both a view of the system, one allowing to replay the situation that led to a crash, one that offers a view of the state of the crashed system. In none of those systems is possible to have, at the end of the execution, a complete view of all the state changes, i.e. any variable set that happened during the execution. In order to have a complete view of the system at any moment, an *omniscient debugger* can be used [Lew03].

An omniscient debugger stores all the variable changes and the function calls that happened during the execution. At the end of the execution the developer can open a debugger and navigate through the state of the program at any moment in time, seeing when a particular variable was set or changed, or when a particular event happened.

As in the *replay debuggers*, recording all the execution causes high overhead and produces a sensible amount of data when the execution time of the debugged program increases. Also querying and navigating that data must be responsive to the user. For example, to tackle this problem, in TOD [PrT09] a database is used to store all the events, and a query manager is provided to go through all the events in an efficient way.

Conclusion

Offline debuggers are undoubtedly diffused in the context of distributed applications, and, especially *replay* and *omniscent* debuggers can provide a clear view of the system to debug a failure. However, the overhead introduced to produce a trace of the execution is high [MH89]. The fault is debugged when the system finished running, that in a data intensive application could mean hours of computation lost [GIY+16].

2.5.2 Online Debugging

Opposite to offline debugging, *online debugging*, often called **breakpoint-based debugging**, controls the execution of the program through different facilities like pausing/resuming execution and step-by-step execution. The main operation offered by an online debugger is the possibility to place a **breakpoint**. A breakpoint pauses the execution of the program and allows the developer to investigate the current state and, depending on the language, to access some data to analyze the flow of the computation, like a *stack-trace*. After stopping via a breakpoint, the developer can *step-in* a particular call to check what its code does, or *step-over* a call to directly jump to its result. Some debuggers also provide a *step-out* command that executes until the end of the current executing function, stepping to where it returns.

A developer can insert a **conditional breakpoint**, that will be activated only when a specific condition is fulfilled. In some systems, the developer can add a **watchpoint**, which will halt whenever the value of a particular variable changes, without needing to provide a specific condition. An online debugger also offers an **evaluator** that allows to evaluate a particular expression in the halted context, which is really useful to inspect the state of the program.

When applying these debuggers to a distributed system we find solutions like CDB [WCS02], TotalView [Got09] and REME-D [GNV+11] which allow to place breakpoints in distributed nodes, stopping their execution and allowing to analyze the state.

Interactive Debuggers

Live programming platforms like Smalltalk offer an interactive environment [Gol84] in which the developer can directly interact through reflection with all objects of the system, including classes, instances, environments and contexts. This structure of the environment makes it possible for a debugger to intercept a failure and show to the developer a complete browsable view of the system in that moment. In these systems, introducing a code change in the debugger will automatically reflect on the running environment, allowing incremental updating while debugging and easing approaches like *test driven development (TDD)* [ABF05].

A representative example of these techniques is the Pharo Debugger [BNDP10], a debugger for the Pharo SmallTalk implementation[Pha]. It offers a full debugger with the breakpoint and stepping commands. It also offers a complete view of the stack, which is represented by an accessible object, and allows to restart the execution from a particular context of the stack, hot-swapping the updated code when necessary.

Conclusion

We believe that online debuggers are more suitable to debug data intensive application, or, more in general, applications that should not crash before we can find a bug in them. However, the main concept of an online debugger is a breakpoint, which stops the execution allowing to use the stepping primitives that, considering an application that is reading from an unstoppable stream of data, might lead to a loss of data. These arguments show that classic online debuggers are not perfectly suited to debug such applications.

2.6 Remote debugging

When debugging an application we normally have two logical processes running: one that is the application itself, and one that is the debugger. We talk about **remote debugging** when the process of the debugger is running on another machine than the debugged process. [PBFD15]

Non-stoppable applications are normally deployed in a distributed system, hence it is interesting to analyze this debugging technique.

Remote debugging is orthogonal to offline and online debugging. So, there exists a number of techniques that provide remote debugging for online debuggers and offline debuggers that we review below.

Offline remote debugging

Offline debugging happens once the application finished by analyzing and inspecting the application logs and dumps. Thus, offline remote debugging can reuse the same techniques explained in Section 2.5.1.

In the case of a log-based offline debugger, the failed execution will probably generate a log. Having access from a remote machine to that log is enough in order to apply the same debugging constructs as a developer would do locally (i.e. going through the log).

If the offline debugger is a replay debugger , we would need to transfer the produced trace to the machine where the debugger will run, to then execute the replay on it. The developer produces a fix on the debugging machine. That fix needs to be transferred and compiled on the machine that is executing the debugged program.

Online remote debugging

An online debugger, as we saw in Section 2.5.2, allows to interrupt the process of the application and interact with it by means of debugging operations such

as *step-in*, *step-over*, etc. In a remote setting the debugger process is in a separate machine than the debugged program, which means that a solution is needed to control directly what is executing on a remote machine without altering its results. The remote program needs then to be updated, better if automatically, with the fix provided by the user.

2.6.1 Online remote debuggers

Different solutions in different programming languages exist for remote debugging, some of which are listed in [PBFD15]. Many mainstream languages offer remote debugging facilities.

The Java Virtual Machine offers a debugging framework called JPDA [Orab], which consist in a remote interface called *JDI* [Oraa], a communication protocol and a debugging support on the debugged virtual machine, implemented at virtual machine level. A system to automatically update modified code on the remote machine is not provided: the user has always to manually run the application on the remote virtual machine in debug mode, and set up the two virtual machines to communicate.

GDB [GNU] offers remote debugging facilities for Objective-C, that can be activated compiling the application in debug mode on the remote machine and activating the *gdb-server*. It also offers a limited mechanism to update the code on the remote machine through a patch of the executable memory.

Microsoft also provides remote debugging facilities similar to the Java ones in its framework .NET through its IDE Visual Studio [Micb].

Similarly, Pharo Smalltalk [Pha] also offers a remote debugging facility, which will be further analyzed in the next section.

Remote interactive debugger

Mercury is an interactive remote debugging model based on reflection [PBFD15]. The Mercury prototype is currently named PharmIDE [Kud] and is part of the remote tool suite of the Pharo language. Mercury uses mirrors [BU04] are used from the debugger machine as interface to represent the objects of the debugged machine.

Mirrors are objects that encapsulate all the meta-level facilities of an object. They make the meta-level facilities independent from the implementation, keeping a direct correspondence with the structure of the object they are mirroring. As we can see in Figure 2.4, every mirror in the development machine contains a reference to an object on the debugged machine, referenced as target in the figure.

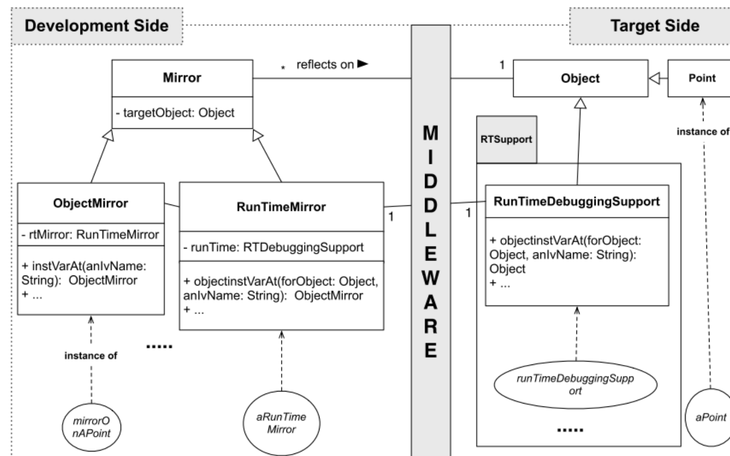

Figure 2.4: Core model of Mercury (Extracted from [PBFD15])

Every call done through mirrors from the developer side are applied in the correct reference on the debugged side, with a slight difference from a normal execution: every value returned to the developer side is returned as a mirror. This approach will basically wrap all the debugged environment in mirrors, allowing from the debugger machine a total manipulation of the objects in the debugged side handled from the meta-level.

We have to consider that in Smalltalk everything is an object, including classes, which means that for example modifying a method means modifying the object correspondent to the method, that actually contains the compiled bytecode to be executed. Changing a method on a class normally reflects to an immediate recompilation of the method, which from then on will be executed with the new bytecode.

In order to have run-time evolution of the code the mirrors have to be *causally connected* to the objects in the debugged image, meaning that any change in one of the sides has to be reflected in the other. Applying changes to a mirror means that they will be immediately be changed also on the remote reference, so in the debugged machine. The model of Mercury supports both *structural reflection* and *computational reflection*. *Structural reflection* reifies any change to the structure of the program (such as class addition, etc.). *Computational reflection* is the reification of the contexts and processes [Mae87].

Debugging remote promises

When debugging remote processes it could be useful to have a clear relation between the operation that is debugged and the one that spawned it. Leske

and al. [LCN16] proposed a model which extends Mercury to debug remote promises. Remote promises represent a result of a remote computation, implemented with proxies in their model.

When a failure happens during this remote computation they propose to use the Pharo remote debugger to debug it. While debugging, they show that normally only the stack of the remote process will be shown, not the one of the main process that executed the call.

This makes debugging more difficult because the cause of the bug could be in that previous call so they propose a model where, relying on the current remote debugging infrastructure, they combine the stack of the caller with the stack of the remote process that failed, giving a more clear overview of what caused the bug.

Conclusion

Remote debugging is a really powerful tool, especially when dealing with distributed systems or clusters, which would be difficult to access singularly to be debugged. Remote debugging allows to use all the classic online debugging capabilities in a remote machine, but, as a normal online debugger, it will stop the execution on the debugged machine when a breakpoint or a failure is encountered.

2.7 Debugging non-stoppable applications

In this section we review existing approaches to debug non-stoppable applications, such as data intensive applications. These applications are normally deployed in cluster systems like Hadoop MapReduce [Apac, DG08] and Spark [Apab] which allow high parallelization of jobs.

To date, those programming platforms offer very few debugging support. A recurrent approach is to debug those application post-mortem using logging. Instrumenting the code is easy and it introduces few overhead. However, the resulting logs are extremely difficult to debug, especially because of the amount of events being logged due to the flow of big amount of data. In the remainder of this section we review some solutions for offline and online debugging of data intensive applications.

2.7.1 Offline debugging

As for classic distributed applications, some of the post-mortem solutions include *replay debuggers*. The systems studied in this thesis allow only to have parallel tasks not directly dependent on each other, and not modifying a shared

memory, since they are programmed in a functional way. This property reduces the trace that has to be recorded, needing to store only the input values of particular tasks, that then can be re-executed and will give the exactly same result, since they do not read from a shared state that might have been modified before the re-execution.

Graft [SSK+15] is a debugger for Apache Giraph [Apaa] that allows to select particular jobs to capture their execution, and then replay the execution later, but requires previous identification of the faulty nodes.

Arthur [DZSS13] allows to perform a post-mortem analysis on the failed process, requiring multiple re-executions to access the intermediate results that caused the failure. This is pretty time-costly when debugging it, since it also requires to write particular queries in order to replay the correct tasks.

2.7.2 Online debugging

Post-mortem approaches are surely useful and scalable to this kind of application, but the bug might be found after many hours of computation lost [GIY+16]. Daphne [JYB11], a debugger for DryadLINQ [Mica], tries to tackle the problem, allowing a runtime view of the system and the query nodes generated by a LINQ query. It allows to add breakpoints to inspect the state and start and stop commands through the Visual Studio remote debugger. Debugging is done directly on the client where the breakpointed node is executing, interrupting it in order to debug it.

We believe that the most advanced debugger in data intensive applications is BigDebug [GIY+16]. Being the closest related work, in the next section we will analyze it in details.

2.7.3 BigDebug

BigDebug [GIY+16] is a debugger for Apache Spark [Apab] that offers a set of interactive real-time debugging primitives, that allow to perform debugging on Apache Spark with a considerable performance improvement from other debugging platforms for Spark like NEWT [LDY13], a tool for capturing dataflow that allows replay.

BigDebug relies strongly on the architecture of Spark, especially on *Resilient Distributed Datasets (RDD)* [ZCD+12]. RDD is a data structure that allows to store a variable as set of transformations from an initial value. When a user writes its function that manipulates an input data, this is interpreted as a series of RDD transformations, which are grouped by stages, as we can see in Figure 2.5.

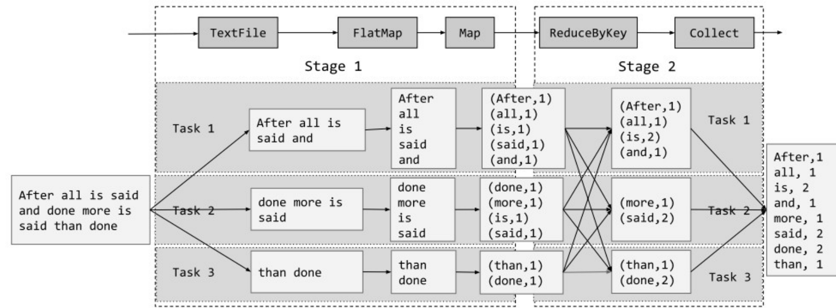

Figure 2.5: Data transformations of a word count (Extracted from [GIY+16]).

RDDs are stored between stages, in memory or fully persisted depending on the configuration, and can be used to restart a particular stage without needing to recompute all the previous stages, but only the lineage of the RDD when needed.

Simulated breakpoints and guards

BigDebug introduces the concept of *simulated breakpoint*, which can be placed between any transformation of a stage, as we can see in Figure 2.6

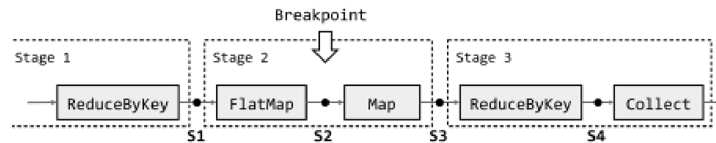

Figure 2.6: Simulated breakpoint in a stage (Extracted from [GIY+16]).

It is called *simulated* because it does not stop the execution, nor freezes all the different nodes in the system waiting for the resolution of the breakpoint. It stores the information necessary to replay the environment, namely the RDD in the point `S1`, so the state of the input of the stage where the breakpoint is, and the different steps to get to `S2`, and notify a driver node, then continue the execution. When stepping in that breakpoint, the node will restart from `S1` and execute until the point of the breakpoint (`S2`). The debugger will then show the state of the variables in that moment. After inspecting the state the user can provide a fix, which will be hot-swapped in the worker before the execution is resumed.

They BigDebug also allows to add a **watchpoint** to an RDD, using *guards*. In a watchpoint a function is provided by the user to check whether an RDD satisfies a predicate, and in that case send the data back to the driver, from which the user can inspect the intermediate state.

Crash remediation

When a crash happens in a Spark node, normally all the stages are aborted, with their intermediate results. BigDebug allows those stages to continue for all the pending tasks, while waiting for the intervention of the user. If there are multiple crashes BigDebug accumulates them, waiting for a fix from the user. Assuming that the provided fix is a correct extension of the old (not working) function, all the crashed tasks are re-executed from their last stage before crashing. At this stage it is not possible to add a completely new function. This is only possible when fixing the code from a simulated breakpoint, and an output will not be shown to the user if the tasks fails.

Conclusion

The approach adopted in BigDebug is definitely the closer to the requirements of the non-stoppable applications. BigDebug offers simulated breakpoint, that do not stop the execution of the other workers, and concludes its jobs at the moment of a crash. It also allows to remotely debug distributed nodes, which is convenient on a distributed setting and automatically propagates a code fix to all the nodes in the system. However we believe this approach has some limitations:

- A complete code fix can be provided only through a breakpoint, not after a crash.
- When debugging a simulated breakpoint, developers debug directly on the node: this means that the node is freezed, responding to our debugging operations.
- If developers provide a wrong code fix, they cannot properly test that it works before applying it to all nodes.

2.8 Problem Statement

The overall goal of this thesis is to study debugging support for distributed non-stoppable applications. As shown in Section 2.5, many solutions exist to debug distributed and concurrent programs (online or offline, in place or remotely). We observed that nowadays they are often implemented using programming models such as MapReduce [DG08] and Apache Hadoop [Apac]. Developers of those platforms typically make use of execution logs through the so called *printf debugging*, which does not add any overhead at run-time, but it often leads to hours of computation loss because of corrupted data and other time

lost to go through the logs, as reported in [FDCD12]. More advanced *post-mortem* debuggers exist, like Arthur [DZSS13], that allows to replay failed execution. However, replay debuggers do not avoid the possible crashing of nodes, and add high overhead when recording a trace.

Alternatively, some online debuggers implementations like (BigDebug [GIY+16]), exist for the platform Apache Spark [Apab]. It offers a limited set of debugging primitives in order to debug by means of breakpoints and watchpoints, remote nodes that might cause a crash. BigDebug does not provide crash avoidance, but rather, it enables analysis of a crash culprit to re-create the same conditions that led to the error. BigDebug tackles this problem by persisting in different points the state of the computation, and then replaying it when needed. The only difference with a classic replay debugger is that they replay from intermediate points of the computation to then stop in a certain point. This is possible because Spark applications are programmed in a functional style, setting up tasks does not share state and perform pure parallel computations.

In this thesis, departing from a pure functional style of programming such applications, we will employ object oriented programming for implementing data intensive applications. The goal then is to study a debugging support for an object oriented language that needs to tackle different problems that cannot necessarily be expressed in the means of parallel independent tasks nor as functional programming constructs.

From the analysis of the characteristics of non-stoppable applications and the conclusions from the literature review, we argue that the debugging support should be able to locally debug remote exceptions. It should also be able to deploy the changes the developer applies during the debugging session. We detail those aspects in the sections that follow.

Locally debug remote exceptions

When an exception occurs on a remote machine that we are debugging, we want to be able to debug while this remote machine keeps executing other jobs. Debugging directly on the remote machine might require physical access to it, and eventually would stop the execution in order to properly debug. Using a classic Remote Debugger [Pap13] might resolve the problem of the physical access, but does not provide a solution to debug a machine while that is still executing other processes. A work-around could be to execute the debugger in a parallel process to the executed jobs, but then, other than introducing some code update problems (analyzed in Section 2.8), we would need to wait that the machine finishes a job to then reschedule the process we are debugging.

Ideally a debugger for data intensive applications should allow us to see locally on the debugger machine what happened remotely, without affecting the remote machine during this process. It is clear that a Remote Debugger with mirrors would influence directly the objects on the remote machine, so we need a system that takes a snapshot of the exception happened and sends it to the debugger, which then will reconstruct it and debug it locally.

Being in Object-Oriented programming and considering that we are debugging non-deterministic applications, this brings the problem of handling a stack, which represents the current state of the program when an exception happened. We need to extract the stack and the different referenced variables in the heap and send it to the debugger machine, so a full picture of the exception can be seen. To this end, we believe that an interactive debugger as the one employed in Pharo provides a good foundation to allow such manipulations.

Deploy changes

When remotely debugging one machine, the code changes provided in the debugging section need to be applied also in the remote machine we are debugging. We saw that only few remote debuggers (like GDB) offer this facility, while is more common in debuggers for distributed clusters like BigDebug [GIY+16].

In systems like those, the updated code will automatically start running on all the nodes: in case the new code does not introduce any new bug, this will not give any problem. Instead, if the new code produces new bugs, all the nodes might suddenly start producing errors or crash. To avoid this, we need a debugger tool that allows the developer to try a fix before deploying it on a running system, eventually multiple times in order to provide a correct fix.

Out of place debugging

The proposed solution allows to debug remote applications **out of their original place**. In other words, the proposed debugger transfers the execution state of the remote application to the local developer's machine. The developer proceeds then to debug as if the application was originally a local application. Anyhow, the remote application is not influenced by the operations of the developer until he decides to resume the failed processes with an updated version of the code. We call this debugging technique **out of place debugging**.

2.9 Conclusion

In this chapter we presented the problems targeted by this thesis, describing different types of non-stoppable applications that we aim to debug. A *non-stoppable application* is an application that is run remotely and cannot be stopped to be debugged, nor can stop working because of a failure.

We analyzed then the state of the art, to check for existing solutions both in online and offline debugging which could suit *non-stoppable applications*. We found interesting approaches for remote debugging, such as Mercury [PBFD15], and for debugging *data intensive applications* (i.e. BigDebug [GIY+16]). Anyhow, those approaches do not totally fulfill the requirements of non-stoppable applications because they still allow the execution to stop on a failure.

We propose IDRA, a debugger designed specifically for non-stoppable applications. IDRA is designed to be employed in a distributed application, and does not stop the nodes where a failure happens. Instead allows to debug them in another node of the system using an out-of-place technique. A committing system is used to propagate code changes to all the nodes of the system.

An interactive debugger such as the Pharo Debugger [BNDP10] (cf. Section 2.5.2) represents a good base for IDRA because its implementation is provided in Pharo and is easily extendible. Also, the Pharo language reifies the concept of call stack, which allows us to directly manipulate execution information.

However, Pharo has limited support for distributed programming for particular applications, such *data intensive applications* that we treat in this thesis. To this end, we implemented a framework for the *master/worker* architecture in Pharo, which will be explained in the next chapter. Afterwards we will give a detailed overview of IDRA.

Chapter 3

A distributed programming model for Pharo

Data intensive applications normally run on distributed architectures that allow high parallelization of computation. In Pharo Smalltalk, the platform we selected to implement our approach, such architecture is not yet available. As such, we designed and implemented a distributed programming model to Pharo to this end. In this chapter we analyze the concept of a *master/worker* architecture and how we designed it for such distributed model.

3.1 Master/Worker Architecture

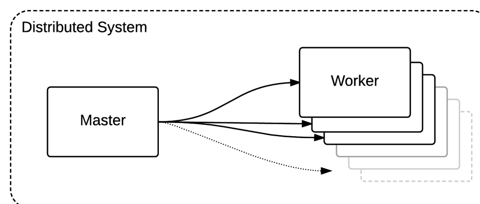

Figure 3.1: Overview of *master/worker* architecture.

The *master/worker* architecture is a concurrent and distributed model that allows to execute parallel jobs in a coordinated way. In this architecture a node of the system takes the role of *master* and coordinates the other nodes, assigning jobs to them and retrieving their results [ANF03].

An overview of the *master/worker* architecture is shown in Figure 3.1. Considering a distributed system where we want to perform parallel computation we can assume that a single node, called the *master*, will be instructed

on which tasks have to be computed on which data. The master node orchestrates the rest of the other nodes in the system, that we call *workers*, and sets up a communication channel with them to this end. The master can instruct *workers* to execute a certain computation on certain data by means of a *task*.

The goal of the *master* is to evenly divide the work (i.e. tasks) between all its *workers*, so all the computational power of the system can be exploited. In order to accomplish this goal, as shown in Figure 3.2, the *master* includes a *scheduler*, that might be implemented as component or as scheduling function, which extracts a task from the task list (2) and selects a *worker* (3) on which the task will be executed (4). After the workers process a task, they might return a value that has to be handled by the *master* (5). Depending on how the master was instructed and how it divided the work, it might feed the return value into another task or persist that value somewhere in memory.

A scheduler can select workers in different ways. Tasks can be divided by time slots, by their complexity or, in more advanced distributed systems, they can also be assigned by distance and ping time.

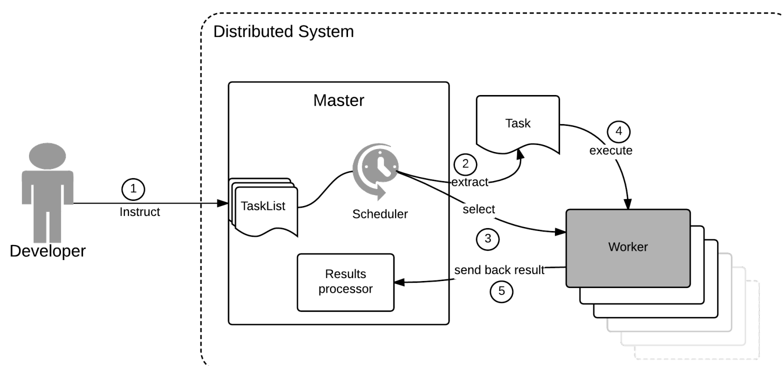

Figure 3.2: Worker selection phase in *master/worker* architecture.

Advantages and limitations

The *master/worker* architecture is an easy approach to divide the work between multiple workers. Most of the complexity is in the scheduling function, that can be tuned depending on which platform this architecture is deployed on [GKYL01, ANF03].

It is, however, a centralized architecture where the *master* plays a decisive role which makes it the single point of failure of the system. Moreover, a failure in a *worker* can be easily handled by the *master* node, but if the *master* fails or is disconnected from the network the whole system could stop working. As

consequence, the *workers* will not receive more tasks from a *master* node and will not be able to return the processed results.

This limitation can be overcome by having several masters in the system, which can take control once one master fails, or introducing a load balancer which selects and instructs the master.

3.2 Design and Implementation of Master/-Worker Framework in Pharo

To the best of our knowledge, there is no *master/worker* framework implementation on top of the Pharo language[Pha]. Papoulias [PBFD15] proposed an approach that enables Pharo to communicate with a remote machine through proxies, but we did no encounter an example of a distributed architecture such as *master/worker*.

We propose an implementation of *master/worker* architecture as framework, which manages the communication layer and leaves to the developers the specifics of task scheduling an execution. For this, our framework provides different abstract superclasses, following the concepts of object oriented programming. The API is composed by two classes, `Master` and `Worker` that provide all the infrastructure for scheduling, connection and communication between *master* and *workers*.

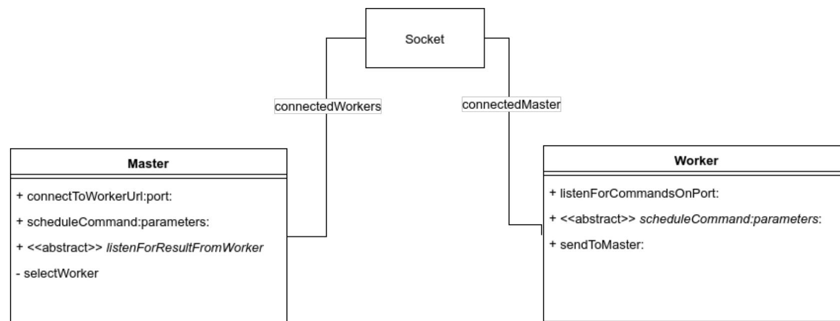

Figure 3.3: Class diagram of the *master/worker* Pharo API.

Figure 3.3 shows an overview of the API. In our implementation a task is identified by a `command`. When a `Worker` receives a `command`, it will execute the code associated to that command in a new task.

A `Master` offers the following functionalities:

- `connectToWorkerUrl:port:` allows to connect a `Master` to a `Worker`. The URL and port number of the `Worker` are needed.

- `connectedWorkers` holds a reference to all the communication channels with the `Workers`. A `Master` can be connected to many `Workers`.
- `scheduleCommand:parameters:` is called to schedule a task in a `Worker`. This is done specifying a particular `command` and some parameters that will be sent to a `Worker`, which will then construct a task using that `command` and those parameters.
- `selectWorker` implements the actual scheduling, selecting an available `Worker` to which a command can be sent. It is used internally by `sendCommand:parameters:`.
- `listenForResultFromWorker:` is called when a new connection is setup with a `Worker`, and listens for values from the `Worker`. This method is *abstract*, so it needs to be implemented in the desired implementation of a `Master`.

A `Worker` provides:

- `listenForCommandsOnPort:` used to setup a communication channel to which a `Master` can connect.
- `connectedMaster` is a reference to the communication channel setup when a `Master` connects.
- `scheduleCommand:parameters:` is called when a worker receives a particular command through the method `listenForCommandsOnPort:`. This method needs to recognize the command sent by the master and processes it, creating and executing a task. It is an *abstract* method, so an implementation of this method needs to be provided if a `Worker` is implemented, so the user can specify different commands and ways to execute them when extending this API.
- `sendToMaster:` is provided to send back to the master a particular result for a task.

3.2.1 Communication between Master and Worker

Being in a distributed system, *master* and *worker* will be running in two different logical processes. This means that an inter-process communication layer needs to be implemented. In Pharo we can assume that all the instances of a particular session are running in a unique virtual machine, which can run many Pharo processes.

Each logical process contains all the objects representing the classes of the system and it holds reference to *global instances*. These global instances include all the global objects, such as an internal scheduler, the UI process, and many other objects that are necessary to run the logical process. One logical process also contains local instances of a particular process that is run on the system, and among this local instances we have on the *master* process a *master* object, and on the *worker* process a *worker* node.

TCP Sockets are used to provide communication between two logical processes, that might be executed on the same machine or also on different machines connected in a network.

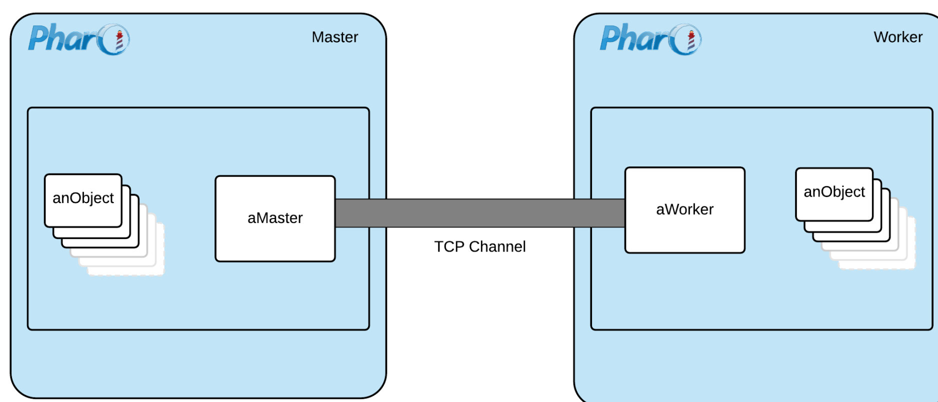

Figure 3.4: Communication of distributed *master* and *worker* on Pharo images.

Figure 3.4 shows an overview of a *master* and a *worker* node, with the two *master* and *worker* entities connected through a TCP channel. A *worker* starts listening on a port of its choice, and then a master can initialize a TCP connection on this port. From then on, all the communication will happen through that socket channel, and objects will be serialized to be sent through a library called *Fuel* [DPDA11].

3.2.2 Scheduling tasks on the Worker

`Master` and `Worker` are actually run each on a logical Pharo process, on two different virtual machines. In our implementation, each logical process includes a scheduler. When a worker receives a message it will delegate the execution of the task to the user defined code. A possible implementation of this is to start the execution of the task in a separate thread to not block the thread managing the communication. In Pharo, threads are implemented as green

threads. Green threads are used to execute different logical threads on the same virtual machine with some internal scheduling.

Our *master/worker* architecture employs the TaskIt library [Bra] to improve the scheduling on the worker. TaskIt allows to easily schedule tasks in new processes on the same virtual machine, handling the order in which tasks are executed and managing the concurrent execution or the execution one by one. The user can choose to use this functionality using the instance of `TKTRunner`, offered by the TaskIt API, present in the `Worker` class.

Note that TaskIt does not provide distribution, but improves the local scheduling of new processes within the same virtual machine. In our use cases we can use it as internal scheduler in the `Worker`, allowing it to receive different commands from a `Master` and to schedule the associated tasks one after the other.

3.3 Conclusion

In this chapter we introduced the *master/worker* architecture, widely used to deploy data intensive applications. It consists of a distributed model containing one node called *master* that controls the other nodes of the system, called *workers*. The *master* can schedule different tasks on the *workers*, allowing a parallel execution of different tasks.

Since, to the best of our knowledge, there is no distributed model implemented in Pharo, we implemented this model to allow us to build data intensive applications in Pharo. Therefore, this is a technical contribution of the thesis.

In the next chapter we will give an overview on IDRA, an out-of-place interactive debugger capable to debug data intensive applications running on architectures such as the one described in this chapter.

Chapter 4

IDRA

In this chapter we present IDRA, an interactive out-of-place debugger designed for data intensive applications and long running systems. In the first part of the chapter we describe the architecture and the components of *IDRA debugger*. The second, third and fourth sections describe how exceptions are handled, how they are reconstructed and how code changes are committed to other machines in the system, respectively. The last section shows the front end of IDRA in Pharo.

4.1 IDRA's Overview

Data intensive applications and long running processes both require a debugging technique that does not stop the execution of processes and allows to debug them remotely. As we argued in section 2.8 a debugger for such applications should handle more than one exception raised by one and many machines, and this debugger should allow to examine the program state at the moment that each of those exceptions happened, letting the running processes continue. To this end, we designed and implemented IDRA, an out-of-place debugger that allows to remotely debug those application without stopping them. IDRA is deployable in architectures such as the *master/worker* architecture (cf. Section 3.1) and in simpler remote setups where is difficult to access a machine that is executing long running processes.

IDRA allows developers to place a special kind of breakpoints in the code, which will not stop the *whole* program execution but rather just *one* particular process and allow the developer to perform step-by-step execution of the program from that point on.

While debugging, the developer is able to change the code of the application and to try this fix on the existent program state that lead to the bug being debugged. The code changes made by the developer during the debugging

session do not affect the running system, because they could introduce new bugs, that then would spread in the system causing more exceptions, or even block the whole application.

Once the developer is satisfied with the bugfix, such fix should be deployed to all the nodes of the distributed system which will then start executing the updated code.

In short, IDRA divides the debugging session in three main phases:

- **Handling exceptions/breakpoints**: operations necessary to handle an exception or breakpoint in the program when they occur. These operations do not impact the execution of the overall program, allowing it to continue executing.

- **Reconstructing exceptions/breakpoints**: operations necessary to reconstruct the state of the program upon an exception or breakpoint on a different remote virtual machine. This allows the developer to debug the exception or breakpoint without affecting the running program.

- **Fixing bugs and committing**: operations that record on the remote virtual machine the changes a developer makes to fix the bug and propagates them to the running system (i.e. all the processes of the system).

4.1.1 Debugger monitor and manager

To further explain IDRA we will employ a concrete example of distributed architecture on which IDRA can be deployed. Let us consider a simple distributed system composed by two machines, one that is executing different tasks, that we call the *worker*, and one that is the machine of the developer wanting to debug those tasks, that we will call the *developer*, as shown in Figure 4.1. In this example, the *developer* machine can be considered as *master* and runs an instance of IDRA. We call the instance of IDRA on the *developer* machine a *manager*. The manager allows the developer to retrieve and debug all the exceptions that might happen on the *worker* machine.

On the other hand, the *worker* machine also contains an instance of the debugger that we call *monitor*, listening for the exceptions to send to the developer machine.

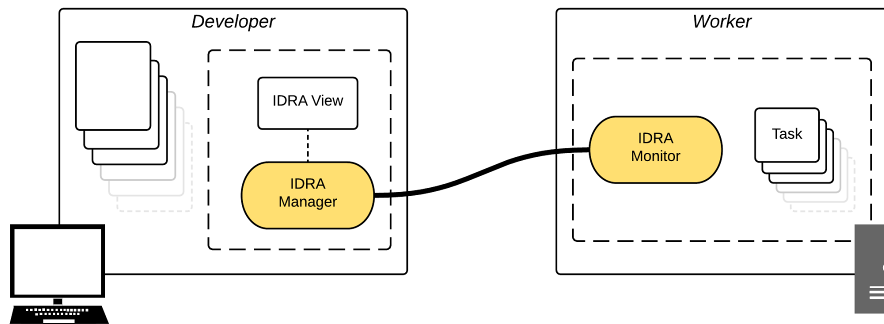

Figure 4.1: Representation of IDRA instances, manager and monitor, in a distributed system of two machines.

Figure 4.1 shows how IDRA can be deployed in the *developer/worker* model.

We now detail the conceptual steps that both monitor and manager performs during a debugging session. On the one hand, the *IDRA Monitor* needs to:

1. Intercept and handle all the exceptions and breakpoints happening on the monitored tasks.

2. Store the exceptions and the relative stacks and variables.

3. Send to an *IDRA Manager* all the exceptions, one by one.

4. Recover the failed tasks and re-execute them on command of the developer.

On the other hand, the *IDRA Manager* at the developer machine needs to:

1. Listen for exceptions happening on the connected *IDRA Monitor*(s).

2. Reconstruct the received exceptions and their relative stack.

3. Allow the developer to debug one by one those exceptions.

4. Allow the developer to re-execute the failed functions with an updated version of the code.

5. Allow the developer to interact with more than one exception at the time to verify the updated version of the code.

We will now consider a more complex distributed architecture where IDRA can be generally deployed. More concretely, we discuss a distributed master/-worker architecture, a recurrent pattern employed by big data applications. Debuggers such as BigDebug [GIY+16] are also based on this kind of setup.

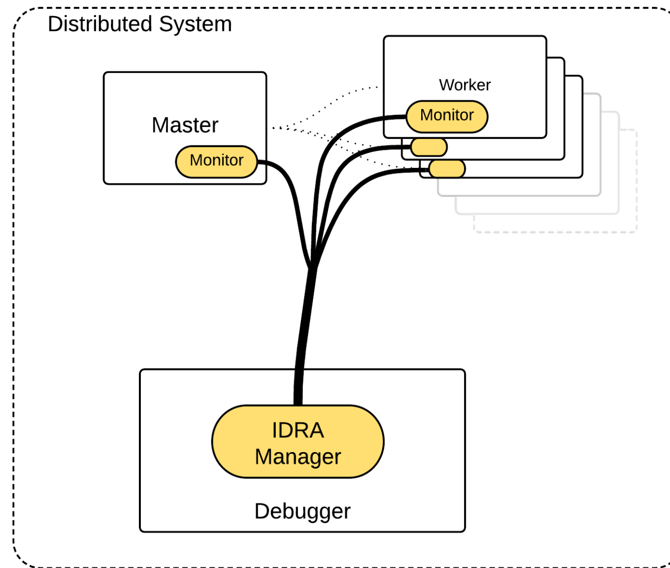

Figure 4.2: Representation of IDRA instances in the master/worker architecture.

Figure 4.2 shows how IDRA can be deployed in a distributed master/worker architecture. In this architecture there are one *master* node and different *worker* nodes.

In this case, the *IDRA manager* is placed in one of the nodes of the system, called *debugger*. The *debugger* node is the one that the developer needs to access in order to debug all the other nodes of the system. The *IDRA manager* node is connected to all the *workers* nodes of the system, and it is responsible of handling all the exceptions, as well as offering a developer interface to the developer to debug the application.

In the following sections we analyze in detail how the different phases are executed and we detail the different components that compose IDRA.

4.2 Handling exceptions or breakpoints

In this section we will describe with a concrete example how IDRA provides handling for exceptions and breakpoints both on the *monitor* and *manager*. Consider again the simple distributed system with one *developer* and one *worker* machine described in Section 4.1.1. In this scenario, a *worker* is running different tasks, which might generate an exception or trigger a breakpoint inserted by the developer. In the next sections we describe in detail the structure of *IDRA Monitor*, *IDRA Manager* and *IDRA Changes Handler*.

4.2.1 IDRA Monitor

The *IDRA monitor* provides the infrastructure to handle the exception before triggering the crash of the *worker*, and stores it to allow the developer to debug it. When an exception happens it is caught by the debugger and the *worker* will proceed to the next task (as IDRA allows the overall program to proceed). In order to capture the exception, the state of the execution needs to be extracted and copied.

After the *worker* continues its execution other exceptions might happen. This is the reason why an *exceptions queue* is kept in the *IDRA Monitor*. When an exception happens, the thrown exception, with all the relative data, will be added to this queue, waiting to be processed, as shown in Figure 4.3. The figure shows how the *IDRA Monitor* updates its state during this process.

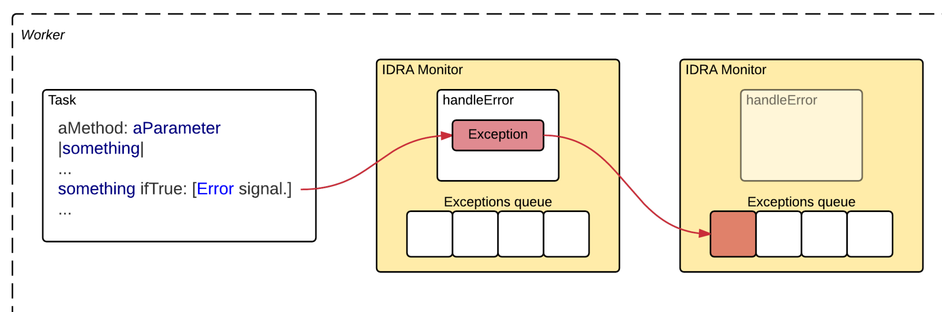

Figure 4.3: Handling of an exception in *IDRA Monitor* - Part 1.

Every time an exception is raised, it is added to the same exception queue. The *IDRA Monitor* checks continuously on the queue and takes the elements to be sent, one by one, to the *IDRA Manager* associated. Exceptions cannot just be forwarded to the *IDRA Manager*, but they have to be stored in a different queue, called *sent exceptions*. This allows to restart the execution from the point in which an exception happened in the worker after the code is

fixed. Hence IDRA avoids losing the data that was being processed when the execution stopped because of an exception.

Figure 4.4 shows a state diagram of the exceptions queue handling, showing how the *IDRA Monitor* updates during the different phases, though the function *processError*, which retrieves an exception from the *exceptions queue*. After retrieving it, it stores the exception in the *sent exceptions queue* and calls the function *sendException* to send it to an *IDRA Manager*.In the last step the exception is sent to the developer machine.

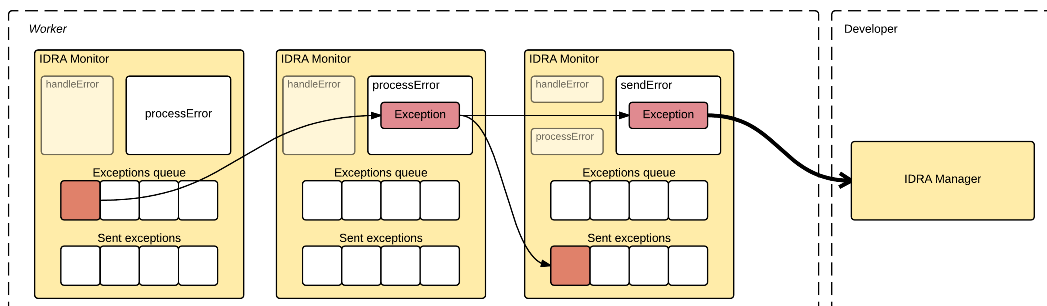

Figure 4.4: Handling of an exception in *IDRA Monitor* - Part 2.

4.2.2 IDRA Manager

Recall that on the *developer* machine an instance of *IDRA Manager* is running, listening for *exceptions* from the connected *worker* machine. When an exception is received, a debugger session is opened on it so that the developer can see what happened on the *worker* machine. This means that at the next exception received, another debugger session will be opened. However, if many exceptions arrive, many debugger session open on the developer machine. As a result it becomes difficult for the developer to debug.

To alleviate this issue, the *IDRA Manager* employs a queue where the exceptions arriving from the workers are kept, allowing only one debugging session to be opened at a time. When a developer terminates debugging one exception, another exception will be processed from the queue. Figure 4.5 shows an overview of these operations in the *IDRA Manager*. The figure shows how the state of the *IDRA Manager* is updated through different functions, namely *receiveError* and *handleError*, after receiving an exception from the *IDRA Monitor*. In the final stage a debugging session is opened.

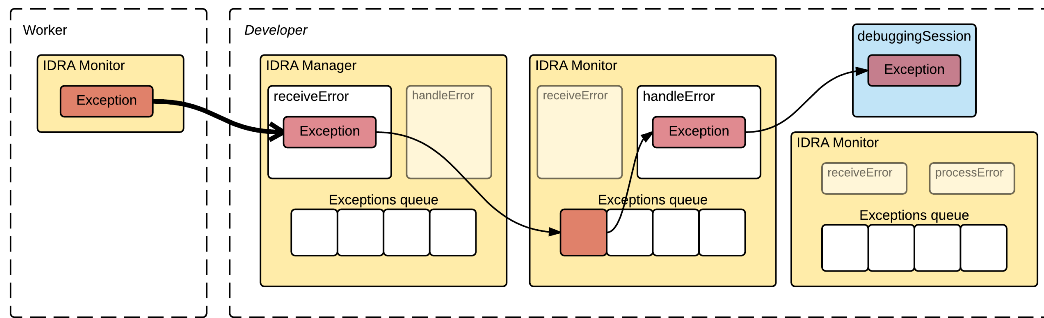

Figure 4.5: Overview of exception handling operations in *IDRA Manager*.

4.2.3 Breakpoints

IDRA offers online debugging support (i.e. breakpoints and step by step execution) in a similar way that it handles exceptions. A breakpoint pauses the execution of a program and allows the developer to inspect the state of the variables through step-by-step execution. In fact stepping primitives like *step-in* or *step-over* allow developers to analyze how the state of the program changes during its execution at each code statement (cf. Section 2.5.2).

As required by the non-stoppable applications presented in this thesis (cf. Section 2.3), IDRA does not simply stop the overall execution when a breakpoint/exception happens, but it allows workers to continue executing tasks. Recall from section 2.7.3, BigDebug [GIY+16] provides a *simulated breakpoint* to avoid to stop the execution of the program. This will make a task proceed after the breakpoint, but store the execution state in order to allow the developer to restart, in the debugger, the execution from that point. While simulated breakpoints are a really powerful instrument to remotely debug a running process, IDRA provides a novel kind of breakpoint that completely stops the execution of one process, and continues it only after a fix is provided. We call such a breakpoint a *stop-and-go breakpoint*.

In contrast to a simulated breakpoint, the *stop-and-go breakpoint* behaves like a classical breakpoint. After storing the information necessary to re-create the situation, it does not let the execution proceed from after the breakpoint. Instead, a *stop-and-go breakpoint* will suspend the execution of the task and let the *worker* proceed to the next task. Only when the developer will decide to restart all the breakpoints the suspended tasks will be executed from that point on, possibly using an updated code.

Offering both kind of breakpoints leaves more choice to the developer on how a worker should behave when it encounters one breakpoint. On the one hand, a *stop-and-go breakpoint* allows the developer to better inspect the state while the computation is interrupted, so if an exception is going to be raised

it will happen directly on the machine being used to debug. On the other hand, the *simulated breakpoint* allows the developer to inspect the state at the moment of the execution of that process, and to analyze what he/she did instead of what it is going to do.

4.3 Reconstructing exceptions or breakpoints

In order to correctly reconstruct an exception on a different debugger instance, we need to store and exchange different information about the state of the program. This step is necessary since applications such as the Twitter scenario (cf. Section 2.4.1) and the Yesplan testing scenario (Section 2.4.2) execute different tasks that are non-deterministic and may depend on different variables. The state of these variables at the moment of the exceptions cannot be lost, since it is the only way to correctly reconstruct the environment that led to the exception.

Most of the object oriented programming languages encode state in objects which are stored in the heap or stack. Each call within a function generates a new stack frame, which will evaluate, eventually generate other stack frames, and then return to the context that created it. As we can see in Figure 4.6, each level of the call stack corresponds to a context, and each context contains a pointer to the precedent context (Priv) in the stack and the eventual successor (Succ). Each context also holds references to memory correspondent to its local variables and, more in general, to all the variables used in that particular context.

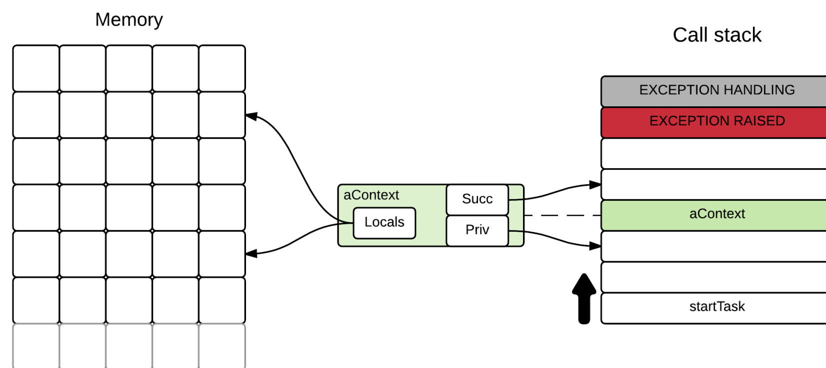

Figure 4.6: Representation of a call stack and a context.

When an exception occurs, the top of the stack stores the context that generated the exception. Some exception handling functions may be then executed on top of the failed context. When debugging an exception, developers

are not interested in such contexts. This is why many debuggers when showing the call stack hide the exception handling contexts. We use the same approach in our debugger.

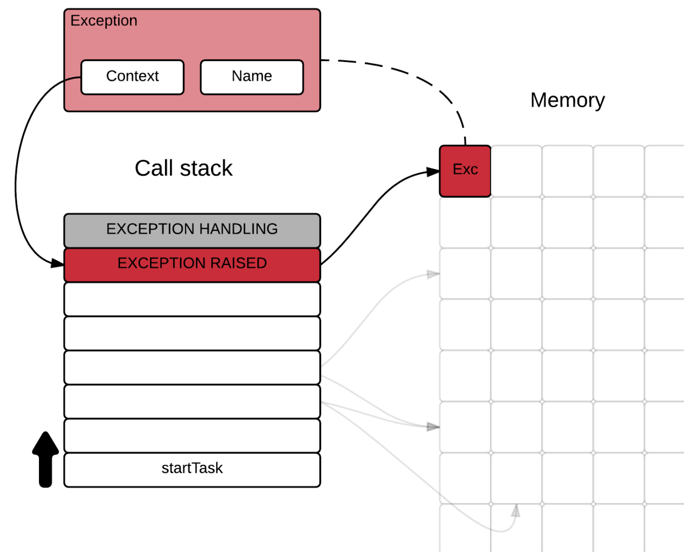

Figure 4.7: Representation of a call stack and an exception.

When an exception is raised, an object representing the exception is instanciated in memory. This object contains some information about the exception and a pointer to the call stack. This allows the debugger to bind an exception to the stack where it was generated.

4.3.1 Handling the state

In order to debug an exception in another process, the exception needs to be transferred to the second process. In our model an exception has a reference to the context that generated it, called signaler context. This signaler context is necessary to reconstruct the exception on another process, as the previous context stored in it. Since any context, except the starting one, has a reference to its calling context , all the context of the call stack are necessary to properly reconstruct the exception.

Note that each context may reference one or more local and global variables. To fully reconstruct the situation that led to the exception, all those variables need to be transferred to the other process. All the *exception handling* contexts, normally presented on top of the context that caused the exception,

can be discarded when transferring the stack, since they are not needed to reconstruct the exception.

In short, in order to send an exception to an *IDRA monitor* all the variables reachable from the exception are needed to be copied. This allows to store the state in the moment of the exception, and avoids other agents to modify the values of the variables that will be sent.

Handling a breakpoint

A breakpoint is a point in program in which execution can be paused or halted. If the debugger needs to reconstruct the context that led to a breakpoint, it is important to note that the context of a breakpoint is at the top of the call stack. This means that a breakpoint has a context, and this context links to the whole call stack, as we can see in Figure 4.8.

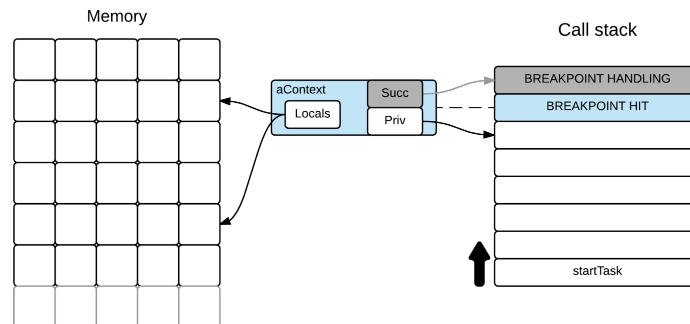

Figure 4.8: Representation of a call stack and a breakpoint-context.

In order to reproduce the context of a breakpoint in another process, the call stack and all its referenced variables will need to be copied and sent to that process. The infrastructure built for *exceptions* is reused to support *breakpoints*, i.e. handling a *stop-and-go breakpoint* happens exactly in the same way as an exception. The *simulated breakpoint* could be also handled as an exception but, since the execution does not need to be restarted afterwards, we can avoid to store them in the *sent exceptions queue*.

4.4 Fixing and committing

IDRA offers the necessary infrastructure to debug remote exceptions executing a local debugger on local objects, not influencing the overall running system. This allows developers to safely debug not worrying on the side effects that

the their changess would have on a remote worker. On the other hand, this poses a new challenge: the debugger needs to keep the code base updated.

Note that approaches like BigDebug [GIY+16] and Mercury [PBFD15] the code base is constantly kept up to date. As discussed in Section 2.6.1, in BigDebug the code base is physically the same for the distributed system, while for Mercury a mirror communication is setup. In both approaches the exceptions are debugged directly on the remote machine, and it is impossible that the code changes made in the local machine do not affect the remote one. This means that if other tasks of the same kind are running, they will immediately start running with an updated version of the code. This approach works perfectly if the fix produced by the developer is correct, but if the fix introduces a new error all the workers of the system will start executing faulty code.

To solve this issue, in our approach, there is not a complete correspondence between the code of the master and the worker. IDRA debugger allows developers to change the code from their local machine without directly affecting the debugged code when an exception raised or a breakpoint was hit. IDRA incorporates a mechanism to propagate code changes from the debugging machine to all workers after developers have the opportunity to verify that those changes work. Such a mechanism is implemented by the *IDRA Changes Handler*.

4.4.1 IDRA Changes Handler

The IDRA Changes Handler offers the following functionalities:

- Detects the changes made to the code base in the developing environment.
- Produces a code-patch (or fix) applicable to all the workers of the system.
- Communicates with other IDRA changes handler sessions in the system to propagate the fix.
- Applies a fix received from another IDRA Changes Handler

Figure 4.9 shows the IDRA Changes Handler in the architecture of IDRA in the scenario of one *developer machine* and one *worker* introduced in Section 4.1.1. All machines where IDRA works contain an instance of the IDRA Changes Handler, then all the connected machines can be updated with the code fix produced in the debugger. When the IDRA Changes Handler is in the developer machine, it records the code changes that happen during a debugging session such as class and method creation, class and method modification,

etc. When a change is detected, the change is stored in a list. Changes are not propagated to other machines until the developer decides it. This is why this list keeps growing until the developer decides to commit the changes.

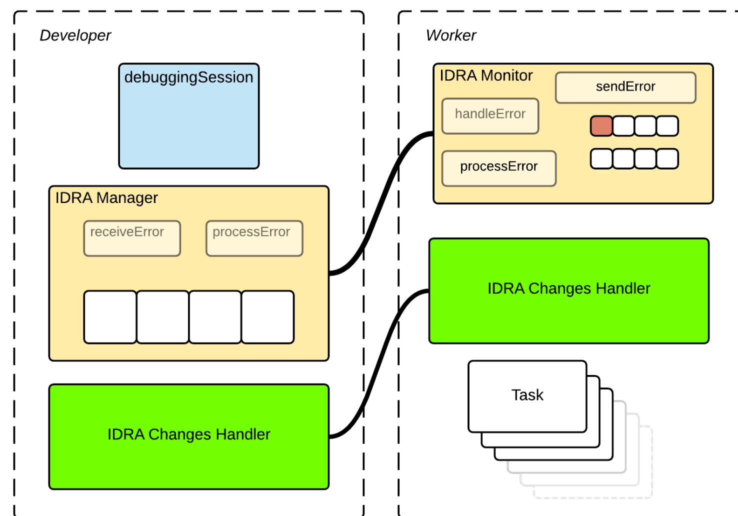

Figure 4.9: Overview of IDRA architecture with an IDRA Changes Handler.

Figure 4.9 shows an overview of the architecture containing both IDRA debugger and changes handler. On the developer side we have an instance of *IDRA Monitor* and a *IDRA Changes Handler*.

Figure 4.10 shows how an *IDRA Changes Handler* reacts to user changes and commits. First the user fixes the code. The change is detected and added into a queue. When the user decides to commit the changes, the fix is produced and sent to another *IDRA Changes Handler*.

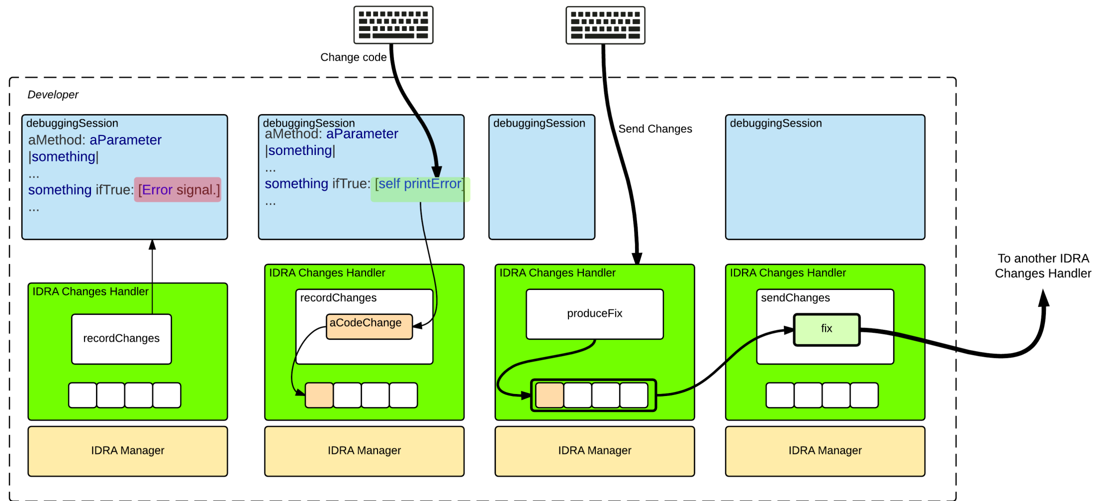

Figure 4.10: Recording changes of a debugging session

4.4.2 Applying changes

When an instance of *IDRA Changes Handler* receives a fix from another, it extracts the set of code changes made by the developer. After extracting the single changes, it applies them to the code base of the machine in which is executing. In our example the instance of *IDRA Changes Handler* running on the developer machine will create the fix and send it to the *IDRA Changes Handler* running on the worker. Figure 4.11 shows an overview of this process.

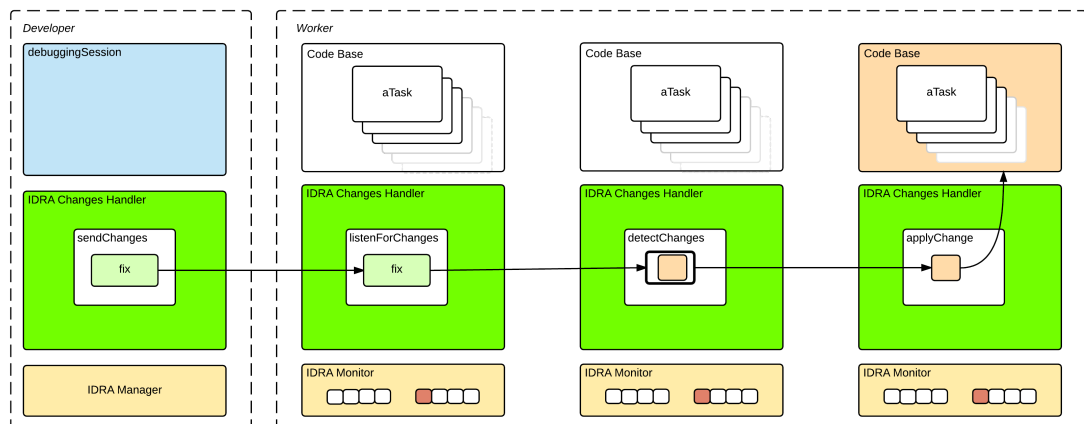

Figure 4.11: Applying changes in a remote machine.

4.4.3 Detected changes

The *IDRA Changes Handler* allows to propagate the changes made in one machine to all the machines of the system. To be complete it has to support a set of changes that is fundamental to keep both versions of the code synchronized, and that satisfy all the structures of the object oriented programming paradigm. Those changes include:

- change of an instance method
- change of a class method
- addition and removal of an instance method
- addition and removal of a class method
- addition and removal of instance variables
- addition and removal of class variables
- addition and removal of a class

This set of changes is necessary to cover all the possible changes in a object oriented program. Non considering any of this changes will lead to a compilation or execution error.

4.4.4 Restarting

The developer can change the code on his local machine and restart the single exceptions to verify that it fixes the problem. He can then decide to use the *IDRA Changes Handler* to propagate the updated code to the other machines in the system.

When an *IDRA Changes Handler* receives a new version of the code, it will apply the update and look for an instance of an *IDRA Monitor*. When it finds one, it will notify it that the code base has changed, and the *IDRA Monitor* will restart the exceptions present in the *sent exceptions* list.

IDRA supports the following different *restarting strategies* which denote the way in which an exception can be restarted.

- **Default strategy:** is used by default, and can restart the contexts in two ways:
  1. Restart from the bottom of the stack, re-executing the whole task.
  2. Restart from the context used to restart in the *IDRA Monitor*.

The second option is possible because, when an instance of *IDRA Changes Handler* propagates a change, it checks for an instance of *IDRA Manager*. If an *IDRA Manager* is present, the *IDRA changes handler* extracts the last restarting information from the *IDRA Manager* and sends it to the other *IDRA Changes Handler* with the changes.

- **Task strategy:** When IDRA is used to debug an application deployed with the *master/worker* framework (cf. Chapter 3), an exception happening during the execution of a task will be able to restart with this strategy. This strategy identifies the original task in the stack and reschedules the task, not really restarting the exception.

- **Test strategy:** Analogue to the task strategy, if an exception happens during the execution of a unit test or if the test fails, the test will be detected from the task and will be re-executed again.

- **NoRestart strategy:** This strategy is used if the developer does not want to restart the enqueued exceptions, possibly because the data that was going to be processed can be discarded. In this case, the stack is simply discarded.

Assigning a restarting strategy

A restarting strategy allows developers to restart the enqueued exceptions in different ways, depending on the problem the developer is debugging, and can be different in any exception. An exception queue can contain different exceptions with different restarting strategies, and the debugger will restart them depending on their restarting strategy.

Hence, the exceptions are 'tagged' with a restarting strategy in the moment they are caught by the *IDRA Monitor*, depending on the context in which they happen. An overview of the operation is shown in Figure 4.12.

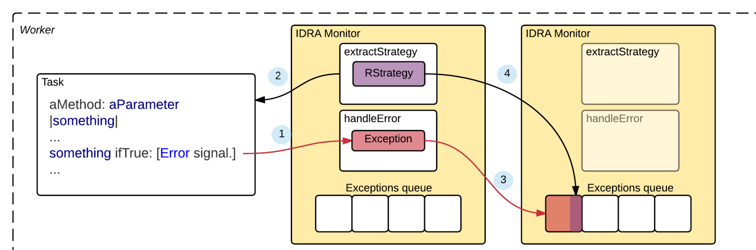

Figure 4.12: Adding a restarting strategy to an exception.

4.4.5 Atomicity of changes

The *IDRA Changes Handler* does not always guarantee the atomic consecutive application of the changes. When a set of changes is received the handler will apply the changes one by one, according to the order in which they were produced. This happens in a process concurrent to the other tasks that run in the machine and the instances of IDRA. This makes it impossible to determine if all the changes will be applied one after another, or if the scheduler of the machine will decide to execute different code in between of the changes. Instead, it is guaranteed that the changes will be all executed, and all in the order that were applied by the developer. This is an open research problem in the context of dynamic software update techniques [PDB+15].

Atomicity is only guaranteed if the *IDRA Changes Handler* is deployed in a *master/worker* setup, following the framework described in Chapter 3. In this case, if an instance of `Worker` is detected, the *IDRA Changes Handler* can communicate with it, and create a *task* which corresponds to the whole code update. In this case, since a `Worker` executes tasks one after another, it can be instructed to execute only the *update task* between two other tasks. In this way the update will be executed atomically with regards the other tasks, and no tasks will execute with partially updated code. Figure 4.13 shows an overview of this setup. We can see the three nodes, *master*, *worker* and *debugger*, which all contain an instance of *IDRA Changes Handler*. The debugger contains an *IDRA Manager* connected to both the *IDRA Monitor* present in master and worker. When the worker receives the change, it will schedule a code update task between other tasks.

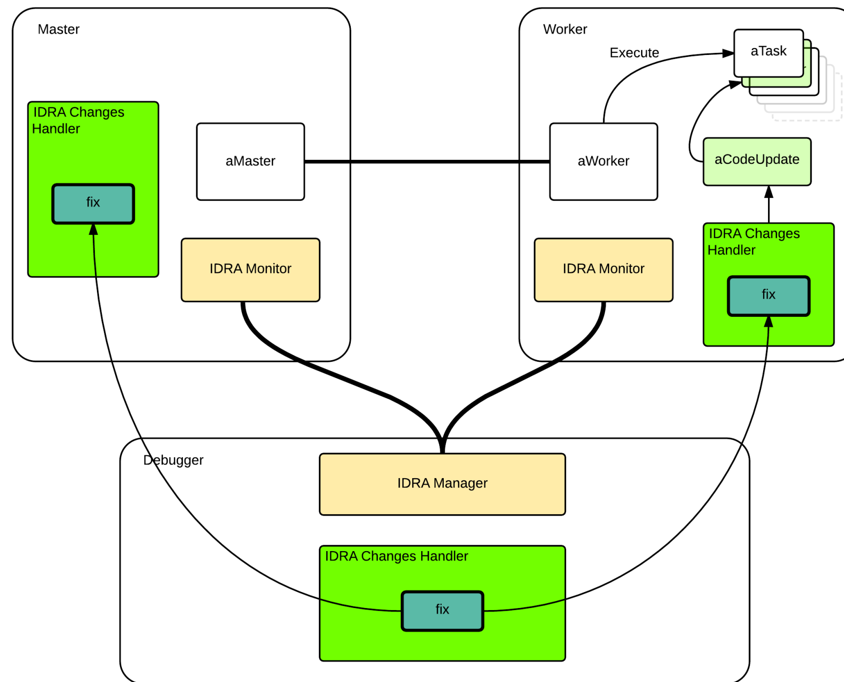

Figure 4.13: Atomic apply of changes when using *master/worker* framework.

4.5 Overview of IDRA architecture

IDRA allows to locally debug remote exceptions, reconstructing the context in which the exception happened on a different process, and allowing to debug different processes in a distributed system.

The main functionalities of IDRA are divided in three components:

- **IDRA Debugger Monitor**: which handles exceptions on a machine. It is responsible for detect exceptions in the system, copy their state information and send them to the *IDRA Debugger Manager*.
- **IDRA Debugger Manager**: which allows to debug all the exceptions happening in the different monitors. It has to reconstruct received exceptions and open a debugger session on them, allowing the developer to change it.
- **IDRA Changes Handler**: running on all the machines, allows to detect and propagate code changes. It detects the code changes (i.e. method or class addition, modification, deletion) made by the developer and, on commit, propagates them to the other *changes handler* in the system.

Figure 4.14 shows an overview of the architecture.

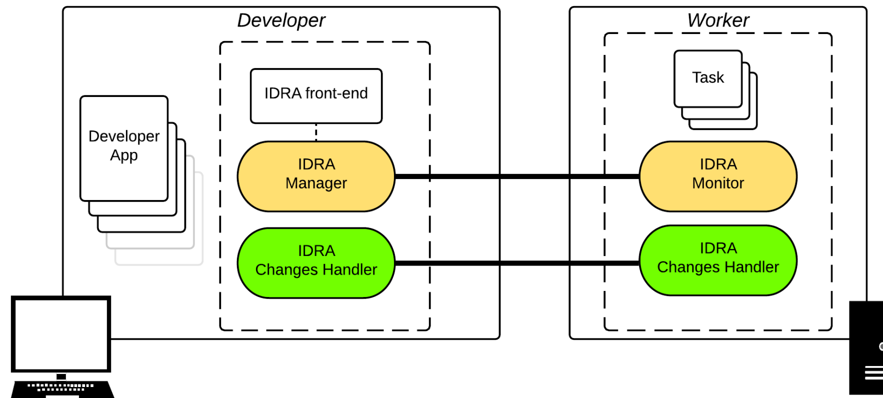

Figure 4.14: Overview of IDRA components in a simple *developer/worker* setup.

In the following section we will analyze the front-end of the debugger, which offers different debugging operations.

4.6 IDRA front-end in Pharo

We implemented IDRA as an extension of the Pharo Debugger [BNDP10], allowing all its interactive debugging operations and adding some functionalities to interact with the *exceptions queue*. This section details first the debugging functionalities offered by the Pharo debugger, then the extensions we provided to the Pharo User Interface to interact with IDRA.

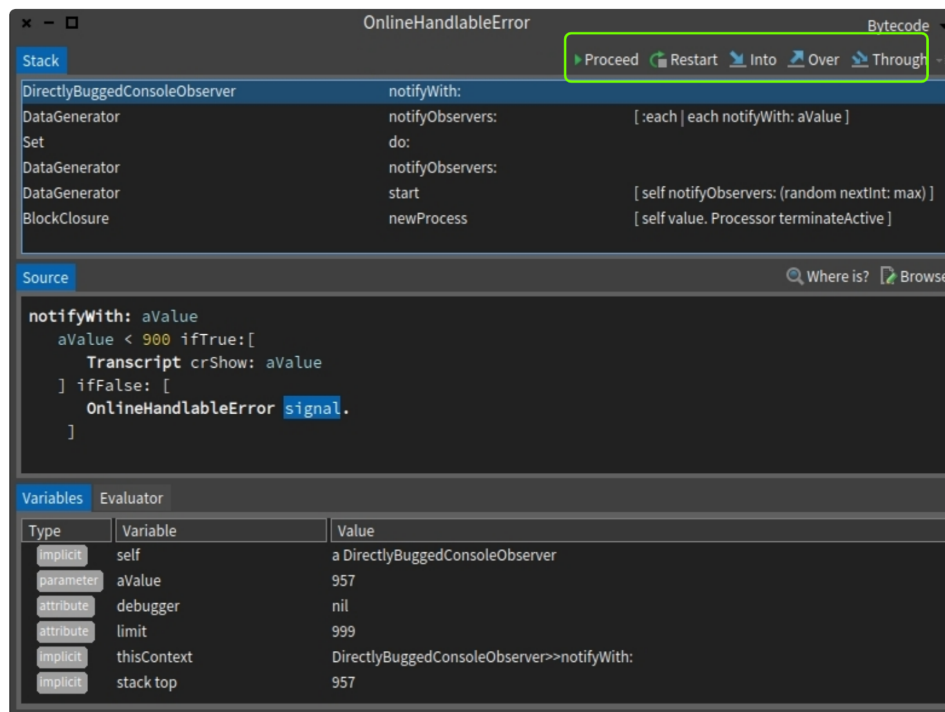

Figure 4.15: The Pharo Debugger.

Figure 4.15 shows the developer interface of the default Pharo Debugger. On the top right we can see five debugging operations:

1. Proceed.
2. Restart.
3. Step into.
4. Step over.
5. Step through.

The view is then composed by three parts:

- The upper one shows a representation of the **call stack**.
- The middle part shows the **source code** of the function executed at the selected level of the call stack.
- The bottom part shows the **state** of the variables in that level of the stack. It can be changed into an **evaluator** to evaluate arbitrary code in the selected context.

Most of the operations offered by the Pharo debugger are common on-line debugging operations, except the second one: the *restart*. Restart allows to replay the execution from the selected point in the stack. This means that an arbitrary context can be selected in that stack, and a click on restart will allow to replay, with a debugger open, to inspect the behavior of the program. This offers to the Pharo Debugger replay capabilities, typical of offline debuggers.

When restarting from a particular context, the code will be re-compiled. This means that, from that moment on, if the developer changes the code of the functions involved in that computation, the new code will be used.

4.6.1 Interacting with the exception queue of IDRA

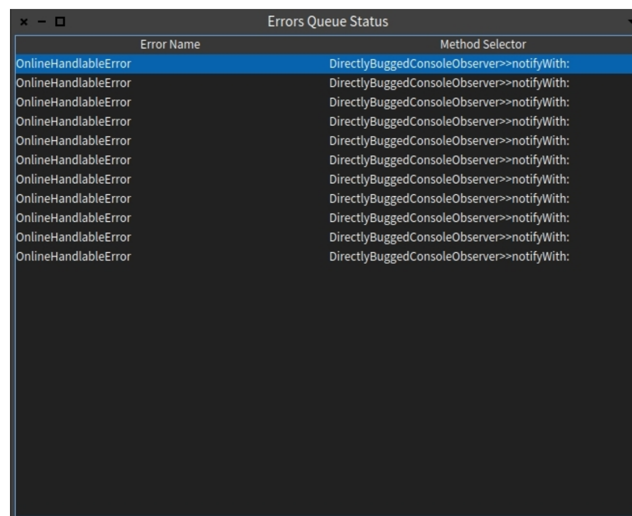

Figure 4.16: The view on the *errors queue*.

When executing an *IDRA Manager* connected to one or more *IDRA Monitor* different exceptions might be added to the *errors queue*, as explained in Section 4.1.

Figure 4.16 shows the view a developer has on these exceptions, which indicates the name of the exception and where it is. A developer can open a debugger on any of those exceptions and debug it with the default Pharo Debugger operations, but he might want to interact at the same time with more exceptions in the queue.

Let us motivate the need for a new visualization to interact with exceptions by means of a concrete example. Consider the case shown in the same figure, where all the exceptions in the queue are equivalent. In this case, probably all exceptions are triggered from the same bug but by different data. The

developer can change the code of the faulty method and provide a fix. Then he can restart the exception that he is debugging, and proceed the execution. If no other error is raised, the bug can be considered solved.

At this point, IDRA would open another debugger on the next exception that happens to be the same as the one the developer just solved. In this case the developer will need to manually restart, one by one, all the exceptions to verify his fix works. This is why we provide an extended version of the Pharo Debugger which allows to directly interact with the queue.

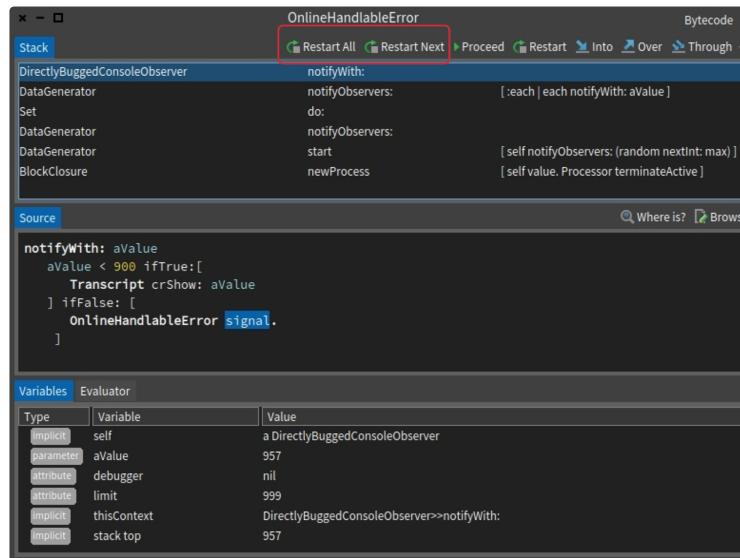

Figure 4.17: The IDRA debugger developer interface.

Picture 4.17 shows how we extended the default Pharo Debugger. We added two operations, underlined in red on the top right of the interface:

- **Restart Next**: This operation allows to restart the current exception from the selected point in the stack, and will automatically proceed with its execution. When the next debugging session will open, it will restart it from the same point in the stack that was previously selected by the developer. It allows the developer to have the updated version of the code in the new exception, and eventually to use stepping operation such as step in or step over, to verify the new behavior. If the exception is not the same or the two stacks are shaped differently, the new debugger will not be restarted and will normally open on the stack level of the exception.

- **Restart All**: This operation sequentially applies a *restart next* to all the exceptions in the queue that have a call stack with the same shape as

the first one the developer restarted. This operation allows an automatic restarting process on all the exceptions.

When using both operations might happen that the new code generates an exception. That exception will be enqueued in the same debugger, therefore information on the new exception will not be lost and the developer will be able to debug it again.

4.7 Conclusion

In this chapter we introduced IDRA, a out-of-place debugger for remote applications that cannot be stopped, such as data intensive applications and long running processes. IDRA allows to remotely debug an exception on a local machine, and to locally test a fix without affecting the other machines in the system. Moreover IDRA enables to deploy the made changes to all the machines and to restart the halted executions. The operations of IDRA can be grouped in three phases:

1. **Handling exceptions/breakpoints**.
2. **Reconstruct exceptions/breakpoints**.
3. **Fixing bugs and committing**.

IDRA executes those phases in three different processes

- **IDRA Debugger Monitor**: is responsible for the phase one.
- **IDRA Debugger Manager**: is responsible for the phase two.
- **IDRA Changes Handler**: handles the phase three.

IDRA is built specifically to debug *non-stoppable applications* (cf. Section 2.3). Those applications cannot be stopped when an exception or breakpoint happens, and this is enforced in the first phase. The second phase allows the developer to debug an exception or breakpoint in a different environment that the one where the exception happened. This avoids interference with the computation happening on the remote machine. The third phase allows to keep the code base of the whole system updated after a developer provides a change to the code. In general all the phases avoid an eventual loss of data, by means of restarting strategies inserted by the *IDRA Debugger*.

In the next section we will analyze the implementation details of IDRA.

Chapter 5

Implementation

In this chapter we present some of the implementation details of our solution. These details include the communication architecture used, as well as the implementation of *IDRA Monitor*, *IDRA Manager* and *IDRA Changes Handler*.

5.1 Communication architecture

Some of the tools and applications developed in the context of this thesis, such as the *IDRA Debugger*, *IDRA Changes Handler* and the use cases, are deployed remotely. Therefore, they are deployed in another process that might be executed on another machine. Those remote instances demand a communication layer to exchange information between local and remote instances. The instances that need to communicate are:

- One *IDRA Manager* and one or more *IDRA Monitor*.
- Different *IDRA Changes Handler*.
- A `Master` and a `Worker` in the *master/worker* architecture described in Chapter 3.

They also need a library to serialize all the objects to transfer them between two processes. The tool needs to be able to:

- Serialize and de-serialize objects.
- Include in the serialization all the reachable objects from a given one, following all the references it includes.

5.1.1 Communication layer

We use **TCP Sockets** as communication layer because they allow to connect processes running locally, on local network and on the internet, and are widely implemented in all the common operating systems. An implementation of TCP Sockets is present in Pharo, under the class `TCPSocket`.

A TCP connection has normally one server listening on a port for requests and one or more clients that connect to that port. As many mainstream languages, Pharo offers the right classes to deploy a TCP connection, with classes such as `ConnectionSocket` for a server and `TCPSocket` for a client. It allows multiple connection spawning a new `InteractionSocket` when a server receives a connection. The TCP handshake is transparent to the developer.

We deployed this in the *IDRA Debugger* and in the *IDRA Changes Handler*, as well as in the *master/worker* architecture (cf. Chapter 3). For example we setup an *IDRA Manager* as server and an *IDRA Monitor* as client. Figure 5.1 shows an overview of this connection.

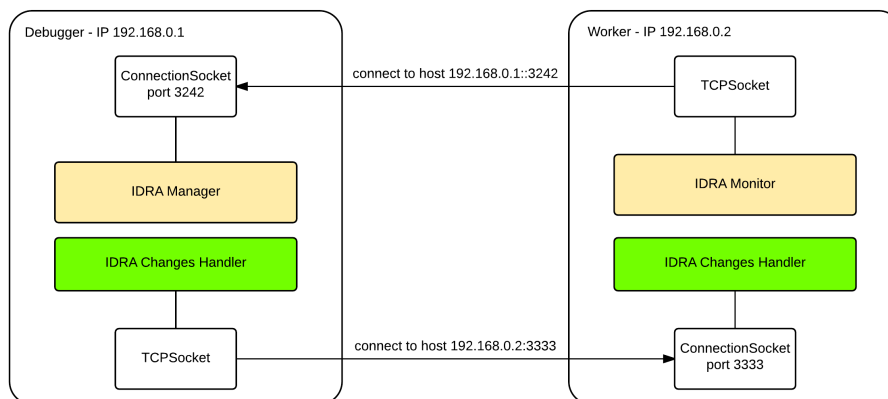

Figure 5.1: An overview of the connection between *IDRA Manager* and *Monitor*, and two *IDRA Changes Handler*

We can see that an *IDRA Monitor* has a `TCPSocket` which can send a connection request to the *ConnectionSocket* of the *IDRA Manager*. In the same way two *IDRA Changes Handler* can connect to each other. In our implementation the instance of changes handler on the worker machine listens for a connection from another one, i.e. the one running on the developer machine.

5.1.2 Communication protocol

In order to exchange objects between the remote instances we use Fuel ([DPDA11]), a library for serialization of objects. Fuel can serialize any object in the system, and all the objects that are reachable from that. Fuel is customizable to avoid serialization of objects, since we want to avoid serializing the instances of the debugger and offer a small API to interact with.

Fuel allows both binary and textual serialization of the objects, and offers acceptable performance on both serialization and deserialization. However, the serializer is not heavily linked to the implementation. In fact is easily replaceable, in the current implementation, by means of overriding the two methods responsible for the communication. The rest of the application logic is not dependent on the serializer.

5.2 IDRA Debugger

The *IDRA Debugger* is the central point of this implementation. It provides all the structure to locally debug remote exceptions, from error handling to the debugging session. By design, each process in the network can run one instance of the debugger, either in *Monitor* or in *Manager* mode. Figure 5.2 shows an UML representation of the `IDRADebugger` class.

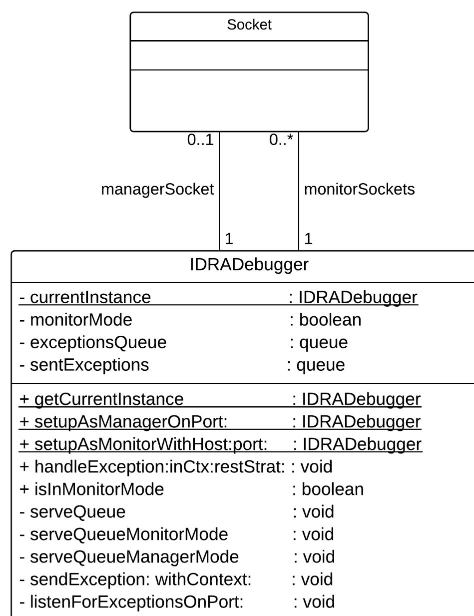

Figure 5.2: Class diagram of IDRADebugger.

The class `IDRADebugger` follows the singleton pattern to ensure there is a single instance per process and clients cannot create new instances of it. It does not expose a constructor and the unique instance can be retrieved through the static method `getCurrentInstance`. Most of the core functionality of this class is implemented in the following methods and instance variables:

- `monitorMode`: A boolean specifying if the debugger is executed as an *IDRA Monitor* or as an *IDRA Manager*.

- `setupAsManagerOnPort:` and `setupAsMonitorWithHost:port:` Two static methods that allow to initialize a new instance of the debugger, which will substitute the old one, in one of the two modes. Figure 5.3 and 5.4 show the code used to setup IDRA as Manager or Monitor.

```
1 setManagerModeOnPort: aPort
2   self resetInstance.
3   monitorMode := false.
4   ↑ ([self getInstance startAsServerOnPort:  aPort] fork).
```

Figure 5.3: Code to setup IDRA as Manager

```
1 setMonitorModeWithHost: serverIP port: aPort
2   | instance |
3   self resetInstance.
4   instance := self getInstance.
5   monitorMode := true .
6   instance openClientSocketWithServerURL: serverIP port: aPort.
```

Figure 5.4: Code to setup IDRA as Monitor

- `handleException:inContext:restartingStrategy:` It is used in both `manager` and `monitor` mode and it wraps an exception in tuple, called `ExceptionTuple` in Figure 5.2 with a particular `RestartingStrategy`. The strategy is inserted by an `ExceptionHandler`, as explained in Section 5.2.1. Figure 5.5 shows how the exception information are wrapped in a tuple.

```
1 handleError: anError inContext: aContext restartingStrategy: aStrategy
2     "create a tuple and add it to errorsQueue and view queue"
3     errorsQueue nextPut: {anError . aContext . currentVersion . aStrategy}.
4     sentExceptions nextPut: { anError . aContext . currentVersion . aStrategy}.
5     "refresh the view"
6     table refresh.
```

Figure 5.5: Code of `IDRADebugger` >> #handleError:inContext :restartingStrategy:

- `serveQueueManagerMode`: reads from the `exceptionsQueue` and opens a debugger on the first of the queue.
- `serveQueueMonitorMode`: reads from the `exceptionsQueue` and sends the `ExceptionTuple` through the method `sendException:withContext:`, as shown in Figure 5.6.

```
1 serveQueueClientMode
2    | |
3   eventsQueue isEmpty
4     ifFalse: [
5               tuple := eventsQueue nextOrNil.
6          self sendException: (tuple first) withContext: (tuple second).
```

Figure 5.6: Code of `IDRADebugger` >> `#serveQueueMonitorMode`

- `sendException:` it is an utility function that sends an `ExceptionTuple` to the `managerSocket`.
- `listenForExceptionsOnPort:` is used from the initializer of the *monitor* mode, and continuously listens on a `monitorSocket` for new exception.
- `managerSocket`: is a `Socket` used when the debugger is in *monitor* mode. It holds the connection to a `IDRADebugger` in *manager* mode.
- `monitorSockets`: is a list of `Socket` used when the debugger is in `manager` mode. It holds a connection for each `IDRADebugger` in *monitor* mode connected.

5.2.1 Breakpoints and exception handlers

Adding an *IDRA Debugger* to the system is not enough to start handling all the exceptions that might happen. This is because to handle exceptions in

Pharo we have to hook into the system's exception handling mechanism. We provide two ways to do so: one is less invasive, and is about changing the exception handling of a single exception. It was employed in a development stage to explicitly avoid catching all the exceptions. We also provide a change to the default exception handling system in order to finally handle all the exceptions of the system.

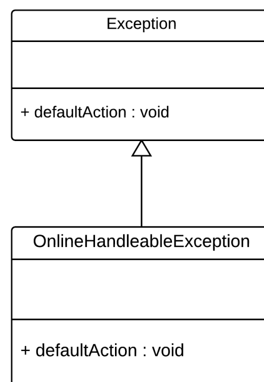

Figure 5.7: Class diagram corresponding to the implementation of exceptions

The first approach shown in Figure 5.7, is to avoid catching all the exceptions, for development purpose, and re-directing to *IDRA Debugger* only custom types of exceptions and breakpoints.In order to implement this, a class `OnlineHandlableException`, extending `Exception`, has been created, which automatically forwards the exception to *IDRA Debugger*. This is possible because in Pharo any unhandled exception in the system is managed by a main exception handler. In the case that exception handler is not configured to handle a particular exception, it will delegate the treatment to the exception itself, sending it the #defaultAction message.

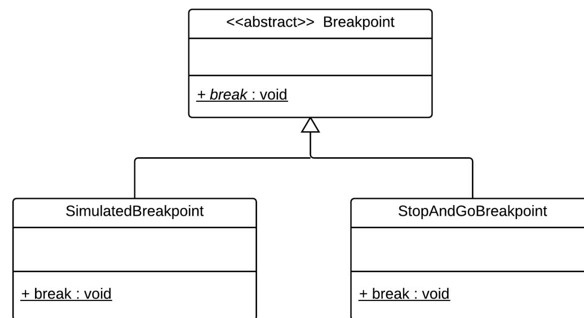

Figure 5.8: Class representation of the breakpoints.

The breakpoints are implemented using the same kind of concept as the exceptions, as in Pharo a breakpoint is itself an exception. Figure 5.8 shows the two classes `SimulatedBreakpoint` and `StopAndGoBreakpoint`. They the two kind of breakpoints supported by *IDRA Debugger*, and can be activated calling the static method `break`.

Figure 5.9 shows, in lines 7 and 10, how the developer can place both types of breakpoint.

```
1  parseFromJSONString: aString
2    | tweet |
3    tweet := self fromMapper: (NeoJSONReader on:(aString readStream )).
4    tweet coordinates: (TweetCoordinates fromDictionary:(tweet coordinates)).
5    SimulatedBreakpoint break.
6    tweet entities: (TweetEntities fromDictionary: (tweet entities)).
7    tweet place: (TweetPlaces fromDictionary: (tweet place)).
8    StopAndGoBreakpoint break.
9    tweet quoted_status: (Tweet fromDictionary: (tweet quoted_status)).
10   tweet user: (TweetUser fromDictionary: (tweet user)).
11   ↑tweet.
```

Figure 5.9: Placing a breakpoint in the code. Code extracted from *Twitter analyzer*.

Exception Handlers

To handle the calls to `IDRA Debugger`, we created different *exception handlers*. An `ExceptionHandler` allows to handle an `Exception`, and calls the *IDRA Debugger* after extracting the context from the exception. It then cuts the extracted context to exclude the contexts related to the exception handling itself and selecting a `RestartingStrategy`. There is one `ExceptionHandler` for each `RestartingStrategy`.

```
handleException: anException
  | context |
  context := self copyContextOfException: anException.
  (self isBreakpoint: anException)
    ifTrue: [ context := context sender sender.
      [ IDRADebugger getInstance
        handleBreakpoint: anException
        inContext: context
        strategy: DefaultStrategy ] fork ]
    ifFalse: [ [ IDRADebugger getInstance
        handleError: anException
        inContext: context
        strategy: DefaultStrategy ] fork ]
```

Figure 5.10: Implementation of `DefaultHandler` >> `#handleException:`.

Figure 5.10 shows the implementation of the method `#handleException:` of the `DefaultHandler`. It gets a copy of the stack through an utility function, which will remove the exception handling generated context. It then checks if is a breakpoint, and calls the debugger with the right method.

If we want a particular exception handler to handle exceptions during a try catch, we can now write the code in Figure 5.11. In fact it is enough to instanciate an `ExceptionHandler` and ask it to handle the produced exception.

```
[ ... ] on: Error do: [ |exception|
DefaultExceptionHandler new handleException: exception.].
```

Figure 5.11: Handle an exception with an `ExceptionHandler`.

Handling all the exceptions

If we want to handle all the exceptions of the system, a particular class needs to be modified: `UnhandledError`. In fact, if an `Exception` is not handled, the method `defaultAction` of the class `UnhandledError` will be called. The call to `IDRADebugger` needs to be placed in that methods, and an `ExceptionHandler` is used to make the call to the debugger. To do so, an instance of `IDRADebugger` also holds a reference to an `ExceptionHandler`, which can be set at any moment with the code shown in Figure 5.12.

```
IDRADebugger restartingStrategy: DefaultStrategy.
```

Figure 5.12: Setting a default strategy to an `IDRADebugger`.

This allows to set a default handling mode for all the exceptions handled by the debugger. Furthermore, `IDRADebugger` allows to enable or disable the option to debug all the exceptions via the field `debugAll`. Figures 5.13 and 5.14 show the default implementation of `UnhandledError >> #defaultAction`, and our modified version.

```
1  "Default exception handling in Pharo"
2  defaultAction
3     [ ↑ UIManager default unhandledErrorDefaultAction:
4             self exception ]
```

Figure 5.13: Default implementation of `UnhandledError >> #defaultAction`

```
1  "check if the debugger is debugging all if true get"
2  "the exceptionHandler to handle the exception."
3  "otherwise, call the default handler of pharo."
4  defaultAction
5     [ ↑ OnlineDebugger getInstance isDebuggingAll
6       ifTrue: [ OnlineDebugger getInstance exceptionHandler new
7               handleException: self exception ]
8       ifFalse: [ UIManager default unhandledErrorDefaultAction:
9             self exception ] ]
10      on: Error
11      do: [ ↑ UIManager default unhandledErrorDefaultAction:
12            self exception ]
```

Figure 5.14: New implementation of `UnhandledError >> #defaultAction`

5.3 IDRA Changes Handler

The *IDRA Changes Handler* is an object that can detect code changes that happen during a debugging session, and can propagate them to other instances of *IDRA Changes Handler*. It is also able to apply a set of changes received from another instance. Figure 5.15 shows an UML class diagram of an `IDRAChangesHandler`.

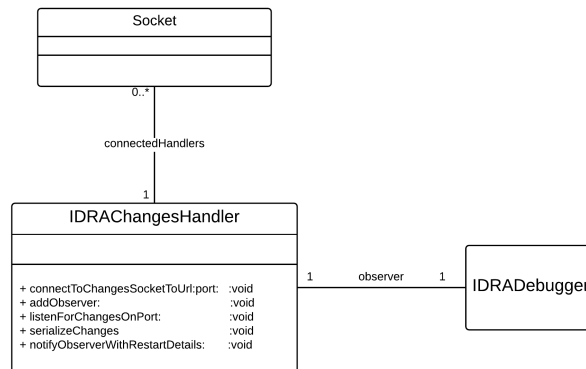

Figure 5.15: Class diagram of a `ChangesHandler`.

The main functionalities are implemented in the following methods and instance variables:

- `connectToChangesSocketToUrl:port:` this method allows to connect a `IDRAChangesHandler` instance to another one to which it needs to send the changes.
- `listenForChangesOnPort:` this method is used to initialize a `Socket` on which that instance of `IDRAChangesHandler` will listen for changes. This is also where the changes are applied to the source base.
- `serializeChanges`: is the method that needs to be called when the user wants to distribute the changes to all the connected instances of `IDRAChangesHandler`.
- `addObserver:` this method can be used to attach an `IDRADebugger` to the `IDRAChangesHandler`.
- `notifyObserverWithRestartDetails:` is the method used to notify all the attached `IDRADebugger` that the code has been updated and that it should restart the exceptions stored. The restarting details can be retrieved from the received changes, and normally indicate which is the context used to restart the exceptions in the *IDRA Manager* instance.

5.3.1 Changes detection tool

The *IDRA Changes Handler* offers the infrastructure to handle changes, but a library called *Epicea* [DCD13] is used for the changes detection and application.

Epicea logs high level information about those changes, it can store intermediate operations and easily separate system events from user changes. For

example the creation, removal or modification of classes and methods, refactors, the execution of unit tests, etc. It allows to record, apply and roll-back code changes, satisfying the requirements of our solution listed in Section 4.4. It also offers an accessible API, from where changes can be retrieved and handled.

In Pharo, Epicea is always active, logging all the changes that happen in the IDE. This means that using this tool will not add recording time and space overhead to our solution.

Epicea is organized in sessions, allowing to separate the changes happening during a debugging session from other ones. A session is accessible from the Epicea API, and contains all the changes that happened since it was started.

Retrieving and applying changes

Epicea offers a class `EpMonitor` which is the one holding a reference to the current *changes store*. The *changes store* is the data structure in which all the changes are stored. This data structure can be retrieved at any moment, and will give a set of `EpEntries` representing the changes of the current session. The *changes store* is totally accessible, and its changes can be re-applied using a visitor pattern. Epicea provides several visitors that simplify the interaction with the changes, such as `EpApplyVisitor`, the `EpUndo` and the `EpRedo`. It is important that the changes are applied in the same order they were retrieved, to avoid missing dependencies.

In the context of the `IDRAChangesHandler` the changes are retrieved from an `EpMonitor`, then these changes are serialized using Fuel [DPDA11] on the communication channel with other `IDRAChangesHandler`s and finally the store is reset, starting a new Epicea session. The reset is necessary because otherwise the next changes will include the old ones.

On the other hand, when an `IDRAChangesHandler` receives a set of changes, it can iterate them and use the `EpRedo` visitor to apply them in that code base.

5.4 Conclusion

In this chapter we presented the implementation details of *IDRA Debugger* and *IDRA Changes Handler*.

In short, *IDRA Debugger* is represented by a singleton class, which can be instanciated only once. It can be both *IDRA Monitor* and *IDRA Manager* (cf. Section 4.1.1), using an internal variable to specify in which mode it is working. We implemented specific exceptions, breakpoints and exception handlers as extensions of the ones present in Pharo, and we hooked to the

Pharo default exception handler to call *IDRA Debugger* when an exception happens.

All the communication happens through TCP sockets, and a library called *Fuel* [DPDA11], which is used to serialize objects on the network.

The *IDRA Changes Handler* allows to detect and distribute changes made during a debugging session. In order to record the changes, we use *Epicea* [DCD13], a library present in Pharo to this extent.

The remote communication between all the entities happens over a TCP connection.

In the next chapter we present the evaluation of our solution, explaining how we implemented some use cases and showing the results of our benchmarks.

Chapter 6

Evaluation

In this chapter we analyze the use cases we developed for this thesis, used scenarios of our benchmarks. We present then an overview of the evaluation and present our different benchmarks. We designed seven benchmarks divided in four categories: *micro-benchmarks*, *network overhead benchmarks*, *IDRA overhead benchmarks* and *IDRA scalability test*. For each category we analyze what is the benchmarks setup, which benchmarks are executed and their result.

6.1 Benchmarking scenarios

In this section we will analyze the implementation of the *twitter analyzer*, *yesplan testing* and *sensor monitoring* application described in Section 2.4. We will also introduce a *Buggy observer*, a simple application which continuously generates exceptions.

6.1.1 Twitter analyzer

The *Twitter analyzer* application is built on top of the *master/worker* framework described in Chapter 3. The framework provides a `Master`, a `Worker` and implements a communication layer and a task scheduling policy. The Twitter analyzer is composed in four main parts

1. The `TwitterMaster`
2. The `TwitterWorker`
3. The Tweet parsing logic
4. The Tweet analyzing logic.

TwitterMaster and TwitterWorker

`TwitterMaster` and `TwitterWorker` extend the classes `Master` and `Worker` of the *master/worker* framework. Figure 6.1 shows an UML representation of the classes.

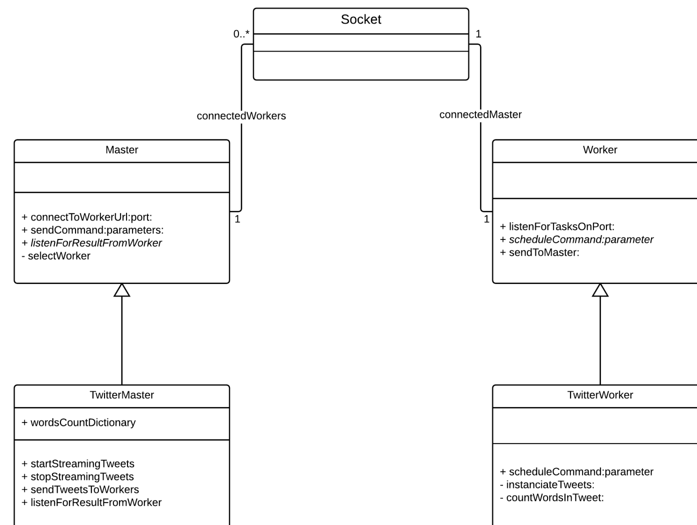

Figure 6.1: UML class diagram of `TwitterMaster` and `TwitterWorker`.

The `TwitterMaster` is responsible for the communication with the Twitter Streaming API [Twi] and for separating the stream of data arriving from the API. The API will continuously return tweets in JSON format. The `TwitterMaster`, in its method `startStreamingTweets` will communicate through HTTP and SSO with the API. The communication layer is provided by Zinc [Zin], a library included in the Pharo Smalltalk Distribution.

Handling results

The `TwitterMaster` handles the results coming back from the remote `TwitterWorker`(s). In the case the result comes from the `instanciateTweets` command, it will send a `countWordsInTweet` command to the `TwitterWorker`. Instead, if the result was generated from a `countWordsInTweet` the local words dictionary of the `TwitterMaster` will be incremented.

Tweet analysis

The `TwitterWorker` provides all the functionalities to initialize and analyze a tweet in the methods `instanciateTweets:` and `countWordsInTweet:`.

To do those operations, an object model of the tweets is used. Figure 6.2 shows an UML representation of some of the classes involved.

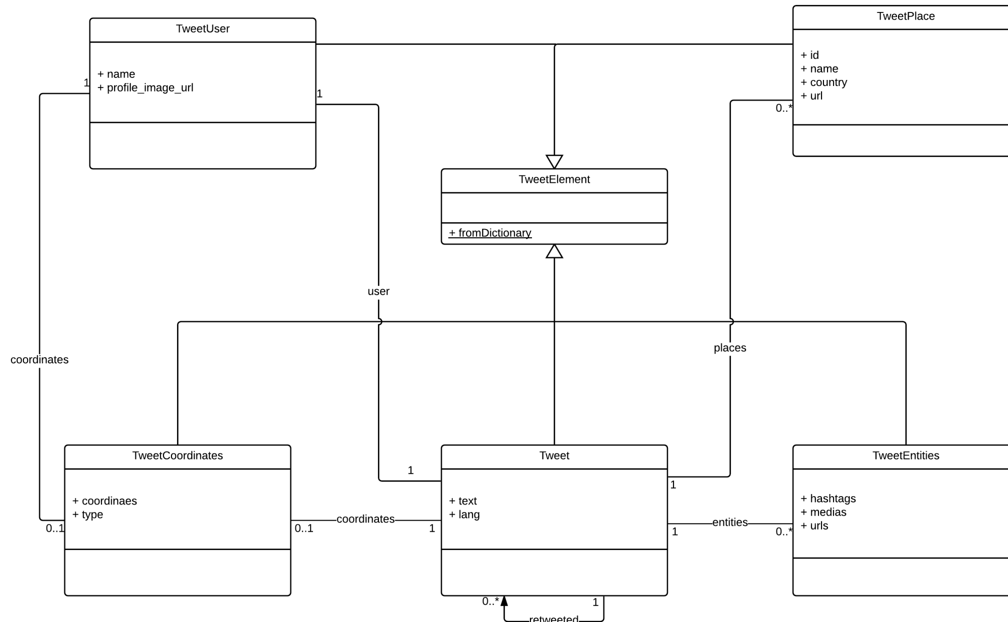

Figure 6.2: Class diagram of `Tweet` and the related classes.

A `Tweet` represents a tweet from a user. It has many attributes, which are not all shown in the figure because of importance. It has different elements also represented by objects, which all extend a common class `TweetElement`.

The method `fromDictionary:` is a static method which can instantiate one `TweetElement` from a dictionary, and is used after parsing the JSON notation of the tweet that has to be initialized. They are necessary to correctly initialize tweets and allow all kind of analysis on them.

However, for our example only the `text` attribute of `Tweet` will be used, since it represents the text of the tweet that contains the words we have to count.

The bug

The *tweeter analyzer* application can present different bugs especially in the stage of parsing and instantiation. For example, a tweet in a language which uses an unicode encoded characters, such as Japanese, can generate an exception because such characters can be recognized as escaping functions. Furthermore the Twitter stream also contains removed tweets, that are encoded in a

different JSON format and produce a parsing error.

6.1.2 Yesplan testing

The *Yesplan testing* demands a framework to remotely execute tests, on which *IDRADebugger* can be initialized to capture test failures. To execute tests Pharo provides a `TestRunner`, in which a user can select tests and execute them. At the end of the execution shows the results of the tests, and allow to re-execute them again. We created a `RemoteTestRunner`, which extends the `TestRunner` and allow to execute tests in a `ControlledTestRunner`.

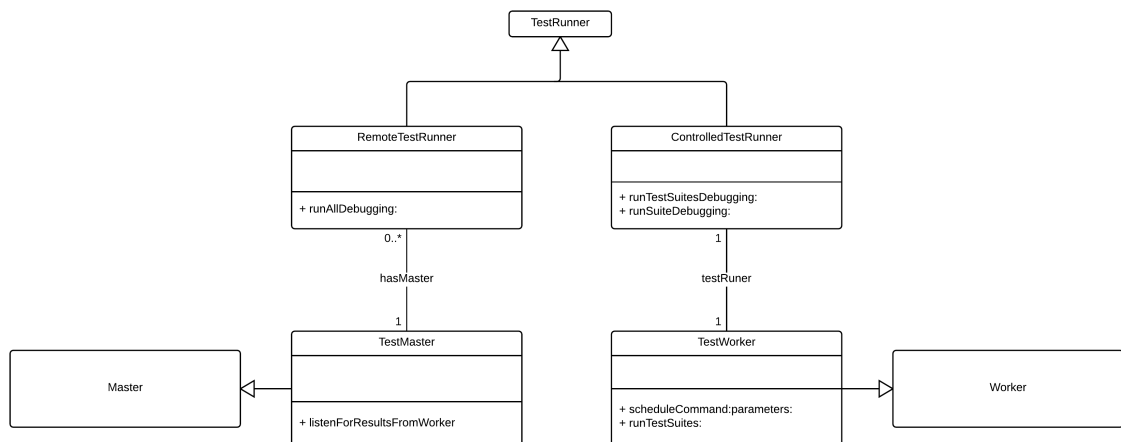

Figure 6.3: Class diagram of `RemoteTestRunner` and `ControlledTestRunner`

As we can see in Figure 6.3, a `RemoteTestRunner` has a reference to one `TestMaster`, which extends the `Master` class of the *master/worker* architecture (Chapter 3). Also the `ControlledTestRunner` holds a reference to one `TestWorker`.

Commands

To comply with the *master/worker* framework, `TestMaster` and `TestWorker` use the command `runTestSuites`, which will trigger the corresponding method in the `TestWorker` class.

Execution of the tests

The tests that have to be executed can be selected in the machine with the `TestMaster`, using the `RemoteTestRunner` view. When the execution command is activated, the selected tests are sent to the `TestWorker` which will

execute them. Test failures will be collected and sent to the developer machine as they happen.

Test failures in Pharo are a special exception, which will be normally debuggable using IDRA.

6.1.3 Sensor monitoring application

The sensor monitoring application consists in a monitoring application running on a Cyber Physical System We built this monitoring system using a *Raspberry-pi* computer [1] together with a *GrovePi* board[2]. The Raspberry-pi is a cheap but powerful small computer that can be easily extended with numerous additional hardwares, like humidity sensors or screens. The GrovePi board is a board designed to be put on top of the *Raspberry-pi* and provide easy access to a variety of sensors. GrovePi already ships with specific drivers written in different programming languages. Our monitoring system controls the hardware using the provided driver to ask for sensor input and display data on the LCD screen.

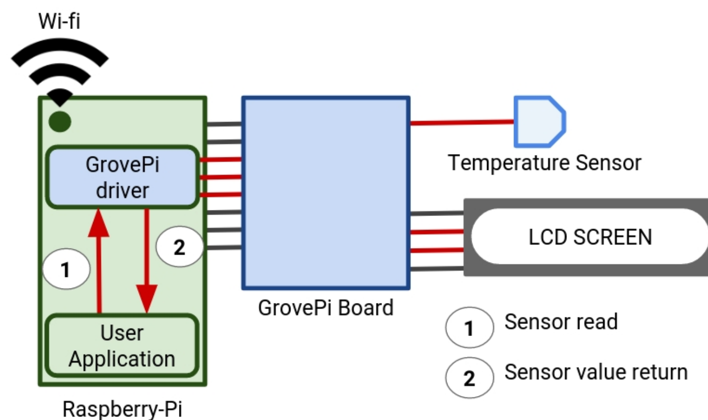

Figure 6.4: Architecture of the sensor monitoring.

Our monitoring software installed in the device is written as a *Pharo* [Pha] application. As shown in Figure 6.4, the user application queries the GrovePi driver for the current temperature with a frequency of 2 Hertz. The driver performs a sensor read and converts the obtained string to a number. The application then shows the current measure in an LCD screen. Our application is also configured with a maximum temperature threshold. While the sensed

[1]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[2]https://www.dexterindustries.com/site/?product=grovepi-starter-kit-raspberry-pi

temperature is lower than the configured threshold, a green LED is turned on. Otherwise, the application sends an alarm to the user. We implemented this alarm for our scenario as a red LED turning on.

It is worth noticing that our design acknowledges the fact that *i/o* errors can occur and that sometimes the sensor coming from the data could be erratic. Because of this, our application validates the value obtained by the sensor before treating it (i.e. the application checks the value is not a null string).

The bug

When testing our application, the device works fine the majority of the time. However, from time to time false alarms are sent to the user. Restarting the device solves temporarily the problem: after an undetermined period of time the bug reappears. Reproducing the bug is not easy because we cannot predict the moment when the bug will appear. In addition, in production mode the temperature monitor works remotely, so when there is a problem we cannot know for sure what is happening. Reproducing the exact conditions under which the bug happens is complicated.

6.1.4 Buggy observer

Buggy observer is an application we created for benchmarking and testing purposes. This application continuously produces an exception, until is fixed by the developer. It is composed by a `Generator` which produces random numbers, and an `Observer`, which receives those numbers and prints them in a console. If the number is higher than a threshold the `Observer` will throw an exception. This application is useful to our evaluation because it can generate an high number of exceptions in a small time, which have to be remotely debugged.

6.2 Evaluation overview and setup

In this section we present the benchmarks that we performed to evaluate our solution. When possible we compare our solution to the PharmIDE [Kud], the last implementation of Mercury [PBFD15] Mercury is a remote debugger for Pharo Smalltalk that we analyzed in Section 2.6.1. We present different kind of benchmarks:

- **Micro-benchmarks**: Micro-benchmarks evaluate the performance of the different IDRA debugging operations and We do so using a simple application which continuously produces exceptions on one machine, and

we will debug them from a different machine. We apply these benchmarks also to the sensor monitoring application, which is a typical application where Mercury is deployed.

- **Network overhead benchmarks**: In these benchmarks we measure the overhead of the network communication generated by an exception and a code change in comparison to Mercury.
- **IDRA overhead benchmarks**: In these benchmarks we evaluate the overhead added by IDRA to the execution of the program in terms of time. We will apply this benchmark to the testing use case.
- **IDRA scalability test**: In these benchmarks we evaluate the number of exceptions IDRA can handle, both on *monitor* and *manager* side, before stopping to work.

We will first describe the hardware setup on which the benchmarks were executed. Then we will explain the benchmarks we executed and present the results. We will first analyze the *micro-benchmarks*, then the *network overhead benchmarks*, the *IDRA overhead benchmarks* and finally the *IDRA scalability test*

6.2.1 Benchmark setup

For the execution of our benchmarks we used different machines in different setups, depending on the context. We used two computers, to which we refer as **Machine 1** and **Machine 2**, and one Raspberry-pi, which we will refer to as **Raspberry**. We will now describe the different machines.

Machine 1

- CPU: Intel Core i7 6700HQ @2.60GHz x 8 with Intel Turbo Boost.
- RAM: 16 GB DDR4
- Operative System: Linux Mint 18.1 Serena - 64 bit.
- Pharo: Version 6.0 #60499

Machine 2

- CPU: Intel Core i7 3540M @3.00 GHz
- RAM: 8 GB DDR3

- Operative System: macOS Sierra 10.12.4
- Pharo: Version 6.0 #60499

Raspberry

- CPU: ARMv8 quad-core @ 1.2GHz
- RAM: 1 GB DDR3
- Operative System: Raspbian GNU/Linux 8 (jessie)
- Pharo: Version 6.0 #60499

6.2.2 Benchmark framework

For different timing benchmarks we use a framework for Pharo Smalltalk called *SMark* [Mar]. SMark allows to easily write benchmarks in Pharo and deploy them in a unit-test like. It allows multiple execution of the same benchmark to provide a more reliable sample of data. In the case of IDRA we added different lines of code to count how much data is transferred through network. In the case of the Pharo Remote debugger the class `SeamlessLogger`, which stores information of all the network communication, has been used to retrieve this information.

6.2.3 SMark benchmarks

SMark proposes a set of benchmarks on basic problems which can be executed to test a particular implementation or setup. We will use some of those benchmarks for part of our evaluation (cf. Section 6.5.2).

The selected benchmarks that we will use are:

1. `SendWithManyArguments`: calculates the time of execution of fifty thousand message send with many arguments.
2. `FloatLoop`: calculates the time needed to execute fifty thousand iteration of a loop with float increment of the looped variable.
3. `ArrayAccess`: calculates the time needed to execute fifty thousand accesses and assignments in an array.
4. `InstVarAccess`: calculates the time necessary to access fifty thousand times an instance variable.

5. `Send`: calculates the time of execution of fifty thousand message send with no arguments.

6. `ClassVarBinding`: calculates the execution time of fifty thousand accesses to a class variable.

7. `IntLoop`: calculates the time of fifty thousand iterations of a loop with integer increments of the looped variable.

6.3 Micro-Benchmarks

In the micro-benchmarks we evaluate IDRA in comparison to the Pharo remote debugger. We execute single debugging operations on a debugging session and we verify how much time take the execution of single debugging operations. We apply the micro-benchmarks to different applications: first a *buggy observer*, a simple application continuosly throwing exceptions. We also evaluated these benchmarks for the sensor monitoring application.

We will now describe the setup.

6.3.1 Setup

We execute all the experiments for one hundred exceptions received on both debuggers. The y-axis is displayed with a logarithmic scale to ease visualization.

For these benchmarks we will use the *buggy observer* application, which constantly generates exceptions and allows us to test, multiple times, both how much time debugging operations take and how much data is sent through the network. For the second benchmark we will also use our *sensor monitoring* application.

These benchmarks are executed using the *machine 1* as *debugger* and the *machine 2* as *worker*.

The debugging infrastructures are run in different ways, due to the different designs of the debuggers.

IDRA debugger

When using IDRA Debugger, the application is executed in the *worker*, along with an instance of *IDRA debugger* in *monitor* mode and with an *IDRA changes handler* active. The *debugger* executes an instance of *IDRA debugger* in *manager* mode and an *IDRA changes handler* analyzing the changes in the code. Figure 6.5 shows an overview of the setup.

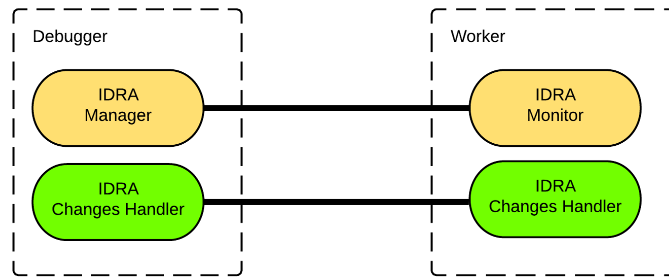

Figure 6.5: Setup of idra on two machines for benchmarking.

Pharo remote debugger

When executing the *Pharo remote debugger*, the application is running in the *worker* machine with an instance of *Pharo remote IDE server* active. On the *worker* an instance of *Pharo remote IDE client* is executed and connected to the server instance in the *worker*. Different benchmarks have been used, which are explained in next section.

Figure 6.6 shows an overview of the setup.

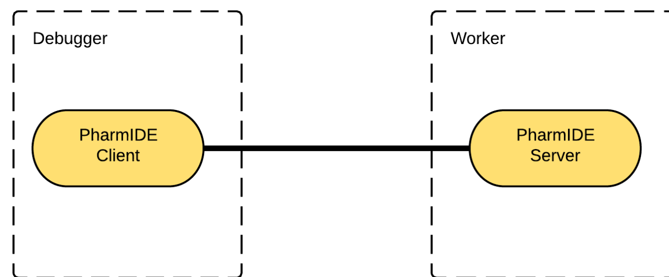

Figure 6.6: Setup of the pharo remote debugger (PharmIDE) on two machines for benchmarking.

6.3.2 Benchmarks

The micro-benchmarks consist in different debugging operations, which measured using both the debuggers and the same application.

These operations are:

Benchmark 1 - Session initialization

When the *debugger machine* receives an exception, it has to open a debugging session on it. We measure the time passed between the the moment when an exception arrives from a remote image and the moment in which a debugging session is opened.

We do not consider the time needed by the user interface to open a debugger. This is totally non-deterministic and not dependent of the two debuggers, since they both use a classic Pharo debugger session.

Benchmark 2 - Stepping operations

When the *debugger machine* receives an exception, a debugging session on it will be opened. At this point different operations can be executed:

- **Restart** the execution from a selected point in the stack.
- **Step Into** the next line of code, shows the code corresponding to the method invoked in that line.
- **Step Over** the next line of code, executes the next line and goes to the following.
- **Step Through** the next line of code, executes parameters evaluated and steps into the proper code execution.
- **Proceed** simply continues the execution not debugging.

For each operation we execute:

1. **Restart** from a point in the stack
2. Execute the operation (**Step into/over/through**). This step is not executed in the case of the **Restart operation**.
3. **Proceed** the computation

This actions represent a typical debugging session, except the fact that no code is changed. It is however consistent to evaluate the execution time of the operations on both debuggers.

6.3.3 Results

We will now show the results of benchmark 1 and 2.

Benchmark 1 - Session Initialization

In this benchmark we measure how much time passed between an exception is received and a debugger is opened on it. In the case of the Pharo remote debugger, every exception opens immediately a debugger, while in the case of IDRA only one debugging session is opened at the time. This is why we had to make sure to close the debugging sessions when opened, in order to have a correct evaluation.

Figure 6.7 shows a boxplot of the results. The time is calculated in milliseconds.

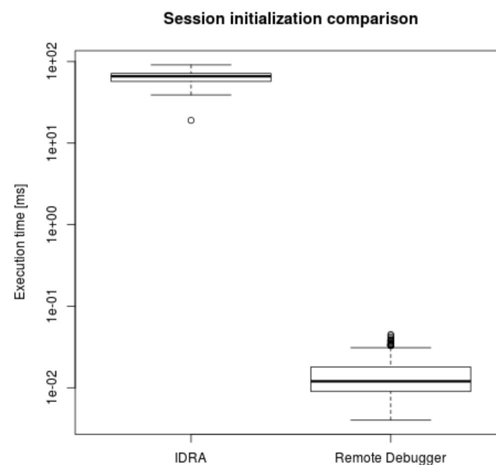

Figure 6.7: Boxplot of the session initialization time for an exception

Figure 6.7 shows that, on average, the Pharo Remote debugger is between a thousand and ten thousand times faster than IDRA. In fact the Pharo remote debugger takes approximately 15 μs on average, while IDRA takes around 60 ms.

This result is expected because of the way IDRA handles arriving exceptions: while the Pharo Remote Debugger immediately calls the user interface to generate a debugger, IDRA puts the received exception in a queue. Another thread reads from that queue, and asks the user interface to open a debugger. This thread reads on the queue every 60 ms, approximately the delay measured in this benchmark.

Benchmark 2 - Debugging operations

In this benchmark we evaluate on both debuggers the time of execution of different debugging operations. Those operations are:

- Restart.

- Step into.

- Step over.

- Step through.

We executed this benchmark both on the *Buggy Observer* and on the *sensor monitoring* application. The time was calculated in microseconds.

Figure 6.8 shows the result of this benchmark on the *buggy observer*. Figure 6.9 shows the result of this benchmark on the *sensor monitoring* application.

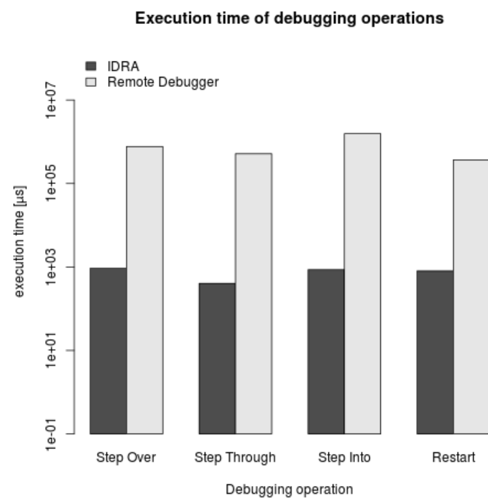

Figure 6.8: Bar plot of the execution time of single debugging operations on the *buggy observer*

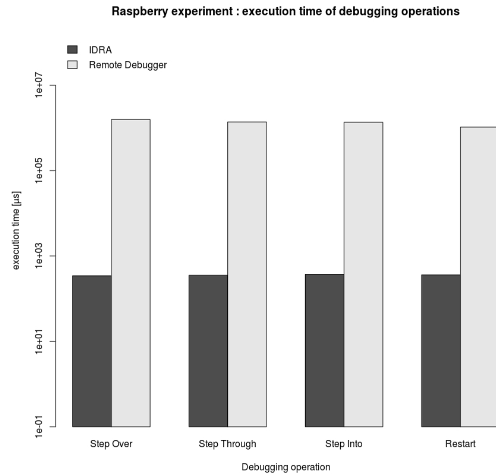

Figure 6.9: Bar plot of the execution time of single debugging operations on the *sensor monitoring* application

It is clear that on both use cases, for all the debugging operations, IDRA is faster than the Pharo Remote Debugger. On the *buggy observer* it is faster between a hundred and a thousand times, while on the *sensor monitoring* application it is constantly more than one thousand times faster.

In fact, in the case of a remote exception handled with IDRA, the exception and all the stack information is copied and sent to the debugger. A debugging session is always opened on a local copy of the exception (and its stack), which makes the debugging session a normal Pharo debugging session.

On the other hand, the Pharo Remote Debugger reconstructs a remote exception by means of proxies of the exception itself and of the related stack. The debugging operations will be executed on the remote machine, introducing communication and network overhead for each of the operations executed.

6.4 Network overhead benchmarks

In this benchmarks we compare the network overhead of IDRA and the Pharo Remote Debugger. We do it executing the benchmarks on both the *buggy observer*.

6.4.1 Setup

The benchmarks setup is analogue to the one used for the micro-benchmarks (cf. Section 6.3.1).

We measure this overhead in different ways, depending on the debugger. In IDRA we measure the size of the data received on the IDRA Manager, since we have control over the TCP connection. Instead, the Pharo Remote Debugger uses an underlining framework called Seamless [PBFD15] which handles the TCP communication. To assess how much data is exchanged through Seamless, we use a logger provided in the framework, which returns detailed statistics over all the communication that happened since it was started.

6.4.2 Benchmarks

Benchmark 3 - Network usage for an exception

This benchmark measures how much data is sent (in bytes) between the two instances of the debuggers: *monitor-manager* in the case of IDRA and *server-client* in the case of the Pharo remote debugger. This benchmark measures this size for a different amount of exceptions, analyzing how the behavior of the two debuggers changes. It can also give an idea on how much communication time is needed to transfer an exception using both debuggers. The communication time is not evaluated since it depends on the network and it would require a notion of distributed clocks to be correctly evaluated. This communication time can be inferred knowing the amount of data transferred and the communication speed of the network.

Benchmark 4 - Size of propagation of code changes

In this benchmark we measure what is the size of different code changes applied to the remote machine. The setup of the two debuggers is different: In the case of IDRA, changes happen locally and are then sent to the remote machine through the IDRA Changes Handler. The changes will be applied in the remote machine only when the user explicitly calls the functionality.

Instead, with the Pharo Remote Debugger the user can open a remote browser on the classes of the remote machine. In this way the user can directly modify in that browser and changes will directly be applied.

In the Pharo Remote debugger we have to consider that the simple opening of a browser and browsing also generates network traffic. This is why all the operations are evaluated only after opening the browser and browsing to the right class.

For different code changes, we analyzed what is the size, in the means of network communication, of propagating:

- **No operation**: no changes are made. A browser is opened and changes are sent.

- **A class addition**: a class named `Test01` is added to the default package.

- **An instance variable addition**: an instance variable named `instanceVariable` is added to `Test01`

- **A class variable addition**: a class variable named `classVariable` is added to `Test01`

- **A method code change**: a method of the class `Test01` is changed adding a line of code.

The code-base is the same in both cases: a basic Pharo 6 image with both Pharo Remote Debugger and IDRA loaded. Figure 6.12 shows the network usage in bytes for each operation.

6.4.3 Results

Benchmark 3 - Network usage for an exception

In this benchmark we measure the amounts of bytes exchanged in the network to handle one exception, happening remotely and debugged on another machine.

We executed this test on the *buggy observer*, sending an increasing number of exception and verifying at each point how much bytes were exchanged at that moment. Figure 6.10 shows the results of these benchmarks. The x-axis shows the number of exceptions and the y-axis the number of bytes exchanged. The y-axis is displayed with a logarithmic scale.

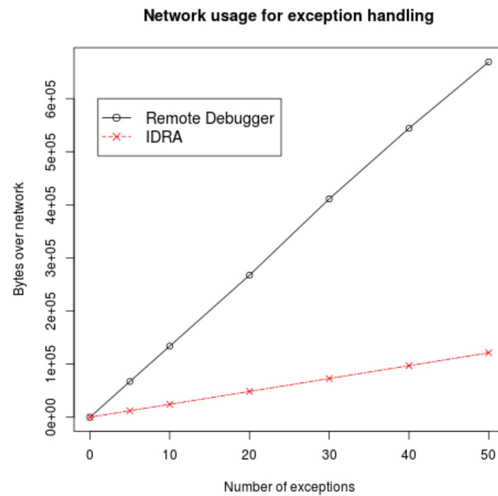

Figure 6.10: Plot of the number of bytes exchanged for an increasing number of exceptions.

Figure 6.11 shows, on average, how much bytes were exchanged for each exception in both debuggers.

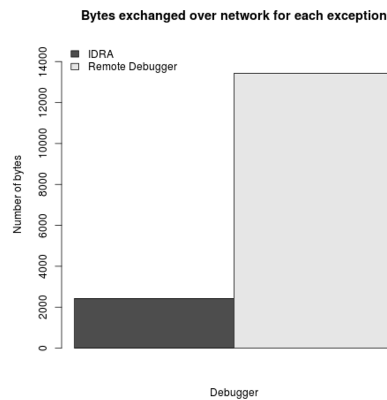

Figure 6.11: Plot of the number of bytes exchanged for one exception.

From both Figures 6.10 and 6.11 is clear that, for this use case, the data exchanged for each exception is sensibly lower in the case of IDRA. For each exception, in a constant way when increasing the number of exceptions, IDRA exchanges over network five times less bytes than the Pharo Remote Debugger. We believe that the reason of this result, in this particular use case, is that the stack associated to the exception is really small, and actually the exception,

and stack itself produces less bytes than all the communication necessary to install and exchange proxies.

Benchmark 4 - Size of propagation of changes

We executed this benchmark doing different code changes and verifying the network overhead.

The y-axis uses a logarithmic scale.

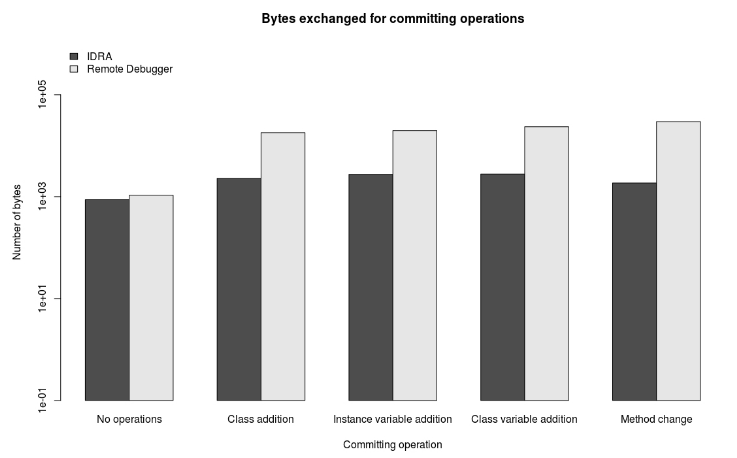

Figure 6.12: Bar plot of the bytes exchanged to commit one change.

The *no operation* category indicates on the IDRA the sending of the changes when changes were not made and the developer sends anyway the command to propagate them. On the Pharo Remote Debugger it indicates the number of bytes exchanged when opening a browser. We can see that the amount of bytes exchanged in this case is approximately the same, not showing a particular difference.

All the other operations show that IDRA uses eight to ten times less network when compared to the Pharo Remote Debugger for simple committing operations.

We believe that these results are due to the fact that the Pharo Remote Debugger uses a remote browser, which contains proxies to many entities of the remote image. Every modification constantly generates a request to the remote image to update, which does not happen in IDRA because the changes are applied to the local code base.

6.5 IDRA overhead benchmarks

In the previous sections we analyzed the performance of IDRA in comparison to the Pharo Remote Debugger with different micro-benchmarks on different debugging operations. In the next sections we will analyze different benchmarks applied on the Twitter and testing use case, along with a small overhead benchmark using the default benchmarks of SMark.

6.5.1 Setup

In these benchmarks we will use always *Machine 1* as debugger machine and *Machine 2* as the machine running the applications that need to be debugged. IDRA debugger is run on both the machines, as *IDRA Manager* in *Machine 1* and as *IDRA Monitor* in *Machine 2*. The two instances of IDRA debugger in the two machines are connected. An instance of *Changes Handler* is also running on both *Machine 1* and *Machine 2*.

All the following benchmarks are executed for one hundred exceptions received on both debuggers.

6.5.2 Benchmarks

These benchmarks are applied on two scenarios. One is the *SMark Benchmarks* and the other one is the *Yesplan testing*. The benchmarks are:

Benchmark 5 - Basic overhead of IDRA infrastructure

In this benchmark we will analyze how much overhead is introduced by IDRA in the execution of the system. We will do it simply running the SMark benchmarks.

There are no exceptions, so the showed overhead is only of the debugger active listening for exceptions, and of the changes handler listening for changes. We performed this benchmark on *Machine 1* setup as *IDRA Manager* and on *Machine 2* setup as *IDRA Monitor*, executing the benchmarks on both machines.

Benchmark 6 - IDRA serialization overhead

: In this benchmark we analyzed how much overhead IDRA adds in the execution of tests in the case of a test failure.

Normally, when running tests in Pharo, when a failure happens it is simply logged. In the case of our *testing use case*, we can remotely run the tests, and debug the failures through our debugger. We will first measure the execution

time of all the test, without failures. Then we will incrementally add failures in the tests to measure how much time is spent to serialize the failure.

In order of not changing the amount of computation executed by the tests, to add a failure we add a basic assertion fail such as the one showed in listing 6.1.

Listing 6.1: Code for a simple assertion failure in a Pharo `TestCase`

```
1 self assert: false.
```

.

We setup the benchmark in the following way: *Machine 1* runs an instance of *IDRA Manager*, one *IDRA Changes Handler* and one *Remote Test Runner*. *Machine 2* runs an instance of *IDRA Monitor*, one *IDRA Changes Handler* and one *Controlled Test Runner*.

The *Remote* and *Controlled* test runners are connected through the master/worker framework introduced in Chapter 3, as explained in Section 6.1.2. We automatized the process of selection of tests on the `Controlled Test Runner` to run the selected package every ten seconds, fifty times. At each fifty run we add a number of test failures in randomly selected test methods. We will run the benchmark with four different amount of failures:

1. No failures
2. 2.25% failures (4 test failing over 154)
3. 5% failures (8 test failing over 154)
4. 10% failures (16 tests failing over 154)

We measure the time passing between the whole test execution is started and is completely finished (all the test methods have been evaluated). We do this measurement on the *Controlled Test Runner* in *Machine 2*, which is the one actually executing the tests.

We executed all the tests of the package AST-Core-Tests, containing 154 tests.

6.5.3 Results

We will now present the results of the *IDRA Overhead Benchmarks*.

Benchmark 5 - Basic overhead of IDRA infrastructure

In this benchmark we measure how much overhead is introduced by an active instance of IDRA over one application not introducing bugs. We do it over

the SMark default benchmarks described in Section 6.2.3. We first executed all the benchmarks without initializing IDRA. Each single benchmark was executed one hundred times. The same operation was done after initializing and connecting IDRA. After the averages were calculated for each operation, and the ratio was calculated dividing the two averages. The resulting overhead is showed in Figure 6.13 and Figure 6.14.

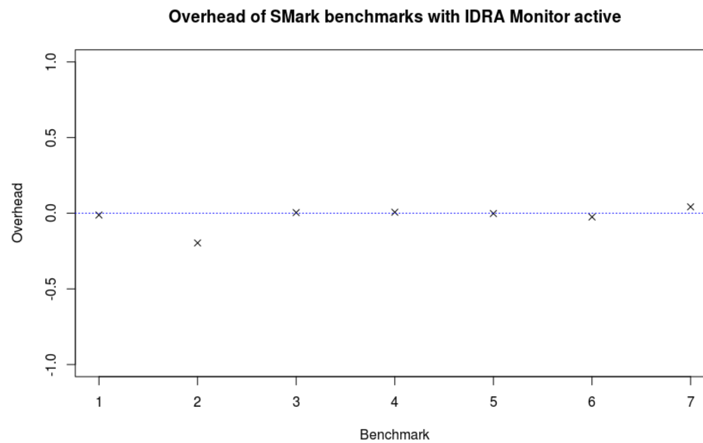

Figure 6.13: Overhead of the execution with active IDRA Monitor.

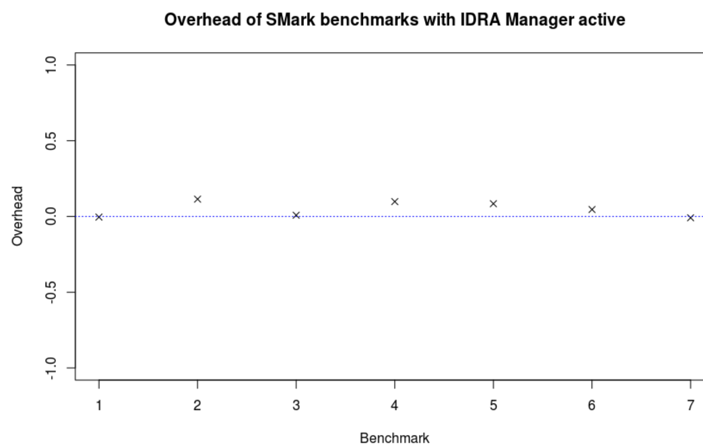

Figure 6.14: Overhead of the execution with active IDRA Manager.

As we can see from figure 6.13 IDRA Monitor does not introduce sensible overhead. Some benchmarks, like the number 2, are even faster when IDRA

is running, which shows that actually if there are any differences, they are not related to the presence of the debugger.

On the other hand, figure 6.14 shows that IDRA Manager introduces a slight overhead in most of the operations. This overhead is due to the presence of one more socket listening for exceptions coming from the connected IDRA Monitor, but is anyway negligible.

Benchmark 6 - IDRA serialization overhead

The goal of this benchmark is to measure which is the time overhead of executing remote tests containing failures. To do this we analyze the execution of the tests of the package `AST-Core-Tests`, which contains 154 tests, normally not failing.

Figure 6.15 shows the running times, on average and with errors, of the tests execution when increasing the number of errors.

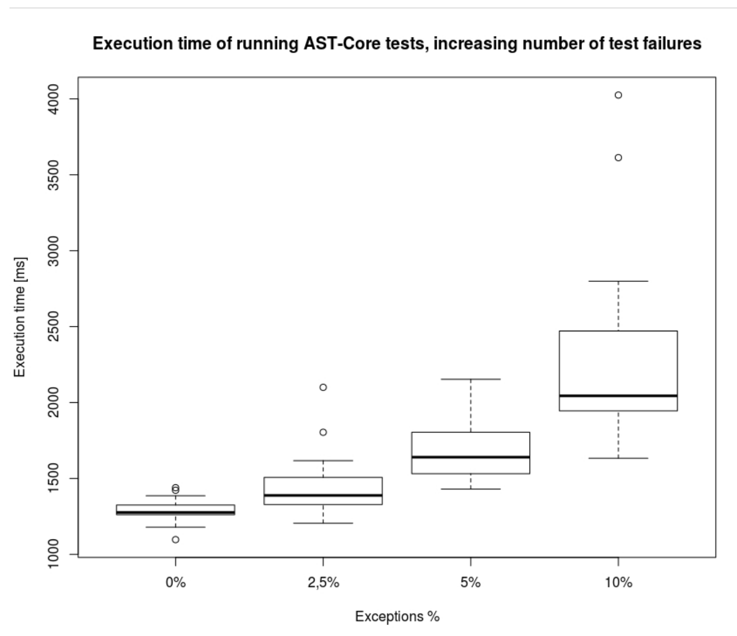

Figure 6.15: Execution of `AST-core-tests`, increasing number of failures.

We can clearly see in Figure 6.15 that the execution time increases when increasing the percentage of failure. This is because of the serialization operations that can take up some processing power in the working machine. Since Pharo is single threaded, even if the the execution of the serialization is done in a parallel green thread, this can clearly impact the time of execution of the tests. The increase is proportioned to the number of exception.

This happens because when a failure is being serialized another failure might happen, slowing down the process because the failure has to be handled while we are serializing already one. We also have to consider that the machines are connected through a local network and each failure, other than the serialization operation, triggers a network communication.

6.6 IDRA scalability test

6.6.1 Setup

We perform this benchmark on *Machine 1* executing two processes: one is the `TwitterMaster` and *IDRA Manager*. The `TwitterWorker` is executing on *Machine 2* and has an instance of *IDRA Monitor* connected to *Machine 1*.

6.6.2 Benchmark 7 - IDRA scalability test

The goal of this test is to evaluate the scalability of IDRA when receiving many exceptions. We use the *Twitter analyzer* to test it, generating an exception for each tweet that is parsed. We will generate the exception in different points in the same stack, running the simulation to see how many exception we can handle before the system stops working. In fact, in the current solution, there is no system to handle such failures.

We manually inserted an exception in the `TwitterWorker`. We decided to insert the exceptions in the `initializeTweet:` method, which takes a string representation of a tweet and parses it to create a `Tweet` object.

We did three tests:

1. Insert the exception as the first line of the `TwitterWorker>>initializeTweet:` method.
2. Insert the exception in the deepest point in the stack possible, so during the parsing of the tweet in `Tweet>>parseFromJsonString:`.
3. Insert the exception as the last line of the method `TwitterWorker>>initializeTweet:`.

6.6.3 Results

Benchmark 7 - IDRA scalability test

The goal of this benchmark is to assess how many exceptions we can receive in an IDRA Manager before any of the processes in the system stops working.

We did two executions for each test, and they produced the following result:

- **Test 1 and 3** - The process running the *IDRA Manager* crashed approximately after ten thousand exceptions received on average.

- **Test 2** - The process running the *IDRA Monitor* crashed after approximately five thousand exceptions sent on average.

The crashes of test 1 and 3 are produced by an **out of memory** exception from the Pharo Virtual Machine. This means that the Pharo Image running in that instance of virtual machine reached its maximum size. Running a 32 bit virtual machine, the maximum addressable memory index is 4,294,967,295. If we analyze the log of our execution, we can see that the instance of *IDRA Manager* received, on average, ten thousand exceptions.

The crash of test 2 happened for a different reason: the execution on the machine instantiating the actual tweets produced a **segmentation fault**. We believe that this happened because the virtual machine tried to access or allocate memory outside the area of memory allowed by the operative system. From the logs we can also see that this crash consistently happened during the serialization process, where different buffers are allocated. Compared to the tests 1 and 3, the test 2 has the deepest stack because the exception is inserted during one of the functions called in the parsing.

We also have to consider that *Machine 2*, where the `TwitterWorker` instantiating tweets was running, has less RAM memory than *Machine 1*. This means that the operative system of *Machine 1* will normally allow one of its processes to use more RAM than one running in *Machine 2*.

6.7 Conclusion

In this chapter we analyzed different benchmarks we applied to evaluate our solution.

The micro-benchmarks, executed on a simple application such as the *buggy observer* prove that IDRA can be a valid alternative to the Pharo Remote Debugger:

- Benchmark 1 shows that the initialization time for a session is bigger, but because of the queue-handling of exceptions that is not present on the Pharo Remote Debugger.

- As analyzed with benchmark 2, all the debugging operations are around one thousand times faster on IDRA then on the Pharo Remote Debugger, because they do not involve any network communication.

The network overhead benchmarks show good performance comparing IDRA to the Pharo Remote Debugger:

- Benchmark 3 shows that on a small stack like the one considered for the *buggy observer*, the amount of network communication in IDRA is lower than in the Pharo Remote Debugger.

- The amount of network communication when executing basic code changes is around ten times lower in IDRA than in the Pharo Remote Debugger, as measured in benchmark 4.

We then applied the IDRA overhead benchmarks on the SMark benchmarks and the *Yesplan testing* scenarios. The results show that:

- Benchmark 5 proves that IDRA does not introduce considerable overhead to a normal execution.

- With benchmark 6 we measured that the execution of the tests becomes slower when adding failures. The overhead increases when increasing the number of failures.

Finally, the IDRA scalability test (benchmark 7) shows that IDRA is pretty robust handling (without user interaction) between five and ten thousand exceptions.

Overall, these benchmarks show that IDRA is improvable on some points, like time of the serialization or maximum number of exception handled, but they also show that it is usable in practice with better results, on some benchmarks, than the Pharo Remote Debugger.

In the next chapter we will give a final overview of the problem statement. We will then briefly give and overview of IDRA and present some limitations and future work.

Chapter 7

Conclusion

In this thesis we studied the problem of debugging applications that should not be stopped, such as *data intensive applications* and *long running systems*. To this end, we employed three different non-stoppable applications. The *twitter analyzer* scenario is a data intensive application which continuously analyzes tweets retrieved from an internet stream. The *Yesplan testing* scenario is a test running environment for the testing process of the Belgian company Yesplan, requiring the execution of long-lived tests which depend on non-deterministic data. The *sensor monitoring* scenario is an example of cyber physical system, which preforms reading operations on a sensor.

In what follows, we first revisit our problem statement, and then introduce the main highlights of IDRA, an implementation of the proposed out-of-place debugging technique for non-stoppable applications. We then recapitulate the evaluation our work based on the three aforementioned scenarios. We then draw final conclusions of this thesis analyzing contributions, limitations and future work.

7.1 Problem statement revisited

To debug *non-stoppable applications* we propose a new model of debugging remote applications, called out-of-place debugging. This debugging technique allows to remotely debug different processes, without stopping their execution when a failure happens. Out-of-place debugging introduces a new approach in the debugging techniques of those applications, and distinguishes three different phases a debugging session consists of:

Handle exceptions and breakpoints. On a remote machine, when an exception or a breakpoint is found, the debugger extracts the necessary information to debug and resumes the execution of the program.

Reconstruct exception and breakpoints. Debugging information is transferred to the developer's environment, allowing her developer to debug the problem locally.

Fix the code and atomically commit them. The developer can, if he desires to, produce a fix in the code and commit all related changes changes to the remote machine. Then the executions suspended because of the breakpoint or exceptions are resumed.

In short, out-of-place debugging allows developers to debug program execution for a task in a different machine but on the environment where it happened. The developer can locally modify the application without affecting the execution of the machine where it was triggered, while using traditional on-line debugging features like stack inspection and step-by-step execution.

7.2 IDRA: An Out-of-place Debugger

We implemented out-of-place debugging in IDRA, a debugger for non-stoppable applications written in Pharo. In this section we will give an overview of the different concepts of IDRA, namely the *IDRA Debugger* and the *IDRA Changes Handler*, independently of the concrete implementation in Pharo.

7.2.1 The IDRA debugger

The IDRA debugger is the component of IDRA that traps exceptions and manage all the debugging operations. The IDRA debugger has two main subcomponents:

IDRA Monitor. The *IDRA Monitor* Runs in the same process than the debugged application. It captures all unhandled exceptions happening on the debugged process. When it captures an exception, it suspends the failing process that produced it and pushes the exception in a queue for later processing. Regularly, it sends the queued exceptions to an IDRA Manager. A transferred exceptions includes:

- The stack trace of the failed process, from the top of the stack to the point where the exception was generated.
- All objects reachable from such stack, including the ones referenced by local and global variables.

This way of handling exceptions allows the machine to continue executing other processes, without being influenced by the debugging operations.

IDRA Manager. The *IDRA Manager* runs in the developer's machine and is connected to one or more IDRA Monitor instances, listening for incoming exceptions. When an exception arrives, it reconstructs a debugging session on it with the debugging information transferred by the IDRA Monitor. The constructed debugging session allows to debug the same code and data that produced the exception, without influencing the remote machine where it happened. This is the core concept of out-of-place debugging.

If another exception arrives while another session is still open, it is enqueued for later processing. Enqueued exceptions will be debugged after the previous exception has been debugged.

A graphical interface offers a clear view of the incoming exceptions.

Both *IDRA Monitor* and *IDRA Manager* are able to handle *breakpoints* in an analogue way as *exceptions* (Section 4.2.3).

7.2.2 IDRA changes handler

The *IDRA changes handler* is responsible of recording all changes that a user makes to his code-base, and to propagate them to all the machines connected to it. It records all the changes made by the developer to an object oriented code base, including addition, changes and removal of classes, methods, instance variables and class variables. It stores all changes done during a debugging session until the developer decides to propagate them. At that moment all session changes are sent to all IDRA changes handlers that are connected to it.

When the remote IDRA Changes Handler instances receive a set of changed from the developer's side, they apply all the changes in order updating their code base.

This allows a developer to debug its exception and change the code base on his machine, as if he was developing the application locally. All the changes he makes are applied in the same moment when the developer decides it, allowing him to test his solution before deploying to all the other nodes of the system.

7.2.3 Evaluation

We evaluated IDRA in two different ways. First, we demostrated that IDRA can be applied to three different case studies that are representative of non-stoppable applications. Second, we conducted different benchmarks to assess

the feasibility of our approach and compare it to state of the art remote debugging techniques. To this end, we use the concrete implementation of an out-of-place debugger in Pharo.

We measured the execution overhead introduced in a normal application, testing its maximum charge, and comparing its performance against the Pharo Remote Debugger [Kud]. Our measurements show that IDRA introduces negligible overhead on the running time of applications which do not have errors. When failures are present, like in the tests use case, the overhead is more visible, since it takes on average 50% more time to execute the tests of a package when there are 10% of failures. We observed also that IDRA can be heavily charged with exceptions, because it is only limited to the virtual machine memory size limit and the memory assigned to the process by the operating system.

In comparison to Pharo's Remote Debugger, our evaluation shows that IDRA's implementation in Pharo has a higher session initialization time than the Pharo remote debugger, due to the presence of the queue in which all the exceptions are added before a debugging session is opened. On the other hand all debugging operations (*e.g., stepping into, over, resuming*) are on average around one thousand times faster on IDRA then on the Pharo Remote Debugger. We also observed that the amount of data exchanged for exchanging exceptions with a small stack is more than five times lower in IDRA then in the Pharo Remote Debugger. Overall, IDRA is comparable to the Pharo Remote Debugger, and offers room for improvement when coming to overhead in the execution.

7.3 Contributions

This thesis makes the following contributions:

1. We identify the debugging characteristics of what we called *non-stoppable applications*, applications that remotely execute time and data-sensitive processes. These applications cannot be stopped to be debugged, nor cannot fail and stop working.

2. We propose and implement a debugging technique called *out-of-place debugging* to remotely debug such non-stoppable applications. In contrast to traditional remote debugging, this technique allows to debug a failure that happened remotely in an another machine reconstructing there the necessary runtime environment. The use of a different environment makes it possible for the different processes of the application to con-

tinue working, while the debugging and code change for bug fix a faulty process happens on a different machine.

3. We implement the out-of-place debugging model in IDRA, a debugger specifically designed for non-stoppable applications written in Pharo. It can handle multiple exceptions, enqueueing them, and allows the use of classic online debugging techniques, introducing two kind of breakpoints and different ways to handle exceptions.

4. We demonstrate that IDRA is a good alternative to state of the art remote debuggers, in particular, PharmIDE [Kud], the last implementation of Mercury [PBFD15]. In fact it outperforms PharmIDE on execution time of debugging operations, being one thousand times faster, on average. on the applications presented in this thesis. Although, IDRA introduces a negligible overhead to the execution of a program and shows a consistent overhead when comparing the initialization time of a debugging session to PharmIDE.

5. As technical contribution in the Pharo community, we provide an implementation of a framework for the *master/worker* distributed architecture. This framework allows to construct distributed applications following the master/worker architecture in Pharo, abstracting the communication between the different nodes.

7.4 Limitations and future work

In this section we analyze some limitations of the IDRA prototype in Pharo and future work.

Local Resources Unavailability

In our prototype, external resources (such as local files, sensors at the local machine's hardware) are not shared between the IDRA Manager and the IDRA Monitors. When debugging directly (or via proxies) an exception that happened in the same machine this problem does not appear. On the other hand, in our solution we reconstruct an environment on a separated machine, so we could have a problem of resource unavailability.

For example, if a developer runs IDRA on the sensor monitoring application, he has to avoid trying to access a sensor from the developer's machine. This is because the sensor is not physically present on the machine he is using to debug, and a call to get information from it will fail. This problem does This is a problem akin to *code mobility*, and many possible solutions can be

found in the literature [FPV98]. For example proxies can be used for those instances that cannot be reproduced.

Serialization

The serialization library used in this thesis (*Fuel* [DPDA11]), provides serialization of all the objects reachable from a certain starting point, namely, in our case, the exception and its context. While this is a desirable property for a serializer, this may become problematic when employing it for an out-of-place debugger. Indeed, some times the object graph reachable when execution stops at an exception or breakpoint, may have references to global objects and include objects that the developer did not expect. Note that serialization has a direct impact on the size of the exchanged stack traces.

Avoiding the serialization of such object can be solved analyzing and optimizing the serialization process to handle better some references, for example references to processes. For example in PharmIDE [Kud], the current implementation of Mercury [PBFD15], the serialization process is optimized to serialize proxies on the TCP network.

Network stability

Communication in our IDRA prototype happens over TCP/IP and we rely on its mechanisms for network failure handling. This means a slow network or machine can lead to errors because of too short timeouts. We did not implement any high level failure handling, nor a robust disconnection/re-connection mechanism either. This might lead to stop the execution of IDRA.

In order to solve this problem, we could implement a better communication protocol to avoid using timeouts, and build an automatic system for re-connection of disconnected nodes. Furthermore, adding a visualization of the connected machines would help the developer to have a clearer view of the system.

7.4.1 Future work

The technical future work plans are to make IDRA stable and usable, and release it within the Pharo community. The issues of network stability and serialization need to be tackled before doing so, but different improvements are possible. For instance, a better handling of the queue, e.g. grouping similar exceptions avoiding to transfer them multiple times, could significantly reduce the communication overhead. Tackling the unavailability of resources will make IDRA fully compliant with the applications it is designed to debug.

In terms of out-of-place debugging, a querying system on the exception queue would improve the debugging experience, especially when many exceptions are present. The debugging experience could also be improved with a better visualization of the state of the debugger and of the different nodes, and the jobs they are executing. Being in object oriented programming, we would also like to extend the support of out-of-place debugging to shared state applications.

Moreover, a deeper analysis of the different debugging features necessary for data intensive applications is needed. We will then try to apply IDRA to other real-world applications. Therefore, we have concrete plans with Yesplan to use IDRA to debug their tests, in their real setup, and check, in practice, if IDRA could help deal with the non-determinism of their tests.

We will also work to enforce an atomic and more dynamic application of changes. To this extent we plan to continue working with Steven Costiou, extending the sensor monitoring scenario to use a system of dynamic layers adaptation that he his developing [CKDP17].

Finally, we plan to investigate if the mechanism underlying IDRA can be used in broader contexts than debugging.

Bibliography

[ABF05] Alex Abacus, Mike Barker, and Paul Freedman. Using test-driven software development tools. *IEEE Softw.*, 22(2):88–91, March 2005.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[ANF03] Kento Aida, Wataru Natsume, and Yoshiaki Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *Proceedings - CCGrid 2003: 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 156–163, 2003.

[Apaa] Apache. Apache giraph. http://giraph.apache.org/. Accessed: 2017-05-10.

[Apab] Apache. Apache spark. `http://spark.apache.org/`. Accessed: 2017-05-12.

[Apac] Apache. Hadoop. `http://hadoop.apache.org`. Accessed: 2017-04-12.

[BNDP10] A.P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Open Textbook Library. Square Bracket Associates, 2010.

[Bra] Santiago Bragagnolo. Taskit. `https://github.com/sbragagnolo/taskit`. Accessed: 2017-05-26.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on*

Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM.

[CKDP17] Steven Costiou, Mickaël Kerboeuf, Marcus Denker, and Alain Plantec. Unanticipated debugging with dynamic layers. Accepted for publication (Apr. 2017), Live Adaptation of Software SYstems, 2017.

[DCD13] Martin Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. *CoRR*, abs/1309.4334, 2013.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DPDA11] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Clustered serialization with fuel. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 1:1–1:13, New York, NY, USA, 2011. ACM.

[DZSS13] Ankur Dave, Matei Zaharia, Scott Shenker, and Ion Stoica. Arthur: Rich post-facto debugging for production analytics applications. *Technical report, University of California*, 2013.

[FDCD12] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with Big Data Analytics. *ACM*, 1072(5220):50–59, 2012.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[GASS06] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[GGSW08] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41(4):30–32, April 2008.

[GIY+16] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[GKYL01] Jean-Pierre Goux, Sanjeev Kulkarni, Michael Yoder, and Jeff Linderoth. Master–Worker: An Enabling Framework for Applications on the Computational Grid. *Cluster Computing*, 4(1):63–70, 2001.

[GNU] GNU. The gnu project debugger. `https://www.gnu.org/software/gdb/`. Accessed: 2017-04-14.

[GNV+11] Elisa Gonzalez Boix, Carlos Francisco Noguera Garcia, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. Reme-d: a reflective, epidemic message-oriented debugger for ambient-oriented applications. In *SAC'11 The 2011 ACM Symposium on Applied Computing*, volume 2, pages 1275–1281, New York, NY, USA, 4 2011. ACM.

[Gol84] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Got09] Chris Gottbrath. Deterministically troubleshooting network applications. *Technical report, TotalView Technologies*, April 2009.

[HS01] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM SYSTEMS JOURNAL*, 41:4–12, 2001.

[JYB11] V. Jagannath, Z. Yin, and M. Budiu. Monitoring and debugging dryadlinq applications with daphne. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1266–1273, Anchorage, AK, USA, May 2011.

[Kud] Denis Kudriashov. Pharmide: Pharo remote ide to develop farm of pharo images remotely. http://dionisiydk.blogspot.be/2017/01/pharmide-pharo-remote-ide-to-develop.html. Accessed: 2017-05-10.

[LCN16] Max Leske, Andrei Chiş, and Oscar Nierstrasz. A promising approach for debugging remote promises. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 18:1–18:9, New York, NY, USA, 2016. ACM.

[LDY13] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 17:1–17:15, New York, NY, USA, 2013. ACM.

[Lew03] Bil Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, December 1987.

[Mar] Stefan Marr. Smark. `http://smalltalkhub.com/#!/~StefanMarr/SMark`. Accessed: 2017-06-01.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[Mica] Microsoft. Dryadlinq. https://www.microsoft.com/en-us/research/project/dryadlinq/. Accessed: 2017-05-10.

[Micb] Microsoft. Remote debugging. `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`. Accessed: 2017-05-12.

[MM80] D. R. McGregor and J. R. Malone. Stabdumpa dump interpreter program to assist debugging. *Software: Practice and Experience*, 10(4):329–332, 1980.

[Net93] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '93, pages 1–11, New York, NY, USA, 1993. ACM.

[Oraa] Oracle. Jdi - java debug interface. http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/ index.html. Accessed: 2017-05-10.

[Orab] Oracle. Jpda - java platform debugger architecture. http://docs.oracle.com/javase/7/ docs/technotes/guides/jpda/. Accessed: 2017-05-10.

[Pac11] David Pacheco. Postmortem Debugging in Dynamic Environments. *Commun. ACM*, 54(12):44–51, 2011.

[Pap13] Nikolaos Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. Theses, Université des Sciences et Technologie de Lille - Lille I, December 2013.

[PBFD15] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, and Marcus Denker. Mercury: Properties and Design of a Remote Debugging Solution using Reflection. *Journal of Object Technology*, 14(2):36, 2015.

[PDB+15] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse, and Max Mattone. Virtualization support for dynamic core library update. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 211–223, New York, NY, USA, 2015. ACM.

[Pha] Pharo. Pharo smalltalk. `http://pharo.org/`. Accessed: 2017-04-14.

[Pro] Selenium Project. Selenium - browser automation. http://www.seleniumhq.org/. Accessed: 2017-05-10.

[PrT09] Guillaume Pothier and ric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26:78–85, 2009.

[RK98] M. A. Ronsse and D. A. Kranzlmuller. Roltmp-replay of lamport timestamps for message passing systems. In *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on*, pages 87–93, Madrid, Spain, Jan 1998.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[SSK+15] Semih Salihoglu, Jaeho Shin, Vikesh Khanna, Ba Quan Truong, and Jennifer Widom. Graft: A debugging tool for apache giraph. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1403–1408, New York, NY, USA, 2015. ACM.

[Twi] Twitter. Twitter streaming public api. `https://dev.twitter.com/streaming/public`. Accessed: 2017-04-14.

[WCS02] Xingfu Wu, Qingping Chen, and Xian-He Sun. Design and development of a scalable distributed debugger for cluster computing. *Cluster Computing*, 5(4):365–375, 2002.

[WPP+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[Yes] Yesplan. Yesplan. `http://www.yesplan.be`. Accessed: 2017-04-14.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[Zin] Zinc. Zinc. `http://zn.stfx.eu/zn/index.html`. Accessed: 2017-05-26.