A debugging approach for Big Data applications in Pharo

Matteo Marra
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
mmarra@vub.be

Clément Béra
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
clembera@vub.be

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Abstract

Big Data applications are more and more popular; they typically analyze big sets of data from different domains. Many frameworks exist for programmers to develop and execute their Big Data applications such as Hadoop Map/Reduce and Apache Spark. However, very few debugging support is currently provided in those frameworks. When an error happens, developers are lost in trying to understand what happened from the information provided in log files. Alternatively, few solutions allow to replay the execution, but they are slow and time-consuming. In this paper, we present an online approach to debug Big Data applications. We first introduce Port, a framework on top of Hadoop Yarn that allows to deploy and execute Pharo Map/Reduce applications. We debug applications deployed on such framework using IDRA, a novel online debugger for Pharo applications. With IDRA the running application can be debugged in a centralized way, and the code of the application can be dynamically updated to fix bugs.

Keywords Distributed systems, software tools, debugging, Big Data, Map/Reduce

1. Introduction

Hardware advances in storage capacity and CPU processing have given rise to the concept of Big Data, characterized by the so-called 3 Vs (Volume, Velocity and Variety). As a result, novel software platforms have emerged to analyze and store such large data sets in a scalable way. The two most prominent programming models are Hadoop Map/Reduce [6] and Apache Spark [2], which typically embrace a batch-oriented data processing to achieve a high parallelisation of data analysis. Current trends indicate that the volume, velocity and variety of data are increasing quickly due to an explosion on diversity and number of sources of information (as a result of the digitalization of data, e.g. smart objects and sensors, interconnectivity of data and popularity of social media data [18]). This poses challenges for Big Data frameworks to be able to meet the new requirements of the emerging real-time streaming data processing applications. The 2017 Hadoop perspective annual report by Syncsort [23], a leading company in (Big) data integration, estimates the need of new tools to simplify the interaction of the programmers with different evolving frameworks and datasets.

Recent work has shown that Big Data platforms provide little or no support for debugging software failures [12]. Developers mostly rely on log files. However, such *post-mortem* debugging technique requires many hours of analysis just to spot a simple problem [9]. Even though specialized tools to visualize and analyze logs for Big Data platforms exist [20], the generated logs can grow to the order of terabytes of data. This makes it extremely difficult to understand production failures, since it is often tricky to extract enough information about a failure from log files [21]. To overcome the use of log files, recent work was focused on replay debuggers, such as Arthur [5]. Replay debuggers allow to replay the execution of the program after it failed to understand where the problem lays. Replay times, however, can increase exponentially in such systems, and it might take hours and multiple replays to spot a particular bug [12]. In addition, it is difficult to apply such techniques on data processing applications that continuously analyze a stream of data.

Online debugging, often called breakpoint-based debugging, is a debugging technique that allows to debug a program while it is executing. Examples of online debuggers are GDB [11], the Python Debugger [10] and the Pharo Debugger [4]. Online debugging can avoid such tedious replay steps, that may take up to hours in complex distributed applications. In this paper, we propose a novel online debugging technique for debugging distributed applications. More concretely, we adapt out-of-place debugging[17] for Big Data applications. An out-of-place debugger transfers the debugging session to an external process, in which the developer

IWST '18, September 10–14, 2018, Cagliari, Italy

ISBN ——…$15.00

DOI: https://doi.org/——

can debug in a safe way. Debugging is centralized, allowing which different nodes to be connected and debugged from an unique point. Our technique also provides dynamic code-update of the running system, allowing to update it without tedious deployment times.

In prior work, we built an out-of-place debugger in Pharo called IDRA[17]. This paper focuses on applying IDRA to a Big Data environment. To this end, we introduce Port, a Smalltalk framework on top of Hadoop Yarn [3], a widely-used resource manager for Big Data. Port can deploy and manage different Pharo environments in different nodes of an existing infrastructure, abstracted using Yarn. To model the distributed computation, we introduced an extensible Master/Worker framework in Pharo. An example of extension is a Map/Reduce framework, that we use to model our Big Data applications. Finally, we model IDRA to support such execution model.

In Section 2 we present some state of the art in debugging Big Data applications. Section 3 gives a background on Yarn and presents our distributed framework: Port. In Section 4 we present the execution model used by Port applications and we provide a debugging scenario derived from [12]. Section 5 introduces IDRA, our out-of-place debugger, and how we extend and use it to debug our debugging scenario.

2. State of the Art

In literature, there are different solutions to debug Big Data applications. Most of them are purely post-mortem, providing debugging capabilities only after the execution of the application failed. Such approaches normally need more than one iteration to actually capture the context of the bug and allow debugging. Among these debuggers we can find Arthur [5], a debugger for Apache Spark, where multiple replays are necessary to find the point of failure. Another solution is Graft [22], a debugger for Apache Giraph [1]. When using Graft, the developer needs to indicate beforehand which particular points of the execution to record, to then be able to replay them afterwards. More recently, Daphne [13] and BigDebug [12] combine replay debugging with some interesting online debugging capabilities.

Daphne is a debugger for DryadLINQ [19] which provides a runtime view of the running system and of the query nodes generated by a LINQ query. It allows developers 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. The execution of a specific node can be replayed locally to analyze its execution using the same debugging primitives.

BigDebug is a debugger for Apache Spark [2] which introduces the concept of a *simulated breakpoint* that does not stop the execution nor freezes the system waiting for the resolution of the breakpoint. Instead, it stores the information necessary to replay the environment and then continues the execution. After the simulated breakpoint, the developer can proceed debugging in a sort of step-by-step execution on the remote node. BigDebug also provides *watchpoints* using *guards*. A watchpoint watches some expression using a predicate function. The developer can then lively visualize the value of the watched expression when the predicate is satisfied.

While Daphne and BigDebug feature some online debugging capabilities, they still fail to correctly capture the context that produced a bug. For instance, BigDebug allows developers, through the use of simulated breakpoints, to remotely debug the application. However, when an exception is raised, the execution stop and the debugger does not capture immediately the context of the bug. Instead, a crash analysis will be used to detect the latest checkpoint before the bug, to then replay the execution from there to reach the bug.

Using BigDebug developers can change code during debugging of a replayed failure, but only to clean a crash including record. Such changes are not applied to the overall system. Major code changes that modify the behaviour of the application need to be done offline and re-deployed on the system. When debugging after placing (and reaching) a breakpoint, some minimal dynamic code update are allowed. However, the new changed function cannot, for example, return a different type than the original one.

In this paper, we study online debugging techniques that can capture the context of the bug when it happens without the need of replay steps. The debugging cycle should also include updating the code of the application while it is still running, enabling to fix some code without stopping, redeploying and restarting all of the components of the system. We prototype these functionalities in IDRA, an out-of-place debugger for applications written in Pharo. We believe that Smalltalk provides a good platform to fast prototype novel tooling for Big Data applications. Nevertheless, the ideas of IDRA could be translated to other programming platforms that (1) provide reflective capabilities that reify the execution stack and provide access to a debugging interface, or (2) allow to introduce virtual machine modifications (when reflective capabilities are limited).

3. A Big Data framework for Pharo

Before explaining our solution to debug Big Data applications, we first provide details about the underlying deployment platform used for Big Data applications.

Rather than creating a Big Data framework from scratch, we decided to build on an existent Big Data infrastructure, namely Hadoop Yarn [3]. Yarn handles the configuration and execution of the system, providing scheduling and nodes management. It is widely used in industry because of its scalability (it can support thousands of nodes) and is commonly used to deploy frameworks such as Map/Reduce and Spark, especially when the size of the system increases. Using Yarn

allows us to abstract on the properties of the system (available memory, available CPU, general availability of a node, ...).

Figure 1 shows the deployment architecture of Yarn. When deploying an application, an application master is set up. The application master can then spawn different containers (using an arbitrary shell command). Such containers are used to execute code of the actual application, and they only report their state to the application master.

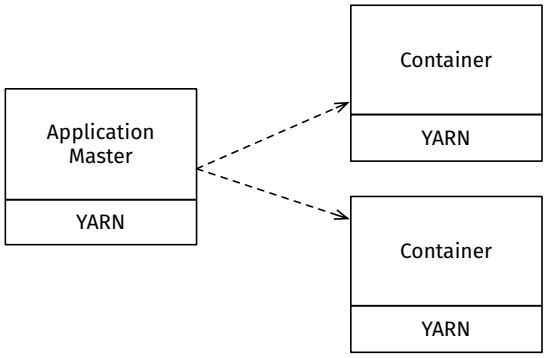

Figure 1. Architecture of Yarn

Upon allocating a container, the application master interfaces with the Yarn framework to know where (i.e., on which node) and when to allocate the container. All the resource constraints (e.g., requested memory vs available memory) are handled by Yarn. If a container cannot be deployed in a particular moment, it will be deployed when the resource constraints are satisfied.

Note that Yarn can deploy any kind of shell-command, i.e. it can execute any kind of application. In this work, we built a framework called *Port* that allows developers to deploy parallel Pharo applications on top of Yarn. Port is composed of two components:

PHOY (PHaro On Yarn) is an instance of a Yarn application master that can spawn different containers running Pharo images. The application master offers a REST interface to handle external monitoring and requests. This interface allows developers to deploy new containers with a particular image and parameters. PHOY also provides an interface to know which containers are running, their IP addresses and other useful information.

Lighthouse is a Pharo tool to control the different containers deployed by PHOY. It interfaces with PHOY using simple HTTP Post requests, and handles the different containers. Lighthouse allows to submit different applications to the system.

4. Programming Big Data applications in Pharo

In order to distribute a Pharo application on top of a Big Data framework like Yarn, we also need to extend Pharo with a parallel programming model. In this work, we model Big Data applications in Pharo using a Master/Worker model. Such an approach is akin to the one used in Apache Spark [2]. The Master/Worker model consists of one master process which acts as coordinator, and many workers ones. The master is responsible for assigning work to the workers and coordinating results. The workers execute code instructed by the master, and may return to it the result of the computation.

The Master/Worker framework is good to model the execution, but does not provide many abstractions to actually build applications. Is is a good basis where to build other parallel models on top of it, such as Map/Reduce [6]. Hence, we extended our Master/Worker programming model to allow Map/Reduce-like applications.

A Map/Reduce application is mainly composed by two functions: a *map* function, that is mapped to all the elements of the input collection, and a *reduce* function, executed after the map, that can reduce all the results to the final one. In our model, a Map/Reduce master can schedule map or reduce tasks on the Map/Reduce worker and handle their result. The master is responsible of parallelizing the computation between the different workers.

Map/Reduce by example

In order to show how Pharo developers can implement Map/Reduce applications, let us consider an election pool analyzing application. The application has been proposed and employed by BigDebug as debugging scenario [12]. It analyzes election pools logs in a parallel way. The analyzed log contains records with the preference of the interviewed person, its region of residence and an UNIX timestamp of when the interview was made. For example: `Toscana Rossi 1517702400`.

Listing 1 shows the core code of the election pool analyzing application. The `map:` method checks if the timestamp of the interview is valid, and filters the interviews for a region (Toscana). The `reduce:` method reduces all the valid entries into an unique dictionary, which will include the information on the preference for each candidate.

Listing 1. The core code of the election poll analysis application.

```
1  PollsAnalyzer>> map: aLine
2  | splitted |
3  (line includesSubstring: 'Toscana') ifTrue: [
4     splitted:=aLine substrings: ' '.
5     (((splitted at: 3) asInteger)
6        >1517616000 ifTrue: [
7        ↑splitted . ] ] .
8     ↑nil .
```

```
9
10 PollsAnalyzer>> reduce: aSetOfVotes
11  | candidate |↑aSetOfVotes
12    inject: Dictionary new
13    into: [ :dict :line |
14       line ifNotNil: [ candidate:=line at: 2 .
15       dict at: candidate ifPresent: [ :val | dict at:
16          candidate put: val+1 ] ifAbsentPut: 1 ] ]
```

Enabling Map/Reduce in Pharo. In practice, the Map/Reduce programming model is enabled by an internal library. In this library, an application is represented by a class that implements the `map:` and the `reduce:` method as shown in listing 1. The execution is modelled on top of the master/-worker model. The master partitions the data and instructs the different workers of the computation that they have to do. The master can both send the data to be processed directly to the worker, or instruct them on where to find the data on the distributed file system.

5. Debugging Port applications

The Port framework described in section 3 can deploy Map/Reduce Pharo applications such as the election pool analyzing application. We now describe the deployment of the election pool analyzing application which features the same bug as described in the original implementation in BigDebug[12]. We first describe the bug, then our debugging approach and finally how to solve the bug with our solution.

5.1 Debugging scenario

When the developer tests the election pool analyzing application on a local instance of Port, on a reduced copy of the dataset, the application works fine. However, when the application is executed on a cluster using Port, one of the worker fails. With the default Pharo/Yarn log-based debugging support, such failure is pretty difficult to handle: containers are deployed remotely, running without an user interface, and by default crash if an error is raised and produce a log.

The bug is actually caused by one of the records, `Bianchi Toscana 02-03-2018`. While the program was expecting a (numeric) UNIX timestamp, the record presented a String-based timestamp that cannot be parsed by `asInteger`, returning a *nil*, hence causing a `doesNotUnderstand:`.

The stack-trace provided in the crash-report shows that there was a `UndefinedObject doesNotUnderstand: #>`. However, it does not provide any information on the data that caused the exception. Alternatively, we could have tried to install a remote online debugger such as TelePharo [14] upfront, to monitor the execution. Deploying many instances of such remote debugger can be tedious: having 1000 containers would mean opening 1000 different debuggers connections, on 1000 different ports on a single remote machine to handle possible errors.

5.2 IDRA

To debug Big Data programs such as the one described in the previous section, we propose to use centralized online debugging which we prototype in IDRA [17], an out-of-place debugger for Pharo applications. In a nutshell, IDRA supports online debugging by transferring the execution state of the debugged application to the local developer's machine. The developer proceeds then to debug as if the application was originally a local application. The remote application can then continue executing the next task that should process. In previous work, we developed IDRA as an out-of-place debugger for long running applications and cyber-physical systems [15, 16].

In this work, we apply IDRA to debug Map/Reduce applications. We will first explain the key concepts in IDRA, then how we adapt it to support debugging of Map/Reduce applications.

IDRA architecture. Figure 2 depicts the architecture of IDRA when working on its default configuration: a single-threaded application runs on an application process which is monitored by IDRA, and the debugger process hosts the front-end of the IDRA debugger.

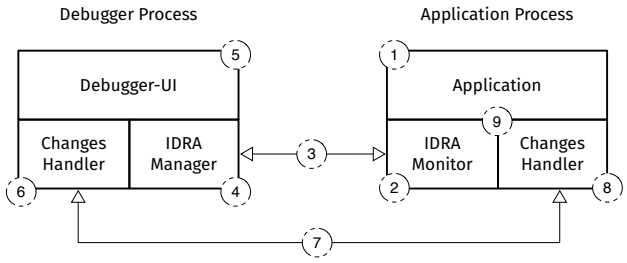

Figure 2. Representation of IDRA instances, manager and monitor and changes handler, in a distributed system of two machines.

When the application monitored by an IDRA Monitor throws an exception or stops in a breakpoint (step 1), the IDRA monitor serializes the program execution state (step 2) and transfers it to the developer's machine (step 3), where the IDRA manager reconstructs the debugging session (step 4). The developer can then proceed to debug locally an exact copy of the original program at the moment of the exception (step 5). If the developer discovers the cause of the bug, he can modify the application code locally to create a bugfix (step 6). At any time, the developer can decide to send all the changes of a bugfix in a single *commit* step to the debugged application (step 7). These changes are applied in the remote application (step 8) and it is finally possible to resume the execution of the suspended point of the application (step 9).

IDRA leverages on Fuel [7] for the serialization of the debugging session and on Epicea [8] for the detection of code changes. All the communication can happen (depending on the configuration) both via direct TCP connections or us-

ing an HTTP Server and asynchronous HTTP post with Zinc [24].

Distributed capabilities of IDRA. The architecture of IDRA is naturally distributed. It is designed in such a way that more than one IDRA Monitor can be connected to a Manager. As a result it is possible to debug different connected applications from an unique point. This is achieved using different queueing systems to handle asynchronously more than one exception. IDRA offers the possibility to selectively choose which debugging session to open, and provides some primitives to, for example, restart all the similar exceptions that it captured. This allows developers to immediately test a new code solution in the local machine on different remote exceptions.

The Changes Handler is also designed to be connected to multiple instances of other Changes Handlers. Changes produced in the debugger can be propagated to the other nodes of the distributed system when the developer decides to do so. Thanks to the code update capabilities of Smalltalk, the code base of different connected nodes can be updated without stopping the application. This reduces debugging and deployment time.

The fact that changes are committed when the developer decides it also allows him to apply to the system only code that he is confident about. We will later describe how, combining IDRA with the Port programming model, we can schedule such updates in the nodes in a safe way.

5.3 IDRA on Port

As we explained in Section 3 the architecture of Port allows developers to deploy a Map/Reduce application, such as the election pool analyzing application, on top of Yarn. In order to debug such application, we deploy IDRA on the same infrastructure.

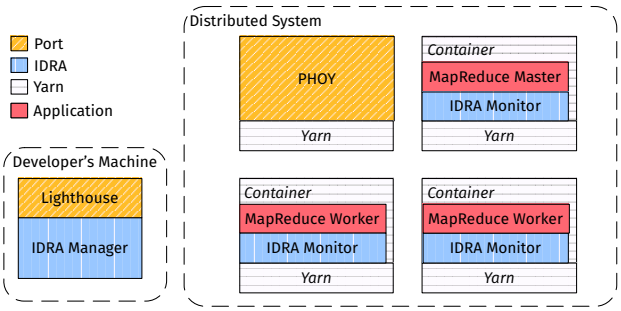

Figure 3. Architecture of Port and IDRA deployed on Yarn

Figure 3 shows the resulting architecture. Components of the same colour-pattern communicate with each other. The developer's machine runs an instance of Lighthouse to control the remote PHOY, and the IDRA Manager, that will allow to debug the remote application if a bug appears or a breakpoint is inserted. The distributed system includes different processes, all running on top of Yarn. One process contains PHOY, correspondent to an Yarn application master. The other processes contain an instance of either the Map/Reduce Master or one of the Worker and, in any case, an IDRA Monitor. Such instances could be deployed on one or different nodes of the distributed system. Port abstracts this leveraging on Yarn. Together with the IDRA Manager and Monitor instances, there is always an associated IDRA Changes Handler (not shown in this figure for simplicity).

This architecture is scalable since all the processes are executed in independent Yarn containers. This means that the scalability of Port is linked to the one of Yarn, known to support thousands of connected nodes. The API of Yarn can be queried to report the status of the different containers, and the instance of Lighthouse can retrieve this information from the remote PHOY instance. In this way Lighthouse allows developers to have a complete view of the state of the system, with its different containers, master and workers.

With the current implementation, PHOY needs to be submitted to Yarn. After PHOY is running, Lighthouse can reach it at a predefined URL. Lighthouse can then be used to deploy different containers with a master or a worker. We assume that (1) Pharo is installed at a known path in all the nodes of the distributed system, and that (2) a Pharo image is present at a known path of the distributed system. The developer can deploy containers by specifying the path to the Pharo image they need to run.

5.4 Debugging scenario in IDRA

Thanks to Lighthouse, Pharo developers can just run their Big Data applications on Yarn with a statement such as:

```
Lighthouse run:PollsAnalyzer on:'/data/polls'
```

where `'/data/polls'` is the path to the data that has to be analyzed[1]. The application will then be automatically run on the system. If it correctly terminates, the Map/Reduce master executes the method `#handleResult:` of the application with the result of applying `#reduce:`.

If some of the computation fails, a debugging session opens at the developer's machine (running the IDRA Manager instance as shown in Figure 3). The workers will continue executing, analyzing all the records even if some of them cause a failure. This feature is extremely useful, especially if the application fails only for some specific records (e.g. our faulty record with a non-UNIX timestamp). But, for the cases this behaviour is not desired, this feature can be turned off, and the application will stop executing at the first failure.

Applying code changes. During debugging the developer will change code to solve the problem. In our debugging scenario that means handling also string-based timestamps. The developer can the restart that particular execution locally,

[1] Our framework supports the Hadoop Distributed File System (HDFS) through our Pharo-HDFS library (`https://gitlab.soft.vub.ac.be/Marra/pharo-hdfs`)

checking that it correctly terminates. Afterwards, he can decide to commit his bugfix to the running application. This bugfix will be applied in all the nodes containing an IDRA Monitor. After applying the changes, the different workers that presented a failure can restart the failed computations with the updated code. In this way the records that caused a failure will be, as well, included in the final result.

In order to avoid applying changes while a worker is executing the code of the application, our Changes Handler can instruct it to apply the changes only between executing different tasks. For instance, if the worker is scheduled to execute two different map tasks, he will schedule a special updating task between the two (or when both are finished, depending on the configuration).

Comparison with BigDebug. In contrast with BigDebug, IDRA captures immediately the bug context, without any replay step. Furthermore, it does not limit the code that the developer is allowed to change, enabling more complete updates of the code.

6. Conclusion

In this paper we presented an online debugging approach for distributed applications in Pharo. We first described Port, a distributed framework for Pharo running on top of Hadoop Yarn, a mainstream resource manager used by Big Data frameworks. Port models the execution with an extensible master/worker model, that we extended with a Map/Reduce model in this work. We show how to implement Big Data applications in Port by means of the election pool analyzing application, originally described in the work of Gulzar *et al.* [12], and we show how we can debug it using IDRA, our out-of-place debugger for Pharo. The main characteristics of IDRA are:

1. It completely moves the debugging session from the application process to an external process, allowing to debug it in an isolated environment while the application still runs.
2. It is deployable on a realistic distributed environment and allows to debug different nodes in a centralized way.
3. It provides dynamic code updates facilities to propagate code changes in the distributed system.

Overall, IDRA enables centralized online debugging of (distributed) Pharo Map/Reduce applications. Furthermore, Port is deployable on state-of-the-art clusters thanks to the use of Hadoop Yarn. We believe that IDRA is a good starting point to develop new debugging techniques for such complex Big Data systems.

As future work, we would like to improve the debugging model to have more knowledge of the application that it is debugging. In fact, in the current implementation, IDRA treats the different processes of the application (e.g. different Map/Reduce Worker) as single applications. It can debug (and update) all of them in a centralized way, but it does not have the necessary knowledge to treat them as an unique running application. We are also interested to track eventual dependencies, both of data and computation, between the different processes and tasks of the distributed application. Furthermore, we want to improve the code updating capabilities of our system, to handle, for instance, a partially updated distributed system. In general, the system should avoid executing a certain application flow with different version of the code. At the framework side (Port), further work is need to optimize the data transfers over the network.

Acknowledgments

Matteo Marra is a PhD-SB fellow at the Fonds Wetenschappelijk Onderzoek - Vlaanderen - Project number: 1S63418N. We would like to thank the anonymous reviewers for their helpful comments.

References

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

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

[3] Apache. Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, . Accessed: 2017-08-24.

[4] A. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Open Textbook Library. Square Bracket Associates, 2010. ISBN 9783952334140. URL `https://books.google.be/books?id=5ok3AgAAQBAJ`.

[5] A. Dave, M. Zaharia, S. Shenker, and I. Stoica. Arthur: Rich post-facto debugging for production analytics applications. *Technical report, University of California*, 2013.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

[7] M. Dias, M. M. Peck, S. Ducasse, and G. 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. ISBN 978-1-4503-1050-5. doi: 10.1145/2166929.2166930.

[8] M. Dias, D. Cassou, and S. Ducasse. Representing code history with development environment events. *CoRR*, abs/1309.4334, 2013. URL `http://arxiv.org/abs/1309.4334`.

[9] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker. Interactions with Big Data Analytics. *ACM*, 1072(5220):50–59, 2012.

[10] T. P. S. Foundation. The python debugger. `https://docs.python.org/2/library/pdb.html`. Accessed: 2017-11-30.

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

[12] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. 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. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884813.

[13] 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. doi: 10.1109/IPDPS.2011.268.

[14] D. Kudriashov. Telepharo. `https://github.com/dionisiydk/TelePharo`. Accessed: 2017-11-30.

[15] M. Marra. Idra: an out-of-place debugger for non-stoppable applications, 2017. Vrije Universiteit Brussel. Brussels, Belgium. `http://soft.vub.ac.be/Publications/2017/vub-soft-ms-17-01.pdf`.

[16] M. Marra, E. G. Boix, S. Costiou, M. Kerboeuf, A. Plantec, G. Polito, and S. Ducasse. Debugging cyber-physical systems with pharo: An experience report. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 8:1–8:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5554-4. doi: 10.1145/3139903.3139913.

[17] M. Marra, G. Polito, and E. G. Boix. Out-of-place debugging: a debugging architecture to reduce debugging interference. *To appear in The Art, Science and Engineering of Programming*, 3(2), October 2018.

[18] C. K. Mayer-Schönberger, V. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. London: John Murray., 2013.

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

[20] M. Mohandas and P. M. Dhanya. An approach for log analysis based failure monitoring in hadoop cluster. In *2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE)*, pages 861–867, Dec 2013. doi: 10.1109/ICGCE.2013.6823555.

[21] D. Pacheco. Postmortem Debugging in Dynamic Environments. *Commun. ACM*, 54(12):44–51, 2011. ISSN 0001-0782. doi: 10.1145/2043174.2043189.

[22] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. 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. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2735353. URL `http://doi.acm.org/10.1145/2723372.2735353`.

[23] Syncsort. Syncsorts third annual hadoop survey uncovers big iron to big data trends to watch in 2017. http://www.syncsort.com/en/About/News-Center/Press-Release/Syncsort-Hadoop-Survey-for-2017. Accessed: 2017-08-22.

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