A debugging approach for live Big Data applications[⋆]

Matteo Marra[a,∗], Guillermo Polito[b], Elisa Gonzalez Boix[a]

[a]*Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium*
[b]*Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 , CRIStAL, Lille, France*

Abstract

Many frameworks exist for programmers to develop and deploy Big Data applications such as Hadoop Map/Reduce and Apache Spark. However, very little debugging support is currently provided in those frameworks. When an error occurs, developers are lost in trying to understand what has happened from the information provided in log files. Recently, new solutions allow developers to record & replay the application execution, but replaying is not always affordable when hours of computation need to be re-executed. In this paper, we present an online approach that allows developers to debug Big Data applications in isolation by moving the debugging session to an external process when a halting point is reached. We introduce IDRA$_{\text{MR}}$, our prototype implementation in Pharo. IDRA$_{\text{MR}}$ centralizes the debugging of parallel applications by introducing novel debugging concepts, such as composite debugging events, and the ability to dynamically update both the code of the debugged application and the same configuration of the running framework. We validate our approach by debugging both application and configuration failures for two driving scenarios. The scenarios are implemented and executed using Port, our Map/Reduce framework for Pharo, also introduced in this paper.

Keywords: Online Debugging, Big Data, Map/Reduce, Live Programming

1. Introduction

Hardware advances in storage capacity and CPU processing have given rise to the field of Big Data. This field is characterized by the so-called 3 Vs: Volume, Velocity, and Variety [28]. Big Data applications analyze a constantly increasing Volume of data, coming at an increasing Velocity and from a day by day bigger Variety of sources. 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 for Big Data are Hadoop Map/Reduce [6]

[⋆]This work is an extension of Marra et al. [17]
[∗]Corresponding author
Email addresses: `mmarra@vub.be` (Matteo Marra), `guillermo.polito@univ-lille.fr` (Guillermo Polito), `egonzale@vub.be` (Elisa Gonzalez Boix)

and Apache Spark [2], which typically embrace batch-oriented data processing to achieve a high parallelization 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, the interconnectivity of data and popularity of social media data [19]). This poses challenges for Big Data frameworks to be able to meet the requirements of the emerging real-time streaming data processing applications. For example, the 2017 Hadoop perspective annual report by Syncsort [27], a leading company in (Big) data integration, estimates the need for 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, that can easily grow to the order of terabytes of data. Even though specialized tools to visualize and analyze logs for Big Data platforms exist [22], it's often extremely difficult to understand production failures from such log files [24]. As a result, such *post-mortem* debugging techniques may require many hours of analysis just to spot a simple problem, such as a minor bug in the application or a configuration error [9].

To overcome the use of log files, recent work has proposed replay debuggers ([5, 26, 21]), which allow developers to repeat a recorded execution. Once a program execution failed, if it was recorded, it can be replayed. Replay times can, however, increase exponentially in such systems, and it might take hours and multiple replays to spot a particular bug [12]. Online debuggers, on the other hand, can potentially shorten the time to find a bug by avoiding replay steps. They control a program's execution by placing breakpoints in specific points of interest during the execution and stepping until the bug is hit. However, typically traditional debuggers (like GDB [10] and the Java debugger [23]) pause the entire execution while debugging. This solution is not always feasible for long-running applications such as Big Data applications.

In this paper, we propose a novel online debugging solution targeted to Big Data applications. In prior work, we proposed out-of-place debugging[18], an online debugging architecture which transfers the debugging session to an external process, in which the developer can debug in an isolated way. Based on this, we augment out-of-place debugging with dedicated features to allow debugging of Map/Reduce applications. With our solution, developers can debug within the same integrated development environment (IDE) both application failures and the so-called configuration failures present in Map/Reduce applications. We also present novel debugging features to combine and relate errors that happen across the parallel execution of Map/Reduce applications. In particular, using our solution developers are able to debug only the failed parts of the computation, with a clear knowledge of which data caused a certain exception. Furthermore, they are also able to propagate code changes to the execution environment without needing to re-deploy or restart.

We prototype our solution in Pharo. For the development of applications, we rely on Port, a Map/Reduce programming model in Pharo introduced in prior work [17] which we augmented with a framework for dynamically deploying Pharo on state-of-the-art Hadoop clusters. We prototype our debugging solution for Map/Reduce applications in IDRA_{MR}, an out-of-place debugger for Pharo applications. We validate our solution by describing three different debugging experiments through two inspiring scenarios, showing how IDRA_{MR} can help to debug both application and configuration level bugs in prototypical Big Data applications.

This paper complements our previous paper [17] by deploying out-of-place debugging on a Map/Reduce distributed architecture, and by defining new debugging abstractions, such as composite exceptions and debugging of virtual partitions. More concretely, the main contributions of this paper are:

1. We introduce the concept of composite debugging events, which aggregate occurrences of a single exception or breakpoint in different workers on a same parallel execution.
2. We introduce debugging operations on virtual partitions, to debug locally a failed parallel execution.
3. We validate our approach by applying it on two real-case analysis: a polls analysis application and blockchain analysis application.

As a technical contribution, we provide *Port* [1], a Smalltalk implementation of the Map/Reduce programming and execution model, and IDRA_{MR}, a debugger for Port applications based on the concept of Out-Of-Place debugging [18]. Furthermore, we introduce *Pharo on Yarn*, a library to dynamically deploy Pharo images on different nodes of a cluster using Hadoop Yarn [3].

2. Motivation

To show the different problems that can arise when debugging Big Data applications, we present here two concrete scenarios of applications featuring a failure. In particular, we illustrate the debugging of the two most representative types of failures in Map/Reduce applications: *application-level failures*[15], also known as application bugs, and *configuration and installation bugs* which are reported to cause more than half of the bugs in Hadoop clusters [25].

Note that code samples in this paper use Smalltalk. We will explain the necessary features of the language to understand the contributions of this work along with the explanation of code samples using footnotes.

2.1. Application Bugs by Example: Poll Analysis Application

A classic example of a Big Data application is an election polls analyzer, akin to the one presented by Gulzar et al. [12]. This application analyzes a dataset containing the results of the election polls and computes, for one region, the number of votes received by each of the candidates. This application actually boils down to a word-count computation, which lies at the core of many

[1]Soon available at `https://github.com/Marmat21/Port`

other applications in Map/Reduce [6]. We implemented again the application as described by Gulzar et al. [12], in which the data is formatted as file entries with the following fields:

`{Region Name Timestamp}`

We also introduced in our application the same bug as in BigDebug [12]: when the application is executed on a Hadoop cluster, a single worker fails. Since the application is deployed remotely without a user interface, this failure produces a log. The stack-trace provided in the crash-report shows that there was a parsing error (a number between 0 and 9 was expected), providing the stack trace shown in Listing 1, but not providing any information on the data causing the exception.

```
2019−04−16T16:51:56.532637+02:00
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
[ :el | self map: el ] in VoteCountingMRApplication(MapReduceApplication)
     >>applyMapTo: in Block: [ :el | self map: el ]
Array(SequenceableCollection)>>collect:
VoteCountingMRApplication(MapReduceApplication)>>applyMapTo:
```

Listing 1: Stacktrace in the log file of the failing polls analysis application.

This particular bug is caused by the record `{Bianchi Toscana 02-03-2018}`. Indeed, the program was expecting a numeric UNIX timestamp while the record presented a String-based timestamp. Our poll analyzer code was parsing the timestamp using `asInteger`, which returns *nil* with the non-numeric record, causing an unexpected exception.

In this concrete example, finding the error would be trivial if the developer was provided with the contextual information about the state of the application, *i.e.,* which record caused the exception.

Since the bug did not manifest in the test-set of the developer, she could try to add insightful log statements to, for instance, print the runtime values of the arguments to detect which one causes the error. This, however, would require to re-deploy and restart several times the execution to find the bug. Moreover, printing runtime values of arguments may fill the log with extra information which will not help the developer to find the bug (e.g., information about not faulty execution). Alternatively, the developer could use more advanced techniques, such as data provenance [11], to detect which of the records caused the error. However, such a technique also requires various replays of the execution untill debugging can happen. In short, using post-mortem debugging to reproduce the bug requires, after an initial analysis of a log file, different re-executions, losing hours of processing as the analyzed data set grows.

2.2. Configuration Bugs by Example: Blockchain Analysis Application

A second representative example of a Big Data application is a Map/Reduce application that analyzes an existing Blockchain platform (*i.e.,* Ethereum) and indexes each of its blocks, storing an association (*index* $\Rightarrow$ *hash of the block*) in a relational database. When done sequentially, such an analysis takes days of computation. As a result, many times Blockchain analysis techniques are limited. For instance, BlockSci [14] scoped their bitcoin analysis to only 22GB of transactions. We implemented this analysis as a Map/Reduce application in a Hadoop cluster, taking 7 hours to process 266GB of transactions [4].

Listing 2 illustrates the pseudocode of such an application. The `map` function queries the blockchain to obtain the data related to a block index. The `reduce` function takes the result of the map on several indexes (*i.e.,* a partition of indexes), and stores them all in a centralized database with a bulk insert. Both the blockchain and the database are accessible at known network addresses through drivers loaded in the runtime environment.

```
map(blockIndex) {
  return blockIndex−>hash(blockchain.at(blockIndex))
}

reduce(pair){
  storeInDatabase(pair)
}
```

Listing 2: Pseudocode of the blockchain indexer.

While developing and executing this application, we faced different configuration bugs that invalidated the results generated by minutes (or hours) of computation. For example, one bug made the application fail when attempting to store the associations in the relational database. After analyzing the logs of the failed application (included in Appendix C) we realized that the application developer forgot to drop the existent tables in the database, making all the stores fail because there was already data with the same primary key from previous executions. Such a bug is representative of the case of a production environment that is not fully ready to execute the application. Similar bugs can also happen when a library is missing or mis-deployed in the execution environment.

Fixing this kind of configuration bug is relatively easy as it does not require extra coding but only the re-deployment of the right configuration files or libraries, or the restart of a service like a database, or, in this specific case, executing a script to drop existent tables in the database. Identifying that we are facing a configuration bug is, however, much more difficult because the root cause of the failure is not in the application. This means developers need to analyze logs that contain information about other components of the framework they are using to implement their applications, requiring them to understand implementation details of the framework to figure out what it is being reported as a failure.

2.3. Problem Statement

Debugging Big Data applications is difficult because of different factors. On the one hand, their distributed nature and the size of the systems and data that they analyze complicates the process of identifying a root cause of a failure. On the other hand, not only do programs fail because of application-level errors accidentally introduced by developers, but they also fail because of mis-configuration (of both the application and the execution environment) and initialization errors [25]. These problems, qualified by Zeller [29] as *minor* and *trivial* problems, can be easily solvable in local applications using interactive debugging tools. However, when present in Big Data programs, solving them with the current state of the art debugging tools becomes a time-consuming task even though the fix may be trivial. As Fischer et al. state in their 2012 article [9]:

> *It is frustrating to wait for hours only to realize you need a slight tweak to your feature set.*

In particular, replay debuggers for Big Data applications would restart the full execution even when the fix only affects a part of the failed execution, possibly replaying the execution for hours. A checkpointed-based debugger like BigDebug [12] can alleviate that issue since it allows one to replay only a part of the application process from the lastest checkpoint [12]. However, what both of the presented debugging scenarios have in common, is that the bug becomes apparent when the developer can control the execution of the application and have access to its state when it fails. Moreover, none of the current debugging approaches for Big Data applications feature support to expose both types of failures and deploy fixes for them without restarting the system.

We believe that these shortcomings of the state of the art motivate the need of a novel debugging approach that allows developers to (1) expose both application-level and configuration failures in their Map/Reduce programs executing remotely in one environment (to avoid searching for the root cause in logs in different software technologies involved), and (2) provide primitives to deploy code fixes without restarting the whole system, including deploying library code and changing the configuration of the framework itself.

2.4. Online Debugging of Big Data applications

In this work, we propose an online debugging technique for Big Data applications. In particular, we apply previous research on *out-of-place debugging* [18] and augment it with novel features to tackle the aforementioned shortcomings for Map/Reduce applications.

Out-of-place debugging is an online debugging technique in which debugging happens by transferring the execution state of the remotely debugged application to another machine. The developer proceeds then to debug as if the application was originally a local application. In previous work, we successfully applied this approach to debugging long-running applications and cyber-physical systems [18, 16]. Such a debugging technique suits Big Data applications since it allows production code running on a cluster to continue processing tasks while the failing tasks can be debugged in an external machine. Once the failing tasks are fixed, developers can commit the code changes and restart the now fixed tasks in the production environment.

In this work, we extend out-of-place debugging by customizing its deployment to a Master/Worker architecture which supports a Map/Reduce programming environment, and introduce different abstractions to compose and debug exceptions happening across the parallel execution. Figure 1 provides an overview of the debugging architecture for Map/Reduce applications.

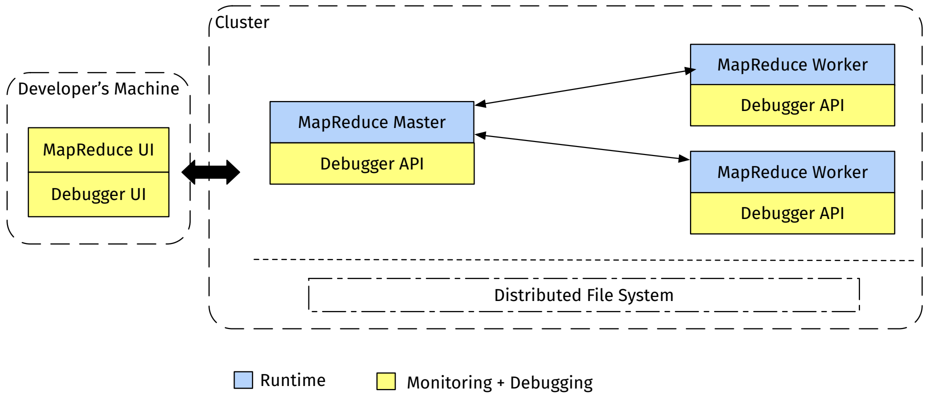


Figure 1: Overview of an out-of-place debugging architecture for Map/Reduce applications

The architecture includes, on the left side, the *developer's machine*, so the machine used by the developer for remote monitoring and debugging of the program execution. The developer's machine thus has an IDE with the MapReduce UI, to monitor the state of the workers, and the Debugger UI. The developer's machine is connected over the network to a cluster running the application. In particular, the cluster runs different processes containing the Map/Reduce Master and different Map/Reduce Workers, that will manage the application execution as explained in the following section. We assume that the different nodes (i.e. master and workers) do not share memory, but that all of the nodes have access to a shared distributed file system (e.g., HDFS in Hadoop clusters). Finally, all of the nodes have a debugger API, detailed later in Section 4 which is used to control and steer execution during debugging. Section 4 will detail this architecture in the context of our prototype implementation in Pharo.

3. Port: A Big Data Framework for Pharo

Before delving into our online debugging approach for Big Data applications, we introduce Port: the programming environment which we employ in this work to write Big Data applications using the Map/Reduce computational model. We also provide the necessary background information on both Master/Worker and Map/Reduce models.

3.1. The Master/Worker Model and Map/Reduce

Port models Big Data applications using a Master/Worker model, akin to the one used in Apache Spark [2]. The Master/Worker model consists of one master process which acts as coordinator, and many worker processes performing tasks. The master is responsible for assigning work to the workers and coordinating results. The workers execute tasks instructed by the master and returns to it the result of the computation. The Master/Worker framework is suitable for modeling the execution but does not provide high-level abstractions to actually program applications. Hence, we introduced a Map/Reduce programming model [6] on top of it.

A Map/Reduce application is mainly composed of 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 intermediate results to a final one. Our Master/Worker framework creates a *Map/Reduce Master* process which is responsible for scheduling map or reduce tasks on different *Map/Reduce Worker* processes and handling their results.

3.2. Map/Reduce by example

A Map/Reduce application in Port is defined as a Pharo class implementing the methods `map:` and `reduce:`. Listing 3 shows the core code of our election polls analyzing application[2]. The `map:` method first filters the interviews for a region (in this case, Abruzzo). It then checks if the timestamp of the interview is valid, reading it from the string as a UNIX timestamp and checks if the date is greater than yesterday.

The `reduce:` method reduces all the valid entries into an unique dictionary, which will include the information on the preference for each candidate.

```
1  PollsAnalyzer >> map: aLine
2      | splitted |
3      splitted := aLine substrings: ' ',
4     (splitted at: 1 includesSubstring: 'Abruzzo') ifTrue: [
5        ((DateAndTime fromUnixTime: (Integer readFrom: (splitted at: 3) )) >
         DateAndTime yesterday) ifTrue: [
6           ↑ (splitted at: 2) -> 1.
7            ]
```

[2] In this code example, you can find syntax that is specific to Smalltalk. `PoolsAnalyzer >> map:` indicates that this is the implementation of the method `map:` in the class `PoolsAnalyzer`. Please note that Smalltalk methods make use of keywords: `line substrings:' '` (cfr. line 3) is equivalent to calling `line.substrings(" ")` in canonical syntax. Methods with multiple parameters (cfr. line 4) are called using a composition of keywords. `splitted at: 1 includesSubstring: '...'` is equivalent to `splitted.includesSubstringAt(1, "...")` in canonical syntax.

```
    ] .
  ↑ nil -> nil.


PollsAnalyzer >> reduce: aSetOfVotes
  | dict |
  dict := Dictionary new.
  aSetOfVotes
    do: [ :vote |
      vote key
        ifNotNil: [ dict
            at: vote key
            ifPresent: [ :val | dict at: vote key put: val + 1 ]
            ifAbsentPut: 1 ] ].
        ↑dict.
```
Listing 3: The core code of the election poll analysis application.

When the application is run, each entry in the input log files is first mapped by the `map:` method and the results of each `map:` invocation is passed as an argument to `reduce:`. Eventually, the poll application returns a set of dictionaries with the number of votes of each candidate.

3.3. Handling input data

A Map/Reduce application accepts different data sources: *i.e.,* (i) an arbitrary collection in memory, (ii) a file on the local file system, and (iii) a file on the distributed file system (*e.g.,* HDFS).

To provide parallel execution of the map and the reduce methods, Port splits the original data into different partitions, that it then assigns to the different workers. In the case of an arbitrary collection, such collection is split equally between the workers and serialized over the network. Instead, when executing on a file (either in the local or distributed file system), the master will instruct the single workers to read each a part of the file, and then to execute the analysis.

Note that in classic Map/Reduce frameworks, the result of the map should always return data in the form of key/value pairs. In Port, map methods are not constrained to return key-value pairs. However, returning key-value pairs becomes mandatory when using *reduce by key* instead of *reduce*.

3.4. Handling intermediate results

Once the map is finished, the partial results of the maps executed in different workers need to be reduced. The developer configures the application to either send the partial results back to the master, store them on an intermediate file on the distributed file system (approach akin to classical Map/Reduce), or keep them in the memory of the workers (approach akin to Apache Spark's workflow).

Before scheduling reduce tasks, a shuffling step [3] might be needed to correctly reduce by key. As other Map/Reduce frameworks, Port handles the eventual shuffling of the data in a transparent way for the developers.

4. Debugging Port Applications with Out-of-Place Debugging

The Port framework described in Section 3 deploys Map/Reduce Pharo applications such as the election polls analyzing application. To debug such applications, in this paper we propose an online debugging technique based on out-of-place debugging [18]. In this section, we first introduce the necessary concepts of out-of-place debugging, then we explain how we applied it to a Big Data context, and finally, we describe the new kind of debugging events devised to debug Map/Reduce applications.

4.1. Out-of-place debugging in a nutshell

Figure 2 depicts the out-of-place debugging architecture. An application runs on a process monitored by the debugger, and an external debugger process hosted in the developer's machine presents the front-end of the debugger. When the application monitored by a debugger monitor stops due to a breakpoint or an exception (step 1), the debugger monitor serializes the program execution state (step 2) and transfers it to the developer's machine (step 3), where the debugger manager reconstructs the debugging session[4] (step 4). The developer then proceeds to debug locally an exact copy of the original program at the moment of the exception (step 5). When the developer discovers the cause of the bug, she modifies the application's code locally to create a bugfix (step 6). Finally, the developer sends all the changes of a bugfix in a single *commit* step to the debugged application (step 7). The explicit commit operation gives the developer control to deploy only code that she is confident about. These changes are deployed in the remote application (step 8) and it is finally possible to resume the execution of the suspended point of the application (step 9).

The out-of-place debugging architecture is naturally distributed: a single debugger manager can connect to multiple debugger monitors at the same time, making it possible to debug different connected applications from a single point. When the debugger manager receives a halted execution from one of the connected debugger monitors, it queues a new debugging session instead of blocking the debugger process by opening multiple sessions. The developers then choose which debug session to open (if more than one is available). Eventually, the developer resumes the execution or cancels the original application process, with the possibility of applying the same operation to all similar debug sessions. This

[3]After a map computation, the resulting key/value pairs are physically at the worker that performed the map. In order to easily reduce by key, key/value pairs that have the same key should be moved to the same worker.

[4]By *debugging session* we mean a practical Pharo debugging session, with a copy of the call-stack and variables as the original debugging session created by the normal execution.

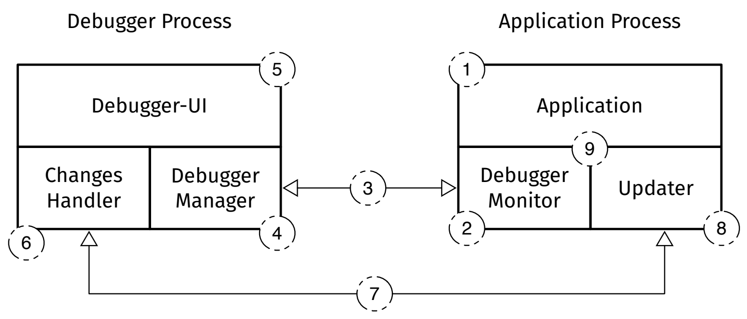


Figure 2: Representation of out-of-place debugging instances, manager and monitor and changes handler, in a distributed system of two machines.

interactive workflow allows developers to inspect and modify a debug session to find and correct bugs in a live way. Code changes produced in the debugger process can be propagated to the application nodes when the developer does a *commit* operation. Such code changes include adding/modifying/removing of both methods and classes. Similar to the debugger manager, the changes handler supports connections to multiple updaters at the same time.

4.2. Out-of-place debugging on Port

In this section, we describe how we adapt the debugging infrastructure (shown in Figure 2) to be deployed on our Map/Reduce runtime (cfr. Section 3). To this end, we build our Big Data debugger, called IDRA$_{\text{MR}}$, by extending the existing implementation of an out-of-place debugger for Pharo Smalltalk applications.

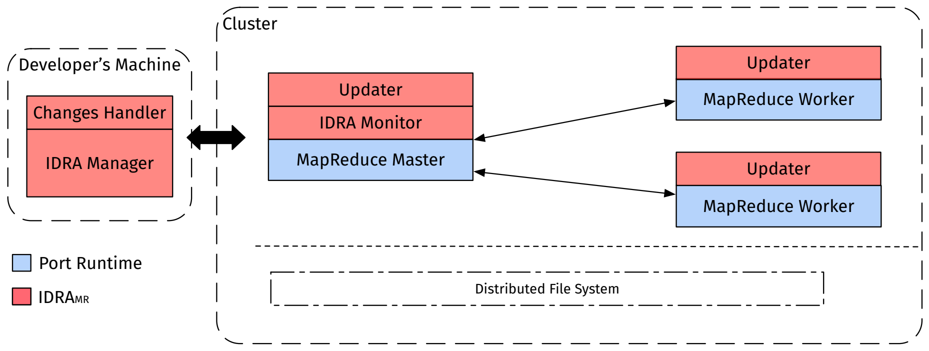


Figure 3: Architecture of Port and IDRA$_{\text{MR}}$ deployed on a cluster

Figure 3 shows the overall architecture of Port when deployed with IDRA$_{\text{MR}}$. Concretely, the different Debugger API instances shown in Figure 1 represent the different instances of IDRA Manager, IDRA Monitor, and Updaters.

The node running the Map/Reduce Master also runs an instance of the IDRA Monitor. Moreover, the Map/Reduce Master node and all the Map/Reduce Worker nodes also run an updater, enabling code updates from an out-of-place debugging session.

External to the cluster, the developer's machine runs an IDRA Manager instance and a Pharo IDE. The IDRA Monitor propagates the exceptions occurring in the cluster to the IDRA Manager instance. The IDRA Manager UI governs the IDRA Manager instance and uses the Port API to communicate with the instances running on the cluster. It presents dedicated UIs to display new debugging features for Big Data applications, which we detail in the remainder of this section.

4.3. Debugging Events and Halting Points

During the execution, the application may reach different halting points. A halting point is a point of the execution in which the execution stops either because of a breakpoint inserted by the developer, or because of an unhandled exception. When a halting point is reached in a Map/Reduce worker, the worker notifies the master. The Map/Reduce master then extracts all of the information needed for debugging from the worker and notifies the IDRA Monitor of a new *debugging event*. A debugging event contains all the contextual information about the halting point. More precisely, it holds an identifier, a copy of the call stack at the halting point, the configuration of the application (*e.g.,* partitioning information), and the data partition that was being analyzed when the event happened. Since the Map/Reduce Master has complete knowledge of the distributed program execution and state, not only of the failed worker(s) but also of the rest of the running tasks of the application, it has access to all the information necessary to construct such debugging event.

As different Map/Reduce Workers are performing parallel map and reduce tasks, the same bug may raise multiple exceptions while analyzing different portions of data in different workers. For example, in the case of the polls analyzer application, if more than one record has the wrong format in the dataset, then the same failure will occur many times during the parallel execution. This will generate many individual debugging events sent to the IDRA Monitor at the Map/Reduce Master process. All these events, however, conceptually belong to a single failure which manifested in different portions of data.

To ease the debugging of such failures, the IDRA Monitor aggregates *all* concurrently raised debugging events related to the same failure into a unique *composite debugging event*. This composite event is then sent to the IDRA Manager at the developer's machine for debugging as if it was one single debugging event. Two or more individual debugging events are aggregated if their call stacks are structurally the same, *i.e.,* the halting point is the same and the call stack frames preceding the halting are called in the same sequence. The composite debugging event will then construct a single call stack which can be further debugged as if it was one failure.

4.4. Composing Events by Example

We now detail how composite events work in the context of debugging the poll analysis application described in Section 2.1. Recall that a composite event is generated when the master receives from the worker(s) the same exception more than once. Figure 4 shows a simplified version of the stack associated with the error in the poll analysis application.

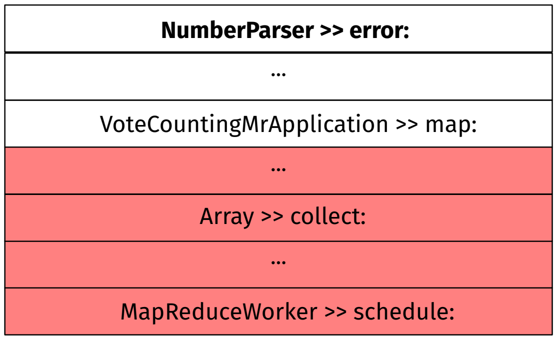

NumberParser >> error:
...
VoteCountingMrApplication >> map:
...
Array >> collect:
...
MapReduceWorker >> schedule:

Figure 4: The simplified stack of the exception. Depicted in red the framework frames. Depicted in white the application frames.

When the worker handles the `NumberParser` exception, it first removes the stack frames related to the framework methods to avoid noise and concentrate on the specific application debugging information (i.e. all frames from `map:` and up). The removed frames are depicted in red in Figure 4. The worker then extracts the meta-data that identifies both the faulty record and its partition and sends it together with the stack in a unique debugging event to the master. The master, in turn, forwards it to the IDRA Monitor.

When a debugging event arrives at the IDRA Monitor, it checks if there are other events related to the same execution. The first time that the IDRA Monitor finds two events to be structurally equivalent, it will generate a composite debugging event for them with a unique id carrying a unique call stack.

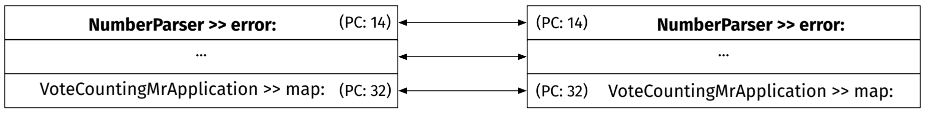
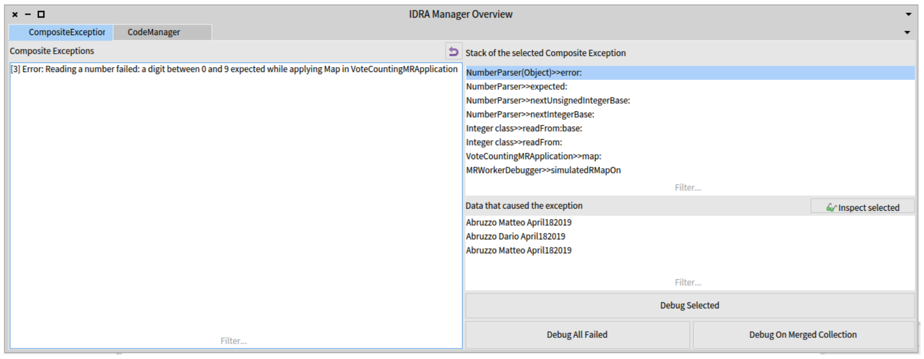


Figure 5: Two structurally equivalent stacks related to the same exception.

Figure 5 shows a simplified representation of the stacks of two structurally equivalent events. We consider two events to be structurally equivalent when (i) they are generated by the same operation (e.g. `map:` in this case) and (ii) each of the stack frames, in order top to bottom, has the same method selector and points to the same program counter (PC). At this point, only one copy of the first stack is stored in the composite event, together with the meta-data of



```
[3] Error: Reading a number failed: a digit between 0 and 9 expected while applying Map in VoteCountingMRApplication
```

```
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
MRWorkerDebugger>>simulatedRMapOn
```

```
Abruzzo Matteo April182019
Abruzzo Dario April182019
Abruzzo Matteo April182019
```

Figure 6: A screenshot of the IDRA Manager UI when handling a composite event. Figures 8 and 9 show, in more detail, the left and the right part of this figure

each of the events. Using such meta-data IDRA$_{\mathrm{MR}}$ is then able to reconstruct the exception for each record that caused it.

When successive structurally equivalent debugging events arrive at the IDRA Monitor, only the first one contains stack information. All the rest contain their unique meta-data and share the identifier of the composite event they belong to. The IDRA Manager will then identify it as a part of a single composite event.

Composite events do not only provide developers with a higher debugging abstraction tailored for the parallelism exhibited by Map/Reduce applications, but they also reduce the amount of memory and network used by IDRA$_{\mathrm{MR}}$. More specifically, the IDRA Monitor hooks into the exception handling of Port, extracting the necessary data from the stack associated with each individual debugging event, to then verify if the stacks are structurally equivalent.

4.5. The Debugging Cycle

A *debugging cycle* in out-of-place debugging denotes the stages from the point in which the developer's machine is notified of a halting point (due to an exception or breakpoint) in a Map or Reduce task of an application until the execution of the halted task is resumed. In this section, we describe the debugging cycle of an application error that manifested in an exception during the execution of the poll analyzer application described in Section 2.1. For a screencast of such debugging cycle, we refer the reader to `https://tinyurl.com/SCPDemo2019`.

Figure 6 shows a screenshot of IDRA Manager UI at the point the developer is signaled of an exception occurring in a *map*:. On the left side, we see the list of distinct exceptions that happened: only one in this case. The number 3 between square brackets denotes that the exception was actually raised three times, meaning this a composite debugging event for the three exceptions.
On the right side, the developers see the stack and the three different records that caused the exception.

Recall from Section 3.3 that the data is split into different partitions, hence when an error happens because of a specific record, such record is part of an associated partition. Through the buttons in the bottom right side of the window, developers can then perform three different debugging operations:

Debug a single halted record. Developers start debugging the map on one of the records for which the execution was halted (denoted those as *halted records*). Once the map on the associated record returns, the developers continue debugging on the rest of records in the same partition.

Debug a virtual partition with all halted records. The developers debug the map on a virtual partition containing only the halting records, regardless of their original partition. For instance, in our example this operation will construct a virtual partition containing all records visible in the bottom part of Figure 9 and let the developer debug the map on such virtual partition.

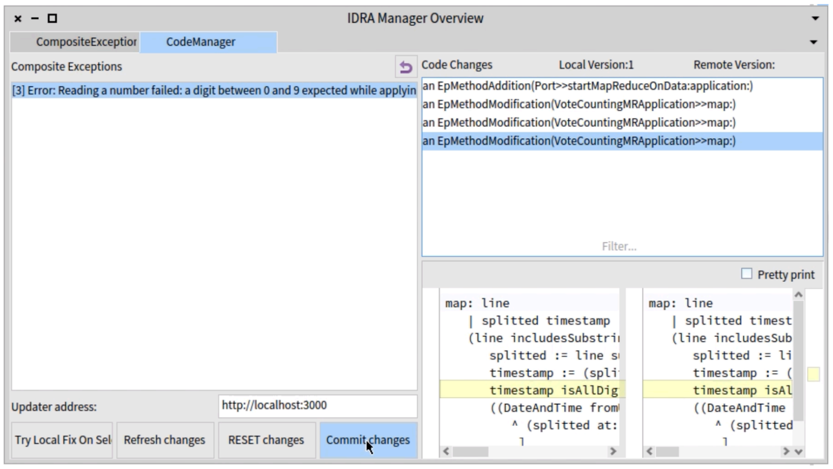
Debug a virtual partition with all halting partitions. The developers debug the map on a virtual partition which is the union of all of the partitions that contain at least one halted record. This virtual partition will contain all records in those partitions, including those that do not halt.

IDRA$_{\mathrm{MR}}$ creates a debugging session by transferring the data required for debugging the requested partition, including the data originally referenced by the stack, the analyzed partition and the current index. Once created, developers use the IDRA$_{\mathrm{MR}}$ UI on the reconstructed debugging session. The IDRA$_{\mathrm{MR}}$ UI extends on Pharo's default online debugger to add dedicated debugging operations for Map/Reduce tasks. More concretely, it provides a new stepping operation that jumps to the map of the next element of the partition, a new operation to resume the execution of all of the remaining elements, and a new operation to halt and inspect the intermediate state.

A debugger UI is created when the application receives a debugging event. Once the debugger UI appears, a developer uses classical debugging operations (step into, step over, resume execution) of the Pharo debugger to debug the reconstructed failed execution on the local machine.

Let's consider that during the debugging session, the developer found the bug and applied a fix. In our concrete scenario, this means modifying the code to manage also string-based timestamps. Those changes are tracked in the *Code Manager* tab of IDRA, displayed in Figure 7. The right side of the code manager shows all of the changes made by the developer while debugging, and the diff of such code changes to the original versions. By clicking on the *commit changes* button, the developer sends the bugfix to the local changes handler.

The bugfix is then immediately propagated by the Changes Handler to the updater instance running alongside the Map/Reduce Master. The Master schedules a task in itself to apply the updates and sends the code fix to the updater instances running alongside each of the Map/Reduce Workers. Note that the



```
[3] Error: Reading a number failed: a digit between 0 and 9 expected while applyin
```

```
an EpMethodAddition(Port>>startMapReduceOnData:application:)
an EpMethodModification(VoteCountingMRApplication>>map:)
an EpMethodModification(VoteCountingMRApplication>>map:)
an EpMethodModification(VoteCountingMRApplication>>map:)
```

```
http://localhost:3000
```

```
map: line
  | splitted timestamp
  (line includesSubstri
    splitted := line s
    timestamp := (spli
    timestamp isAllDig
    ((DateAndTime from
      ^ (splitted at:
        ]
```

```
map: line
  | splitted timest
  (line includesSub
    splitted := li
    timestamp := (
    timestamp isAl
    ((DateAndTime
      ^ (splitted
        ]
```

Figure 7: A screenshot of the code manager tab of IDRA.

update propagation does not happen atomically in all workers at once since each worker will apply the updates only when it finished executing the current task.

Once the code changes are deployed the debug session is finished with one of the following operations:

1. Re-schedule all partitions that halted. This avoids the re-execution of tasks that finished with success.
2. Re-schedule the application from the start on all the partitions. In case the modified code requires an entire re-execution.

The first option is particularly useful when only a small part of the computation failed due to few failure-inducing records. In this case, the developer preserves most of the execution, avoiding tedious replay times, and restart only the failed part of the computation.

4.6. Debugging a configuration error

While explaining the debugging cycle, we focused on application-level failures that are be fixed in the code of the application. However, to debug configuration errors, developers need different debugging operations. IDRA$_{\mathrm{MR}}$ allows developers to (i) load and change code of libraries locally, and propagate the changes as a normal code-update and to (ii) execute arbitrary code directly on the runtime environment of the master and workers. The former leverages on the previously explained code-update capabilities of IDRA$_{\mathrm{MR}}$, the latter makes use of debugging hooks provided by the Port framework itself.

Port allows developers to execute arbitrary expressions in the context of the master, a single worker or all workers. Configuration errors are fixed by executing expressions that modify internal configuration of the nodes or affecting the global running environment. For example, this can be used to programmatically re-start a database in the cluster. We will present more details on how this functionality can be used in Section 5.4.

5. Validation

We validate our approach through three different experiments that highlight the utility of the different debugging features presented in this paper. Throughout the different experiments, we use the two scenarios described in Section 2: the polls analyzer and the blockchain indexing application.

5.1. Experimental Setup

We execute the experiments with Port deployed using Yarn (cfr. Section 6.1) on a 10-nodes cluster. The cluster is composed of one *root* node and ten identical *slave* nodes. Each slave node presents the following specifications:

- Processor: Intel Xeon CPU E3-1240 @ 3.50GHz (4 cores, 8 threads)
- Ram: 32 GB
- Storage: 200 GB SSD

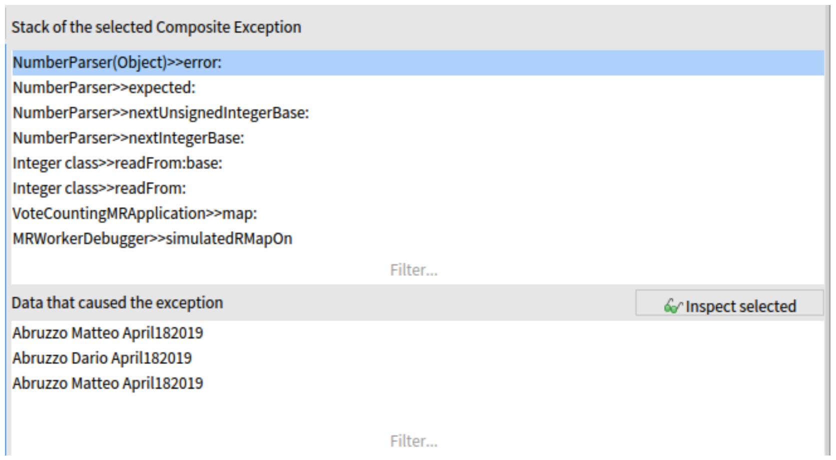
The root node has the same specification as the slave nodes, but it has enhanced storage. All the nodes are connected through a 1 Gigabit local network.

HDFS is running as namenode in the root node, and as datanode in the ten slave nodes. For running the blockchain application, one of the ten slave nodes is exclusively running Geth, a blockchain data node. In addition, the root node is running an instance of the Postgres database.

5.2. Experiment 1: Debugging a Composite Exception

In the first experiment, we compare the debugging cycle for an exception that happened multiple times using log files or using our approach which features composite debugging events. Consider the application level failure of the polls analyzer application described in Section 2.1. Analyzing the log file does not allow developers to get enough information over the execution to know which register or partition caused the failure. Furthermore, if the same exception happened in parallel on more than one map tasks, the log file gets much more complex, and partially replicated, and it still does not give developers enough information. The reader can find such a complete log file in Appendix B. Even if the developer would add explicit log statements to log the intermediate state of a variable, retrieving such log would require multiple executions, and, in order to spot the right record that caused the exception, the developer will need to do a thorough read of the log to find the right statement.

When debugging using our approach, the IDRA monitor will immediately be notified by the exception(s) happening in the different map tasks and will transfer the debugging session, in the form of a composite debugging event, to the


Stack of the selected Composite Exception

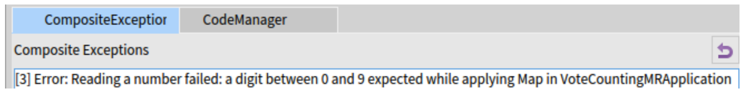
```
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
MRWorkerDebugger>>simulatedRMapOn
```

Data that caused the exception

```
Abruzzo Matteo April182019
Abruzzo Dario April182019
Abruzzo Matteo April182019
```

Figure 9: A detail of the IDRA Manager UI when handling a composite exception. Detail of Figure 6

IDRA manager running on a different external machine, providing centralized debugging and the different debugging operations as described in Section 4.5.



Composite Exceptions

```
[3] Error: Reading a number failed: a digit between 0 and 9 expected while applying Map in VoteCountingMRApplication
```

Figure 8: A detail of the IDRA Manager UI when handling a composite exception. Detail of Figure 6

Figures 8 and 9 show in detail the IDRA Manager handling the exception (cfr. Figure 6). Figure 8 shows the name of the exception, how many times it happened (the number 3 between squared brackets) and where it happened (*i.e.,* the map). Figure 9 shows the shared stack, and shows the different data samples that caused the exception, three in this concrete case. With these visualizations, we can already see that they share the timestamp in a string format, and not in the numeric UNIX timestamp format. The developer can choose one and debug it through the *Debug Selected* button (cfr. Figure 6), or debug a custom set of the data as described in Section 4.5.

In this case, online debugging features such as accessing the state of the application helps to immediately identify the problem. Moreover, the enhanced online debugging features for Big Data applications of IDRA$_{\text{MR}}$, like fix and resume only failed tasks, avoid re-executions of the application to reproduce and fix the bug.

5.3. Experiment 2: Debugging a Configuration Error

In this second experiment, we describe how IDRA$_{\text{MR}}$'s debugging cycle avoids tedious times of re-deployment in case of configuration bugs. Configuration bugs are a classic kind of bug when using big data frameworks, and reported to raise many bugs in Map/Reduce [25]. In this experiment we consider again the polls analyzing application. Consider that, after debugging the application with IDRA$_{\mathrm{MR}}$ and fixing the application level failure as described in experiment 1, the framework correctly finishes executing the program. The code for the application including the fix is shown in Appendix A. When the application finishes, Port will store the final results to HDFS, as indicated in the code of the application (cfr. lines 10-16 in Appendix A).

Consider a configuration and installation error in our poll analysis in which we forget to deploy our HDFS library on the worker nodes at the cluster. This is akin to not correctly packaging a library jar in Hadoop Map/Reduce or Apache Spark. Our correct poll analyzer application will now fail after the reduce task is completed, when the master is handling (and storing) the final result. The program will fail because the class representing the HDFS File System access is not loaded in the worker's execution environment (i.e. The package is not loaded in the image that is running the Map/Reduce Worker). While classic approaches crash the application and require log analyses to find the problem, Port reports to IDRA$_{\mathrm{MR}}$ an exception, in the same way as a classic application exception. The developer then proceeds to load locally the HDFS FileSystem library and IDRA$_{\mathrm{MR}}$ will capture the associated code changes. Such code changes can be committed in order to update the codebase of Port, and the execution of the result store can be restarted. In this particular case, the developer will need to restart the reduce phase to trigger again the result store. This is because, otherwise, the master would need to keep in memory the result that triggered the failure.

Debugging such configuration errors with IDRA$_{\mathrm{MR}}$ and its code updating capabilities avoids the restarting of the whole system. The support for library code update avoids the hassle of packaging errors and related re-compiling and re-deployment steps. This is particularly useful, especially when configuration bugs appear only in a late stage of the computation, as in this example.

5.4. Experiment 3: Debugging cycles

Consider now the blockchain analysis application failing in the reduce phase because of a misconfiguration error of the database as described in Section 2.2. Listing 4 shows the code of the `map:` and `reduce:` methods of the application.

```
MRIndexingApp>>map:blockIndex
  | ethereumBlock mappedProperty |
  ethereumBlock := FogBlockChain at: blockIndex.
  mappedProperty := ethereumBlock
    get: #hash.
  ↑ blockIndex −> mappedProperty

MRIndexingApp>>reduce:pairs
  PostgresDatabase storeIndexedValues: pairs.
```

Listing 4: Map and Reduce methods of the Blockchain indexing application.

The Map/Reduce is initially called on a collection of indexes, meaning numbers from 1 to the maximum number of blocks in the blockchain. The `map:` method executes on a single index, and uses `FogBlockChain`, a global reference to the driver for the blockchain data node, to retrieve a particular block from the blockchain. It then extracts the hash of the block and returns a key/value pair with the index and the associated mapped property (*i.e.,* the hash of the block). In the `reduce:` method, the application takes a set of key-pairs, returned by applying the map to a partition of the collection of indexes, and calls the database, accessible globally, to do a bulk insert of the data.

Note that the `reduce:` method assumes that the database is empty, otherwise a primary key clash error occurs. If the database is not correctly reset, all the reduces fail, after the execution of the map completed without errors. IDRA_{MR} will receive it as a composite event, akin to experiment 1. However, the error will be raised in the call to `PostgresDatabase`, making it clear to the developer that such error is not directly related to the application code, since the code of the reduce is correct, but it is due to a wrong initialization that in turn caused a configuration error.

To solve this configuration error, developers can use Port's remote code execution infrastructure (cfr. Section 4.6) to submit a script performing the database initialization to the node containing the database. They then test the database to check its correct initialization and finally resume the execution.

Since all of the reduces failed because of this error, all reduces need to be resumed. However, by using IDRA_{MR} we avoid re-executing all of the maps. This is crucial in the case of the blockchain indexing application where the entire map execution time for analyzing 266GB of transactions takes more than 6 hours, accounting for 90% of the total execution time of the application [4].

Since out-of-place debugging does not block the whole application but allows to debug and fix individual tasks in an isolated environment, developers can correctly re-initialize the database with the procedure described above, and reschedule the execution only of the reduce operation, saving 6 hours of replaying the computation.

Also in this case, similar to experiment 2, debugging with the remote code execution capabilities of Port and IDRA_{MR} avoids a complete restart of the system, saving precious computational time. The remote code execution also allows developers to inspect intermediate state of the master and of the workers, or change the configuration of Port while it is deployed.

5.5. Discussion

While applying out-of-place debugging on a Map/Reduce architecture provides advantages when debugging parallel applications, it also presents several challenges that we discuss in what follows.

First, debugging a configuration error in a different environment can be tricky, even using our approach. Consider the example of the database configuration error of Section 2.2. While the debugger would show an error produced by the database driver, if the developer restarts locally the execution on her machine, a different error will appear: the database is not available at the developer's machine. This shows a more general problem of code mobility. In the original paper on out-of-place debugging for long-running applications [18] , we solved these particular situations by employing proxies. In particular, a proxy can be added during serialization to objects that are known not to be movable (e.g., a connection to the database, a file, a socket, ...). In this paper, due to the nature of current Big Data applications, we believe that this is a minor limitation to debugging such use cases, as shown in Section 5.4. In fact, such applications often interact (in a stateless way) just with the source of data, having a limited interaction, normally known to the developer, with other external sources.

Second, the presented debugging approach has been devised to work with a Master/Worker and Map/Reduce models. While the Master/Worker model is widely used in frameworks that provide parallel execution, relying on a Map/Reduce programming model may limit the applicability of our approach to different Big Data frameworks (e.g., Spark). For instance, the generation of composite events and the handling of its meta-data, are really coupled, in their implementation, to debugging the `map:` and `reduce:` methods. Applying it to a more extended programming model such as the one of Spark may require different kinds of abstractions. It is ongoing work to extend composite events to a Spark-like programming model.

We discuss technical limitations related to our current prototype implementation in Section 6.4.

6. Implementation

In this section, we describe technical details of our approach, including the deployment of Port and the libraries we rely on, as well as a complete architecture when deploying Port on Hadoop Yarn with IDRA$_{\mathrm{MR}}$.

6.1. Deploying Port on Clusters

Port can be deployed in three ways:

1. **Locally**: with different processes (including master and workers) running on the same machine.
2. **Standalone**: deploying manually the master and the different workers across a distributed system, to then provide a specification to the master to know where the workers are.
3. **On Yarn**: using Hadoop Yarn [3] and our library Pharo On Yarn to deploy the different master and workers on a cluster.

While both local and standalone modes are good for small testing environments, deploying Port on a cluster brings different challenges including how to handle resources, monitor nodes, share data between the different nodes, etc. To this end, we decided to rely on the popular resource manager Hadoop Yarn [3]. Yarn handles the configuration and the deployment of the system. It is commonly used to deploy frameworks such as Map/Reduce and Spark, especially when the size of the system increases since it can scale to thousands of nodes. Yarn allows us to abstract over the properties of the underlying hardware like available memory and CPU, availability of a node, etc.

Figure 10 shows an overview of Port deployed on a cluster using Yarn, including the IDRA$_{\text{MR}}$ debugging infrastructure. As mentioned, resource management is leveraged on Hadoop technology (*i.e.,* Yarn and HDFS), while the execution environment layer we use our Port framework including the Map/Reduce Master and the different Map/Reduce Workers to execute a Map/Reduce application.

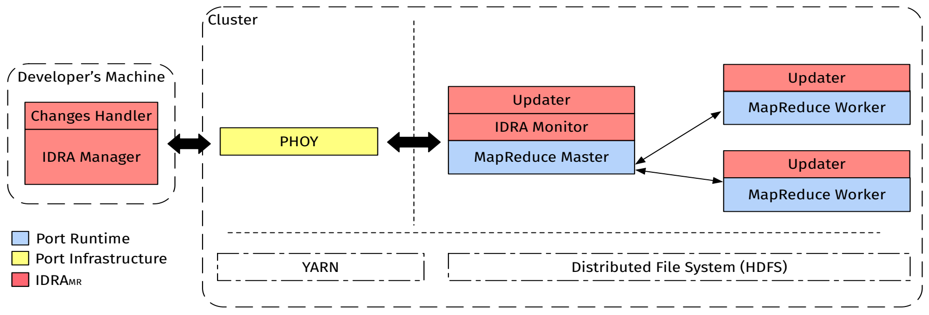


Figure 10: High level view of deploying Port on a cluster

Pharo on Yarn (PHOY). PHOY is a Yarn application used to dynamically spawn different isolated execution environments so-called *containers*. In our case, each container runs either a Port Map/Reduce Master or a Port Map/Reduce Worker. While Yarn takes care of where and when to allocate a new container, PHOY introduces an API to instruct Yarn to deploy new containers and to query information about existing containers. For instance, the Map/Reduce Master will be able to use PHOY to know if a particular Map/Reduce worker is still running.

Supporting a Distributed File System. Typically, on a cluster, a distributed file system allows easy sharing of data between nodes. Such a distributed file system is used both by the developers to store data and result and by the Big Data frameworks to store intermediate results or share data between the different running components. To this end, we provide the *Pharo-HDFS* library which enables Pharo developers and the Port framework itself to access the HDFS file system, the popular Distributed File System of Hadoop. In particular, Port uses Pharo-HDFS to start the execution on a file stored on HDFS and to store intermediate information when needed.

6.2. Handling Composite Events

Composite events (described in Section 4.3) are generated in the IDRA Monitor and sent to the IDRA Manager. Such events contain one copy of the generated stack, as it would be using classic out-of-place debugging, and the metadata of the single exceptions. In the current implementation, such metadata includes the full partition that had the record causing the exception and the index of the faulty record in the partition. This information is enough to reconstruct the exception in the IDRA Manager every time the developer selects a particular virtual partition to debug.

6.3. Communication and libraries

Both Port and IDRA$_{\text{MR}}$ leverage on Fuel [7], a common Pharo library for the serialization of object graphs, to serialize debugging sessions, data, and operations.

Epicea [8] is used in IDRA$_{\text{MR}}$ for the detection and application of code changes. Epicea is a Pharo library for logging, un-doing and replaying code changes made at runtime. It stores the single code changes in an external file.

All communications happen by using HTTP requests with Zinc [30], an HTTP framework for Pharo that allows, among other things, to both manage an HTTP Server and act as an HTTP client.

Thanks to the code-update capabilities of Smalltalk, the code base of different connected nodes can be updated without stopping the application, reducing debugging and deployment time.

6.4. Limitations of the prototype

Both the prototype of IDRA$_{\text{MR}}$ and of Port present different limitations due to implementation choices. We now discuss the most important technical limitations of our prototype in what follows.

When there is an exception, the IDRA Monitor sends to the IDRA Manager the exception, and some meta-data. As explained in Section 6.2, such meta-data includes the full partition that the worker was analyzing when the exception happened. In the case this partition is particularly big, it may cause (i) delays, (ii) the IDRA Monitor or the external IDRA Manager to run out of memory. This can be solved by sending data on-demand.

For serialization, we heavily rely on the Fuel [7] library. While using Fuel spared us much work (e.g., to define a serialization protocol), Fuel is sometimes slow to serialize, and, in some corner cases, it will try to serialize some objects that are not really needed in the debugging session. This introduces delays during serialization and network communication, which could be avoided by optimizing the serialization engine.

7. Related Work

In literature, we can find two well-known families of debuggers: online and offline debuggers [20, 24]. Online debuggers manage the execution of an application at the moment of failure. They allow developers to interact smoothly with a running application, offering breakpoints, watchpoints and stepping operations that give immediate feedback to the developer. Offline debuggers (or post-mortem debuggers), on the other hand, try to help the developers understanding, or sometimes reconstructing, the context of a bug from a failed execution. Such solutions analyze or replay log files, code dumps and/or execution traces to help the developer discover the source of the problem. Reproducing a bug with these techniques can be tedious and time-consuming, especially because many debugging cycles are required before the error happens again as argued in Section 2.

While this paper focuses on devising an online debugging solution, in what follows we compare our approach to the closest related work in debugging approaches for Big Data applications, both for offline and online techniques.

Most of the debugging solutions for Big Data are the so-called *event-based debuggers* [20] that record and store events of one execution for later inspection and or replay. 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 [26], 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 afterward. More recently, Daphne [13] and BigDebug [12] combine replay debugging with some interesting online debugging capabilities. We detail below both approaches and how they compare to our solution.

Daphne is a debugger for DryadLINQ [21] 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. Contrary to IDRA_{MR}, debugging is done remotely directly where the breakpointed node is executing, while IDRA_{MR} moves the debugging session to an external debugger process. Interestingly, Daphne allows to debug also locally but still requiring a replaying step that IDRA_{MR} avoids by moving the debugging session as soon as a halting point is reached. Furthermore, IDRA_{MR} can handle both breakpoints and exceptions in an online way, while Daphne requires a replaying step in case of an exception.

BigDebug is a checkpoint-based 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 in a snapshot (i.e. a checkpoint) and then continues the execution. After the simulated breakpoint, the developer can proceed to debug in a sort of step-by-step execution on the remote node.

When an exception is raised in the application, the execution stops and the BigDebug debugger does not capture immediately the context of the bug, letting the application crash. Crash analysis is then used to detect which part of the execution failed, and then a replay step is required (in the best case from a stored checkpoint). This is avoided by IDRA_{MR}, offering an online debugging session to the developer reconstructing the application context when the failure occurred. Although BigDebug provides some support for hot-fixing the code, it is only limited to one particular execution (the replayed one), and such code fix can only change a particular function (e.g. the lambda that is mapped). For instance, the developers cannot change the type returned by the mapped lambda. This functionality aims to fix a particular crash inducing record, instead of fixing the application. Major code changes that modify the behaviour of the application need to be done offline and re-deployed on the system. In comparison, IDRA$_{\mathrm{MR}}$ can propagate both minor and major code updates in a live and transparent way.

8. Conclusion

In this paper, we presented an online debugging approach for Map/Reduce applications, by the use of online debugging and of debugging abstractions such as composite exceptions. Since our prototype is based on our Map/Reduce implementation, we first described Port, a distributed framework for Pharo. Port models the execution of parallel applications with a master/worker model on top of which we build a Map/Reduce model. We then presented an online debugger for Map/Reduce applications in Port based on the ideas of out-of-place debugging called *IDRA$_{MR}$*. The main characteristics of IDRA$_{\mathrm{MR}}$ are:

1. It completely moves the debugging session from the worker nodes at the cluster to an external process, allowing to debug map or reduce tasks in an isolated environment.
2. It provides dynamic code updates facilities to propagate code changes back to the workers, without requiring stopping the whole distributed system.
3. It centralizes the debugging session, allowing to debug a distributed parallel application from a unique debugger manager.

IDRA$_{\mathrm{MR}}$ introduces also different dedicated online debugging features targeted at Map/Reduce applications. First, IDRA$_{\mathrm{MR}}$ provides *composite debugging events*, as an abstraction of the same event (e.g., an exception or breakpoint) that happened multiple times during the parallel execution of a task. Second, IDRA$_{\mathrm{MR}}$ allows developers to choose three different strategies to determine which kind of data a debugging session operates on (*e.g.,* a virtual partition with all the failing records).

We validate our approach by debugging two concrete cases, an election polls analyzer, originally described in the work of Gulzar *et al.* [12], and a blockchain analysis application. Through three different experiments, we show how our approach can help developers to (i) detect and react to bugs happening in parallel during the execution (ii) discover and fix configuration bugs through remote code execution and (iii) correctly resume the execution of the application with updated code for both application and library code.

As future work, we are planning to generalize our debugging support and operations to other Big Data execution models, such as Spark. We are also planning to consider debugging the dependencies between different operations and data, to improve the debugging experience.

Acknowledgements

We would like to thank Clément Béra for his help in the early stages of this work ([17]). We would also like to thank the anonymous reviewers for their useful feedback.

Matteo Marra is a SB PhD Fellow at the FWO - Research Foundation - Flanders. Project: 1S63418N.

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] S. Bragagnolo, M. Marra, G. Polito, and E. Gonzalez Boix. Towards scalable blockchain analysis. *Proceedings of 2019 IEEE/ACM 2st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, page To Appear.

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

[11] M. Gulzar, S. Wang, and M. Kim. Bigsift: automated debugging of big data analytics in data-intensive scalable computing. pages 863–866, 10 2018. doi: 10.1145/3236024.3264586.

[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] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform. *ArXiv e-prints*, sep 2017.

[15] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5): 1027–1041, Sep. 2005. ISSN 1045-9227. doi: 10.1109/TNN.2005.853411.

[16] M. Marra, E. Gonzalez 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, C. Béra, and E. Gonzalez Boix. A debugging approach for big data applications in pharo. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '18, 2018.

[18] M. Marra, G. Polito, and E. Gonzalez Boix. Out-of-place debugging: a debugging architecture to reduce debugging interference. *The Art, Science and Engineering of Programming*, 3(2):pp. 3:1–3:29, October 2018. doi: https://doi.org/10.22152/programming-journal.org/2019/3/3.

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

[20] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989. ISSN 0360-0300. doi: 10.1145/76894.76897.

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

[22] 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.

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

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

[25] A. Rabkin and R. Katz. How hadoop clusters break. *IEEE Softw.*, 30 (4):88–94, July 2013. ISSN 0740-7459. doi: 10.1109/MS.2012.73. URL `https://doi.org/10.1109/MS.2012.73`.

[26] 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`.

[27] 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.

[28] J. S. Ward and A. Barker. Undefined by data: A survey of big data definitions. *CoRR*, abs/1309.5821, 2013. URL `http://arxiv.org/abs/1309.5821`.

[29] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005. ISBN 1558608664.

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

Appendix A. Code of the polls analyzer application

This appendix provides the code of the polls analyzer application described in Section 2 which is used as running example along the paper. We use the notation NameClass >> nameMethod as a convention in this appendix to denote a method called nameMethod defined at a NameClass class.

```
MapReduceApplication subclass: #VoteCountingMRApplication
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Port−Examples'

VoteCountingMRApplication >> parallelReduce
  ↑ true.

```

```
VoteCountingMRApplication >> handleResult: res
  fs := FileSystem hdfsAtHost: hdfsConfiguration hdfsHost user:
    hdfsConfiguration hdfsUser.
  fileName := fs workingDirectory
    / ('results/result−', DateAndTime now asUnixTime asString , '−' , aResult
    dataId asString , '.' , aResult partition asString).
  fs store createFile: fileName.
  aResult data do: [ :data | fileName writeStream appendAll: data asString ,
    String cr ].

VoteCountingMRApplication >> remotePartitions
  ↑PersistedRemotePartitions.

VoteCountingMRApplication >> map: line
   | splitted |
   splitted := line substrings: ' ',
  (splitted at: 1 includesSubstring: 'Abruzzo') ifTrue: [
    ((DateAndTime fromUnixTime: (Integer readFrom: (splitted at: 3) )) >
    DateAndTime yesterday) ifTrue: [
      ↑ (splitted at: 2) −> 1.
       ]
     ] .
  ↑ nil −> nil.
VoteCountingMRApplication >> isResultKeyable: aCommand
  ↑(aCommand beginsWith: 'applyMap')

VoteCountingMRApplication >> repartitionBeforeReduce
  ↑true.

VoteCountingMRApplication >> reduce: aSetOfVotes
  |  dict |
  dict := Dictionary new.
  aSetOfVotes
    do: [ :vote |
      vote key
        ifNotNil: [ dict
            at: vote key
            ifPresent: [ :val | dict at: vote key put: val + 1 ]
            ifAbsentPut: 1 ] ].
        ↑dict.
```

Appendix B. Log of the failing polls analyzer

This appendix provides the full log filed used in the running example of both motivation (Section 2) and experiment 1 of the validation (Section 5).

```
2019−04−30T13:43:27.442041+02:00 FINISH SCHEDULING OF
     applyMapTo:

HandleResult of applyMapTo:2019−04−30T13:43:27.442168+02:00 MAP
     FINISHED

HandleResult of applyMapTo:2019−04−30T13:43:27.442445+02:00 MAP
     FINISHED

HandleResult of applyMapTo:2019−04−30T13:43:27.442552+02:00 MAP
     FINISHED
2019−04−30T13:43:27.442612+02:00 HANDLING ERROR
2019−04−30T13:43:27.445653+02:00
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
[ :el | self map: el ] in VoteCountingMRApplication(MapReduceApplication)
     >>applyMapTo: in Block: [ :el | self map: el ]
Array(SequenceableCollection)>>collect:
VoteCountingMRApplication(MapReduceApplication)>>applyMapTo:

2019−04−30T13:43:27.445728+02:00 CRITICAL FAILURE

HandleResult of applyMapTo:2019−04−30T13:43:27.445787+02:00 MAP
     FINISHED

HandleResult of applyMapTo:2019−04−30T13:43:27.445858+02:00 MAP
     FINISHED

HandleResult of applyMapTo:2019−04−30T13:43:27.445882+02:00 MAP
     FINISHED
2019−04−30T13:43:27.445909+02:00 HANDLING ERROR
2019−04−30T13:43:27.446234+02:00
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
[ :el | self map: el ] in VoteCountingMRApplication(MapReduceApplication)
     >>applyMapTo: in Block: [ :el | self map: el ]
```

```
Array(SequenceableCollection)>>collect:
VoteCountingMRApplication(MapReduceApplication)>>applyMapTo:

2019−04−30T13:43:27.446262+02:00 CRITICAL FAILURE

HandleResult of applyMapTo:2019−04−30T13:43:27.460139+02:00 MAP
    FINISHED
2019−04−30T13:43:27.493406+02:00 HANDLING ERROR
2019−04−30T13:43:27.493644+02:00
NumberParser(Object)>>error:
NumberParser>>expected:
NumberParser>>nextUnsignedIntegerBase:
NumberParser>>nextIntegerBase:
Integer class>>readFrom:base:
Integer class>>readFrom:
VoteCountingMRApplication>>map:
[ :el | self map: el ] in VoteCountingMRApplication(MapReduceApplication)
    >>applyMapTo: in Block: [ :el | self map: el ]
Array(SequenceableCollection)>>collect:
VoteCountingMRApplication(MapReduceApplication)>>applyMapTo:

2019−04−30T13:43:27.493661+02:00 CRITICAL FAILURE
```

Appendix C. Log of the failing blockchain analysis

In the following we present the log of the blockchain analysis failing during the reduce because of the database initialization problem.

The presented log is only the printed stack, the rest of the log was omitted. Please refer to Appendix B for an example of a complete log.

```
ERROR: duplicate key value violates unique constraint "blocks_hash_pkey"

GAError signal:
GAConnection>>executeAndCheckResult:
GAPostgresDriver>>execute:
UQLLUniqueIndex(UQLLSQLIndex)>>privateSqlExecute:
UQLLUniqueIndex>>registerAll:allongWithFullscanKeys:
UQLLIndexMapReduceApplication>>reduce:parameter:
MapReduceWorker>>scheduleCommand:
ScheduleCommand>>executeOn:
MessageSend>>value
TKTGenericTask>>value
TKTTaskExecution>>doExecuteTask
TKTReadyTaskState>>performTaskExecution:
TKTTaskExecution>>executeTask
```

```
[ self executeTask ] in TKTTaskExecution>>value in Block: [ self executeTask
        ]
[ activeProcess psValueAt: index put: anObject.
aBlock value ] in TKTConfiguration(DynamicVariable)>>value:during: in
      Block: [ activeProcess psValueAt: index put: anObject....
BlockClosure>>ensure:
TKTConfiguration(DynamicVariable)>>value:during:
TKTConfiguration class(DynamicVariable class)>>value:during:
TKTConfiguration class>>optionAt:value:during:
TKTConfiguration class>>runner:during:
TKTTaskExecution>>value
[ self noteBusy.
aTaskExecution value.
self noteFree ] in TKTWorkerProcess(TKTAbstractExecutor)>>executeTask:
      in Block: [ self noteBusy....
BlockClosure>>on:do:
TKTWorkerProcess(TKTAbstractExecutor)>>executeTask:
TKTWorkerProcess>>executeTask:
[ self executeTask: taskQueue next ] in TKTWorkerProcess>>workerLoop in
      Block: [ self executeTask: taskQueue next ]
BlockClosure>>repeat
TKTWorkerProcess>>workerLoop
MessageSend>>value
MessageSend>>value
TKTProcess>>privateExecution
TKTProcess>>privateExecuteAndFinalizeProcess
```