B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 65

Post-Debugging in Large Scale Big Data Analytic Systems

Eduard Bergen[1] Stefan Edlich[2]

Abstract: Data scientists often need to fine tune and resubmit their jobs when processing a large quantity of data in big clusters because of a failed behavior of currently executed jobs. Consequently, data scientists also need to filter, combine, and correlate large data sets. Hence, debugging a job locally helps data scientists to figure out the root cause and increases efficiency while simplifying the working process. Discovering the root cause of failures in distributed systems involve a different kind of information such as the operating system type, executed system applications, the execution state, and environment variables. In general, log files contain this type of information in a cryptic and large structure. Data scientists need to analyze all related log files to get more insights about the failure and this is cumbersome and slow. Another possibility is to use our reference architecture. We extract remote data and replay the extraction on the developer's local debugging environment.

Keywords: Software debugging, Bug detection, localization and diagnosis, Java Virtual Machine, JVMTI, Bytecode instrumentation, Apache Flink, Application-level failures

1 Motivation

Fast and responsive data processing in the era of Big Data separated by the four dimensions, volume, variety, velocity and veracity lead to a high complexity of data management. Complex data management systems offer interfaces to get information about the current system state, but there is a gap of subset observation in an executed large scale big data analytic system (LSA).

When starting an investigation process to find the root cause of a failure, data scientists not only need to investigate a big amount of diverse log-data. Data scientists also need detailed knowledge about the executed processing engine such as Apache Flink to find out where to start the search.

In our application, we are interested in the identification of failure patterns in the data that user defined functions (UDFs) process. Within the identified failure pattern we observed, the subset of the execution graph is of interest. The observation methodology allows doing an extraction of the required information and application values to do a replay. We assume most failure patterns at the runtime layer in the analysis process. Failures, which bring the whole system down such as kernel crashes from the operating system, are not in the scope of this work. Since a kernel crash is not safe and a crashed application cannot detect its own crash without additional and complex tooling.

[1] Department of Computer Science and Media, Beuth University of Applied Science Berlin, Germany, Luxemburger Str. 10, 13353 Berlin, eduard.bergen@beuth-hochschule.de
[2] Department of Computer Science and Media, Beuth University of Applied Science Berlin, Germany, Luxemburger Str. 10, 13353 Berlin, stefan.edlich@beuth-hochschule.de

Currently, the system works as a prototype in a holistic system. The observation, extraction, and replay need multiple steps. In this paper, we pursue the observation of a subset in an LSA such as Apache Flink and emphasize obliviate techniques for a full automated observation process. Although we integrated our prototype in Apache Flink, not all components are optimal and efficient. Hence, response time and performance are not in the focus of this work. The result of our work is a system that replays certain input records on the developer's local machine. This allows a much faster investigation of the big data analytic job and brings us a step forward to find possible execution errors i.e. exceptions in minimal time.

This paper has following structure. First, we briefly present some background regarding observation in Section 2, classify the runtime environment and failure types, address related technologies, and describe their deficiencies. Section 3 is the main part and introduces our approach by filling the gap through isolated local debugging. Within Section 4 we look into our reference architecture, analyze behavior and show a use case. Finally, we conclude our work.

2 Background and Related Work

In recent years more and more LSAs such as Apache Flink [AF16] and Apache Spark [AS16] address fast parallel data processing using distributed system concepts. In distributed systems capabilities to digest and interpret the root cause of an error fail because they are too complex and large to handle failure analysis in production. Additionally, complex error investigation techniques like traces suffer from getting in touch with the faulty environment. Common approaches to discover a failure is through monitoring, remote debugging, data provenance [CAA11] or tracing the lineage [CWW00]. LSAs with limited control over the execution of a process and manipulation of data make it difficult to discover runtime errors. Runtime errors result from code that is syntactically correct but violates the semantics of a programming language.

```
public class SubText {
public static void main(String[] args) {
String printText = "SubText";
System.out.println(printText.substring(7,4));
}
}
```

List. 1: Source code sample *printText* with exceptional part inside the substring method

Typically, LSAs are complex unit interaction environments where messages need to exchange between the units correct, deterministic and fault-tolerant. Inside units, the environmental messages interact through interfaces. The model of an implemented interface contains consistent entities. Every message interacts with a fixed protocol and connected ports also known as channels. Independent failures do occur in units or channels. The challenge is to detect these failures. At the application level, a failure category consists of applicational failure models (AFM) and functional failure models (FFM).

Listing 1 represents a Java source code sample and shows a common scenario for a usual runtime error. For simplicity reasons, we choose this example. In the fourth line of Listing 1 the value of variable *printText* will produce a Java exception of type *StringIndexOutOf-BoundsException* because the value length of variable *printText* is seven and not eleven. The method *substring* tries to access a range of chars between char seven and eleven. The result is an abnormal program execution.

A program break inside LSAs is more than a classical abnormal program execution. Mostly it is time-consuming to finish a broken job if a program breaks in a big cluster setup, because of added additional communication overhead to a job. The common scenario is to identify parts of the cluster as vulnerable manually or semi-automated and wait until these parts restarted to a clean state. Other strategies are defining a model for the provenance of a data item or querying [Vi10].

After introducing some preliminaries, we will briefly recall known methodologies in debugging of distributed systems, specialized on record and replay (RnR). While there are many different RnR systems, proposes because of varied applications such as program debugging, online program analysis, fault tolerance, performance prediction and intrusion analysis probably the most important application is program debugging. The most common operation of debugging is bug reproduction, diagnosis and fixing. Program debugging consists of several imperative steps such as executing the job multiple times, pausing and investigating the state of variables and tracing [HT14].

It is also challenging to debug distributed software which often has complex structures [Wi12]. The reproduction of identical executed jobs requires the same runtime. LSAs like Apache Flink and Apache Spark have multiple signal controls, flows, application state, intermediate results and shared resources. For reproduction purpose the replication of an erroneous behavior of the used LSA [Fe15] becomes very hard because of first a different ordering of data tuples, second the execution resources and third the runtime environment.

There are plenty of tools for debugging distributed systems such as Inspector gadget [OR11] and Arthur [An13], Spark-BDD [Co15], Daphne [JYB11], and NEWT [LDY13]. These tools are specific and run inside the same process as the application itself. Normally the application uses the maximum level of the Java Virtual Machine (JVM) heap. Thus, it is not possible to attach remote debuggers. Furthermore the debugger is using an expensive heap space during the runtime, which often conflicts with the application heap space.

The more critical issue is the garbage collector itself. If the debugger runs in the same JVM and owing to the fact that a garbage collector is working during a Java program termination, in that case the debugger will also terminate. No further control is possible as to introduce a new active polling process. This leads to a bad design where further issues occur as a high CPU usage. Apart from internal observation, an external observation as descriptive debugging information used in DTrace [CSL04] needs detailed knowledge about the application runtime.

DiSL [Ma13] introduces the investigation process on the external and internal level. Basically DiSL uses a Java Virtual Machine Tool Interface (JVMTI) [OC16] implementation

of bytecode instrumentation. Furthermore the DiSL Java framework does bytecode transformation. In DiSL each compilation step sends and receives bytecode data between the native agent and the DiSL framework through a Transmission Control Protocol (TCP) socket connection. DiSL uses the combination of internal and external observation and decouples the bytecode transformation to a separate process. The communication is mainly socket based. If the compiler does not return the correct bytecode, the instrumentation of bytecode is a bottleneck. In general the result of the integrated faulty bytecode leads to an unpredictable JVM process termination.

If there is a need to get the fastest instrumentation of code transformation, implemented prototypes should be able to manage and solve any locking techniques during the instrumentation phase. Thus, the post debugging approach needs to process deterministic in the external observation way.

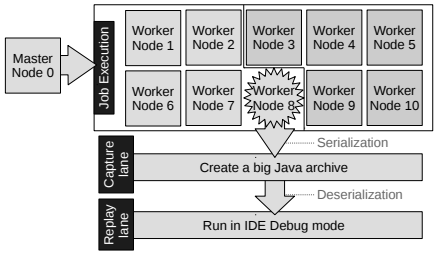

Fig. 1: An overview of our environment extraction and replay methodology

3 Reference Architecture

In this Section, we present the reference architecture according to the requirements described in Section 2. Figure 1 shows the architecture of the prototype. Our implementation of the reference architecture uses open source technologies. There is a plan to open source the reference architecture in several steps.

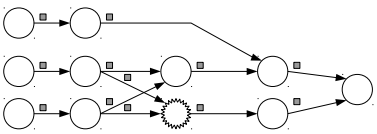

Fig. 2: Example of a job graph in Apache Flink

As depicted in Figure 1 the job fails in worker node number eight. Furthermore Figure 1 shows our reference architecture which consists of two parts: the *capture lane* and the *replay lane*. The *capture lane* allows users to get a subset of a defined workflow of UDFs using not only second order functions of operators such as map and reduce. During the *replay lane* a process in the reference architecture translates the recorded subset.

In Apache Flink there is a master and multiple worker nodes [Ba10]. The Apache Flink *JobManager* is the master node and the *TaskManagers* are the worker. In order to accomplish a record and replay workflow, the replay process requires complete information about the

executed job. First a data scientist ensures that a native agent attaches to a specific worker node. Every time a failure occurs in the attached mode within the cluster, a native agent creates a big Java archive on the filesystem.

This paper primarily focuses on our novel process for subset extraction and replay. Although other systems introspect extractions in the form of a whole job graph, our approach goes a step further by determining low-level characteristics to generate a small environment for later analysis. Additionally, systems exploring and tracing code via remote connections either suffer from getting the right place of a tuple value or need to generate source code to connect with precompiled operators.

While the investigated application handles specific application errors by itself, with small pieces of code we extract and serialize also the environment, additional runtime values, such as current input data of used operator. A native agent appends the recorded information about the environment to the Java archive file during the exception handling in the Java process.

Figure 3 shows the workflow of extraction and replay. Inside the web interface, archived jobs also contain the corresponding subset of the failed job. Hence, users are able to download the created archived jar-File for further local inspection.

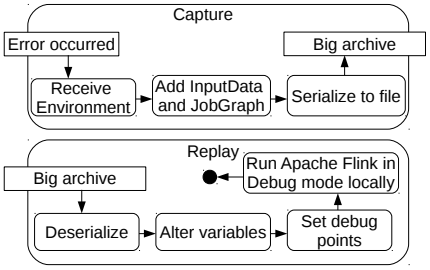

Fig. 3: An overview of extraction and replay activity in Apache Flink

The replay component takes in Figure 3 the archive, extracts the subset and starts a translation process in a local development environment. The translation process consists of three steps. First, the native agent serializes needed variables. Then updates of the configuration follow and at the end there is a need to use specifically overridden classes to inflate current input data for the used operator.

Before the replay process starts, data scientists execute run configurations on a homogeneous and small local machine. After the setup of workflow, users set debug points in specific lines in the source code. Since the current job executes, the attached debugger stops at user defined debug points. Data scientists usually use in debug session an Integrated Development Environment (IDE) to step through the call graph and observe the values of variables. Now that we have introduced our reference architecture, the following Section gives an impact about the usage and behavior of our reference architecture.

4 Experiences

In a RnR system such as our reference architecture that offers comprehensive code coverage, the challenge for data scientists quickly becomes figuring out what not to track. The main technique for filtering noisy data objects is to define transformation classes. In the record component as depicted in Figure 3, an implementation of a JVMTI native agent offers such a possibility.

During the attachment phase at the JVM, the native agent activates common capabilities to start receiving events such as class calls in a bytecode structure. Each time the JVM invokes a callback with the name *ClassFileLoadHook* the native agent executes a registered method. Accordingly, the callback method initiates a class file transformation and basically exchanges current bytecode with the desired bytecode of the transformation class.

In order to enable the class transformation, the native agent requires the desired transformation classes. Thus, the deployment of the native agent bundles compiled Java classes. We have used our reference architecture extensively to understand system behavior mainly in development environments. One issue [AFI14] in which our reference architecture could be useful is a long running Apache Flink job fail because of an *IndexOutOfBoundsException*. We have recreated this exception in a simple case as shown in Listing 1. Instead of using a constant value as in Listing 1, we used to input data from the production environment.

Figure 4 shows the replayed Java stack trace and expresses the proof of a working implementation of our reference architecture. Thus, data scientists begin root-causing as depicted in Figure 4 at line 94 of the WordCount-class, step-through the call graph until DataSourceTask-class and try to analyze why the program breaks in the invoke-method locally.

```
/usr/lib/jvm/java-7-oracle/bin/java ...
java.lang.StringIndexOutOfBoundsException: String index out of range: -4
    at java.lang.String.substring(String.java:1911)
    at org.apache.flink.client.testjar.WordCount$Tokenizer.flatMap(WordCount.java:94)
    at org.apache.flink.client.testjar.WordCount$Tokenizer.flatMap(WordCount.java:82)
    at org.apache.flink.runtime.operators.chaining.ChainedFlatMapDriver.collect(ChainedFlatMapDriver.java:79)
    at org.apache.flink.runtime.operators.DataSourceTask.invoke(DataSourceTask.java:180)
```

Fig. 4: A Java stack trace after replay with our reference architecture in Apache Flink

With the subset record and replay reference architecture, we are able to debug in a local environment and find the root cause. Compared to the time it takes to reproduce the failure in issue [AFI14], with our reference architecture, data scientists and developers are able to focus more on finding the root cause and solving the problem.

5 Conclusion

We have described our reference architecture, a new facility for debugging of LSAs through dynamic bytecode instrumentation. We have described and shown the primary features of our reference architecture, including details of record process and entry parts for further development. Also, we have demonstrated the use in root-causing a given problem. Although it is hard to design a single general approach of value extraction that replays all runtime

failure types immediately, it is still feasible to design application-oriented schemes for specific application scenarios.

6 Future Work

Our reference architecture provides a reliable and extensible foundation for further work to enhance our possibility to observe and post debug. We actively extend and update our reference architecture. Further development of components focuses first on automation in deployment and managing of agents, second in a plug-in system to address different observed program versions, and third in the visualization of subset reduced call graphs.

7 Acknowledgement

This work is generously funded by the Federal Ministry of Education and Research under the reference number 01IS14013D and was created as a part of the BBDC.berlin project.

References

[AF16] Apache Software Foundation: Flink Fast and reliable large-scale data processing engine, 2016, URL: http://flink.apache.org, visited on: 03/15/2016.

[AFI14] Apache Software Foundation: ASF JIRA: [FLINK-1000] Job fails because an IndexOutOfBoundsException, 2014, URL: https://issues.apache.org/jira/browse/FLINK-1000, visited on: 03/15/2016.

[An13] Ankur, D.; Zaharia, M.; Shenker, S.; Stoica, I.: Arthur: Rich Post-Facto Debugging for Production Analytics Applications, 2013, URL: http://ankurdave.com/dl/arthur-atc13.pdf, visited on: 03/15/2016.

[AS16] Apache Software Foundation: Spark Lightning-fast cluster computing, 2016, URL: http://spark.apache.org, visited on: 03/15/2016.

[Ba10] Battré, D.; Ewen, S.; Hueske, F.; Kao, O.; Markl, V.; Warneke, D.: Nephele/-PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC '10, ACM, Indianapolis, Indiana, USA, pp. 119–130, 2010, ISBN: 978-1-4503-0036-0.

[CAA11] Cheney, J.; Ahmed, A.; Acar, U. a.: Provenance As Dependency Analysis. Mathematical. Structures in Comp. Sci. 21/6, pp. 1301–1337, 2011, ISSN: 0960-1295.

[Co15] Condie, T.; Gulzar, M. A.; Interlandi, M.; Kim, M.; Millstein, T.; Tetali, S.; Yoo, S.: Spark-BDD: Debugging Big Data Applications. In: the 16th International Workshop on High Performance Transaction Systems (HPTS). 2015.

[CSL04] Cantrill, B. M.; Shapiro, M. W.; Leventhal, A. H.: Dynamic Instrumentation of Production Systems. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. ATEC '04, USENIX Association, Boston, MA, pp. 2–2, 2004.

[CWW00] Cui, Y.; Widom, J.; Wiener, J. L.: Tracing the Lineage of View Data in a Warehousing Environment. ACM Transactions on Database Systems 25/2, pp. 179–227, 2000, ISSN: 0362-5915.

[Fe15] Ferber, M.: FerbJmon Tools - Visualizing Thread Access on Java Objects using Lightweight Runtime Monitoring. In: Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers. Springer International Publishing, Cham, pp. 147–159, 2015, ISBN: 978-3-319-27308-2.

[HT14] Honarmand, N.; Torrellas, J.: Replay Debugging: Leveraging Record and Replay for Program Debugging. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. ISCA '14, IEEE Press, Minneapolis, Minnesota, USA, pp. 445–456, 2014, ISBN: 978-1-4799-4394-4.

[JYB11] Jagannath, V.; Yin, Z.; Budiu, M.: Monitoring and Debugging DryadLINQ Applications with Daphne. In: IPDPS Workshops. IEEE, pp. 1266–1273, 2011, ISBN: 978-1-61284-425-1.

[LDY13] Logothetis, D.; De, S.; Yocum, K.: Scalable Lineage Capture for Debugging DISC Analytics. In: Proceedings of the 4th Annual Symposium on Cloud Computing. SoCC '13, ACM, Santa Clara, California, 17:1–17:15, 2013, ISBN: 978-1-4503-2428-1.

[Ma13] Marek, L.; Zheng, Y.; Ansaloni, D.; Bulej, L.; Sarimbekov, A.; Binder, W.; Qi, Z.: Introduction to Dynamic Program Analysis with DiSL. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. ICPE '13, ACM, Prague, Czech Republic, pp. 429–430, 2013, ISBN: 978-1-4503-1636-1, visited on: 03/15/2016.

[OC16] Oracle Corporation: JVM Tool Interface Version 1.2.3, 2016, URL: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html, visited on: 03/15/2016.

[OR11] Olston, C.; Reed, B.: Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. PVLDB 4/12, pp. 1237–1248, 2011, visited on: 03/15/2016.

[Vi10] Vicknair, C.: Research Issues in Data Provenance. In: Proceedings of the 48th Annual Southeast Regional Conference. ACM SE '10, ACM, Oxford, Mississippi, 20:1–20:4, 2010, ISBN: 978-1-4503-0064-3.

[Wi12] Wieder, A.; Bhatotia, P.; Post, A.; Rodrigues, R.: Orchestrating the Deployment of Computations in the Cloud with Conductor. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12, USENIX Association, San Jose, CA, pp. 27–27, 2012.