December 2016

An All-in-One Debugging Approach: Java Debugging, Execution Visualization and Verification

Beinan Wang
Syracuse University

ABSTRACT

We devise a widely applicable debugging approach to deal with the prevailing issue that bugs cannot be precisely reproduced in nondeterministic complex concurrent programs. A distinct efficient record-and-playback mechanism is designed to record all the internal states of execution including intermediate results by injecting our own bytecode, which does not affect the source code, and, through a two-step data processing mechanism, these data will be aggregated, structured and parallel processed for the purpose of replay in high fidelity while keeping the overhead at a satisfactory level. Docker and Git are employed to create a clean environment such that the execution will be undertaken repeatedly with a maximum precision of reproducing bugs. In our development, several other forefront technologies, such as MongoDB, Spark and Node.js are utilized and smoothly integrated for easy implementation. Altogether, we develop a system for Java Debugging Execution Visualization and Verification (JDevv), a debugging tool for Java although our debugging approach can apply to other languages as well. JDevv also offers an aggregated and interactive visualization for the ease of users' code verification.

An All-in-One Debugging Approach: Java Debugging, Execution Visualization and Verification

By

Beinan Wang
B.S. Fudan University, 2004
M.S. Fudan University, 2007

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University

December 2016

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisor Prof. Roger Chen for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank the rest of my Ph.D. proposal committee: Prof. Qinru Qiu and Prof. Jim Fawcett, for their insightful comments and encouragement, but also for the hard question which motivated me to widen my research from various perspectives.

My sincere thanks also goes to the other three members of my oral defense committee: Prof. Thong Dang, Prof. Jian Tang, and Prof. Edmund Yu, for their time and great knowledges.

Special thanks to my wife Juan Du, without whom I cannot finish this dissertation research.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Today as software systems grow larger and more complex than ever debugging is becoming an unprecedentedly challenging job due to the nondeterministic nature [1]. Particularly in the case of the multithreaded programs, the behavior on each run for the same input can vary so much that it is almost impossible for bugs to be precisely reproduced and thus identified. In fact, the breakpoints based approach, perhaps the most commonly used debugging approach in both academia and industry, will be hardly usable in the face of such environments as it does not record data and thus does not provide replay for user's code verification.

Due to the lack of efficient debugging tools developers usually have to insert extra statements to print logs to monitor the execution. However, this is quite problematic when dealing with the concurrent programs with large data sets since the logging approach will likely fail to show accurate information of all the internal states during the execution as the values are easily changeable by other threads. Besides, manual inspection through log files in itself will be painfully difficult for developers.

Therefore, we devise a debugging approach with a powerful record-and-playback mechanism to collect as much as possible the information which will help deep- inspect bugs, including intermediate results, method invocations, values of variables, among other things that are relevant to all the internal states throughout the entire execution (altogether referred to Execution Data), thus making replay available in high fidelity of the bugs. However, such non-uniform Execution Data will not be suitable for replay unless properly structured or aggregated. That is why we devise a Two-Step Data Processing method to process these data while in the

meantime keep the overhead at an acceptable minimum level as the Execution Data can be overwhelming in both size and diversity.

Particularly we obtain the Execution Data through injecting our own bytecode at certain places largely related to the intermediate results or method invocation; based on that our debugger would be able to multithreaded-record the generated data for playback. In doing so there will be no need for users to provide any program specification or property and thus even beginners can very quickly utilize such a tool on increasingly demanding coding jobs.

Next comes the Two-Step Data Processing mechanism: firstly, we use an Object-Oriented Data Model (OODM) to group data regardless of whether they share the same data entry or not. Secondly the data will be parallel processed based on the promise theory [2] so as to obtain the Processed Data for the replay and visualization. Particularly we integrate several rather cutting-edge technologies, including MongoDB, Docker [3], Spark [4] and Server-Side JavaScript (Node.js) [5], to best achieve high performance, while keeping overhead at a very low level.

Unlike all other debuggers, ours will provide an interactive and combined visualization consisting of both the source code and the Processed Data, and users can very easily select any class of the code for inspection with just a quick mouse click. Moreover, users can bi-directionally navigate between the source code and the bug information without switching windows, thus providing users with an easy code verification environment. Besides, such visualization can be web-based and deployed to a Cloud to improve the efficiency of teamwork.

In addition, we employ Docker Container and Git [6] to offer a reusable clean running environment. By integrating them with the aforementioned record-and-playback mechanism,

all needed bug information can be efficiently stored for repeated running and monitoring with little impact on the source code to reproduce the bugs with the maximum precision.

Further, our debugging approach can be implemented through loosely coupled modules in JavaScript Object Notation (JSON) format so that any one of the modules can be smoothly replaced or modified. In other words, our approach allows a wide applicability for almost all the procedural languages including the dominant ones such as Java and C++ because of such a modularized architecture.

For the sake of description, we will present the design, implementation and evaluation of the above ideas just for Java programs and call this debugger Java Debugging, Execution Visualization and Verification, i.e., JDevv.

Chapter 2. Related Works

As aforementioned, neither breakpoints nor logs provide a satisfactory approach at dealing with complex concurrent programs. For example, Eclipse, like many other Integrated Development Environments (IDEs) that provide breakpoint-based debuggers, will impose heavy burden on users for multi-threaded programming.

As for the logs, the text files for complicated programs can often be excessively huge to be readable. Even though a number of log processing technologies such as [7] and [8] have been developed to address this file reading issue, the logs still cannot provide all the accurate information of a state during the execution of concurrent program since the values can be changed by other threads. For example as the Figure 2-1 shows, the value of “a”, which is supposed to be equal to “b” plus “c”, can be changed, hence resulting in a confusing log. Even if some lock mechanism is introduced to fix this issue, the deadlock risk and significantly more efforts will become the concerns. In addition, for source code which either does not exist or cannot be modified such as library code, the generated log will contain no such information at all.

```
001  val a = b + c;
002  log("a = b + c", a, b, c)
003
```

Figure 2-1 Logging Code

In such a situation where obtaining and monitoring internal states of an execution of a program is central to debugging, three approaches, yet for Java programs only, have been developed, each of which, however, has its own pitfall.

The first is to use a custom JVM instead of the standard JVM [9][10][11], which tends to result in considerable incompatibility with either source code or the other co-running JVM tools. The second is through bytecode instrumentation [12] with which the modified or injected bytecode triggers predefined events so as to avoid the shortcomings with the custom JVM and thus has been extensively employed in both academia (e.g. Java PathExplorer [13], Mop [14] and jMonitor [15]) and industry (e.g. Chronon [16] and Takipi [17]). However, Java PathExplorer, Mop and jMonitor all require extra code property specification with particular syntax, which in general is difficult for users to use. As for Chronon, it records certain data of execution behaviors but such data are short of further processing and consequently the generated view is usually not sufficient to undertake the code verification in practice. Moreover, Chronon is highly integrated with the Java editor of Eclipse which makes it difficult to support other programming languages while requiring additional effort to stay compatible with each updates of Eclipse. Although Takipi appears to require a rather small overhead, it only reacts to the exceptions and thus will not provide all the needed accurate information about the internal states. The last approach is to suspend the programs for recording as Java Platform Debugger Architecture (JPDA) [18] does. However, by doing so the performance degradation can be unacceptably severe when recording too much data. Therefore the tool based on JPDA such as Jive [19] is still mainly limited to teaching purpose although it provides a rather aggregated visualization. In addition, JPDA requires JVM to open an extra port which can cause a possible violation of security policy.

Chapter 3. JDevv System Design and Theory

3.1 Architecture

We design JDevv by dividing the systems into two sub-systems, the Cloud Server and the Database Server, and distinctly put the Compile-time module and the Run-time module on the Cloud Server to enhance cooperation among various modules of the system.

As you may recall that the common data model of our debugging approach can be implemented in JSON format such that any module can be updated and extended to enjoy a wide applicability. In the case of JDevv we make Java Parser replaceable by converting the parsed data, i.e., abstract syntax tree (AST) into JSON such that JDevv can support all other JVM languages such as Scala, Groovy and JRuby rather than Java alone.

Following the above descriptions, the overview of JDevv system workflow is shown in Figure 3-1. Firstly within the Compile-time module the uploaded source code will be parsed by the Java Language Parser [20] and thereafter be stored into the AST Database for visualization.

Unlike most previous approaches, we directly use standard Java compiler to compile the source code into bytecode for standard Java compiler's high compatibility and reliability. Note that the source code will not be instrumented in this module, meaning that JDevv will not inject bytecode into the syntax parsing results; instead, the Bytecode Injection will occur in the Class Loader to avoid compile error and its impact on the source code.

Next with the JIT Compiler we can perform the execution and obtain the Execution Data. Since the data we have now are not yet suitable for direct inspection, the Two-Step Data Processing mechanism is introduced within the Database Server to obtain the Processed Data. Specifically,

the Execution Data will be semi-structured by OODM first and then be further processed by the promise-based processing framework. Thereafter such Processed Data will be read via the Data Query Library to be correspondingly selected by the user for easy code verification. With an Html5 Visualization module JDevv will produce a clickable sequence diagram which combines

both a source code view and all the needed bugs' information for the replay and visualization.

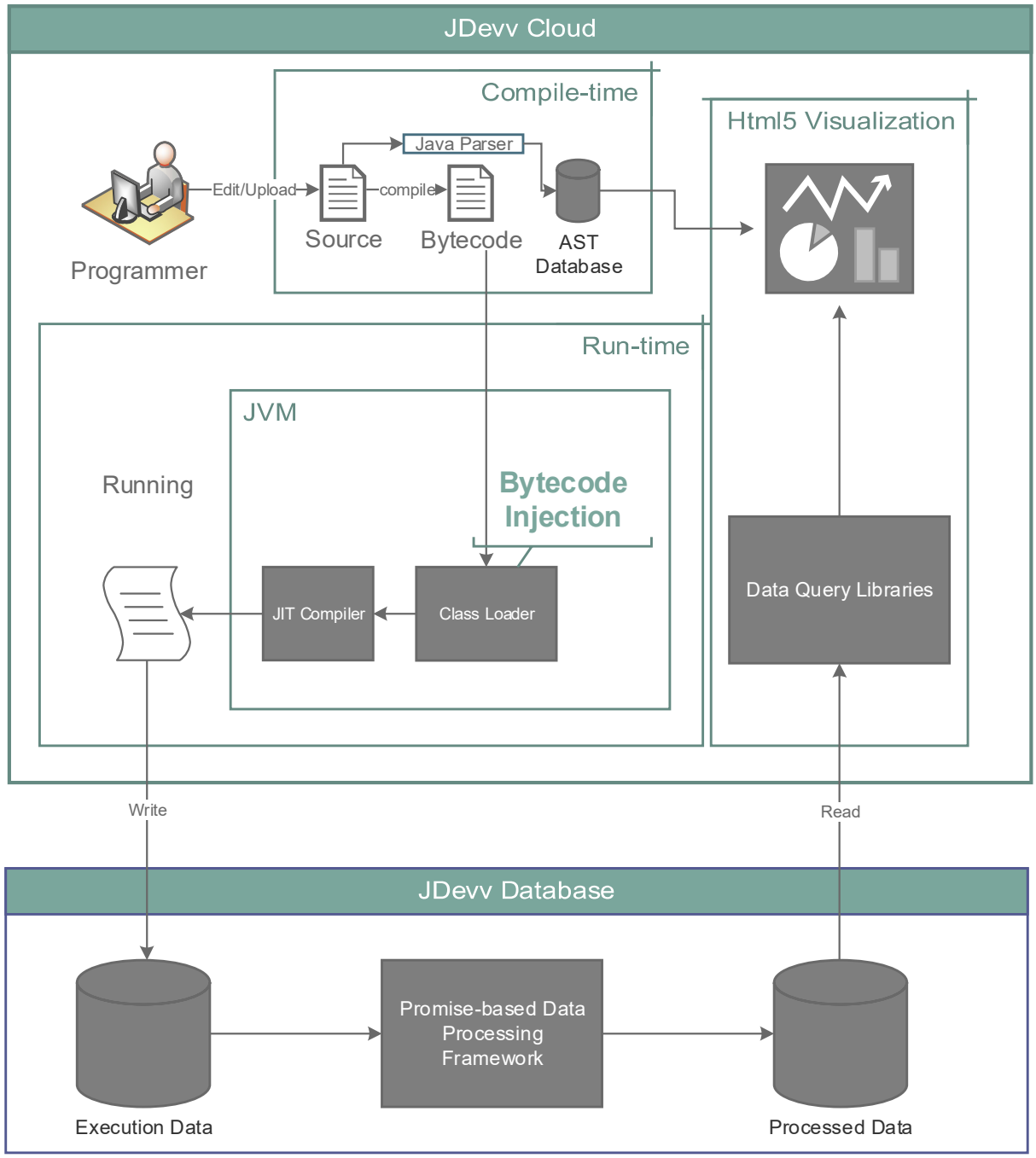

Figure 3-1 JDevv Overview

3.2 Technologies

In this section we will discuss in detail how we use these cutting-edge technologies.

3.2.1 Docker Containers and Git

In order to provide a reusable clean environment for debugging, we choose Docker Container and Git for JDevv. Specifically, Docker Container can work like Virtual Machine while not requiring very high overhead as most of the VMs do, and through Git a separate folder or data entry will be created in the container every time when the source code is uploaded in a Zip file or Git URL. With these two combined every execution will be independent as if the code has been isolated for a brand new running. Besides, Git allows great convenience in finding bugs by comparing different versions when modification on the source code is involved. In other words, not only can users repeatedly run the original source code but also run the modified code as often as needed to undertake the testing and monitoring.

Docker is a technology that provides a low-cost software container, which wraps up the source code and its dependencies so as to allow users to build, ship and run their applications on any physical server. That means, Docker container is a rather isolated virtualization platform like a virtual machine (VM) while performs even better in some respects. Therefore, we chose Docker container for its advantages in meeting our special requirements as follows:

- Sharing the same host operating system (OS) kernel; this allows for more than one container to be created in a physical machine and thus more than one debugging process run in parallel on the same server, meaning a significantly improved utilization

rate of the physical servers. More importantly, such sharing means an instant restart of the container and therefore a very low overhead.

- Easy shipment; this will benefit teamwork greatly where multiple containers are involved.
- Isolated and independent container; this provides a clean environment whenever a debugging process is initiated and thus no adverse effect.

Figure 3-2 depicts how multiple Docker containers will work via Git especially in a teamwork setting. The developers will first submit their code to the repository for testers, who will fetch the code through the JDevv web interface (Docker 1) and start testing, while typically more than one testing process with varied test cases or user input are taking place. Then if the testers initiate two tests the two new and independent containers will be provided as Docker 2 and Docker 3, and both of their data will be stored in the same container (Docker 4). Because of the isolation and independence of Docker 2 and Docker 3 the testers can quickly identify bugs and report them to the developers, who will in turn acknowledge the source of bugs by simply looking at the corresponding Docker. Besides, they share the data with all the executions inside one container, i.e., Docker 4 in this case, and therefore the developers can repeatedly play back the identified bug in a time-efficient manner.

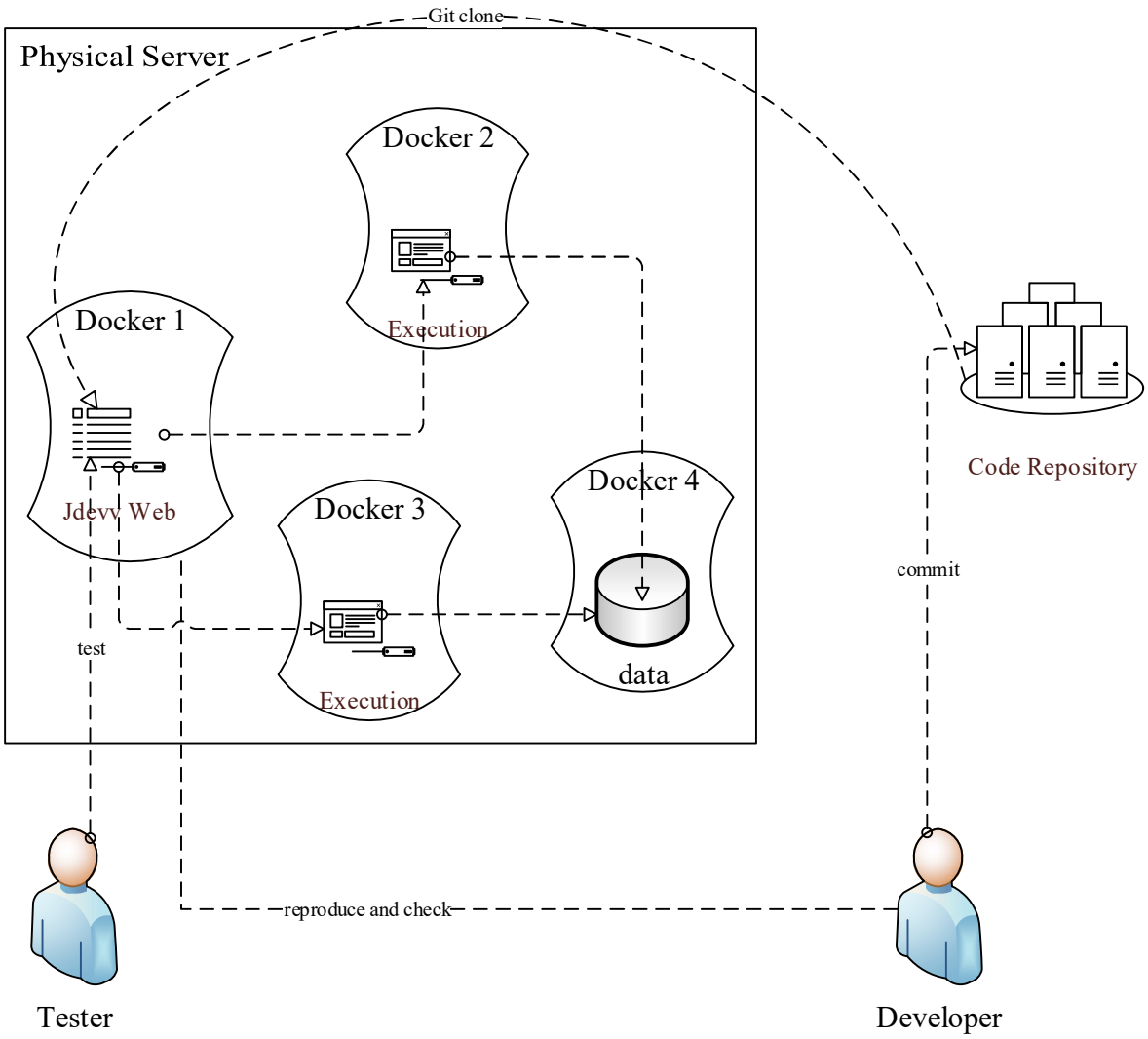

Figure 3-2 Multiple Docker Working Mechanism

3.2.2 Java Agent

The traditional approach of bytecode injection is just to provide a user-defined class loader to simply override the standard class loader of JVM. However, this approach cannot inject bytecode into the classes in the JVM bootstrap libraries such as the standard libraries provided by JDK. In addition, this approach may conflict with many of the popular application servers

(e.g., Tomcat and JBoss), because these application servers usually load classes by their own class loader. To solve these problems, JDevv involves Java Agent technologies instead of the user-defined class loader.

Java Agent[1] is a little known built-in interface of JVM through which we can implement our Bytecode Injection without involving non-standard third-party technology. This seamless compatibility enables our Bytecode Injection to record every operation of JVM without missing anything and thus a fine-grained instrumentation that might not be available otherwise. For example, JDevv is able to inject bytecode into most classes in the Standard Library and local methods where most other debugging approaches will fail.

More than that, the Bytecode Injection can even happen in the loading of the users' bytecode through Java Agent such that there will be no modification on either the source code (i.e., *.java files) or on the users' bytecode (i.e., *.class files) since the Bytecode Injection would merely modify the bytecode loaded in memory and do no writing back to the *.class files. In other words, the source code would remain as it originally was throughout the debugging process and the original bytecode will remain transparent.

3.2.3 MongoDB

Most logging and monitoring tools are using text file to store the logged data. However, it is very difficult to store values with complicated data structure such as embedded objects and arrays. Furthermore, there is no efficient approach to perform a query (e.g., data filtering and

[1] Java Agent is introduced in Java Development Kit 5. See the definition of package 'java.lang.instrument' https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

grouping) in a big text file. Some tools use RDBMS as an alternative. RDBMS can append indexes on the data which can highly improve the query efficiency while requiring a strict schema for the data stored. Therefore, JDevv uses a NoSQL database to store the semi-structured data (i.e., Execution Data and the data processing results).

MongoDB is a well-recognized NoSQL database in industry to handle large dataset and also a Document Oriented Database that stores data in form of document. Specifically, MongoDB uses BSON, a binary JSON-like format to store and transfer the documents and in doing so MongoDB provides a schemaless storage of semi-structured data.

Apart from that JDevv makes use of some other MongoDB's favorable features [6] as below:

- Supporting auto sharding to store data across multiple database instances to allow JDevv to record more data;
- Supporting basic aggregation operations such as Pipeline and Map-reduce to analyze data;
- Supporting index to improve the query efficiency; and
- Supporting server-side JavaScript for an easy implementation.

3.2.4 JavaScript

We mostly employ JavaScript (except for the Java Parser and Bytecode Injection which are implemented by Scala and Java because the existing functions can be employed directly) to

execute both the Cloud Server and the Database Server for its easy implementation, compatibility with browsers, high performance on server side and the newly emerged database side implementation. Firstly the process of constructing web applications can be simplified by using a new runtime environment called Node.js [21]. Secondly, the module loading system of Node.js can allow for the use of functions of all the other dependent modules such that data can be shared among browsers and databases. Thirdly although JavaScript has been viewed as a client programming language for a long time it actually now can be used on server side as well with Node.js for high performance applications [5]. Last but not the least, this (i.e., the JDevv development effort) is perhaps the first time JavaScript is used for a database server as such utilization has not been commonly recognized until very recently when MongoDB was introduced. For all these reasons the Cloud Server is implemented using server-side Node.js and the Database Server was implemented using JavaScript.

Specifically, each Node.js module is a single JavaScript file containing a group of functions and any one of such modules can "depend on" other modules by 'require' statements in JavaScript source file, which means, if any module was loaded by Node.js, all the modules it depends on will be loaded by Node.js as well, such that they will share all functions even though they are separate files.

However, browser and database cannot be modularized or share functions by modules loading as Node.js does unless we devise a mechanism to enable them to. Figure 3-3 shows how this mechanism works in detail.

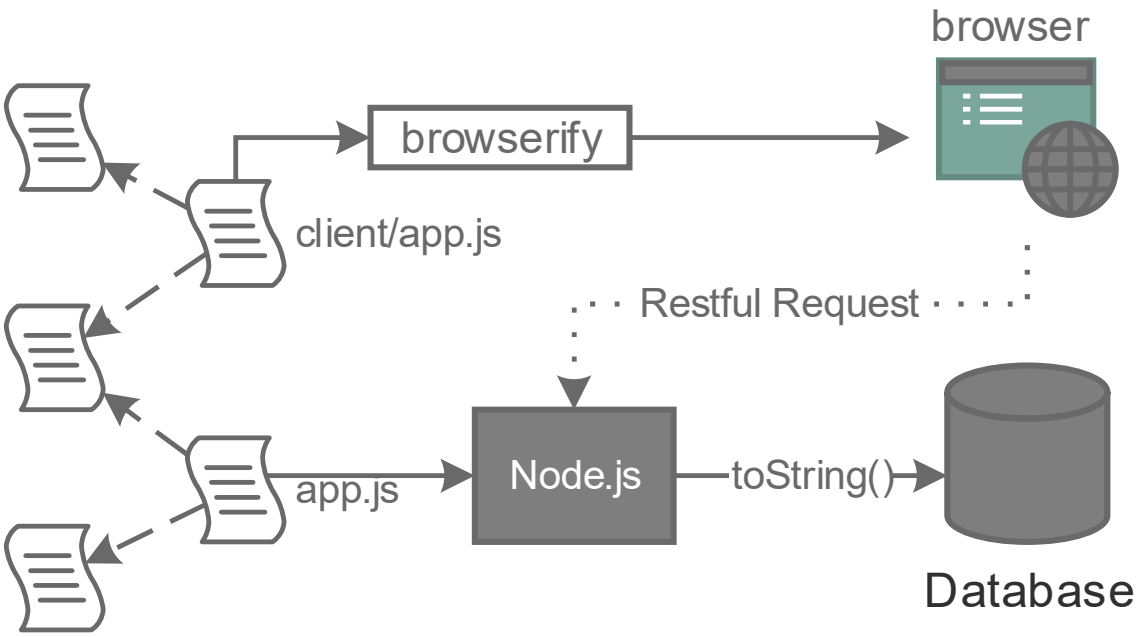

Figure 3-3 Sharing JavaScript code among browser, Node.js and database

First, for the browser we set an entry point to load all the modules it depends on and by loading from this 'client/app.js' the "browserify" will be able to pack up the loaded code and send them to the Browser to read. Similarly, we set another entry point for Node.js to load from "app.js" and then send the functions in a string format to the Database. Also we write the Database functions in Node.js module to allow for a version control system (e.g., Git) and the JavaScript code will not need to be stored in MongoDB as usual. In other words, when we need the Database to execute a JavaScript function, we simply ask Node.js to send the corresponding function in a string format to MongoDB. In addition, the browser will rather easily obtain the needed data for the final visualization through a simple quest. By so doing, not only do we reduce the repetition of the implementation but also unify all the JavaScript code.

Another interesting feature of our use of JavaScript is ‘event loop’, which makes JavaScript never block. In the implementation of JDevv, because the database queries are performed via both events and callbacks we can utilize the ‘event loop’ to achieve the effect such that when the JDevv is waiting for a query to finish it can handle other queries or events in the meantime.

Chapter 4. JDevv Debugging Environment

JDevv contains a full featured cloud-side debugging environment to allow users to debug their program on the JDevv Server. JDevv provides a user friendly web interface to allow user to uploading, retrieving, debugging and verifying their programs via a web browser.

All the modules of JDevv Debugging Environment are loosely coupled by passing data in JSON format so each one of them could be replaced or modified with as minimum impact on others as effortlessly to be updated and extended in the future. Particularly we wrapped the Java Parser in Compile-time Module to convert the produced syntax tree into JSON format so the Java Parser could be replaced as well. In other words, JDevv can support all the other JVM programming languages, such as Scala, Groovy and JRuby, other than merely Java by simply replacing the Java Parser. In the case of C/C++ we could also make it work by replacing both the Compile-time module and Run-time module.

As for the implementation of JDevv, we mostly employed JavaScript for both server-side and client side of JDevv Debugging Environment except for the Java Parser and Bytecode Injection both of which are implemented by Scala and Java.

4.1 Compile-time Module

4.1.1 Code Preparing

Users can upload their project in a Zip file or retrieve the code from a Git (the most popular source code management tool) URL. For each uploading or retrieving, JDevv will create a separate folder and also a separate data entry in the database. And then this separate folder will be added to a Docker container as a data volume (this procedure is just like mount a new hard disk into a Linux system). By doing so, the users' programs can be run in a separate Docker container. If users' programs write some data to the disk, the data will be also in the data volume. In addition, data volumes are independent of the Docker container's lifecycle. That means the data volumes will not be deleted when container is removed, terminated or crashed. So all the code and data can found again after the debugging. The data entry in the database is used by our web debugging interface to help user track all the debugging process.

Therefore, users can debug their programs with different versions at the same time. This feature is helpful for the users to find the bugs which are caused by the modification on a particular version.

4.1.2 Code Compiling and Parsing

For compatibility and reliability, we are using the standard Java compiler to compile users' source code; furthermore, JDevv does not instrument on users' source code so that no compile error will be involved by JDevv.

All the source code will be parsed by a Java Language Parser[20] which currently supports up to Java 8 . Based on the parsing result, for each file a syntax tree is generated in JSON format, which are stored in the MongoDB for the future use of visualization module.

It's important to notice that no bytecode injection is done in this module. Furthermore, JDevv does not inject bytecode based on the syntax parsing result. In order to better compatibility, we inject bytecode based on the parsing of the bytecode from users' code during the run-time.

And the entire users' program will be compiled within a Docker container. In order to provide greater flexibility, multi Java compiler version is supported. Because the Docker provides Virtual Machine like containers while without high cost as most of the VM do, which is allowed to maintain its own Java compiler installation (The java compiler is usually contained in Java Development Kit (JDK)).

4.2 Run-time Module

JDevv would instrument users' code through the Bytecode Injection during the class loading by JVM to obtain all the needed run-time information on bugs, i.e., Execution Data, which includes the method invocation, intermediate results and other things.

Figure 4-1 depicts the process of when JDevv instrument users' bytecode. The JVM invokes the premain method in the JDevv Agent before it loads and runs the user program. The JDevv agent is deployed as a JAR file which includes the implementation of the interface "ClassTransformer" in order to transform the bytecode in users' classes. According to the configuration of users' program, the JVM may use the System Class Loader or User Class Loader

to load users' classes. After the loading, the bytecode which is in an array of bytes will be passed to the "ClassTransformer". The transformation occurs before the class is defined (or we can say used) by the JVM. Therefore, the JVM will execute the transformed bytecode during the runtime.

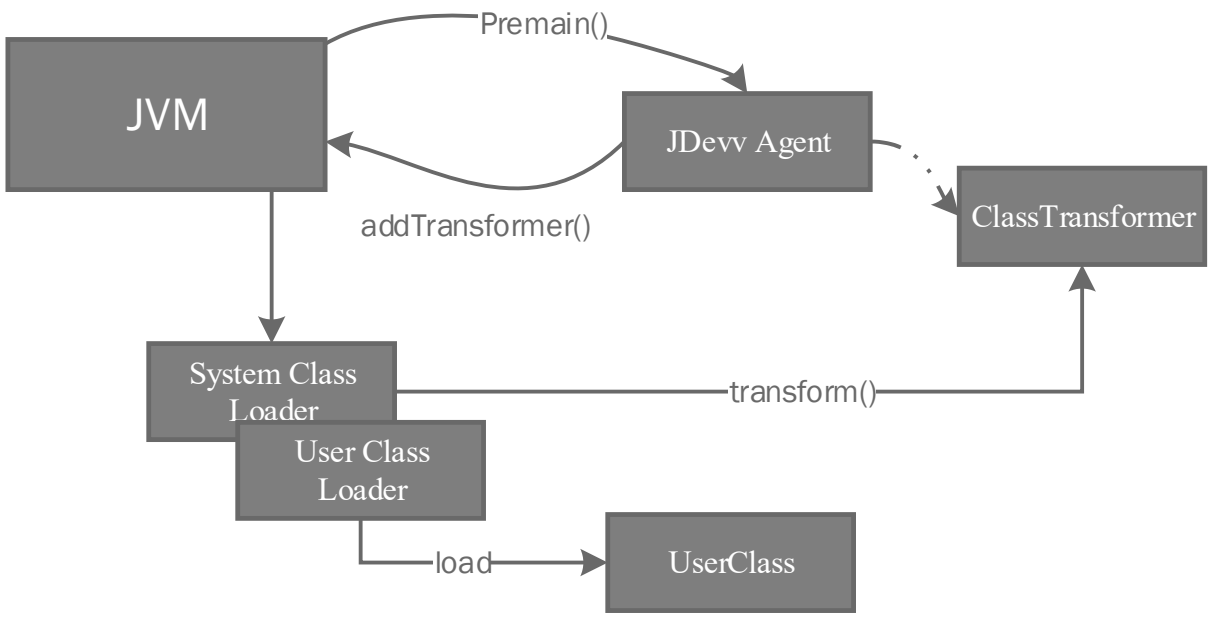

Figure 4-1 JDevv Instrumentation Process

4.2.1 Java Bytecode Instruction

Java bytecode is the instruction set of the JVM. All the source code of Java as well as other JVM languages are compiled into Java bytecode. Then the bytecode is executed by JVM. The JVMs of all of the platform such as Windows, Linux and Solaris are using the same instruction set, or we can say the same bytecode. Taking the advantage of this, JDevv's instrumentation is platform independent.

Each bytecode instruction is composed of one or two bytes for the opcode such is (ILOAD, INC, IADD), along with the parameters (optional, which may take zero or more bytes).

Java Bytecode is a statically typed and accordingly many instructions have a prefix character to refer to the type of operands. **Error! Reference source not found.** lists all of the prefixes of the types supported by JVM 7 and 8. For example, "iadd" will add two integers together. The requirement of the type of "iadd" is strict, which requires that two integer values have to be pushed as the top values of the operand stack. If the top two values in the operand stack is other types such as long or double, the execution will be crashed. However, in the bytecode level, the type information of the current values in the operand stack is not available. The operand stack can only maintain the values of type we listed in **Error! Reference source not found.**. As for the value of an object, the operand stack can only hold the reference of that object rather than the real value of that object.

Table 4-1 Bytecode types and type prefix

Prefix	I	l	s	b	c	f	d	z	a
Type	int	long	short	byte	char	float	double	bool	ref

Error! Reference source not found. lists some of the frequently used bytecode instructions in groups, which will be discussed in the following sections.

Table 4-2 Grouped Frequently Used Java Bytecode

Group	Opcode
Method Invocation	invokestatic, invokeinterface, invokespecial
Operand Stack	dup, pop
Object and Field	new, getfield, putfield
Local Variables and Constant Values	istore, aload, const_x
Arithmetic, Logic and Type conversion	iadd, imul, cmpl, d2i
Control	ifeq, goto

4.2.2 Method Invocation

As for the method invocation, each Java Virtual Machine thread maintains a private Java Virtual Machine stack as Figure 4-2 shows. JVM generates a method frame each time when a method is invoked, meanwhile frames are pushed to the Java Virtual Machine stack of the thread creating the frame. Therefore, the frame at the top of the Java Virtual Machine stack is the frame of the current method invocation which is just executed by a thread. In addition, this frame will be popped when the method invocation is terminated whatever the termination is normal or abrupt (if the method terminated by an uncaught exception). Moreover, the popped frame will be destroyed immediately, which means the information such as local variable and operand stack in the frame cannot be recovered if we don't log them before the method invocation terminates.

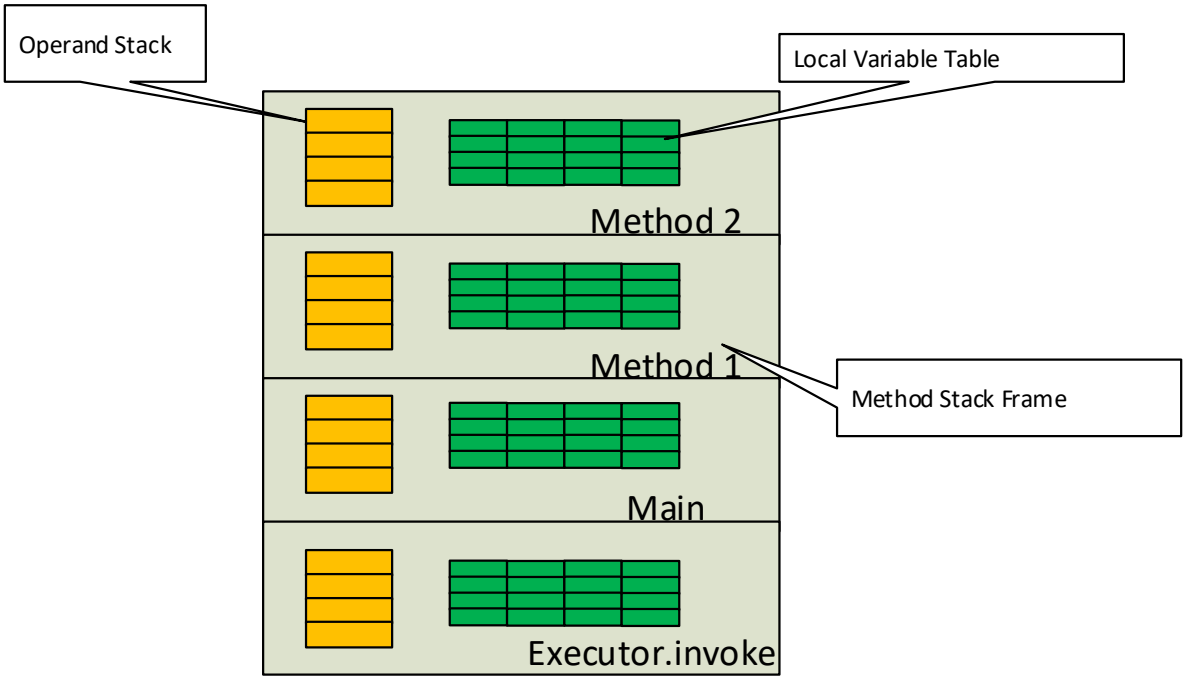

Figure 4-2 Java Virtual Machine Stack

As we mentioned, each frame maintains an Operand Stack to store all the intermediate results, and the most recently retrieved data are actually stored in the top element of the Operand Stack. For example, “iadd” will add two integers together. During the execution, both of the two integer values are popped from the operand stack. And then their sum is pushed back onto the operand stack after the execution. Therefore, all the intermediate results including method return value, variable assignment and field access can be monitored via Operand Stack by the appropriate bytecode injection. Actually it is worth mentioning that Operand Stack has yet not been widely acknowledged and such use of Operand Stack for retrieving those needed data is possibly first-of-its-kind. Particularly upon the use of a local mechanism such as Operand Stack the return value retrieved would be the accurate information rather than the modified value because of other threads while involving no deadlock risks.

It's notable that the maximum depth of the operand stack is a fixed number which is calculated at compile time and is stored in the bytecode of that method. Therefore, if the code we instrumented to users' bytecode will generate extra intermediate results to the operand stack, we have to update the maximum depth in the bytecode.

4.2.2.1 Static Method Invocation

Figure 4-3 lists the code of a standard quicksort method. This method takes three parameters "int[] v1", "int b" and "int e". "v1" is an array of integers and "b" and "e" are just simple integer values.

```
001     public static void quick_sort(int[] v1, int b, int e) {
002         if (b == e) return;
003         …
004         for (int k = b + 1; k <= e; k++)
005         {…}
006         v1[l] = pivot;
007         if ((l - b > 1)) quick_sort(v1, b, l - 1);
009         if ((e - u >1)) quick_sort(v1, u + 1, e);
010         return;
011     }
```

Figure 4-3 QuickSort Example

Figure 4-4 shows the equivalent source code how JDevv instrument this method to log the execution data about how this method invocation is entering.

For a static method, if three arguments are passed in, they are stored in the exactly same manner of the local variables which are numbered 0 through 2 of the frame created for this method invocation.

```
001     public static void quick_sort(int[] v1, int b, int e) {
002         long invo_id = JBDTrace.traceStaticMethodEnter("Quick#quick_sort");
003         JBDTrace.traceMethodArg(invo_id, "Quick#quick_sort", 0, v1);
004         JBDTrace.traceMethodArg(invo_id, "Quick#quick_sort", 1, b);
005         JBDTrace.traceMethodArg(invo_id, "Quick#quick_sort", 2, e);
006         if (b == e) return;
007         …
009         for (int k = b + 1; k <= e; k++)
010         {…}
011         v1[l] = pivot;
012         if ((l - b > 1)) quick_sort(v1, b, l - 1);
013         if ((e - u >1)) quick_sort(v1, u + 1, e);
014         return;
015     }
```

Figure 4-4 Instrumentation for Method Enter

Figure 4-5 shows the equivalent source code how JDevv instrument this method to log the execution data about how this method invocation is returning. Before the method invocation, a "tmp_invo_id" is recorded for tracking the method invocation, and then JDevv also use the value of "tmp_invo_id" to track the method return. By doing so, for each method invocation, we generate a pair of data entries, "method_invoke" and "method_return". In addition, we can use the difference of the created time of these two entries to obtain the runtime of this method invocation.

```
001  public static void quick_sort(int[] v1, int b, int e) {
002        long invo_id = JBDTrace.traceStaticMethodEnter("Quick#quick_sort");
003        …
004        if ((l - b > 1)) {
005          long tmp_invo_id = JBDTrace.traceMethodInvocation("Quick#quick_sort",
     invo_id);
006          quick_sort(v1, b, l - 1);
007          JBDTrace.traceMethodReturn("Quick#quick_sort", "_$void$_", invo_id,
     tmp_invo_id);
009        }
010        ….
011     }
```

Figure 4-5 Instrumentation for Tracing Method Invocation and Method Return

4.2.2.2 Method Return Value

Figure 4-6 shows how JDevv monitors the return value of a method invocation. As was the case with other bytecode instructions, the return value would be at the top of the Operand Stack after the method invocation, JDevv would inject a "DUP" instruction to copy this value and push it back to the Operand Stack. Consequently, two values, the original one and the duplicate would both be stored in the Operand Stack and thus the value would always be there, intact, for the execution.

And then JDevv would inject three instructions to push three values: "method_name", "parent_invocation_id" and "invocation_id" to the Operand Stack. After these values were

ready, JDevv would inject “Invoke JBDTrace.traceMethodReturn()”, which was a static method with four parameters to record all the needed information and pop the top 4 values in the Operand Stack. Thereafter “Save to A” would be executed as it should have been as the original return value was still on the top of the Operand Stack after the invocation of “JBDTrace.traceMethodReturn ()”.

Original	Instrumented
Load b into stack Load c into stack Invoke Method add Save to A	Load b into stack Load c into stack *…//trace method invocation here* Invoke Method add **Dup *//if return type is long or double, using dup2 here*** **Push method_name** **Push parent_invocation_id** **Push invoication_id** **Invoke JBDTrace.traceMethodReturn()** Save to A

Figure 4-6 Instrumentation for Tracing Method Return Value

4.2.2.3 Normal Method Invocation

The normal method invocation means the method JVM invoked is belong to an object. If the object is an instance of a sub-class which is inherited from some super-class, during the runtime JVM has to determine the run-time type of the object. Based on this run-time type, which implementation of the method will be invoked can be decided at run-time. In C++, we are using virtual for the methods which you may want to override in the sub-classes. However, in java, almost all the normal methods are virtual. In bytecode level, such an invocation is implemented using the "invokevirtual" instruction.

As with the static method invocation, the normal method invocation could be monitored by generated two records, "method_enter" and "method_return". The only difference is that for the normal method invocation, the reference of the owner has to be recorded.

4.2.2.4 Special Method Invocation

Special method invocation means the JVM invoke a method by "invokespecical" instruction. Currently, JVM only use this instruction to invoke the constructor method which is a special type of method that is used to initialize the object. The constructor method is invoked only once for each object at the time of object creation. In bytecode level, every constructor method is written as a normal instance method that has the special name <init>. This name is not a valid method name for all the JVM languages, it cannot be written directly in users' source code. Accordingly, it can be only generated by the compiler. In addition, the constructor

methods can only be invoked by the “invokespecial” instruction. JDevv instruments constructor method similarly as the normal instance method we discussed in the previous section.

Another kind of constructor is called static block, which is mostly used for initialize the default values of static variables. This kind of code will be compiled as static method with a special name <clinit> which will be invoked when the class or the interface is loaded in the memory. A class may contain more than one static block, which will be compiled in the same order of which they have been written in the source code. In bytecode level, the <clinit> they are never invoked directly from any Java Virtual Machine instruction, but are invoked only indirectly as part of the class initialization process. The name <clinit> is supplied by a compiler. Because the name <clinit> is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Class and interface initialization methods are invoked implicitly by the Java Virtual Machine; Accordingly, JDevv instrument <clinit> method just like a normal static method.

4.2.3 Java Multi-Threaded Programming

For an application, a single Java Virtual Machine is able to support many threads running at the same time. Each Java Virtual Machine thread maintains its own program counter register (PC). These threads share the memory (heap) of the JVM. That is to say, some of the values may be shared among the threads. In the hardware layer, there threads may be supported by multiple processors (CPU cores). Accordingly, more than one instruction may be executed in parallel. Therefore, multiple threads are very likely to operate the same data at the same time. Even for

a single processor environment, the execution of a multiple threaded program could still be interweaved by time-slicing. Either way would greatly improve the difficulty of debugging.

All the JVM threads are the instance of the Thread class. When a Java Virtual Machine starts a users' application, a thread (main thread) will be started to invoke the static main method in users' class. In this main thread, other new threads (user's thread) can be created by users' code. There are only two approaches to create a new thread. One is to declare a subclass to inherit class Thread. The other is to declare a subclass that implements the Runnable interface. In either way, the subclass has to implement users' business logic in the "run" method which is an abstract method in class Thread, which can be monitored just like a normal method invocation.

During the execution of a thread, we are able to obtain the thread Id of the current thread by calling Thread.currentThread().getId() at any place and any time. In addition, each thread can maintain its own static variables via ThreadLocal class, which means the ThreadLocal will initialize an independently copy for each thread. By taking the advantage of ThreadLocal, we can generate an increasing unique identifiers locally to each thread. For instance, the "invocation_id" we used in the method invocation is just such a ThreadLocal variable.

It is notable that the deprecated methods in class Thread such as stop() and suspend() will not be supported by our system. Because these methods will be removed in the near future, and should not be used any more.

4.2.4 Field Accessing

In order to track the field accessing, JDevv injects bytecode in the two aspects below.

Firstly, for each field in the class, a pair of getter and setter method will be generated as Figure 4-7 shows. For the fieldA in Clazz (line 003), an instance of Locker will be generated in initialized as a new field of this class (line 005). The Locker also maintains the version information of the fieldA. And then based on the type of fieldA, a getter method (line 006 - 009) and a setter method (line 010 - 012) are generated to tracking both the value and the version of the fieldA. Thus, a synchronized block associated with the instance of Locker is generated to guarantee the value and version are matched. In the synchronized block (line 007 and 011), all the information of the field including the field name, class name, owner reference as well as the value and version will be recorded.

```
001  public class Clazz{
002    //original code
003    int fieldA;
004    //generated code
005    Locker __locker_fieldA = new Locker();
006    int __get_fieldA(long parentInvocation_id){
007      sychronized(__locker_fieldA){…}
009    }
010    void __set_fieldA(int newValueOfFieldA, long parentInvocation_id){
011      sychronized(__locker_fieldA){…}
012    }
```

Figure 4-7 Field Getter and Setter Generation

Secondly, all the field accessing instructions (i.e. PutField and GetField) are replaced by the method invocation of the getter and setter method generated above. And also the parent "invocation id" which is generated in the method invocation section would be passed into the getter and setter to make them know who is calling.

4.2.5 Array Element Accessing

As with the field accessing, a pair of getter and setter functions is generated for the accessing of array elements. However, since array is not a class in java, we cannot generate the getter and setter functions as member methods of a class. Therefore, a pair of static getter and setter functions are generated to track the read and write on the element of arrays. These two static functions take two extra parameters for the reference of the array and the index of the element.

It is notable that all the elements of an array will share a single instance of Locker, which means the mutation on the elements will be counted just by one version counter. By doing so, we do not need to maintain a huge number of lockers for the large array, and also the results are more readable for humans.

4.2.6 Thread Synchronization

Java language provide two ways to implement thread synchronization: synchronized block and synchronized method.

4.2.6.1 Synchronized Block

In Java, a synchronized block is enclosed by the “synchronized” keyword. A synchronized block has to be synchronized on some object provided as a parameter of the “synchronized” keyword. Meanwhile, every object has an intrinsic lock associated with it. Therefore, if a thread that needs exclusive visit to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it finishes the read or write operation. So all synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Accordingly, there are two instructions monitorenter and monitorexit for synchronized block in Java bytecode. For each synchronized block, a pair of monitorenter and monitorexit is generated by the compiler to control the lock mechanism on an object among multiple threads. For example, see the Java code below:

```
001     public static void foo(Object v) {
002         sychronized(v){
003         …
004
005         }
006
007         return;
009     }
```

The code above will be compile to the bytecode instructions below. Line 002 and 007 load the reference of “v” to the operand stack, and then line 003 and 009 take it out as a locker. Therefore, this procedure can be easily tracked just like a method invocation.

```
001     public static void foo(Object v) {
002         aload_0         ; load v
003         monitorenter     ; lock object in local variable 0, the first
004 parameter
005
006          …
007         aload_0           ; load v
009         monitorexit       ; release the lock in local variable 0
010         return
011     }
```

4.2.6.2 Synchronized Method

In Java, synchronized method is used to synchronize the entire method. A synchronized instance method in Java is synchronized on the object (owner) owning the method. Only one thread can enter inside a synchronized instance method for one object. If there is more than one object, then there is only one thread per instance.

Compared with normal Java methods, there is no extra bytecode generated by the compiler for synchronized methods. Therefore, as for the execution data of a synchronized method, JDevv just uses three kind of message types, method_invoke, method_enter and method_return to record.

4.2.7 Wait and Notify

When wait is invoked, the current thread releases the lock and suspends. Also when a thread invokes the wait and notify method of an object ‘a’, it must own the lock of ‘a’. In JDevv, these

two indeed are treated just like normal method invocation. The status of the threads will be calculated in the data processing and visualization.

4.2.8 Java Exception

All Java exception are the instance of the classes which are the sub-class of the java.lang.Exception. When a Java exception is throw out, the normal execution of the program is disrupted and the current method invocation terminates if the exception is not catch.

4.2.8.1 Programmatically exception

When an exception is thrown by users' program explicitly, the "athrow" instruction will be generated by the compiler. "athrow" instruction will throw the top value which has to be a reference to an instance (object) of Java Exception in the operand stack. Accordingly, the "athrow" can be tracked just like a method invocation.

4.2.8.2 Other exception

During the execution, there are varied exceptions thrown by the Java Virtual Machine instructions when they detect an abnormal condition. For example, if a value is divided by zero or the array accessing is out of boundary, the corresponded exception will be thrown out. For these kind of exceptions, if JDevv append a try-catch block for every instruction which may throw an exception, it will generate too much bytecode, and also may cause performance issues. Therefore, JDevv do not do any instrumentation on this kind of exceptions. Instead, JDevv detect these exceptions during the data processing.

4.2.8.3 JVM system error

JVM system errors are mainly thrown by the JVM environment rather than users' code. For example, if the JVM runs out of the memory, then an error will arise. Generally, the execution of users' program cannot be recovered from an error, which means the execution will be crashed. In most cases, the errors can rarely be debugged and located to users' code by the current debugging tools including our JDevv. However, the users can still inspect the data recorded by JDevv to indirectly identify the cause of the problem.

However, for some special cases, if the error is thrown by users' code explicitly by an "athrow", then it can be tracked similarly just like an exception.

4.3 Data Type Deduction Algorithm

4.3.1 Algorithm

Because JVM generates a frame each time a method is invoked and each frame maintains an Operand Stack to store the intermediate results, the most recently retrieved data are actually stored in the top element of the Operand Stack. However, since no type information associated with data is stored in the Operand Stack, Type Inference Algorithm is then proposed to infer the types of those intermediate results.

The Algorithm 1 displays how it works: it starts with checking the preceding instruction. If the preceding instruction is a method invocation or GETFIELD, the type should follow the return type or field type. If the type cannot be determined by the preceding instruction, the algorithm

checks the current instruction. If the current instruction is a local variable assignment (ISTORE, ASTORE, DSTORE ...), the type should follow the local variable type; otherwise the type does not need to be determined at the time.

```
Input Data :    <source bytecode>

Result :        < the type of the element at the top of the Operand Stack>

<load bytecode>

for each bytecode instruction do:
   ci ← current instruction
   if ci is the instruction to retrieve value then:
      pi ← preceding instruction
      if pi is a method invocation then:
         type  ← return type of the method
      else if pi is GetField then:
         type ← type of the field
      else if ci is xStore(can be iStore, aStore) index then:
         look up the local variable table by index and current offset;
         type ← type of the local variable found
      else:
         type ← not decidable
      end
   end
end
```

Figure 4-8 Data Type Deduction Algorithm

4.4 Multi-threaded Logging

To reduce the overhead of recording, we devised a multi-threaded recording mechanism for JDevv and the workflow of which is shown in Figure 4-9. When two threads, Thread 1 and Thread 2, each produced a sequence of signals that we needed to record, JDevv would start three threads for the recording and dispatch the signal data randomly to these 3 threads, which was the default setting for JDevv. Besides, although the order of the tracing data would be disrupted for such multi-threaded recording we could solve this problem by simply putting a topological sort.

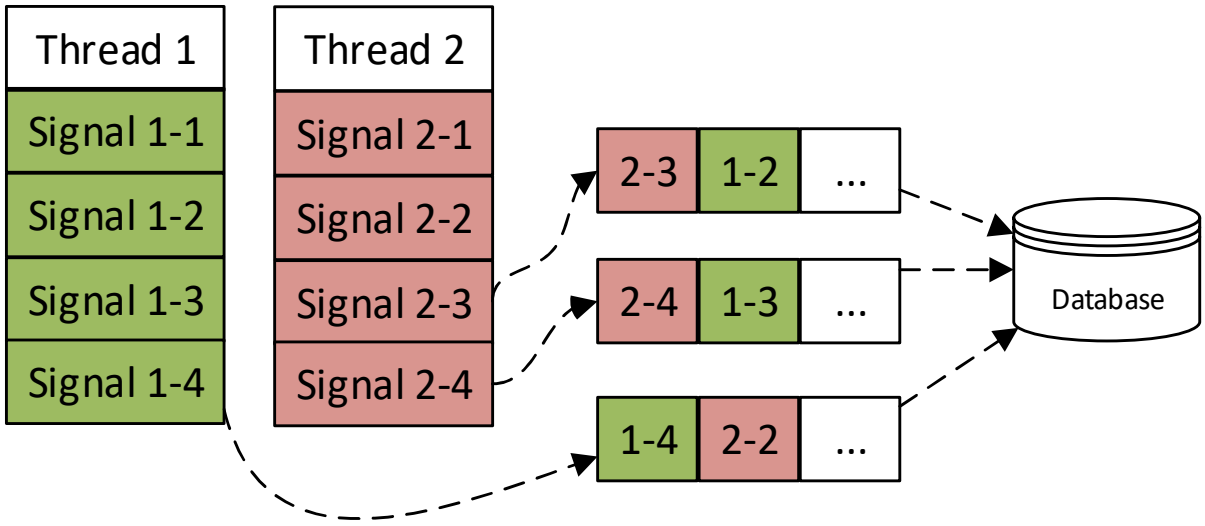

Figure 4-9 Multi-threaded Logging

According to the users' working environment, JDevv supports multiple log recording manners. Furthermore, as MongoDB offers auto sharding to record data across multiple machines and this is good especially for large data set, so we used MongoDB to store the Execution Data

.

4.5 Experimental Results

As for the overhead experiments we executed a simple program (Figure 4-10) n times with different problem size “n”, and then compared the average runtime computed from ten (10) executions under each of three circumstances: the original program without Bytecode Injection; traditional logging approach; and our JDevv performance.

```
001  1 to n foreach( i =>{
002    Thread sleep 1
003    println(i)
004  })
```

Figure 4-10 Simple Scala Code for Monitoring

As a result, our approach would have little effect on the code execution as you could see from the Row 1 (“No Injection”) and Row 3 (“JDevv”) in

Also the Row 2 (“Text”) and Row 3 indicate that although the overhead incurred by JDevv seems slightly bigger than that from the logging when the problem size was rather small (from n=1000), JDevv’s performance would get almost as good as the logging approach as the problem size increases (to n=100000). Figure 4-11 Recording Overhead Comparison Figure 4-11 shows exactly how such differences in terms of percentage got marginally small and even near to 1 percent between these two approaches. In fact, since the overhead introduced by initializing the driver and connection pool of MongoDB is independent of problem size, JDevv probably would exhibit an even better performance when the overly complex code was involved, which seems increasingly commonplace nowadays.

Table 4-3 Logging Overhead

n(Prob Size)	1000	5000	10000	100000
No Injection	1092ms	5481ms	10846ms	108436ms
Text	1133ms	5562ms	10994ms	109632ms
JDevv	1680ms	6237ms	11759ms	110692ms

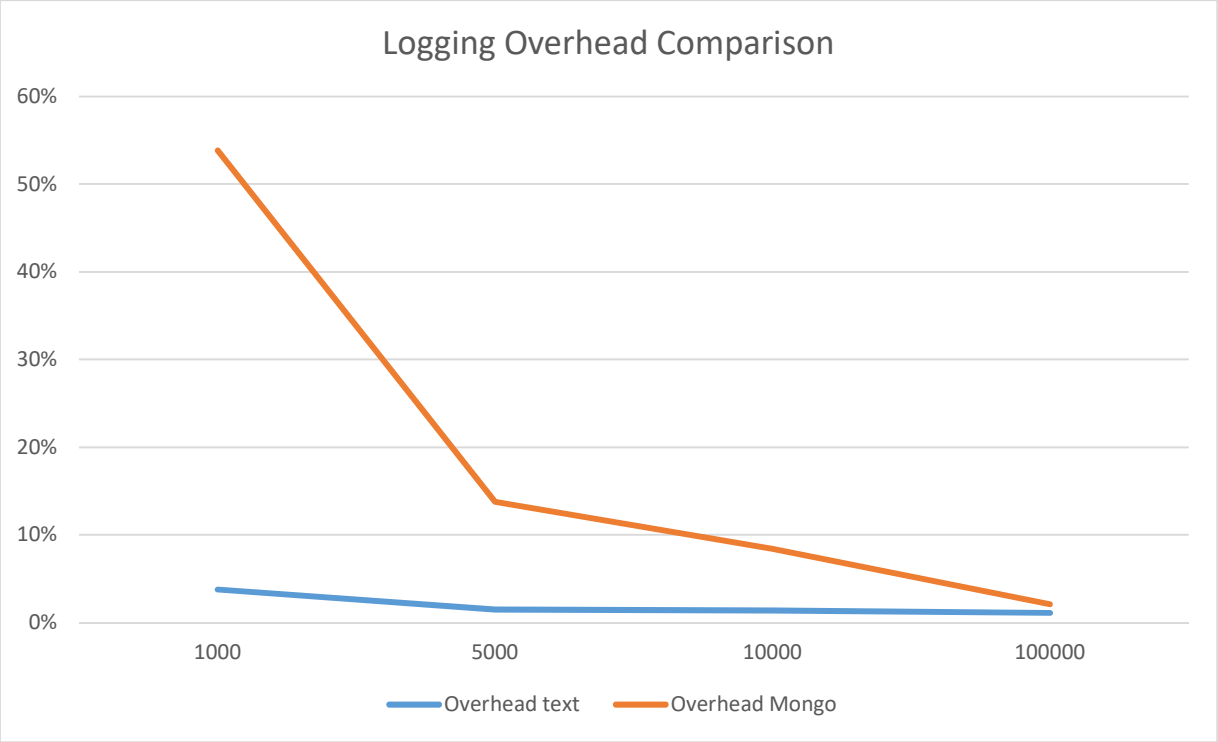

Figure 4-11 Recording Overhead Comparison

To support large data set, JDevv is using MongoDB to store the raw Execution Data (MongoDB Raw Data) as the first choice, if MongoDB is available in users' environment. Because as the data size increase, a single machine may not be able provide sufficient storage space; MongoDB is able to store data across multiple machines by sharding approach.

Chapter 5. JDevv Data Storage and Analysis

Many programmers are trying to implement their own data processing algorithm in parallel for the purpose of better performance. Nevertheless, parallel programming is a very challenging job even for experienced developers. On the other hand, if a user decide to use the existing data-parallel framework to avoid parallel programming such as Spark [4], it often takes a lot of efforts to migrate the existing implementation into the data-parallel framework. To deal with these problems, we propose a JavaScript Promise [22] based data processing framework (JPD) to allow user to implement the data-parallel processing jobs in a conventional single-threaded programming pattern. Moreover, the existing implementations can be reused and then processed in parallel by JPD with the minimum efforts.

Figure 5-1 presents an overview of JPD. Multiple data sources such as text files, database and web services are able to be processed together. During the processing, users can provide the processing policies which are implemented by JavaScript based on their domain knowledges. Moreover, the existing implementation of data processing job such as text parser, Spark jobs and MongoDB aggregation functions can be reused and integrated with users' own data processing jobs. During the processing, MongoDB is used as sink and source to store the intermediate and final results. We also provide a group of Restful Web-Services to provide convenient for connecting with other JPD instance or other data processing platform.

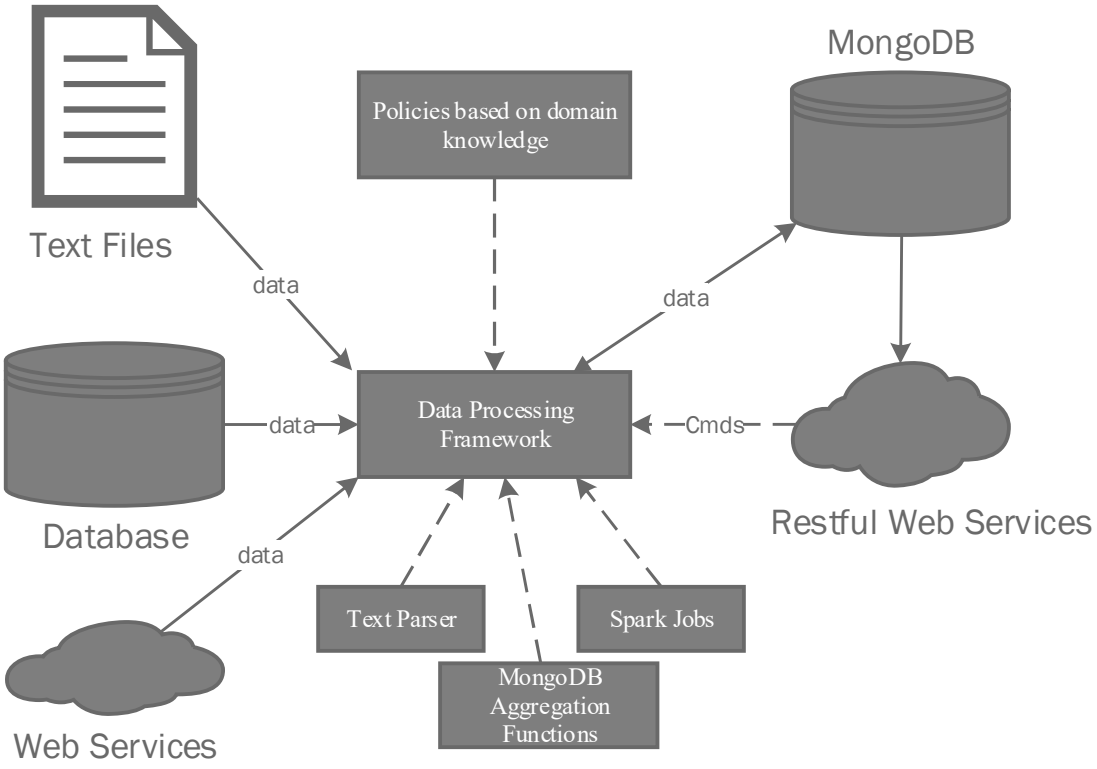

Figure 5-1 Data Framework Overview

JDevv collects a massive set of data generated during the compile-time and run-time of users' program for further visualization and verification. For the purpose of supporting debugging varied kinds of software programs with a good performance, we devised a flexible mechanism for the data storage and analysis. Moreover, a large number of existing and newly developed jobs are involved for the data processing.

The cooperation of these jobs are modeled and implemented base on the promise theory [2]. Data (results) shared among jobs are in semi-structured JSON or BSON (Binary JSON) format. The major two advantages of this approach are that each job only need to concern its own behavior and the result of asynchronous jobs can be handled just like synchronous jobs.

We enhance the standard Promise in JavaScript to support the tasks other than JavaScript functions. It is worth mentioning that the command-line based jobs (e.g. Linux shell command) and distributed jobs (e.g. Spark jobs and MongoDB Pipeline) can also be handled by our framework. Based on the enhanced Promise, we also design and implement a promise guard to supervise the in-processing jobs. By doing so, time-outed and errored jobs can be killed and then completely cleared. Our framework can be flexibly deployed either as a central hub or a separate component in a big data processing project.

5.1 Architecture and Data model

When JDevv monitors a program, the data generated by the execution arrive in a raw format. In general, each record of raw logging data is a reflection of an event such as method invocation, field read & write and array element read & write, which may contain different information in different structure. Furthermore, there is no explicit dependencies between the records. Therefore, this raw data cannot be used directly by the downstream modules such as visualization, which expect well-structured and more aggregated data.

To overcome these issues, we mentioned above and also to improve the performance of JDevv, we devised a two-way data store mechanism for JDevv logging data processing. We store the raw data in Raw Data Model and the processed data in Processed Data Model, also we provide a scalable framework which can distribute the data to Server-Side JS or Spark and MongoDB to process the data in the back.

Figure 5-2 depicts the structure of our design and the data flow. As we discussed in the previous chapter, JDevv supports two data formats: text files (Text Raw Data) and MongoDB data (MongoDB Raw Data). JDevv uses MongoDB Query Engine (which is provided by MongoDB) to retrieve data from MongoDB Raw Data. And then according to the size of the data, Data Processing Framework decides to use the framework itself (Server-Side JS running on Node.js) or Spark to process the data. If the data size is normal (less than 10, 000 raw signal records), JDevv processes data by Data Processing Framework to get a better performance; otherwise, JDevv use Spark to handle the data. If the Execution Data is stored in text files (Text Raw Data), no matter how large the data is, JDevv always uses Spark to process the data, to avoid the big memory usage of parsing and processing text files in Server-Side JS.

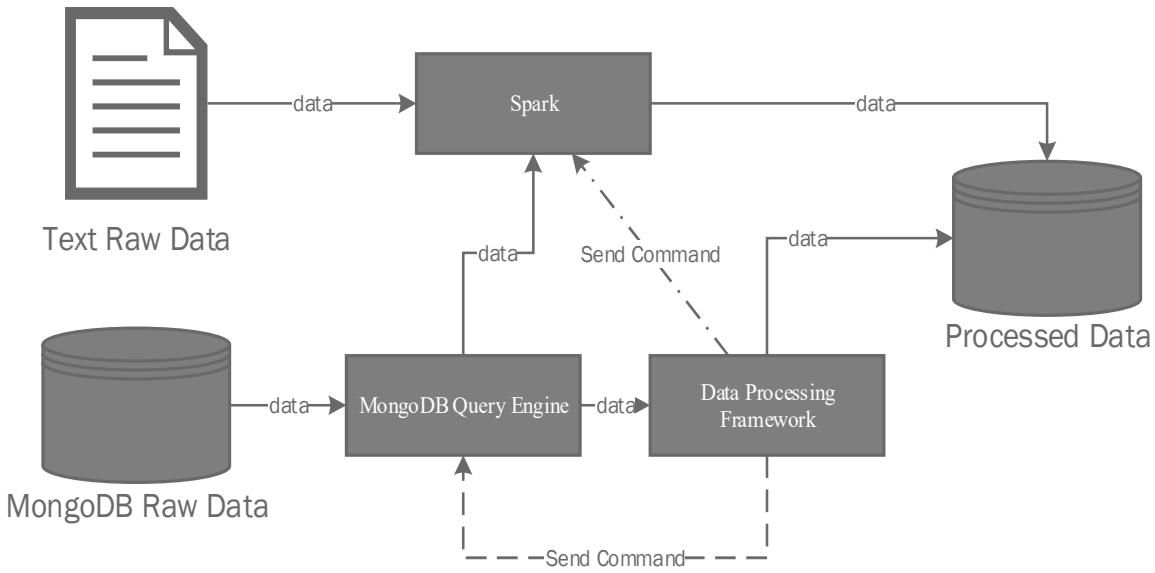

Figure 5-2 JDevv Data Processing Workflow

Based on the features of MongoDB and JavaScript, JDevv adopts Object-Oriented Data Model naturally. More specifically, the basic unit of data stored in MongoDB is a document, which can be mapped directly to a JavaScript object. Therefore, MongoDB and our JavaScript data processing program are able to share the same data model to get rid of Object-Relational (OR) mapping, which is one of the bottleneck in data processing. Both MongoDB document and JavaScript object can be present in JSON format as follows:

```
{name: 'Jim' , age:25,  address: { city: 'New York', street: '111 Pine Dr'}}
```

Each document or object consist of a group of key-value pairs; each key-value pair is a data entry.

MongoDB groups the documents in collections. Collections are the equivalent of tables in Relational Database Management System (RDBMS). A significant difference between collection and table is that collection in MongoDB does not enforce a schema, which mean the documents in one collection are able to contain data entries in different structure.

5.1.1 Raw Data Model

JDevv stores the raw logging data into a MongoDB collection "trace", in which the data is semi-structured to support logging data which generate by varied types of event. Figure 5-3 depicts an overview of this data model. Some of the data entries such as JVM ID, Thread ID and Invocation ID are shared among all the logging data, because these entries are mandatory for the logging data of all types of events. Besides, some of the data entries are shared only in a portion of data. For example, "Owner" and "Owner Ref" are only valid for method invocation

and field read & write, because in Java, all the methods and fields have to be belong to a class (owner), therefore the information of owner is mandatory for these kinds of events. Note that, the "Owner Ref" is only valid for instance method invocations and the instance fields in objects, because as for the static methods or static fields, there is no object reference. Sharing the same data entries allows JDevv to group all the logging data related with one entity such as an object or a class together during the processing. Also, using this semi-structured data model, JDevv can support new event types brought in by new language features with minimum effects.

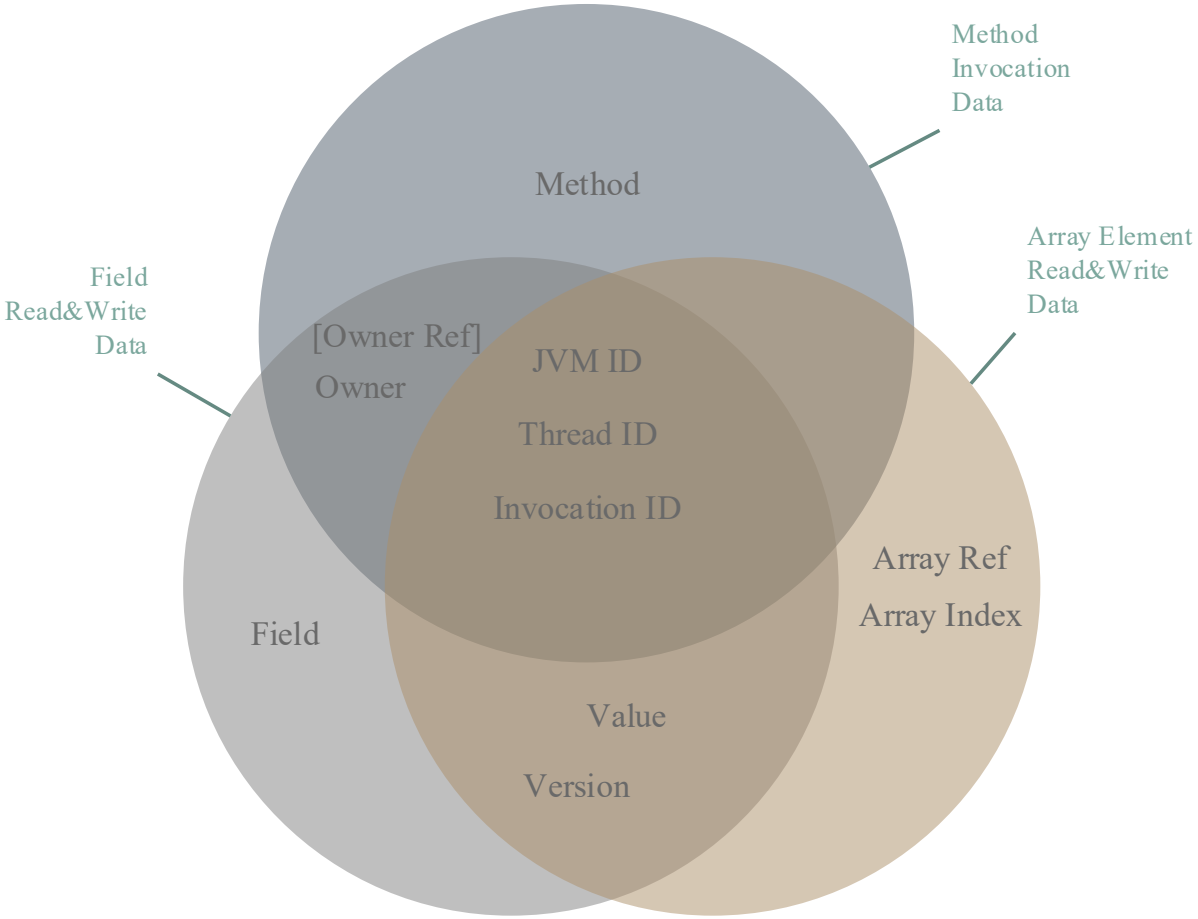

Figure 5-3 Semi-structured Data Model of Raw Logging Data

We do not apply any index on the collection of Raw Data. Because after any write operation, MongoDB will update the data itself, as well as all the index association with the collection. Therefore, to reduce the overhead of writing logs, in the raw data collections, we do not apply any index to improve the performance of writing operations. In addition, this approach does not slow down the read operations much, because most of the read operations on the raw data collections require to perform a full collection scan, which cannot benefit much from the index.

If the data size is too large, we may place the data into multiple database instance using MongoDB auto sharding mechanism.

5.1.2 Processed Data Model

We borrow the entity and relationship concept from Chen's entity-relationship (ER) approach [23] to describe the data structure of our processed data model. Figure 5-4 lists all the data entities and their relationships of processed data model. All the records of each entity contains one unique identifier as the primary key (PK in the figure). The relationships are maintained by foreign key (FK in the figure). The name of the required attribute is written in bold in the figure; the name of the optional attribute is followed by (O).

Unlike raw data model (having data stored from many forms), processed data model contains five kinds of data entities. The records of each entity are stored in a separate MongoDB collection.

UserApp entity represents a user's program which is to be debugged. The 'folder' attribute is for the version control. In reality, user may keep modifying the program. In this case, the program may have many versions. If the user wants to debug the program by comparing the execution of different versions of the program, he can put each versions of the program into separate folders.

JVMProcess entity represents the process of JVM. Generally, a process of JVM represent a process of the execution of user's program. The relationship between UserApp and JVMProcess is one-to-many. Because for a program, the user can run it many times.

Actor entity represents an object or a static class in the memory during the execution. The relationship between JVMProcess and Actor is one-to-many, since each JVM process creates many objects and static classes in the memory during the execution.

Lifeline entity represents an entire process of a method invocation from start to finish. The relationship between Actor and Lifeline is one to many, since the methods of an object or a static class can be invoked many times.

Signal entity represents a signal sent out from a Lifeline (a method invocation). The relationship between Lifeline and Signal is one-to-many, since one method invocation may send out many signals. The attribute "signal_detail" is an embedded object which contains the detail information of this signal. According to the value of "signal_type", the inside structure of "signal_detail" is varied.

It is notable that the "to_lifeline_id" will be null if the signal is not a method invocation. That means the destination of this signal is not an instance of Lifeline entity.

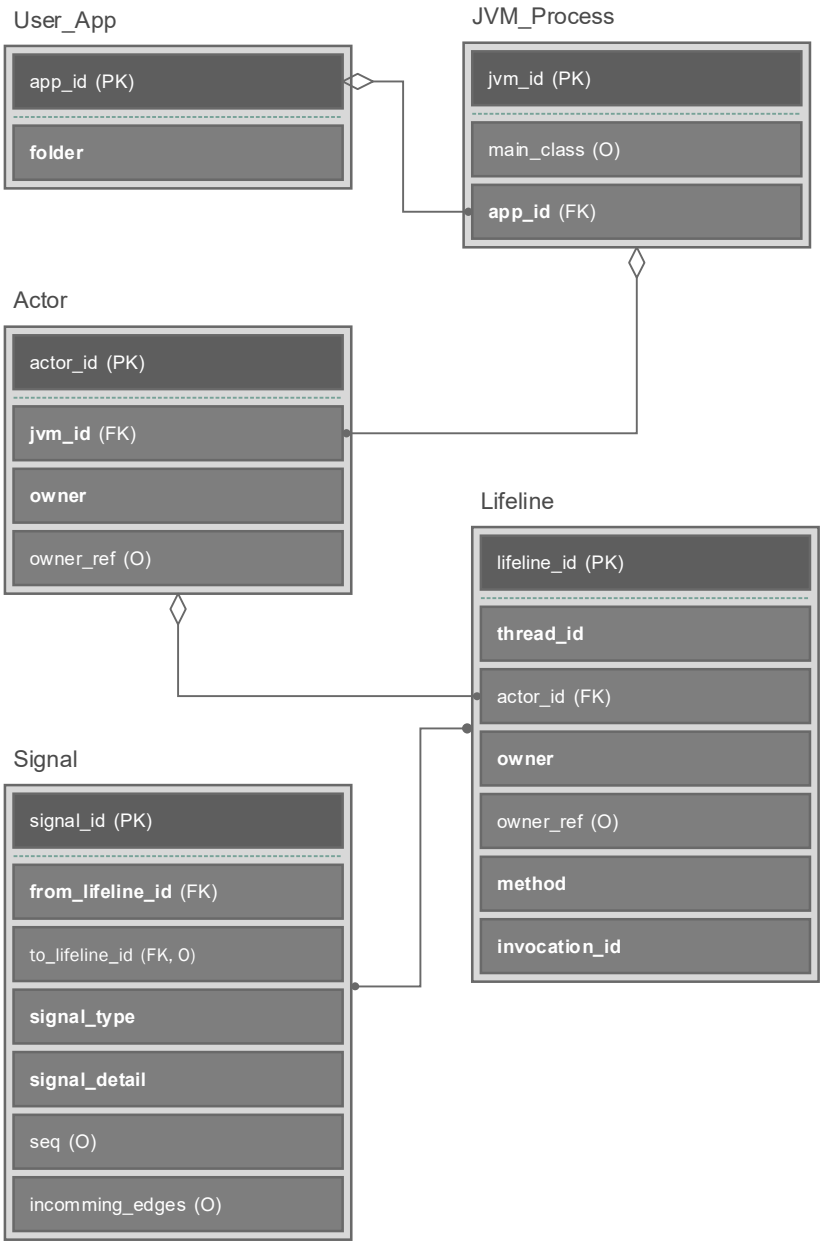

Figure 5-4 Processed Data Entities

5.2 Promise Theory based Data Processing Framework

In this section, we discuss the design and implementation the data processing framework of JDevv. In general, in order to improve the reusability of the code, the entire process of data

analysis is split into many small data processing jobs. Therefore, the major purpose of our framework is to cooperate the schedule of the data processing jobs, e.g. the job of topological sorting cannot start until all the jobs of creating signals are finished. The cooperation of these jobs are modeled and implemented base on JavaScript implementation of the promise theory [24]. The modern JavaScript engine such as Node.js is natively support JavaScript Promise[22] objects. Using JavaScript Promise objects, we can easily wrap a normal JavaScript function for deferred, namely asynchronous data processing. The major two advantages of this approach are that each job only need to concern its own behavior and the result of asynchronous jobs can be handled just like synchronous jobs.

Furthermore, our data processing framework is able to handle diversified data processing jobs other than JavaScript functions, such as MongoDB commands, Shell program or a Spark job. Using JavaScript function to process data is a very convenient and efficient practice. Moreover, wrapping JavaScript function into Promise object is very straight forward. Why do not we just use JavaScript? The major reason is that the Node.js which is the most popular JavaScript engine currently cannot handle large data set in the memory (according to our experiments, the upper limit is around 1.5 GB). Unfortunately, the data set of the logging data may become very large for debugging some kind of program. Due to resource-consuming nature of analyzing these massive data, we developed a group of approaches to wrapping up resource-consuming jobs in JavaScript Promise objects. Taking the advantage of JavaScript Promise, we can implement our data processing in parallel with no need of threads. By doing so, the stability and run-time performance can be improved and the complexity of the system can be highly reduced.

In order to support other data processing job other than JavaScript functions, besides the standard JavaScript Promise which is a build-in feature in JavaScript, JDevv also introduces 4 types of Promise as below:

- Standard Promise: Wrapping normal JavaScript functions and all kinds of Promise objects
- MongoDB Query Promise: Wrapping MongoDB Query and Commands such as map-reduce and pipeline.
- File System Operation Promise: Wrapping file copy, move, read and write operations.
- Docker Promise: Wrapping all the Docker Command.
- Spark Promise: Wrapping all the Spark commands.

Based on all the Promises we list above, the data processing algorithm can be implemented in a very flexible approach, since we can do both parallel and sequential programming very easily.

5.2.1 Promise of Normal JavaScript Function

We involve the syntax of JavaScript to introduce how promise objects work together. Figure 5-5 shows how we simply wrap a plain JavaScript function 'job1' into a promise object. If the 'job1' function finished successfully, the promise will be fulfilled by the "resolve(result)" statement; otherwise the promise will be rejected by the "reject(e)" statement. It is notable that the main JavaScript will not wait for "job1" to finish, but continue to run the next statements.

```
var promise1 = new Promise(function(resolve, reject) {
  try{
    var result = job1();
    resolve(result);
  }
  catch(e){
    reject(e);
  }
});
//next statements
```

Figure 5-5 Wrapping JavaScript Function to a Promise Ojbect

If the promise is fulfilled, the JavaScript engine will call the handler which is assigned to the then () function; if the promise is rejected, the JavaScript engine will call the handler which is assigned to the catch () function. Suppose that, there are three job functions: job1, job2, and job3. If we want to run job1 first, and then if job1 is fulfilled, we start to run job2, otherwise we start to run job3. We can simply wrap job1 into a promise object: promise1. Figure 5-6 shows the JavaScript code which can simply achieve that.

```
promise1.then(job2).catch(job3);
```

Figure 5-6 Promise Objects in a Branch Example

It is notable that the result (return value) of the “job1” function will be passed into the “job2” function; the error information (if there is exception occurred in “job1”) will be passed into the “job3” function.

Similarly, we can make the three functions: job1, job2 and job3 run in sequential order as below:

```
promise1.then(job2).then(job3)
```

Figure 5-7 Promise in Sequential Order

Moreover, Figure 5-8 shows how we can use Promise.All() to wait a group of Promise objects together in parallel. The handler function "job4" we assigned to Promise.All().then() will be invoked when all the Promise objects (promise1, promise2 and promise3) are fulfilled. Like normal promise objects, the results of promise1, promise2 and promise3 will be passed to job4 as well.

```
var promise1 = new Promise(function...);
var promise2 = new Promise(function...);
var promise3 = new Promise(function...);
Promise.All([promise1, promise2, promise3]).then(job4);
```

Figure 5-8 Promise Objects in Parallel

5.2.2 Promise of MongoDB Query

Figure 5-9 depicts how Promise handle the MongoDB Queries in parallel. When the "Query Promise 1...n" is created, we don't have to wait for the result of those queries, instead we assign a handler function to those promises. Then the main process will go the next statement rather than wait. When the query finished, the handler function we assigned will be invoked autsomatically.

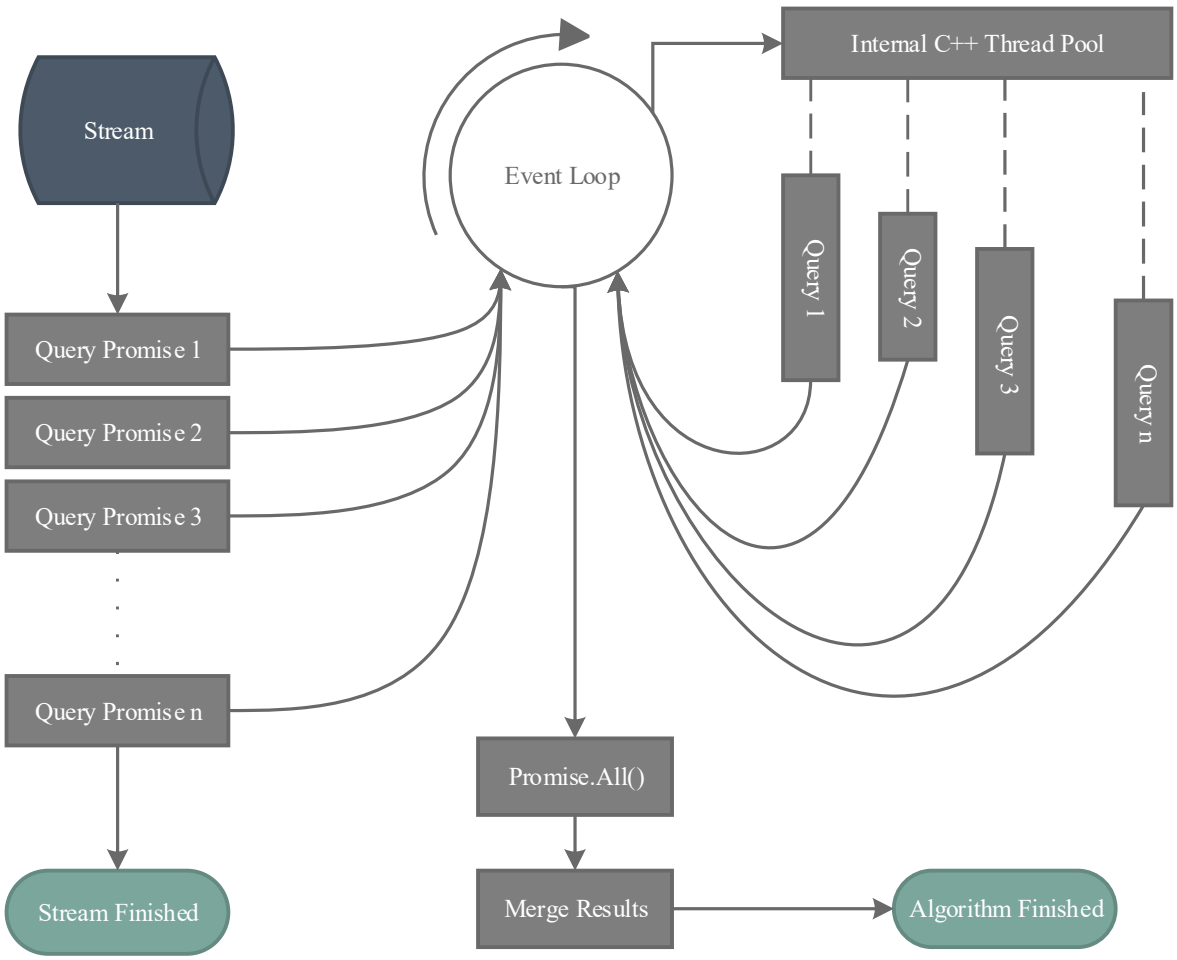

Figure 5-9 MongoDB Query and JavaScript Promise

5.2.3 Parallel functions

As we mentioned, the MongoDB can be deployed as a cluster. With a MongoDB cluster JDevv can use more than one MongoDB server to handle the query or aggregation functions. Taking the advantage of that, the data processing functions can be processed in parallel in the server side.

The data processing frame work of JDevv provides a group of parallel functions which can be processed on the cluster. Unlike Spark using resilient distributed dataset (RDD), our parallel functions are using a single MongoDB collection is as the working set. MongoDB is able to do the data partitioning automatically by MongoDB Sharding. The partitioning can be controlled by setting a sharding key. The operations of our parallel function are implemented based on MongoDB query and aggregation functions and JavaScript promise.

Table 2 shows the parallel functions provided by JDevv as well as the corresponding functions in Spark and JavaScript. Some of JavaScript functions are not supported by the standard JavaScript Library. Therefore, we are using underscore library [25] which provides a group of high performance helpers on collections as a supplement.

Table 5-1 Parallel functions supported by JDevv

JDevv Function	Local JavaScript	MongoDB Query	Spark Function
Filter	Array.filter	find or $match	filter
Map	underscore.map	cursor.map	map
Union	underscore.union	$setUnion	union
Intersection	underscore.intersection	$setIntersection	intersection
Distinct	underscore.uniq	distinct	distinct
Group	customized implementation	group	groupByKey
Sort	sort	sort	sortByKey
Count	Array.length	count	count
slice(begin, end)	Array.slice(begin,end)	skip(begin).limit(end)	only take(n)

Normally, the result of the parallel functions of JDevv are returned in a single BSON document. This format can be converted to JSON which can be used by JavaScript code directly. For large data size of the result, developers can make parallel functions (except count which just returns a single number as the result) write results to a specified MongoDB collection. By doing so, the results are able to exceed the BSON document size limit (16 megabytes).

Figure 5-10 depicts an example of how to use JDevv parallel functions. This example calculates the start time and the end time of a group of method invocations. Line 001 is initialization of the context of the JDevv parallel functions. Line 002 is the filter which only select the signals

which are sending from the given method invocations by a list of method invocation ids stored in “parent_ids”. Line 003 declares a grouping operation based on both thread_id and parent_id. Line 004 and Line 005 define what kind of data will be shown in the result. For each thread and each method invocation, the earliest and latest time of the signals will be collected, which will be useful for calculating the running time of each method invocation. Line 006 defines the result handler which should be a pure JavaScript function or another data processing job just like this example.

```
001   JPD(process_id)
002     .find(parent_id:{$in:parent_ids})
003     .group({parent_id:'$parent_id', thread_id:'$thread_id'}, {
004         start:{$min:"$created_datetime"},
005         end:{$max:"$created_datetime"}})
006     .then(resultHandler);
```

Figure 5-10 JDevv Parallel Function Example

Figure 5-11 depicts the MongoDB query which is generated automatically by JDevv for the example above. The pair of bracket in Line 001 and 012 defines the query is a sequence of sub-queries. Line 002-004 defines the first sub-query which selects the result based on process_id and parent_id. Line 005-011 defines the second sub-query which group the result by parent_id and thread_id.

```
001 [ /*MongoDB Pipeline query generated by JDevv*/
002   {
003     $match:{process_id: process_id, parent_id:{$in:parent_ids}}
004   },
005   {
006     $group:{
007       _id:{parent_id:"$parent_id", thread_id:'$thread_id'},
008       start:{$min:"$created_datetime"},
009       end:{$max:"$created_datetime"}
010     }
011   }
012 ]
```

Figure 5-11 Generated MongoDB Query for JDevv Parallel Function

And then the query we discussed above will be send to the MongoDB server by the “aggregate” command for aggregation pipeline. The aggregation pipeline supports operations on sharded collections, which can make JDevv parallel functions to support huge data source. And the result will be return in a BSON document. To support large result set, we can simple put “$out: {collection_name}” to this query, which takes the documents returned by the aggregation pipeline and writes them to a collection with the given collection name. As required by MongoDB, the $out operator must be the last sub-query in the query sequence. Since there is no limitation of the size of a MongoDB collection, the query is able to return the result set at any size.

5.2.4 Promise of Shell Commands

Generally speaking, JavaScript is a single process and single thread language. However, we can spawn a new child process from the main JavaScript process to run a shell command. The child process is able to emit couples of events to the main process. By doing so, the main process is able to know the status of the child process. Moreover, the main process can collect the output (via the stream of standard output and standard error) of the child process.

Figure 5-12 depict how JDevv create a promise for a shell command in the main process. The main process initializes the output string to empty at the very beginning. And then the main process spawns a child process to execute a shell command. After the child process starts, the main process is free to handle other jobs. During the running of the child process, any output printed out will be sent to the handler in the main process, which append the output of the child process to the output string in the main process. When the child process finishes, the exit code of the child process will be sent to the main process. If the exit code is 0 which means the

command in the child process is finished successfully, the resolve(output) will be called to make the promise fulfilled, otherwise the reject(error) will be called to make the promise rejected. Ultimately, we wrap an execution of a shell command into a JavaScript Promise.

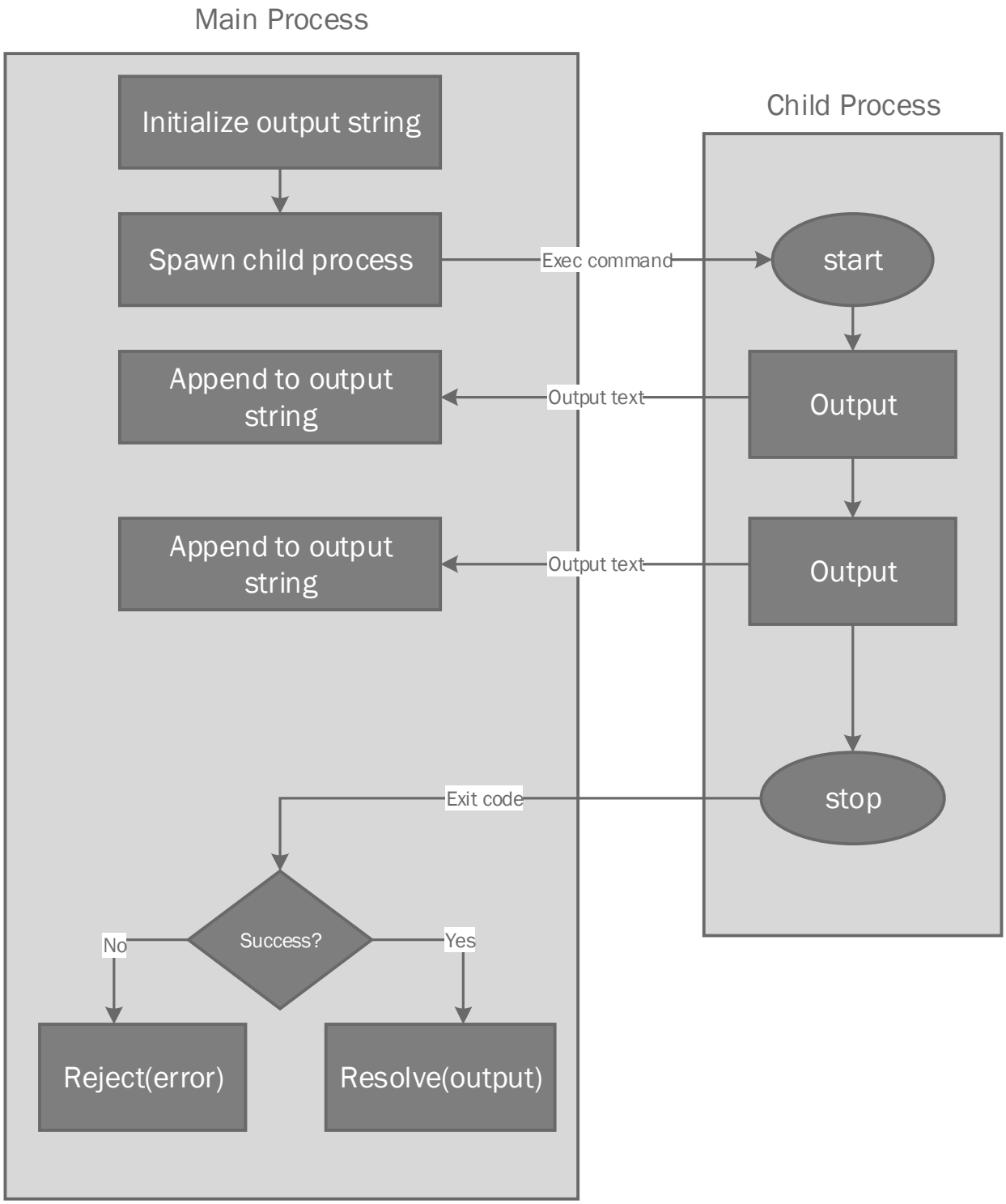

Figure 5-12 Promise Spawns a Child Process for Shell Command

5.2.5 Rerun-able Spark Promise

For the purpose of sharing data with other JPD data processing jobs, our Spark jobs use MongoDB as the input source and output destination. Connecting MongoDB is achieved by MongoDB-Hadoop Connector [26] which is an open-source plug-in of Hadoop and Spark. Moreover, we can specify a query criteria during the data loading, which means we can only load the data needed in Spark. After loading the data from MongoDB to RDD, you can implement your algorithm as usual with all the parallel functions in Spark.

Storing data into MongoDB is straightforward. Nevertheless, the same Spark job may be started by JDevv again and again with different parameters. Because during the debugging process, the users may check the result from a small portion of their code to another small portion. That is why we need to rerun the spark jobs for different portions of data.

Therefore, we may create the same object (data entry) during the rerun. Since we cannot check the existence for a particular MongoDB record by the MongoDB-Hadoop Connector (if we do it manually, it may cause serious performance issues), we are using an "upsert" model to create or update the data in MongoDB.

Figure 5-13 depicts an example of the “upsert” model used by JDevv Spark jobs. The real data is stored in the data object (line 003). Line 002 provides a matching criterion which means the document will be matched by “_id”. By the attribute "upsert:true" (line 004), if a matched document exists in the MongoDB, then the update operation performs the specified update

“$set : data”(line 003); if no matching document exists, then the insert operation performs to create a new document with the data provided in line 003.

```
001  {
002    _id: data._id,
003    $set: data,
004    upsert: true,
005    multi: false
006  }
```

Figure 5-13 Upsert Data Model

With the “upsert” data model we described above, since no duplicated objects will be created, all the JDevv Spark jobs are re-runnable.

5.2.6 Promise Guard

Under certain circumstances, the exit code of the promise of shell command may not be return on time (e.g. the shell command is dead) or the exit code cannot reflect the true situation (Spark jobs may return before everything is done).

Figure 5-14 shows the design of a timeout guard. We register a timeout callback function when the main process spawns the child process. If the child process finished in time, we clear the callback function and then it works just like a normal promise. Otherwise, the timeout callback function will be triggered, which will kill the child process and then make the promise rejected. The dash line from “Register a Timeout Function” to “Timeout triggered” means the current process is able handle other event during the waiting.

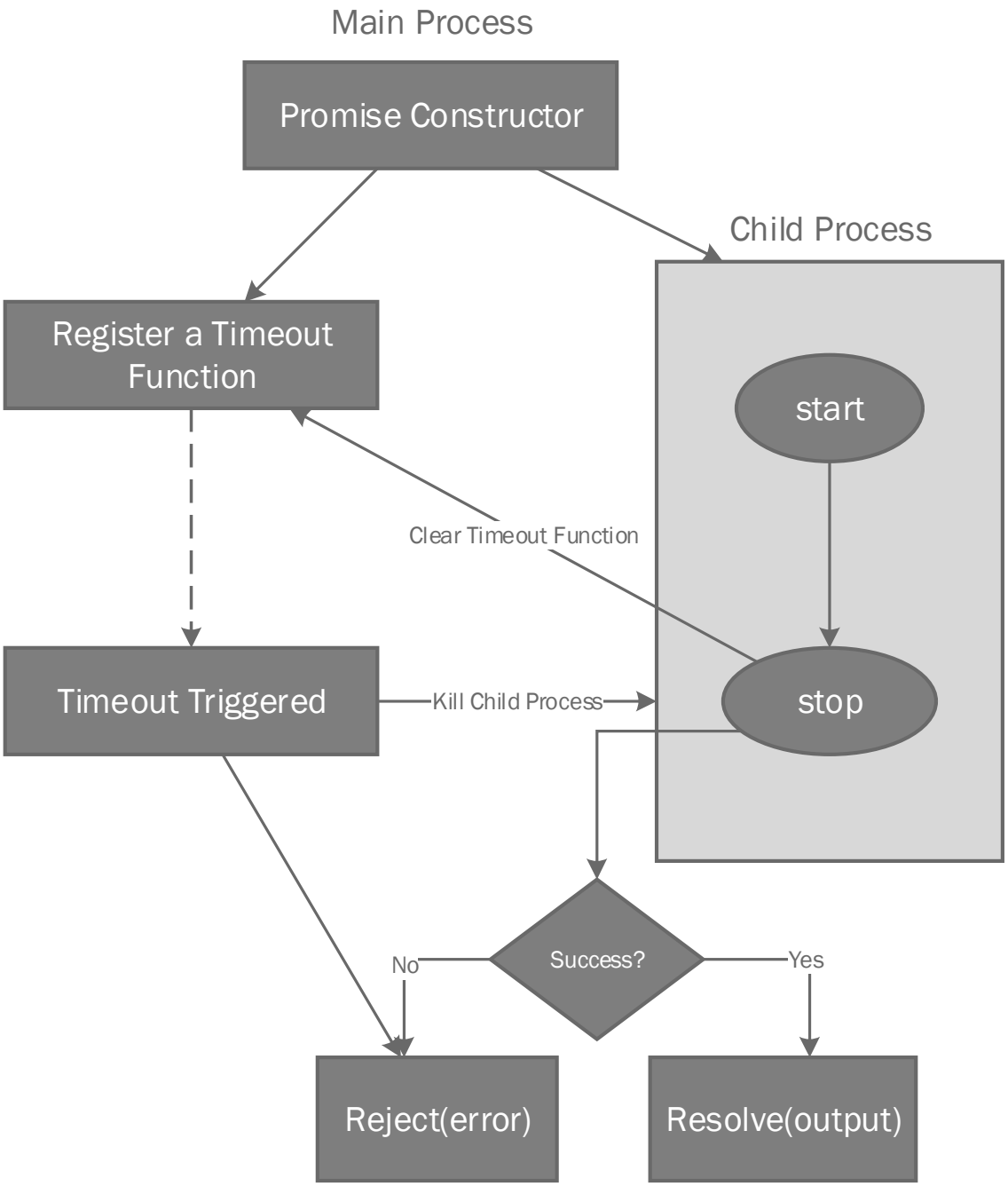

Figure 5-14 Promise Guard

5.2.7 Restful Web Service

On the server side, Our Promise based framework can be easily integrated with any http web server implemented by JavaScript, such as the very simple one proposed in [5] or the full-featured web framework Express [27]).

Figure 5-15 shows how to wrap a JDevv data processing job as a web service. These fifteen lines code can host a separate service for the data processing job we pre-defined in line 006. Line 003 defines the endpoints of the service by a URL "/process" which will be exposed to the outside. The result which is returned by the data processing job (line 007) will be rendered in JSON format as an HTTP response in line 009. If the data processing job is failed, then the problem will be handled at line 011, the web service will render the error information in JSON format as well (line 013). Since all the data processing jobs of JDevv are using JSON as the default format of input and output, there will be no extra overhead for the data format converting in this web service.

```
001  var express = require('express');
002  var app = express();
003  app.post('/process', function(req, res) {
004    var param1 = req.body.param1;
005    var param2 = req.body.param2;
006    processingJob(param1, param2)
007      .then(function(result) {
008        //return result to the client.
009        res.json(result);
010      })
011      .catch(function(error) {
012        //return the error stauts and error information the client.
013        res.status(500).json(e);
014      });
015  });
```

Figure 5-15 Wrapping JDevv Data Processing Job as a Web Service

On the client side, JDevv provides a “postJson” function which wraps an HTTP request in POST method as a normal JavaScript promise. Figure 5-16 shows the implementation of this “postJson” function. Line 003 define this function with two parameters, one is the URL of the remote service and the other is data which will be passed to the remote service. Line 008 renders the data as a string of JSON which can be transferred via HTTP protocol directly. Line 001 defines the timeout which is optional. Line 013 defines the handler for the normal cases and line 014 defines the handler for the failed cases.

```
001 var Promise = require('es6-promise').Promise;
002 var $ = require('jquery');
003 exports.postJson = function(url, data){
004   return new Promise( (resolve, reject) =>  {
005     $.ajax({
006       type: "POST",
007       url: url,
008       data: JSON.stringify(data),
009       contentType: "application/json; charset=utf-8",
010       dataType: "json",
011       timeout: 10000
012     }).then(
013       (data) => resolve(data),
014       (jqXHR, textStatus, err) => reject(err)
015     );
016   });
017 };
018
```

Figure 5-16 Wrapping Http Request as a Promise

Based on both server side and client side implementation, we can establish a distributed data processing work by delegating some job to another instance of JDevv via http web service. Figure 5-17 shows the simple code of a distributed data processing example. Line 001 prepares the data locally. And then the data will send to the remote data processing service which is exposed at "/remote/service". When the remote data processing is finished, the result from the remote end will be passed to a local data processing job which will be started at line 005. Any errors from the remote job and local job will be handled at line 008.

```
001  var data = {…};
002  postJson("/remote/service", data)
003    .then(
004       function(resultFromRemote){
005          localJob(resultFromRemote);
006       }
007    ).catch(function(error) {
008        …//do something to handle the error
009    });
010  )
```

Figure 5-17 Distributed Data Processing Example

5.2.8 Distributed Data Processing Workflow

Figure 5-18 depicts the data processing workflow for the visualization of program execution. Via the web interface, users start the static analysis (PSF) and dynamic analysis (SED) in parallel. PSF contains a JavaParser and an AST convertor which can parse the source code and then convert the result to JSON. SED starts a Docker container which will execute users' program in a virtual machine-like environment. These two jobs are wrapped as Shell-Promises with promise guard, and then store the data collected into MongoDB database. After all the storing job finished successfully, a Data Mapping (DMap) job is started to deal with each record in the database to extract information from some of the string value. Then two grouping jobs groupByField and groupByThread start in parallel. After the grouping job is finished, a graph algorithm Topological sort (TopSort) is started and then all the temporary results of steps above will be cleaned in Clean Temporary Results (CTR) step. DMap, groupByField, groupTheThread and TopSort can be implemented as either a MongoDB promise or a Spark promise.

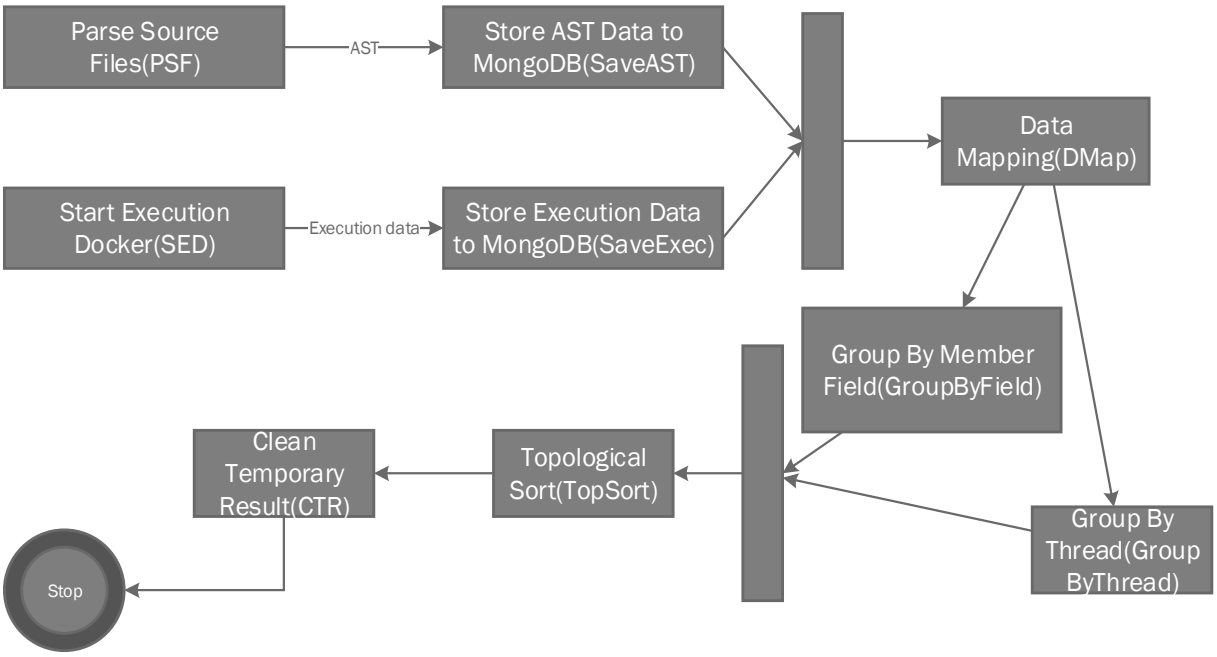

Figure 5-18 Distributed Debugging Data Processing Workflow

The steps are implemented as data processing jobs with varied of technologies, and they are wrapped as JavaScript Promises as we discussed in the previous subsections which make the jobs working together very smoothly.

5.3 Server-Side JS and MongoDB-based Analysis Mechanism

In this section, we will discuss how to use Server-Side JS and MongoDB to implement the data processing jobs we mentioned in the previous sections.

MongoDB natively supports single purpose aggregation operations, aggregation pipeline and map-reduce function to perform data aggregation [28]. Single purpose aggregation operations provide simple access to common aggregation process just in a single collection, such as “group”, “count” and “distinct”. Aggregation pipeline constructs a multi-stage data processing

pipeline which transforms the raw documents into the final results step by step. Each stage of a pipeline is a simple native operation which has been predefined by MongoDB such as group, average, distinct and count. Map-reduce asks users to provide two JavaScript functions to perform the map and reduce operations, so that map-reduce is more flexible and less efficient than the aggregation pipeline. However, either approach cannot satisfy all the needs of JDevv, such as topological sort and sequence diagram building. Therefore, we developed our own processing approach by using server-side JavaScript supported by Node.js. All the complex algorithms are implemented by JavaScript, and the JavaScript will control the query against MongoDB.

In most case, a software developer who wants to debug a big program (which may contain thousands of source files) will start out from a very small portion of the whole program. Therefore, JDevv provides a feature that the user can specify an appropriate filter to highly reduce the data amount needed and then improve the runtime performance of JDevv. Figure 5-19 depicts the how the data filter works in a massive dataset. The green circles represent the data which can be selected directly by the filter; the yellow circles represent the data which is related by the green ones. JDevv only need to process the data which are represented by the green and yellow circles, because the other portion of data are beyond the scope of the software programmer to consider the issue. For instance, assuming that, a programmer is debugging his code under JBoss which is a widely used J2EE container. The JBoss contains more than 20, 000 java classes which may generate huge Execution Data, most of which is irrelevant to the debugging. Therefore, the programmer only need to focus on the data generated by his

own code (green circles) and also the data generated by the libraries which are invoked by his own code (yellow circles). We will discuss this approach in detail in the next section.

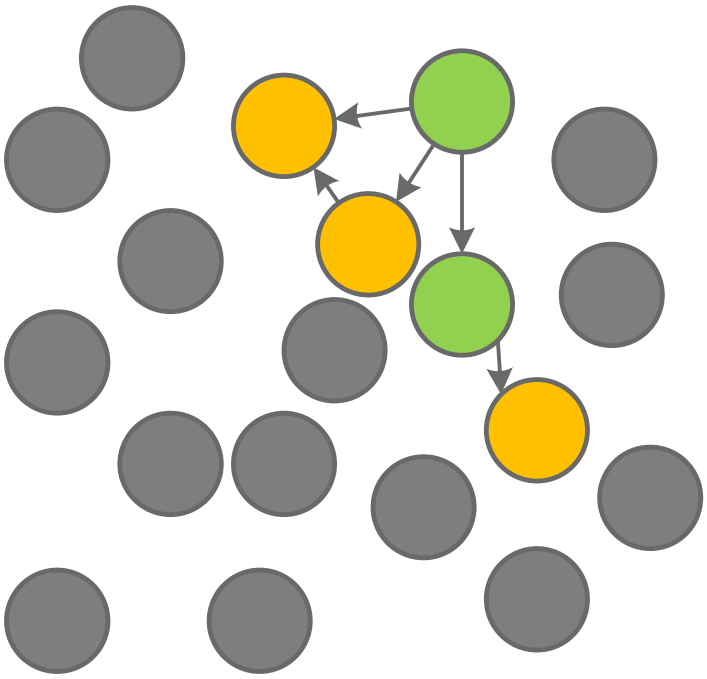

Figure 5-19 MongoDB and Server-Side JavaScript Data Processing Mode

5.3.1 Parallel Sequence Diagram Data Building (Data Mapping)

In this section, we discuss how JDevv generate Actors, Lifelines and Signals based on the Raw Execution Data.

Before the processing, the user can set the filter to specify the scope of the code he wants to debug by providing a group of class names. And then we collect all the records with the type "method_enter" from the classes the user just specified. Based on this idea, a query criterion "Q" can be generated as below:

```
Q = {
        jvm_id : …,
        msg_type:'method_enter' ,
        $or: [{method_desc: {$regex: "^"+class_name_1}}, {method_desc: {$regex:
"^"+class_name_2}, …, {method_desc: {$regex: "^"+class_name_n}}; };]
}
```

Figure 5-20 Query all the "Method Enter" data

Since the value of "method_desc" is a string which follows the format: "{class_name}#{method_name}({method_params}){method_return_type}", we can match all the data we needed by providing a regular expression {**method_desc**: {$regex: "^"+class_name_n}}; }. "^" + class_name_n here means the value of "method_desc" should start with the value of "class_name_n".

The query "Q" is the starting point of our data processing. And then JDevv build Actors, Lifelines and Signals with the algorithm depicted in Figure 5-21. Line 001 queries the database with the criteria we build above and then uses the result as a stream. For each data entry (which is data in line 002) in the steam, the function createActorAndLifeline which returns a JavaScript Promise creates an actor (if the actor does not exist) and a lifeline (line 003). It is notable that the promise returned by createActorAndLifeline will start immediately just like a new thread, and the current thread will go to the next data entry without any waiting. When the promise are fulfilled, an other function createOutSignals which takes the result of createActorAndLifeline as parameters and also returns a promise will be invoked in line 004.

When all the data entry have been processed, the function of the next step will be invoked in line 007.

```
001   var stream = find(Q).stream();
002   stream.on('data', (data) => {
003        …//do some other jobs on the data
004        createActorAndLifeline(data).then( (from_actor, from_lifeline) =>
005            createOutSignals(from_actor, from_lifeline);
006         }
007       }).on('close', function () {
008             //algorithm finished, do next step such as top sort
009       });
```

Figure 5-21 Naive Parallel Building Actor, Lifeline and Signal Algorithm

However, the algorithm we described above may lose a small portion of data during the processing. Because when all the data entries have been processed, we cannot guarantee all the promise created have finished, which means some of the actors, lifelines and signals may not have been written to the database. If we start the function of the next step, these data will be not involved. Therefore, we provide an improved version of this algorithm to build the actor, lifeline and signals in a sequential approach which is depicted in Figure 5-22. Before the createActorAndLifeline has been invoked, the current process pauses the stream (line 003), and then when all the actor, lifeline and signals have been created, current process resume the stream (line 008). The pause and resume function work just like a locking mechanism which can guarantee that when the function of the next step in line 012 is invoked, all the creation of

the actors, lifelines an signals have been finished. But this approach will increase the runtime about 30% percent because it runs the jobs in a sequential rather than a parallel way.

```
001   var stream = find(Q).stream();
002   stream.on('data', (data) => {
003        …//do some other jobs on the data
004        var self = this;
005        self.pause();
006        createActorAndLifeline(data).then( (from_actor, from_lifeline) =>
007            createOutSignals(from_actor, from_lifeline).then(functin(){
008                self.resume();
009            });
010         }
011    }).on('close', function () {
012          //algorithm finished, do next step such as top sort
013    });
```

Figure 5-22 Sequential Building Actor, Lifeline and Signal Algorithm

Therefore, Figure 5-23 depicts an improved parallel building actor, lifeline and signal algorithm without any locking mechanism. Every time, when the createActorAndLifeline function is invoked in line 005, the promise returned by this function will be pushed to an array of promise (line 008). The function Promise.all in line 010 will create a new JavaScript Promise which will waiting for the completion of all the promise in the given promise array.

```
001   var stream = find(Q).stream();
002   var promise_array = [];
003   stream.on('data', (data) => {
004     …//do some other jobs on the data
005     var p = createActorAndLifeline(data).then( (from_actor, from_lifeline) =>
006            createOutSignals(from_actor, from_lifeline);
007     }
008     promise_array.push(p);
009   }).on('close', function () {
010      Promise.all(promise_array).then(function(){
011         //algorithm finished, do next step such as top sort
012      })
013   });
014
```

Figure 5-23 Improved Parallel Building Actor, Lifeline and Signal Algorithm

Another problem is how to avoid to create duplicated actors into MongoDB database in the parallel data processing. If we query the database to check the existence of the actor just before the actor creation, we have to put the two operation query and creation in a database transaction to avoid some other thread to create a duplicated actor just between these two operations. However, involving database transaction may cause severe performance problem. In addition, in MongoDB, transaction is not natively supported. Therefore, based on the features of MongoDB database, we use findOneAndUpdate function to do these two operation together as an atomic operation which cannot be interrupted by other processes. If the actor we want to create has already been in the database, the findOneAndUpdate function will also return the existing one, which is useful for us to create lifelines and signals.

5.3.2 Topological sort

The purpose of topological sort is to reproduce the sequence of states of a multithreaded program. Because the sequence of the logging data cannot reflect the real sequence of the execution of a multithreaded program. However, we don't have to reproduce the exact sequence of the execution, because we don't care about the sequence of the execution of many of the instruction which don't depends on others. See the example in Figure 5-24, if we start two thread to run the method "foo", the instructions in line 002 and line 003 don't depend on these instructions in other thread. In other words, for these two thread, who run these instructions first won't change the result of the program. Therefore, we don't care the sequence of these instructions during the debugging. However, the sequence of execution of the code in line 004 may give us different result of the value of "this.a" in the multithreaded environment. Because this instruction depends on the current value of "this.a" which may be modified by some other thread. Therefore, in the logging data, the sequence of the execution of this instruction has to be sorted in a proper way.

```
001     public static void foo(int start, int end) {
002         int a = start;
003         int b = end;
004         this.a = this.a + a;
005         return;
006     }
```

Figure 5-24 Code Dependency Example

Moreover, for each thread, line 002 and line003 also don't depend on each other, and actually the real sequence of these two instruction during the runtime is unpredictable because the JVM may optimize the execution by JIT. But most users may feel uncomfortable with that the execution of line 003 occurs before line 002. Thus, just in order to provide a less confusing result for the users, we sort the sequence of these two instruction by their natural order in the source code in this case.

Therefore, we construct the dependency graph via the following two rules:

- In one thread, the current instruction depends on the preceding one. In other words, in one thread, all the preceding instruction must be executed before the current one. This rule can be enforced by the invocation_id which is maintained as a thread-local variable of each thread.
- For one field, the current modification depends on the preceding modification; also the current reading depends on the preceding modification. This rule can be enforced by the version information which is maintained by each field.

Figure 5-25 depicts a dependency graph of the execution of a two threads program with no race condition. The solid lines with the arrow indicate the dependency relationship in the same thread. In the source code level, in line 004 of Figure 5-24, the assignment sign "=" which is an operation of writing to "this.a" appears before the reading of "this.a". But in the bytecode level, the read appears before the write. That's why in the figure block (4) is before block (3). In general, the flow of the solid line shows the sequence of one thread. And the dash line indicates the dependency relationship between two threads. In the figure, block (8) read the value of "this.a" which is written by block (3), so block (3) has to occur before block (8). Since block (1) and (2) don't depends on the block (5) and (6), the results of the topological sort of this graph are varied, which can be [1 2 3 4 5 6 7 8], [1 2 5 6 3 4 7 8] or [5 6 1 2 3 4 7 8]. For the debugging purpose, all of these results are equivalent. Because the sequence of the execution expressed by these varied results are equivalent.

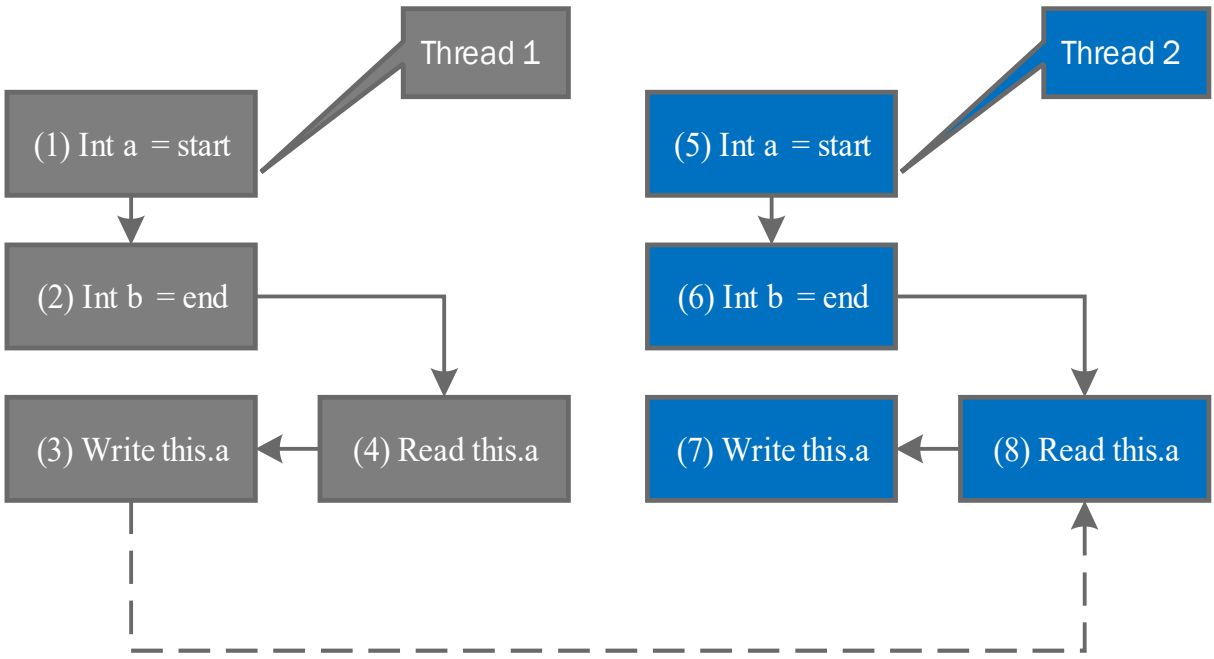

Figure 5-25 Dependency Graph with a Normal Case

Figure 5-26 depicts a dependency graph of the execution of a two threads program with a race condition. In this figure, the reading operation of "this.a" in one thread does not depend on the writing operation of "this.a" in the other thread. If the two threads read the same version of "this.a", the order of these two reading operation cannot be determined. But for the writing operation of these two thread, since they are writing to the same field "this.a", there must be one and only one winner of the writing operation. The winner will overwrite the result written by the other thread. In the figure, if the version of "this.a" in block (3) is greater than the version in block (7), then the block (3) has to occur after block (7).

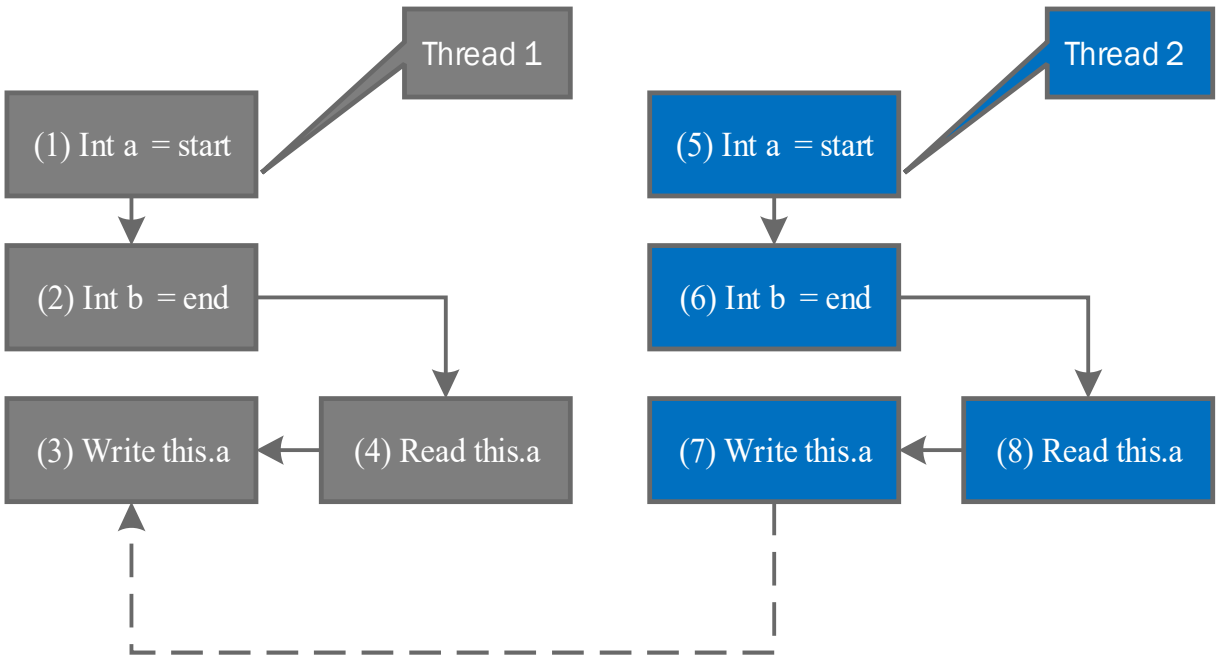

Figure 5-26 Dependency Graph with a Race Condition

The implementation of dependency graph is using a MongoDB array to store the incoming edges of each signal, since each signal may contains zero, one or many incoming edges. To reduce the storage cost, the array just contains the signal ID rather than the entire signal.

Figure 5-27 shows the algorithm of the topological sort based on MongoDB and Promise theory. The algorithm is divided as two steps. The first step is building the edges which are the dependency relationship between signals (line 002). This step is wrapped as a JavaScript Promise which returns the number of the signals which have been built with dependency edges. The second step is reducing the no dependency signals based on the idea derived from Kahn's topological sort algorithm [29]. This step is also wrapped as a JavaScript Promise which returns the number of the signals which have been reduced. The result of these two steps are

collected in line 009 as two attributes “signal_count“ and “reduce_count” in a JavaScript object. If these value of the result are not equal to each other, then the data processing will be terminated because that usually means the dependency graph contains at least one cycle which shouldn’t occur in normal execution data.

```
001 exports.signalTopSort = function(jvm_name){
002   return buildEdges(jvm_name).then(function(count){
003     console.log("top sort start...");
004     var p = new Promise(function(resolve, reject){
005       topSortReduce(jvm_name, resolve, reject, 0);
006     });
007     return p.then(
008       function(reduce_count){
009         return {signal_count: count, reduce_count: reduce_count};
010       }
011     );
012   });
013 };
014
```

Figure 5-27 Topological Sort Algorithm Based on MongoDB

Figure 5-28 shows the algorithm of topological sort reduce. Firstly, the algorithm will find one and only one signals with no incoming edges and the “seq” is unset. At the same time, the algorithm updates the “seq” of the signal just found to count + 1 (line 003). Then the algorithm assigns the ID of this signal to “signal_id” (line 004). If no such a signal can be found, then the algorithm finished and return the count as the final result. If there is such a signal, then the algorithm scans the entire signal collection to find all the signals with the incoming edge

"signal_id". For the signals found, the algorithm removes the "signal_id" from incoming_edges (line 008 and line 009). Then the algorithm recursively invoke itself again with an increased count value to handle the next signal (line 010). Although it is a recursive function, Node.js offers a tail call optimization which make the last function call "topSortReduce" in line 010 won't increase the size of method invocation stack. In our experiments, this implementation can support more than 1 million signals without any stack overflow exception.

```
001  function topSortReduce(resolve, reject, count){
002      Signal.findOneAndUpdate({incomming_edges is empty AND seq is unset}, {set
003  seq to count + 1})
004      Var signal_id = the id of the signal just updated
005      if(no record updated)
006         resolve(count)
007      else{
008        Signal.update({incomming_edges contains signal_id}, {pull signal_id out
009  from incomming_edges}}, { multi: true })
010        topSortReduce(resolve, reject, count + 1);
011      }
012  }
```

Figure 5-28 Topological Sort Reduce Algorithm

The result of this algorithm will be stored in the "seq" attribute of each signal. If we need to re-run this algorithm, the both "incoming_edge" and "seq" attributes have to be cleared.

5.4 Spark-based Analysis Mechanism

JDevv will use the Spark-based analysis mechanism to process the data, if the user cannot specify an efficient filter and the data size is very large. In this mechanism, we do not relay on the query engine of MongoDB much. Instead, we load all the data to Spark partitions, and then process the data with functions implemented by us based on Spark. Figure 5-29 depicts the

data flow of Spark-based analysis mechanism. All the data are loaded into the resilient distributed dataset (RDD) which mostly are in the memory by Spark. Since the physical memory in just one node (a physical server or a virtual machine server) is limited, Spark indeed divided the data into many partitions. Based on these partitions, the data processing functions such as filtering, transforming and grouping can be processed in parallel. After the processing, the result usually is also in the RDD which are also partition across the nodes. Because most of the processing will use the data in the memory, the runtime claimed by Spark team are 10 times to 100 times faster than Hadoop [30] which is one of the most popular map-reducing framework. Indeed, Spark is also preferred to use HDFS which is provided by Hadoop as the data source and result container. In JDevv, we are using MongoDB as the data source and result container to get more flexibility for the logging and visualization. A performance evaluation [31] shows the performance trade-offs about use Hadoop and MongoDB together. Their point of view combined with our experiments shows that using MongoDB as the data source (read only operations) of Spark can get better performance than HDFS in most of our user scenario. However, it may cause severe performance issues when Spark writes large volume of result into MongoDB. Therefore, we design our Spark data processing jobs in the pattern of reading large dataset and then only writing a much reduced result set to MongoDB.

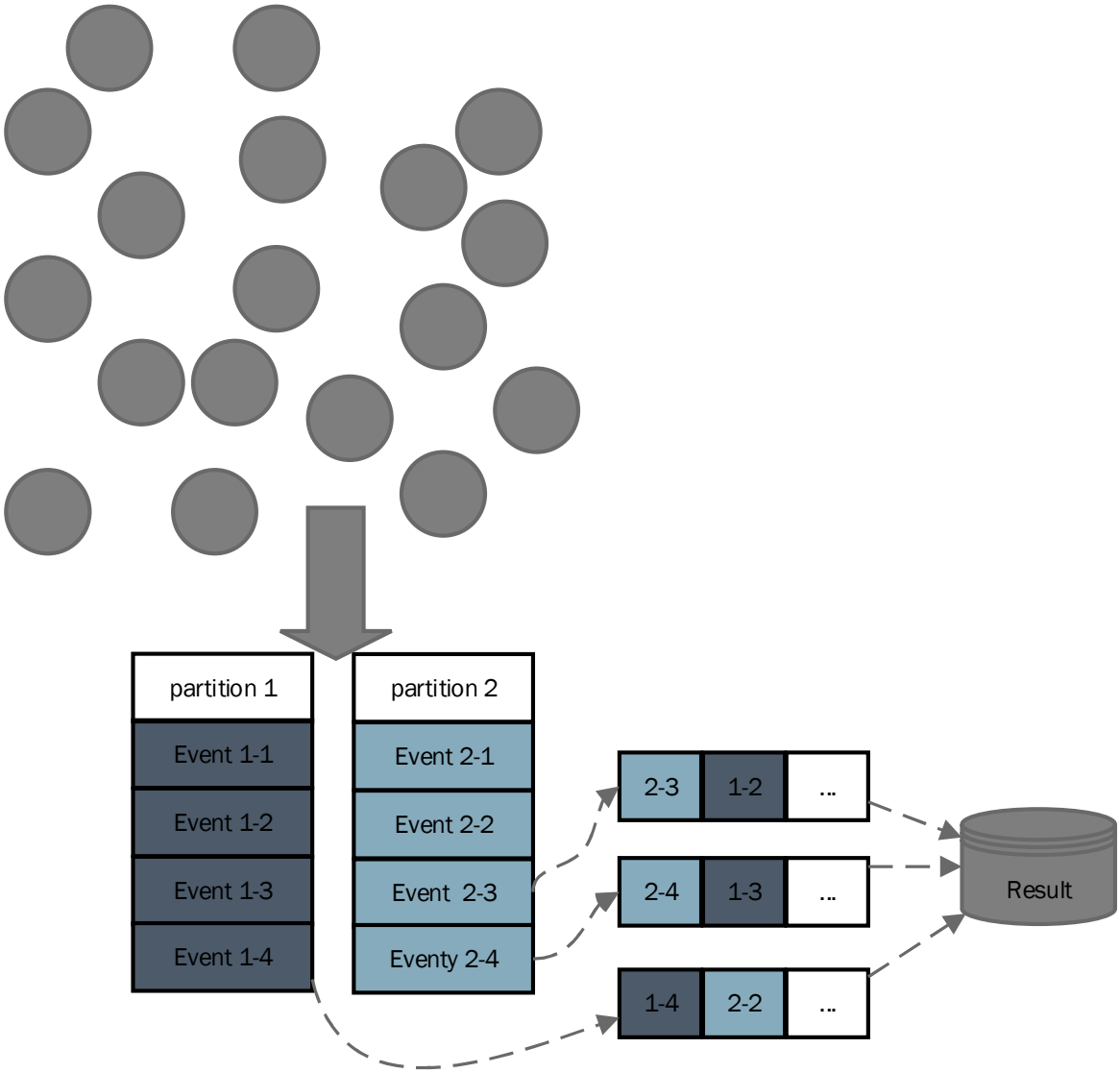

Figure 5-29 Spark-based analysis overview

5.4.1 Sequence Diagram Data Building and Dependency Edges Generation

In this section, we discuss how JDevv generate Actors, Lifelines and Signals based on the Raw Execution Data with Spark. The approach described in this section can get the equivalent result of the approach described in the section 5.3.1.

Before the processing, the user can still set the filter to specify the scope of the code he wants to debug by providing a group of class names. If the user cannot specify the scope, all the raw logging data will be loaded to the spark. Figure 5-30 depicts the data processing flow after all the data loaded. All the "method_enter" data will be reduced by "method_desc" which contains the method information into the data of lifelines, and further the result reduce by the class_name and owner_ref to the data of actors. Meanwhile, other data will be collected and reduced into the Raw Signals. In Raw Signals data, some of the attribute such as from lifeline id is missing. So we have to calculate this kind of attributes by joining the data of raw signals and lifelines. And then we group the Unsorted Signals by thread ids and field name. See the example depicted in Figure 5-25, the result of grouping by thread is two array [1,2,3,4] and [5,6,7,8] after sorting. And then we slice the result of each thread by 2 to generate the dependency edges as a group of signal id pairs [(1,2), (2,4), (4,3), (5,6), (6,8), (8,7)]. For the field writing, the result of group by field name is [3,7] after sorting (since the version of (7) recorded in the Execution Data is greater than the version of (3)). Similarly, one edge (3,7) is generated for field writing. For the field reading, we join the two reading signals [4,8] and two writing signals [3,7] together by the version. Because (8) is using the data written by (3), the version of (3) and (8) are equal. Thus the join result is [(3, 8)] which is the collection of edges for dependency edges between reading and writing (we put writing as the first one, because the reading is depends on the writing). The last, we generate the incoming edges for all of these Unsorted Signals by merge all the edge results above together. Indeed, the edges we generated are the pair (from_signal_id, to_signal_id) of signals. Since the data entry of each edge only contains two ids, it would be good for efficient use of the memory. In addition, the

entire data structure is flat and can be partitioned automatically by Spark. Thus, the entire procedure can be distributed into many Spark nodes to handle larger data set. Also, by using more nodes the data processing jobs can obtain more resources such as CPU cores and memory to get better performance.

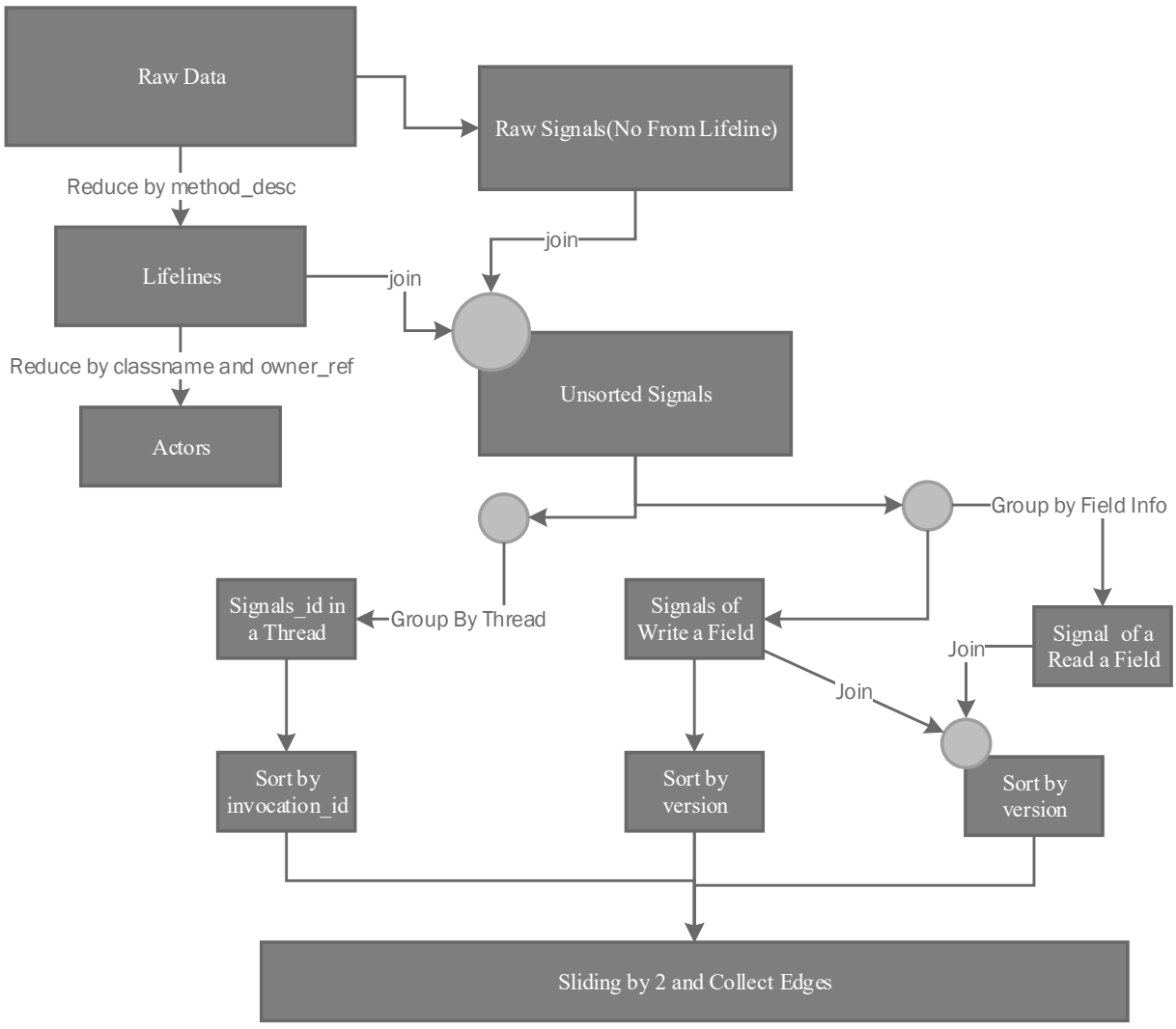

Figure 5-30 Sequence Diagram Data Building and Edges Generation

5.4.2 Topological sort

Figure 5-31 depicts the data processing flow of the topological sort on Spark RDDs. Based on the data we prepared in the previous section, we have two data set, unsorted signal Ids and the data of edges. We join these two data set together by the "to_signal_id". For example, there are two incoming edges [(3,8), (6,8)] for the signal (8). Accordingly, after the joining operation, a data entry will be generated as [8, [3, 6]] which contains the "to_signal_id" and also an array of the "from_signal_id". By the data we generated above, we are able to split the unsorted signal Ids into two data sets (RDDs), the Ids with incoming edges (Remaining Signals) and the Ids with no incoming edges (Independent Signals). Then we use the data set of Remaining Signals to replace the data set of original Unsorted Signal Ids. In the other hand, we use the Edges to join with the IndependentSignals to remove all the edges from the signals in the IndependentSignals, and then the result is the UnprocessedEdges, which is used to replace the original Edges. And also, we collect (means copy the data from multi nodes to just one node) all the data in IndependentSignals and assign an increasing unique sequence number for each signal in IndependentSignals. Then, the result appends to the data set SortedSignalID. Then the data processing restart again and again with the updated UnsortedSignalIds and Edges till the IndependentSignals is empty. If the IndependentSignals is empty, but the Remaining Signals is not empty, the data processing will be terminated with an error because there are cycles in the dependency graph which won't happen for our execution data. When we collected all the signal Ids to the data set of SortedSignalID, we join SortedSignalID and UnsortedSignals together. Because SortedSignalID only contains the information of the Ids and

sequence numbers, while UnsortedSignals does not contain the sequence numbers. And then for each signals, we have a sequence number on it, which means the signals are sorted.

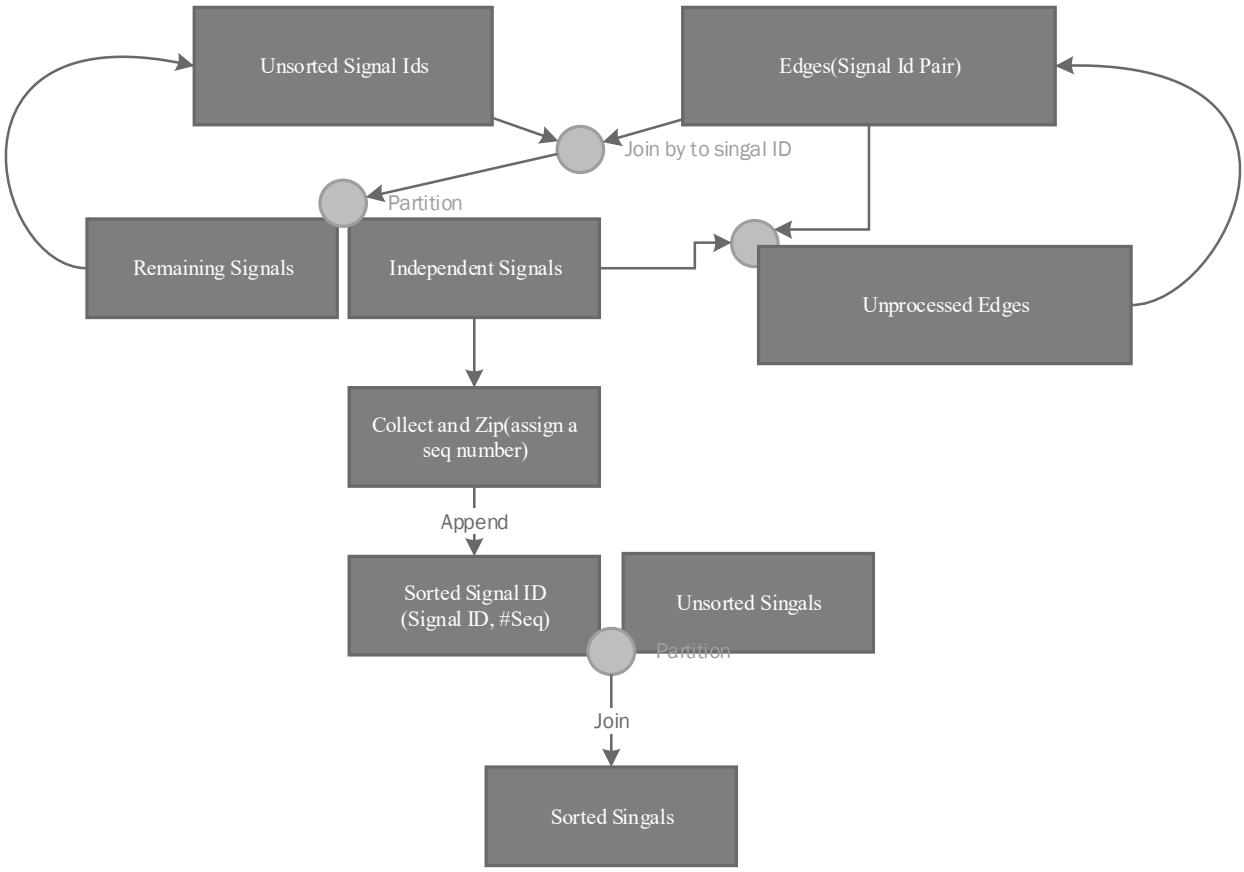

Figure 5-31 Spark Topological Sort Processing Flow

5.5 Experimental Results

All the experiments are carried out on a 16-cpu-core virtual machine with 16GB memory installed. All the instances of node.js, MongoDB and Spark are deployed in the Docker containers to simulate a distributed system environment. To each Docker container, we

assigned 2 processor cores. Moreover, all the memory and disk space are shared among all the Docker containers.

As for the runtime of a single function, we compare the average runtime computed from ten (10) executions for these three implementation: The Spark parallel functions; local JavaScript build-in functions; and our JPD parallel functions. Figure 5-32 shows our JPD parallel outperforms the other two implementations. The main advantage of JPD is that a single parallel function is running on the MongoDB server side, thus there is no need to transfer the data needed between the server and client.

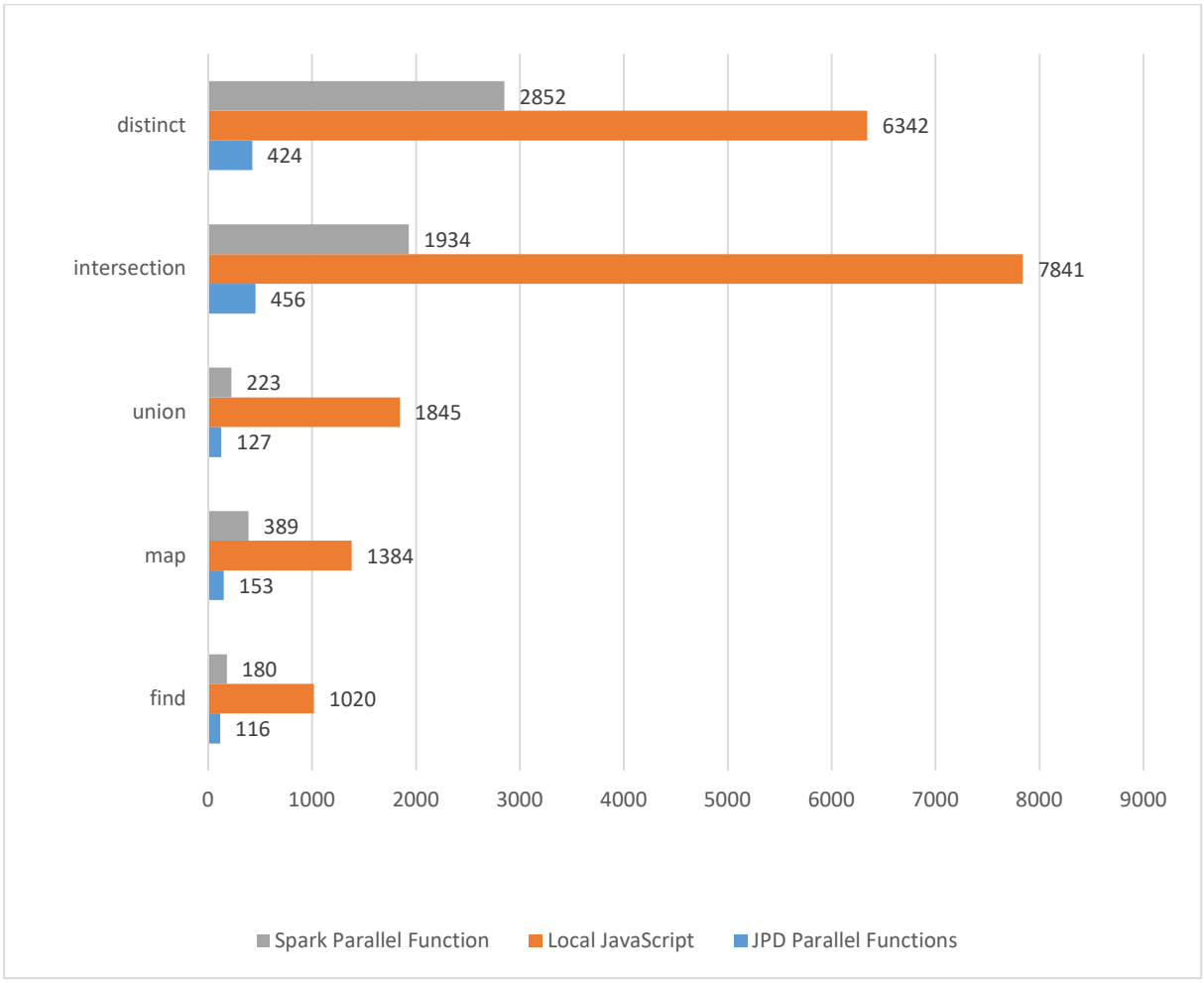

Figure 5-32 Parallel Functions Performance Comparison

Figure 5-33 shows the measured runtime of sort function against different problem sizes to compare Spark and JPD. We prepared two testing environments for Spark, one (gray) is using RDD as the result set and the other (yellow) is using MongoDB. The runtime performance of JPD's sort function is astonishing when applying a proper index, given the fact that the index of MongoDB is implemented based on B-Tree, which keep all the data entries sorted by the index key. If there is no available index, a collection scan will be performed during the sorting, i.e. scan every document in the entire collection. Even in this case, JPD still overperforms Spark for all the problem size we tested. Our result also shows that writing back the results to MongoDB

in Spark will introduce significant overhead. Thus, we recommend only storing the highly reduced result into MongoDB.

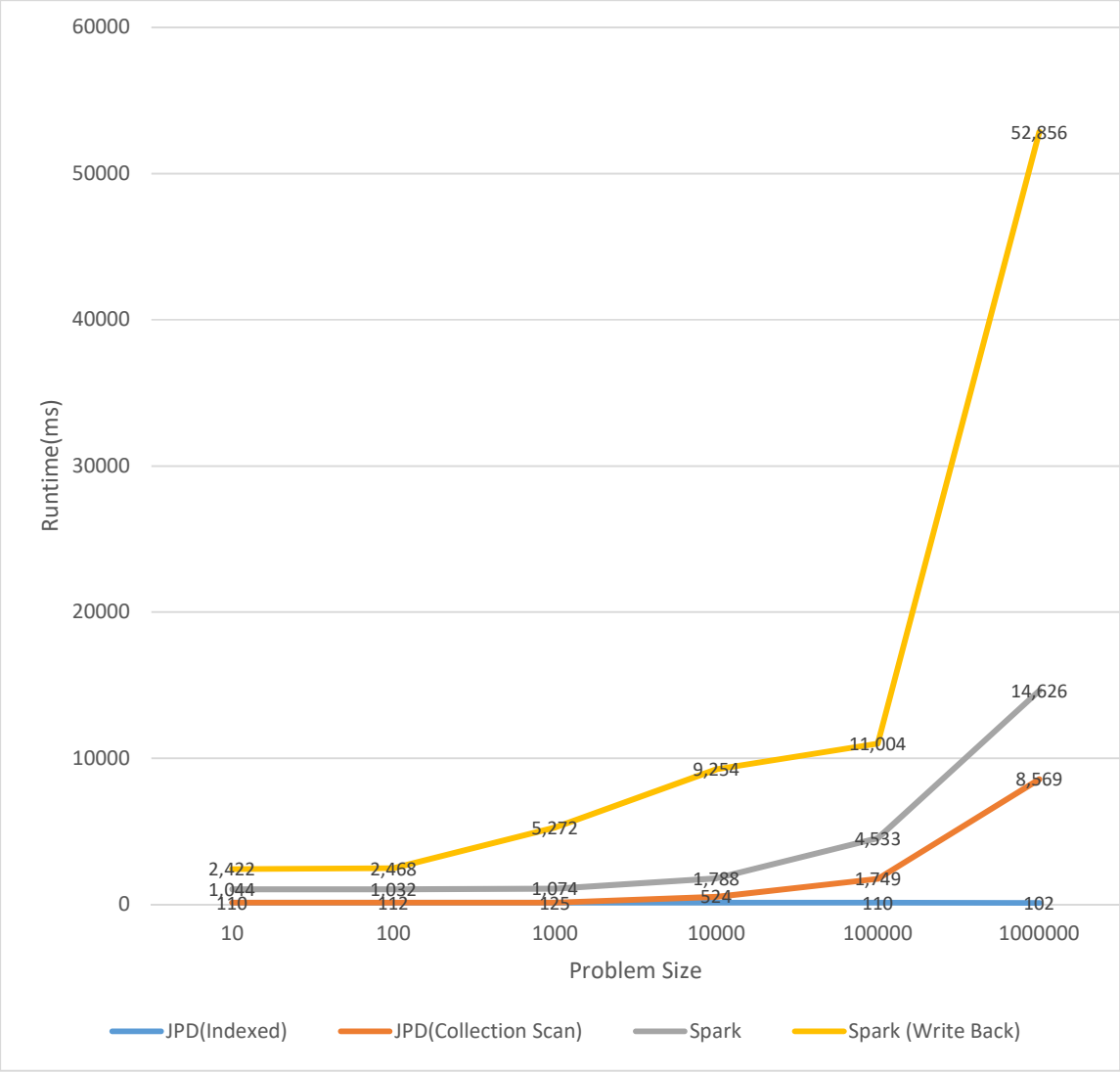

Figure 5-33 Sorting Runtime with Different Problem Size

We also measured the maximum memory usage of JPD and Spark from the operating system point of view during the testing. The four Spark nodes used around 16GB memory in total, and JPD only used 0.8GB (node.js plus MongoDB). Therefore, our JPD will be better suited for the servers in a low memory situation.

As for the runtime of the data processing we would measure JavaScript and Spark respectively through a rather simple multithreaded program such as Quicksort. Table 2 shows that when the number of signals generated increased as the problem size grew, the runtime by JavaScript would get remarkably large while that by Spark would get otherwise. Actually as most of the bugs are independent of problem size JavaScript would be able enough to handle this type of bugs by simply setting the size to a small number if it was too large. On the other hand, Spark would be better able to deal with the bugs related to the problem size, which are not as commonplace as bugs independent of problem size. That is exactly why we set the JavaScript as the default debugging approach while employing both the JavaScript and Spark for the data processing. Consequently, whether the problems come in big size or small JDevv would be able to efficiently deal with all of them.

Table 5-2 Data Processing Runtime

Problem Size	10	100	1000	5000
Num of Signals	521	8114	81650	274663
JavaScript	0.9s	15s	2122s	282492s
Spark	12.8s	34.2s	84s	332s

Chapter 6. JDevv Dynamic Visualization

6.1 Html5-based Visualization

When we check the execution of a program by a visualized view, it is desirable to view the related data from one portion to another. For example, when we are viewing the execution of a method invocation, we may also would like to see the parent method invocation and some of the method invocations which are sent out from the current one. Also, we may want to see what happened before and after this method invocation. Interaction and animation will be a very promising approach to visualize the related data. Previous research [32] [33] shows that dynamic visualization with interaction and animation may be more effective than comparable static graphics in many situations. In addition, using the animation on the multi-stage chart (such as the chart with changing values) can greatly enhance the effectiveness of the visualizations [34].

In order to visualize the execution of users' program, JDevv provides an html5-based visualization framework which can support interactive and animated views. Our framework can transform the processed Execution Data into Html5 element (not only just tables, paragraphs and buttons, but also Scalable Vector Graphics (SVG) such as rectangle, cycle and lines). Indeed, our framework uses SVG instead of bitmap images to present graphical elements. By doing so, our framework can support larger amount of element and more customizable for interactive events and animation.

In addition, our framework is cross platform, since these Html5 elements can be viewed by most of modern web browsers such as Chrome, Firefox and Safari which can be install on varied operating systems such as Window, Linux, iOS and Android.

Our framework also provides a modular programming model to render Component-Based html pages. Based on a html-like syntax extension JSX [35] to JavaScript, our framework wraps html code directly into JavaScript classes as a component. For each component, it can be either stateful or stateless. The state can be the value of any properties of an html element such as width, height, color and text, which can be updated any time after the element is created. Thus, stateful components provide more powerful support to the animation and multi-stage chart and stateless components provide less overhead while updating.

Based on the components, our framework support multi views in one html page to display text compound with graphic shapes. Indeed, each view is a container of the components, which is designed for one or more functionalities of the visualization, such as sequence diagram, value monitor and source code. Each view and component will handle its own event such as mouse click, drag and drop. The event which has to be notify other components, based on the idea derived from Flux [36], will be wrapped as an action which will be sent to an unidirectional action flow to notify all the components registered to this action.

6.1.1 Related Work

The main functionality of our framework is generating and manipulating html element based on the data of JDevv. The main closest analogues is the D3.js [37] which can selectively bind the

data to arbitrary html elements. D3 as well as many other successful JavaScript Libraries such as jQuery manipulating the html elements based on a selection using a simple predicate. For example, d3.selectAll(“p”).style("border", “1px solid black”) will update all the border paragraph element (identified by “p”) to a black and solid line in 1 pixel weight. Our experiments show that, these JavaScript based selection may cause performance issues when it is repeatedly selecting in a huge html document which contains more than 10 thousand elements. However, during the JDevv visualization and animation process, many html elements have to be updated again and again in a short interval. In addition, these selections based libraries usually bind data to a selection of html elements, which means the html elements know the data but the data don’t know what elements are bind. Thus, if one data entry is bind to many html element, when this data entry is updating, it would be difficult to select and notify all the elements bind. To overcome this problem, lots of separate custom event handlers have to be registered to the elements, which may cause difficulties in system design. Furthermore, if some elements highly depend on the status of other element (e.g. the speed of the animation of one chart is synchronized with the animation of another chart), the selection based library cannot provide much help.

6.1.2 Architecture

In this section, we proposed our architecture design of our visualization framework. Figure 6-1 depicts the architecture design of how our framework transform the data in the server to the html elements in an html page. Our visualization framework contains three major parts: the dispatchers, the stores and the views & components. The solid lines represent the data flow and the dash lines represent the event and action flow. The stores load the data from server by

ajax requests in JSON format, and then the stores convert the JSON data to JavaScript objects. The views & components rendering the html elements to an html page based on the JavaScript objects in the stores. When an event occurs in the html page (e.g. user clicks a button), the event will be handled at views & components first. If the event has to be notified to other views and components, it will be wrapped as an action and then passed to the dispatcher which is a singleton class (only one instance in the entire system). The dispatcher will notify the corresponded stores which are related with this action. The store may update the data base on the logic of this action. And then based on the updated data, the views & components will re-render the corresponded html element. For example, when the user clicks on a signal in the sequence diagram, all the other views and components have to be synchronized to this signal also. Thus this event will be wrapped as a “JumpTo” action, which will send to all the stores related via the dispatcher. Then all the stores related will update the current signal to the signal the user just clicked, which will trigger the correspond event handler in all the views and components related such as sequence diagram, source code view and etc. to re-render the html element to “jump” to the signal the user just clicked.

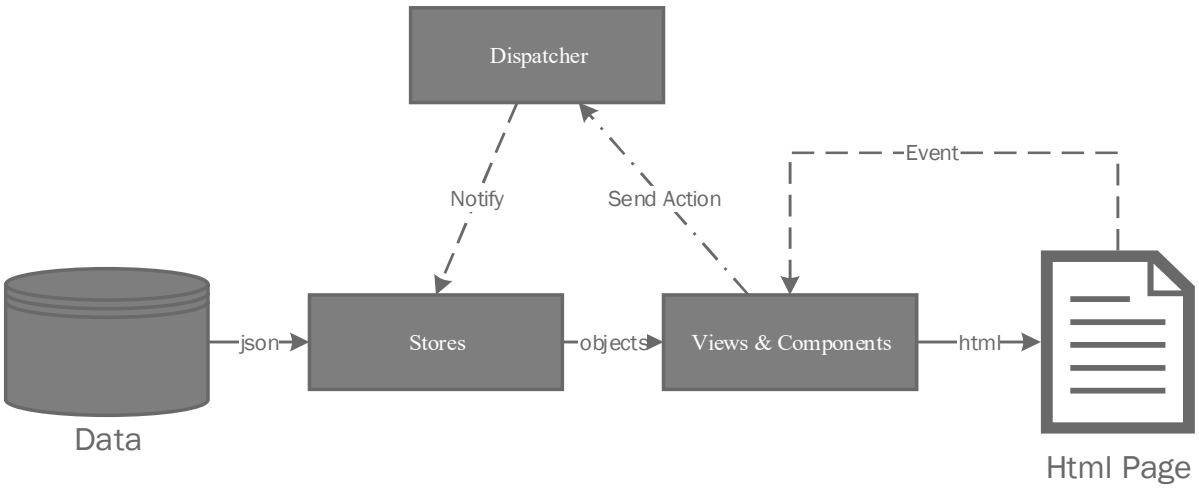

Figure 6-1 Architecture Design of Visualization Framework

6.1.3 Stores (Data Model)

The stores maintain the data, state and logic. Figure 6-2 depicts the design of the stores in our framework. All the stores inherit from the class EventEmitter which can emit event to the event listeners pre-added. CommonStore is the super class for all the stores which provides set, get, delete and merge functions to manipulate the data stored in it. When the data is updated, the CommonStore will fire an event to the listeners. RemoteStore is the super class for all the stores which need to fetch data from the server. To construct this kind of store, a url and request method (such as get or post) will be passed into the constructor. Then the instances of the remote store will load the data from the server via the given URL and request method automatically. All the instances of stores have to be created by StoreFactory which control the life cycle of all the stores. Figure 6-2 also provides tow concrete classes JvmProcessStore and JvmClassMetaStore as an example to show when and how to use the CommonStore and RemoteStore. JvmProcessStore is a sub-class of RemoteStore and it will

load all the information about a JVM process including a group of JVM class “meta” information. JvmClassMetaStore is a sub-class of CommonStore which does not load data from server. Indeed, the instances of JvmClassMetaStore are created based on the data entry loaded by JvmProcessStore. This design can avoid to request the server too many times if the number of the data entries of JVM class “meta” is very big.

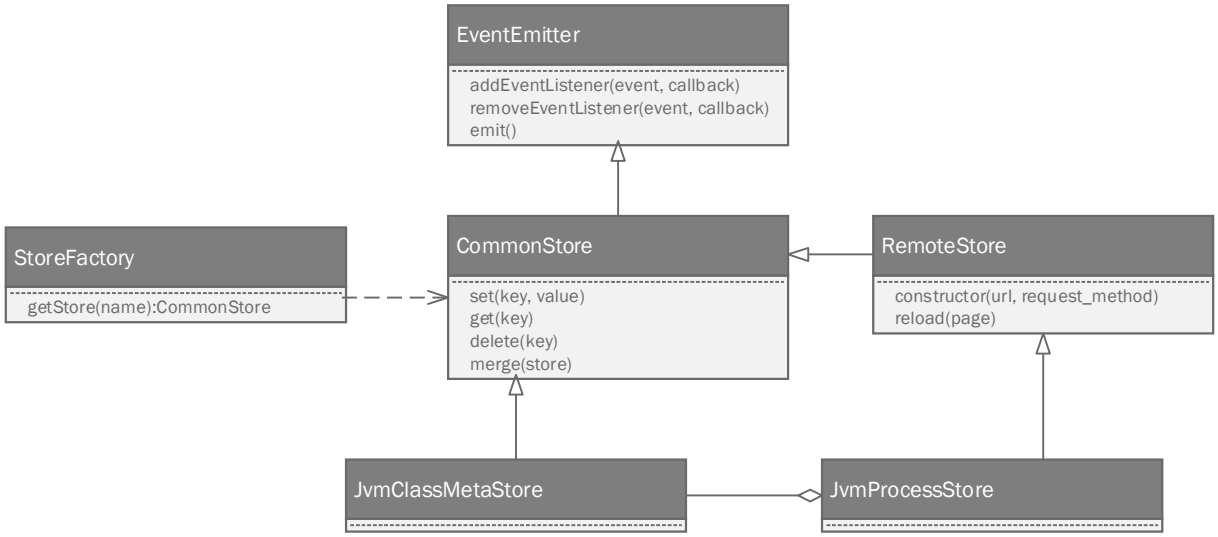

Figure 6-2 Design of Data Stores

In stores, all the data are maintained as key and value pair. The value can be any data type in JavaScript, such as number, string, an object or an array. For the object values with complicated data structure, it may become very difficult to track the mutation. To solve this problem, we are using immutable collections to store data by the implementation of the Immutable.js [38]. Particularly, we are using immutable map to store the object and immutable list to store the array. The immutable collection cannot be changed after it created. For

example, when we add a new element to an immutable map, it would not update the original map but create a new map which contains all the elements in the original map as well as the new element. By doing so, for all the stores, there is only one event called “change” needed to handle all the mutation happened in the current store. And the mutation can be detected easily by shallowly check the object or array reference.

6.1.4 Views and Components

Figure 6-3 depicts the design of views and components. All of the views and components inherit from React.Component [39] which helps render html elements to an html page. The views such as SeqDiagramView provide an abstract container for the components. Each component can maintain couples of properties such as stores or other useful information. For easy mutation detection, all the components can only take immutable data as properties. ImmutablePropComponent implements the mutation detection and re-rendering logic based on shallow object comparison, which is the super class for all the components. PureRenderComponent enhance the ImmutablePropComponent by adding inner state supporting, which is useful for the multi-stage component such as SeqDiagram. Each component may associate with a store which maintain the data needed by the component.

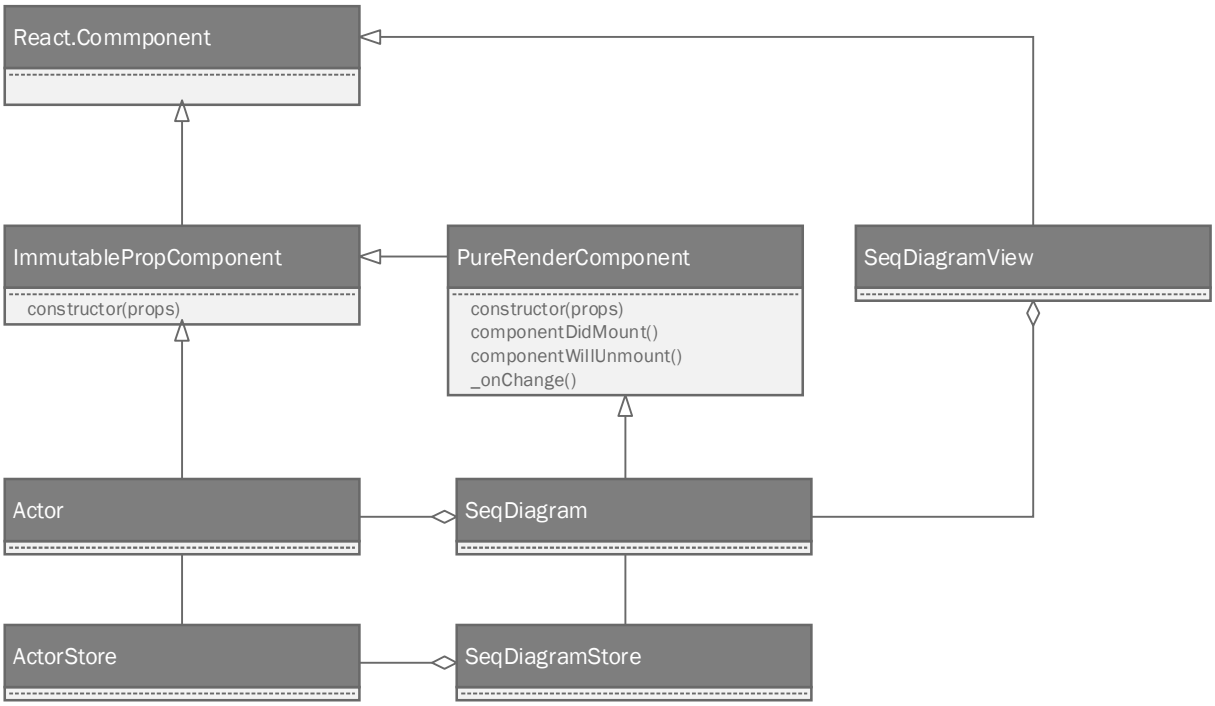

Figure 6-3 Design of Views & Components

All the sub-classes of ImmutablePropComponent only shallowly compares the objects (the properties passed). As we mentioned in the previous section, for the complicated data structures, the immutable collection is used for mutation detection.

By using the components, we defined above, we don't use a template to rendering the page. Using template technologies currently is the mainstream of the html element rendering, which provides a clean separation between the presentation tier (view and component) and business logic. It's very useful and handy to render the html page with the data which follow a fixed structure like a blog page. However, most of our views are rendered based on the data which may can be varied based on our Execution Data. To support this kind of data, we have to put lots of logic expressions such as if statement in the template, which break the original purpose

of using template. In addition, a small portion of the data keep updating during the animation, which will be difficult to use template to just update a small portion of html page.

6.2 User Interface Design

6.2.1 Enriched Sequence Diagram

In this section, we discuss how JDevv to deterministically simulate the execution of Java programs on an enriched sequence diagram.

Sequence diagram is a diagram in software engineering that has been accepted by the majority of programmers. Traditional sequence diagram shows how objects pass messages (method invocation or assignment) to one another and in what order. So it is useful to show the sequence of message passing among objects. However, the traditional sequence diagram cannot illustrate multiple threads on the same diagram.

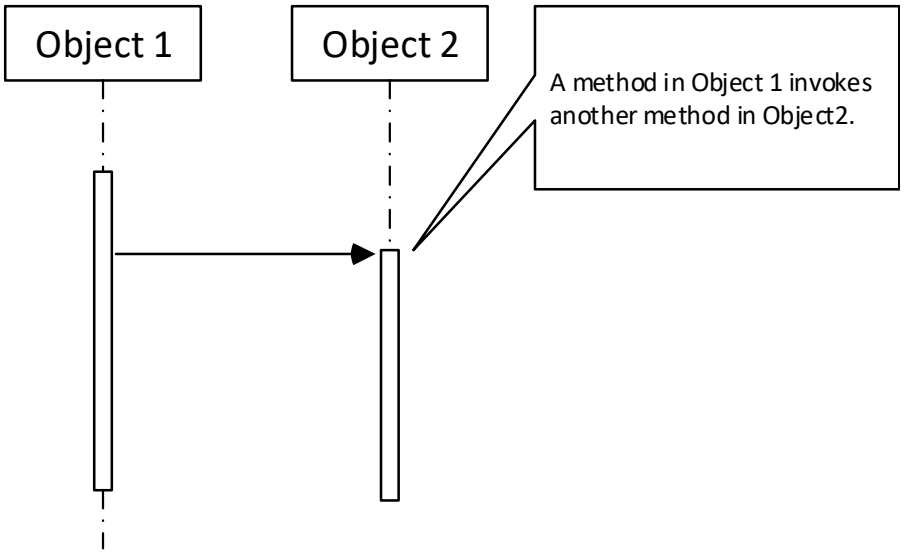

Figure 6-4 Traditional Sequence Diagram

In addition, the traditional sequence diagram cannot show the runtime details (e.g. intermedia results) in a method invocation.

6.2.1.1 Sequence Diagram Placement

Figure 6-5 shows how we enhance the traditional sequence diagram to support multi-threaded programs. The “object1” is shared by two threads, each of which invokes the method which belongs to “object2”. In the method of “object2”, there are a couple of internal signals such as field read and field write which are clickable to jump to the corresponding positions in the source code.

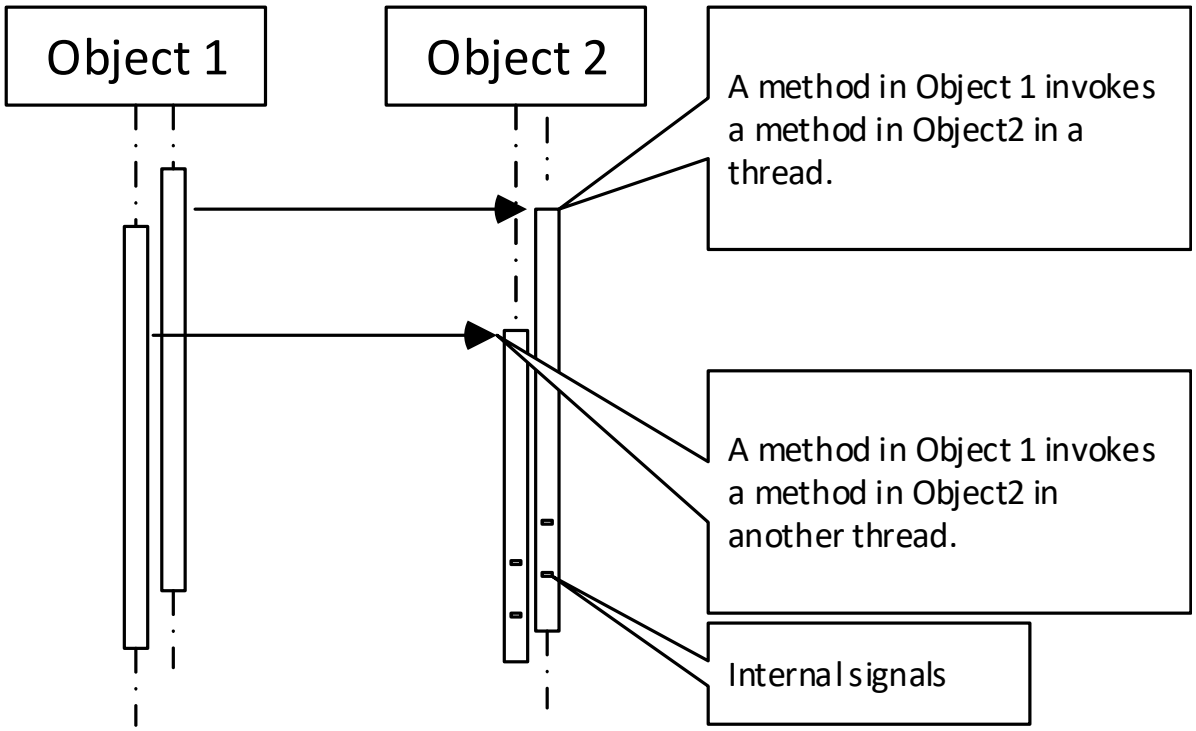

Figure 6-5 Enriched Sequence Diagram Placement

The enriched sequence diagram is also designed for the execution animation of the users' programs. Since the sequence diagram can show the global picture of the execution, user can use this diagram to navigate to the position of interest

6.2.2 Source Code View

Although the enriched sequence diagram is able to presents lots of the details about the execution, users still need to locate the problem found to the source code to fix the bugs. Figure 6-6 shows how the enriched sequence diagram (left side) synchronized with the source code view (right side) during the animation. A horizontal red line is provided to indicate the current signal in the sequence diagram, and the black bar on the source code indicates the position which generate this signal in the source code. If the user moves the red line on the left, the black bar will also jump to the corresponding position.

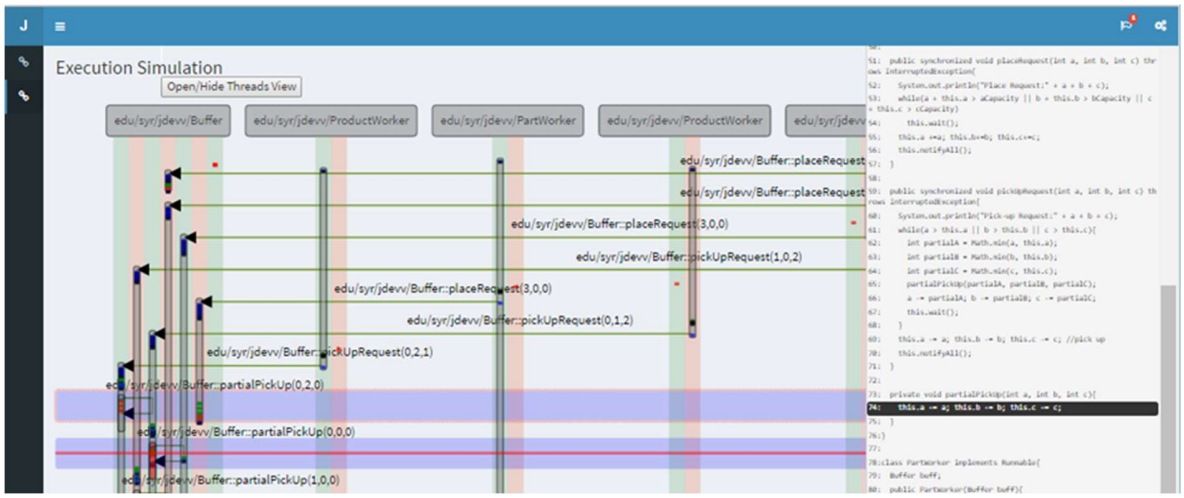

Figure 6-6 Source Code View synchronized with Sequence Diagram

6.2.3 Thread Status View

Sometimes the threads may be blocked by other threads forever (e.g. deadlock or starvation). Therefore, a thread status view is provided to show the status of all the threads together. Figure 6-7 shows an example about how to detect deadlock on the program showed in Figure 6-8 by threads status view. The thread2 and thread3 are blocked by waiting for a monitor lock to enter a synchronized block (line 002 – 006 in Figure 6-8). Meanwhile, the thread1 obtains the monitor lock and then enter the synchronized block. In the synchronized block, thread1 is checking whether “v” is ready to do something. If not, then the thread1 will start waiting. If a thread is waiting, the lock gained by this thread will be temporary released. Thus, the thread2 can get the lock and then enter the synchronized block. Similarly, the thread2 starts waiting and then thread3 gains the lock. If the “v” is still not ready, then all the threads will be in waiting status. Figure 6-7 shows that our thread status view can clearly show this situation. In addition, each cell in the table is clickable which can locate the problem to the source code.

Thread 1	Thread 2	Thread 3
Running	Running	Running
Acquire	Running	Acquire
Running	Acquire	Blocking
Waiting	Blocking	Blocking
Waiting	Running	Blocking
Waiting	Waiting	Blocking
Waiting	Waiting	Running
Waiting	Waiting	Waiting

Figure 6-7 Threads Status View

```
001     public static void foo(Object v) {
002         synchronized(v){
003           while(!v.isReady())
004              v.wait();
005           v.doSomething();
006           v.notify();
007         }
009         return;
010     }
```

Figure 6-8 Deadlock Example

6.2.4 Array/Fields Value Chart

In the source code view the variables are able to be clicked to produce a line chart for users to watch the trends of a group of variables based on a human-cognition model. For instance, as Figure 6-9 shows, the user could select some of the fields to watch if the variable clicked is an object. Moreover, users can click the points on the line chart to navigate to the code where the modification happened.

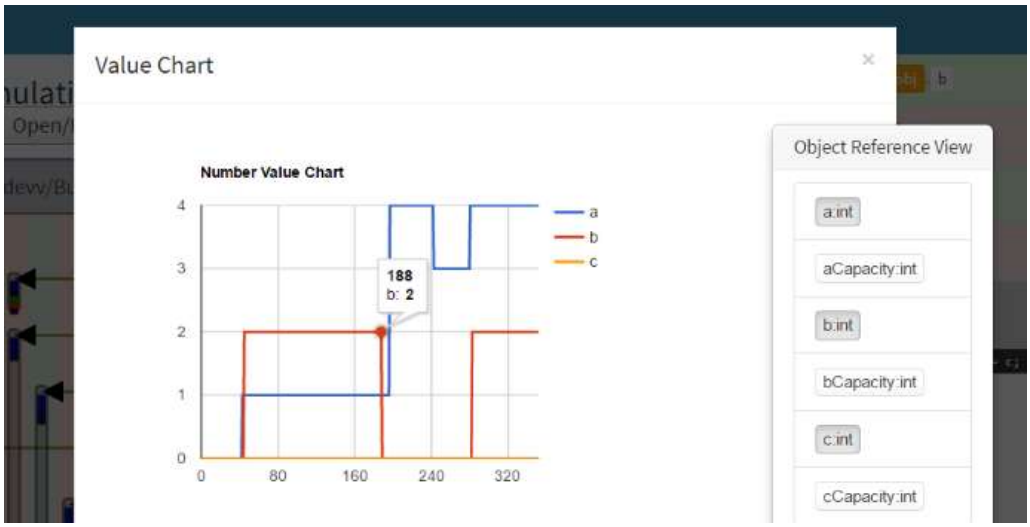

Figure 6-9 Field and Array Value Chart

Chapter 7. Conclusions and Future Work

In conclusion, our all-in-one debugging approach is not only an academic theory with wide applicability but also it can be, and has been indeed, implemented in practical terms into an efficient debugging tool, i.e., JDevv in this case. With the design as we have proposed, JDevv can be readily put into use for both academia and industry. Especially for the latter JDevv can meet almost all practical needs as it allows quick implementation, teamwork environment, strong debugging capabilities especially for complicated programs with enormous data, high performance and ease to get hands on.

Notwithstanding, we still need to conduct more research on how our debugging approach can be implemented for other programming languages such as C/C++, Objective-C and Swift as they are either prevalent as Java or trending nowadays. And although we have already had certain clues in this respect including that we may develop an alternate Run-time Module and Code instrumentation module to support LLVM, which in turn can support the said languages, further work can surely be done toward these valuable goals.

Bibliography

[1] M. Russinovich and B. Cogswell, “Replay for concurrent non-deterministic shared-memory applications,” *ACM SIGPLAN Not.*, vol. 31, no. 5, pp. 258–266, May 1996.

[2] M. Burgess and J. Bergstra, *Promise Theory: Principles and Applications (Volume 1) 9781495437779*. 2014.

[3] D. Merkel, “Docker: lightweight Linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, p. 2, Mar. 2014.

[4] M. Zaharia, M. Chowdhury, and M. Franklin, “Spark: Cluster Computing with Working Sets.,” *HotCloud*, 2010.

[5] S. Tilkov and S. Vinoski, “Node.js: Using JavaScript to Build High-Performance Network Programs,” *IEEE Internet Comput.*, vol. 14, no. 6, pp. 80–83, Nov. 2010.

[6] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. 2012.

[7] S. Weigert, M. Hiltunen, and C. Fetzer, “Mining large distributed log data in near real time,” in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques on - SLAML ’11*, 2011, pp. 1–8.

[8] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, “In-situ MapReduce for log processing,” p. 26, Jun. 2011.

[9] J.-D. Choi and H. Srinivasan, “Deterministic replay of Java multithreaded applications,” in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT ’98*, 1998, pp. 48–59.

[10] K. H. S. P. W. V. Guillaume Brat, “Java PathFinder - Second Generation of a Java Model Checker.”

[11] J. J. Cook, “Reverse Execution of Java Bytecode,” *Comput. J.*, vol. 45, no. 6, pp. 608–619, Jun. 2002.

[12] P. Moret and P. Moret, “Advanced Java Bytecode Instrumentation,” no. January 2016, 2007.

[13] and G. R. Havelund, Klaus, “Monitoring Java Programs with Java PathExplorer,” *Electron. Notes Theor. Comput. Sci.*, vol. 55, no. 2, pp. 200–217, 2001.

[14] F. Chen, “MOP : An Efficient and Generic Runtime Verification Framework ∗,” *ACM SIGPLAN Not.*, vol. 42, no. 10, pp. 569–588, 2007.

[15] M. Karaorman and J. Freeman, “jMonitor: Java Runtime Event Specification and Monitoring Library,” *Electron. Notes Theor. Comput. Sci.*, vol. 113, pp. 181–200, Jan. 2005.

[16] “Chronon | DVR for Java.” [Online]. Available: http://chrononsystems.com/. [Accessed: 01-May-2016].

[17] “Takipi - Detect and fix production errors.” [Online]. Available: https://www.takipi.com/.

[Accessed: 01-May-2016].

[18] “Java SE 7 Java Platform Debugger Architecture (JPDA)-related APIs and Developer Guides.” [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/. [Accessed: 01-May-2016].

[19] P. Gestwicki and B. Jayaraman, “Methodology and architecture of JIVE,” in *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis ’05*, 2005, p. 95.

[20] “JavaParser.” [Online]. Available: https://github.com/javaparser/javaparser. [Accessed: 25-Jan-2016].

[21] E. Begoli and J. Horey, “Design Principles for Effective Knowledge Discovery from Big Data,” in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 215–218.

[22] “Promise - JavaScript | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed: 27-Jan-2016].

[23] P. P.-S. Chen, “The entity-relationship model---toward a unified view of data,” *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976.

[24] M. Burgess, “Ambient Networks: 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2005, Barcelona, Spain, October 24-26, 2005. Proceedings,” J. Schönwälder and J. Serrat, Eds. Berlin, Heidelberg: Springer Berlin

Heidelberg, 2005, pp. 97–108.

[25] “Underscore.js.” [Online]. Available: http://underscorejs.org/.

[26] “MongoDB Connector for Hadoop.” [Online]. Available: https://docs.mongodb.com/ecosystem/tools/hadoop/.

[27] “Express - Node.js web application framework.” [Online]. Available: http://expressjs.com/. [Accessed: 03-May-2016].

[28] “Aggregation — MongoDB Manual 3.2.” [Online]. Available: https://docs.mongodb.org/manual/aggregation/. [Accessed: 27-Jan-2016].

[29] A. B. Kahn, “Topological sorting of large networks,” *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[31] E. Dede, M. Govindaraju, D. Gunter, R. S. Canon, and L. Ramakrishnan, “Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis,” *ScienceCloud*, p. 13, 2013.

[32] B. TVERSKY, J. B. MORRISON, and M. BETRANCOURT, “Animation: can it facilitate?,” *Int. J. Hum. Comput. Stud.*, vol. 57, no. 4, pp. 247–262, 2002.

[33] C. Gonzalez and Cleotilde, “Does animation in user interfaces improve decision making?,” in *Proceedings of the SIGCHI conference on Human factors in computing systems common ground - CHI ’96*, 1996, pp. 27–34.

[34] J. Heer and G. G. Robertson, “Animated Transitions in Statistical Data Graphics,” *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1240–1247, Nov. 2007.

[35] “JSX | XML-like syntax extension to ECMAScript.” [Online]. Available: https://facebook.github.io/jsx/.

[36] “Flux | Application Architecture for Building User Interfaces.” [Online]. Available: https://facebook.github.io/flux/docs/overview.html.

[37] M. Bostock, V. Ogievetsky, and J. Heer, “D^3 Data-Driven Documents,” *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[38] “Immutable.js.” [Online]. Available: https://facebook.github.io/immutable-js/.

[39] “A JavaScript library for building user interfaces - React.” [Online]. Available: https://facebook.github.io/react/.

Vita

Summary

A twelve-year back-end developer with a focus on improving both performance and scalability of web and cloud applications.

A researcher of Java bytecode instrumentation and JVM runtime verification & profiling.

Six years of MongoDB data model design and performance tuning.

An experienced “scrum” player in agile development or lean startups.

Education

08/2012 – 12/2016 PhD Candidate in Computer Engineering Syracuse University, USA

Designed and implemented a cloud-based debugging tool for large volume distributed and concurrent J2EE system via Java Bytecode Instrumentation

(ASM), Logging Data Processing (MongoDB & Spark) and Logging Data Visualization (html5, svg, d3.js and react.js).

02/2005 - 06/2007 Master in Computer Engineering (Part-time) Fudan University, China

08/2000 - 06/2004 Bachelor in Computer Science Fudan University, China

Experience

04/2010 - 07/2012 Architect & Director of Engineering eDoctor Healthcare Information, China

Providedarchitecturesofthecloud-basedproductsusingtechnologiesincluding Java, Scala, MongoDB, Rails, Solr and Html5.

Specifically, designed and implemented a distributed documents service which had crawled, parsed and indexed more than forty million online healthcare publications without a crash in six months; an event voting system which had supported a 1500-person conference; and a general testing framework for varied web apps via automating web browsers.

Led a 40-person team through an agile/scrum transition.

04/2009 - 04/2010 Senior Java Developer Starcite Shanghai Branch

Led the agile development of the user-account and contact services, the two core web-services for meeting/travel planning to be accessed by both the planners and the hotel/meeting suppliers.

Designed a full-text search service using Solr and Apache CXF.

TunedtheWeb-servicesperformancethroughthesourcecodeofJavaServlet (Tomcat), XFire and iBatis.

04/2007 - 04/2009 Architect & Co-Founder YepWorld (Self Start-up enterprise) Shanghai

Developed a college students targeted crowd-sourced traveling sales website

YepWorld.com with PHP, Memcached, Lucene and XFire.

01/2005 - 04/2007 Senior Java Developer Wuerth Phoenix (German), Shanghai Branch

Designed the persistence layer and optimized the data query of the cross-module tasks “purchase reservation”, a major part of the ERP product.

Designed and implemented an automation testing framework for asynchronous workflow nodes using JMS.