Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jakob Joachim

Methodology for Debugging Flink Applications

Fakultaet Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jakob Joachim

Methodology for Debugging Flink Applications

Bachelorarbeit eingereicht im Rahmen der BachelorprÃ¼fung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der FakultÃ¤t Technik und Informatik
der Hochschule fÃ¼r Angewandte Wissenschaften Hamburg

Betreuender PrÃ¼fer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 1. Januar 2345

Jakob Joachim

Thema der Arbeit
Methodology for Debugging Flink Applications

Stichworte
Schluesselwort 1, Schluesselwort 2

Kurzzusammenfassung
Dieses Dokument …
Jakob Joachim

Title of the paper
Methodology for Debugging Flink Applications

Keywords
keyword 1, keyword 2

Abstract
This document …

Contents

Listings

1 The Art of Debugging

As debugging of traditional programs is in many ways similar to debugging Flink applications it is necessary to explore what good debugging is. This chapter is an accumulation of techniques and methods that many people consider essential to debug effectively.

1.1 Word Count Application

This section quickly outlines the "Word Count Application" that is used in the thesis as an example. The Application counts how often each word is in a specific text and returns the results to the command line. An example run of the program would be:

```
$ java -jar wordCountSimple.jar "Hello hello how do you do"
how: 1
hello: 2
do: 2
you: 1
```

Note that the application ignores capital letters in the input. This will be relevant later on.

1.2 The Java Debugger

Flink applications are very complex Java or Scala programs that still share a lot of the characteristics of typical Java programs. As such most methods of debugging Java still apply to Flink and should be used. This chapter will outline some of the ways that help Java developers to find their bugs faster and with less of a hassle.

1.2.1 Basics of the Java Debugger

To understand how the java debugger works we have to first look at how Java runs applications. Java applications run on the JVM, the Java virtual machine, to gain access to the information of the running program, the virtual machine needs to be reachable. Java provides interfaces for these communications, in general, the IDE implements the Java Debug Interface (JDI), and the

JVM Software implements the JVM Tool Interface (JVM TI). To allow communications between the two interfaces, a Protocol is necessary as the two interfaces are not running on the same (virtual) system. Java uses the Java Debug Wire Protocol (JDWP) for that, which specifies which byte holds which information on the byte stream. When launching an application in the "debug" mode, the IDE also starts a JVM TI alongside the application in the JVM and provides the JVM TI with the breakpoint locations. These breakpoints can either be line numbers or method heads that only trigger when reaching that precise method. Once the program hits the particular method (or line), the JVM TI stops the application and notifies the JDI. In general, IDEs provide information about the state of relevant variables; this is done by the IDE itself as it has to send a request to the JVM TI for each variable the user might want to see.

1.2.2 Debugging Principles

There are a few simple principles that help not only to find errors in the code easier, but also to make sure that similar errors won't occur in the future. This section will lay out some of these principles.

Debugging Mindset

The mindset plays a significant role in debugging; often developers see debugging as an annoyance and try to get away from it as fast as possible. Although it is the goal of good debugging to minimise the time a developer spends on it, it should be seen as a way to learn something about why the error occurred in the first place. This learning reduces the time he has to spend searching for a similar mistake later on.

The Programmers Fault

Another important step is to understand that it's most probably the developer's fault when an error occurs. Thus the programmer should always first look into his code to find the bug, only if that fails he should start looking into other possible reasons. It is a common mistake not to recognise one's fault and should always be considered when debugging.

Last Change

Often the error occurs at a piece of code that is not responsible for the error itself. Zeller (2009) describes program errors as follows:

1. A programmer creates a defect in the code.

2. The defect causes an infection.
3. The infection spreads
4. The infection causes a failure

It is often best to first look at the last change that was made in the code, as the failure probably originated there. Obviously, that doesn't always have to be true, as there are multiple other reasons why the failure only just now got discovered.

1.2.3 Conclusion

It is important to note that this section was merely a refresher on debugging and is not enough to understand it in its completeness. For further information on the topic consider reading Zeller (2009). The book covers the subject of debugging much more in depth.

1.3 Why Programs Fail

This section will examine the "TRAFFIC" method of debugging that is published in Zeller (2009)'s book "Why Programs Fail". It consists of seven steps that lead to a quick removal of the failure while still preserving the information on how the failure came to be, as a big part of debugging is not to fix a failure but to make sure that the same or a similar failure won't happen again. The seven steps are:

1. Track the problem in the Database
2. Reproduce the failure
3. Automate and simplify the test case
4. Find possible infection origins
5. Focus on the most likely origins
6. Isolate the infection in the chain
7. Correct the defect

The following subsection explains each step of the method along the example program outlines above 1.1

1.3.1 Track the problem in the Database

Tracking is not the first step, but a good method at each step, logging what happened so that anybody involved knows how far each problem was investigated. It is started once someone finds a problem. Put simply tracking is holding the information on what the problem is and how close the developer is to fix it. This is mostly done on a platform that is accessible for all involved parties to make the communication between the user and the developer easier. This platform is only useful when used permanently as out of date information is more harmful than useful. Most software projects have multiple people working on them, not all of them know every bit of the program. This adds complexity to an already complicated process. It is necessary to split the work to different developers and this process has to be logged otherwise problem reports might get lost. A developer might have 20 problem reports on his desk of which he only knows how to solve a few. Other reported problems might not even be problems, but a wanted state. For example, someone might report that a password field is showing only stars instead of the letters he put in. As this (for the developer) is a security feature, it will not be changed and has to be logged or noted so that the same problem will hopefully not be reported again. To solve these and other issues tracking should be used. Depending on the application these metrics are useful:

1. The State of the Problem - Is the problem new, assigned to a developer, resolved, closed, etc. This is useful for the developer as he can easily see which problems he has to work on and which are already solved. It is also beneficial for the user as he can easily see when his problem is resolved.

2. The Resolution - Is the problem fixed, invalid, won't be fixed (as the example above), a duplicate, etc. This is useful as it lets the user see to which conclusion the developer came.

3. Assigned Developer - Which developer is assigned to the problem. Makes it easy to communicate with the correct person and lets people know that the problem is worked on.

4. Severity - Is the problem crucial or is it only a minor inconvenience. Helps the developer prioritise which problems to solve first.

These are just the most important once, depending on the project more should be added.

1.3.2 Reproduce the failure

The first real step in any debugging activity is to consistently reproduce the problem described in the problem report. This has two important reasons:

1. To observe the problem - The developer has to be able to reproduce the problem to fix it, the developer could also check the source code at a position he thinks could be responsible for the problem without reproducing it, but that makes it unnecessarily hard on the developer as a good problem report should be re-creatable.

2. To check whether the problem is fixed - It is incredibly hard to tell in most situations if a problem is solved or not without being able to rerun the problem without it happening.

Reproducing a problem can be incredibly hard, as the problem is rarely found by the person that has to fix it, but by an individual who doesn't understand how the program works. This makes it difficult for both the finder and the fixer of the problem, as the finder doesn't know which information the fixer might need, and for the fixer as he can only reproduce it with the corresponding information.

Reproducing is done by going through the following three steps until the problem can be reproduced:

1. Reproduce the Problem locally - In the best case scenario, the problem can be replicated on the local machine of the developer fixing it with the information provided by the problem report. This is most successful when the problem state is not connected with many other choices in the program. For example, a button might not work (as reported in the problem report) that starts some function no matter what else happened before the button was pressed.

2. Adopt more Circumstances of the Problem environment - Sometimes the problem can not be reproduced by only following the steps provided by the problem report. In that case, it is necessary to check what else is known about the environment the problem was found in. This means installing the same version of depending software, using the same configuration file, using the same hardware or anything else that could influence the described problem.

3. Contact the Problem Finder or declare the Problem invalid - If the reproduction failed even with all circumstances applied as outlined in the problem report, there are only two options left. Either the problem is not real, or the finder has not provided the necessary information. Depending on the bug this can be solved by contacting the finder and

asking him for more details. It is beneficial for the developer to see the problem live, so it would be a good option to remotely gain access to the computer of the finder or get a video meeting with him. This is obviously not possible or feasible in many situations but extremely helpful if it can be arranged.

At this point, it is important to mark what information were necessary, so that another developer can reproduce the problem easier as well as saving the information as it is not guaranteed that the problem will be solved immediately. This is done with tracking: see 1.3.1

1.3.3 Automate and simplify the test case

Once the problem is reproduced, it is desired to simplify the problem so that it can be replicated by a test case that as well as confirming that the problem is solved, helps avoid building a similar problem at a later time.

Simplification is done by understanding what the root of the problem is. When the problem occurs when pressing a particular button, but the button has no context to other set states of the program it is safe to assume that actions done before the button click can be omitted. Thus leaving only the test case: After button press, function x started? This statement can now be easily modelled by a test case.

The simplification is not only essential for building test cases but also helpful for the developer when trying to locate the problem in the code as it is much easier to find the error when only looking at the relevant pieces of source code.

Why write tests? It is understandable to ask why to write a test when it is already known where the bug is originating from. The developer only has to fix the problem and manually test the program once, what good is a test case here? It is important to understand that finding a problem is giving a lot of valuable information to the developer he might not even realise. We can assume that if the problem got into the code once it might get into the code again. The only way to make sure it does not is to give the developer an immediate feedback when the problem occurs again. As well as helping the Developer in the future it can assist the developer while writing the fix to quickly test if the fix worked or not without having to manually start the application and reproduce the problem all over again.

```
1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class WordCount {
5     public static void main(String[] args) {
```

```
6           Map<String, Integer> result = new HashMap<>();
7           String[] words = args[0].split("\\s+");
8           int numberOfWords = words.length;
9           System.out.println("There are " + numberOfWords + " words in total.");
10          for (String word : words) {
11              int count = result.getOrDefault(word, 1);
12              result.put(word.toLowerCase(), count + 1);
13          }
14          int uniqueWords = result.size();
15          System.out.println("Of these " + numberOfWords + " words " + uniqueWords +
16          System.out.println("The unique words are: ");
17          for (String word : result.keySet()) {
18              System.out.println(word + ": " + result.get(word));
19          }
20      }
21  }
```

1.3.4 Find possible infection origins

Once the failure can be reproduced easily, it is time to search for what part of the code is responsible for the failure. This is often the hardest part of debugging. To make finding easier for the developer the traffic approach suggests to use backtracking to find the relevant piece of code. Backtracking is done by starting at the manifestation of the error, meaning if the program fails with an exception, the line noted in the exception. If the program just produces an unwanted or wrong value, the line is used where that value was returned. Once a starting point is found the next step is to back track all active and passive usages of the variable. As failures can quickly propagate through the application it unfortunately not enough to only look at the variable that was causing the exception we have to also look at all other variables that interact with the first variable and so on. Lines that don't use the variable can be omitted as they cannot be responsible for the failure. The information gained by this technique should be displayed by a control flow graph to make them easier to understand. The graph can often be omitted but should be used once the problem space gets too big to handle without a graph.

1.3.5 Focus on the most likely origins

Once the control flow graph is done, the developer should look at the most likely sources. This can be quite difficult as it is subjective to the developer what the most likely origins are. Some places to check can be:

1. Last Change in the Application - Often problems occur after a change in the software thus it is very likely that problem is a result of the modification. It is, of course, possible that the problem was there before but got propagated through the new piece of code. In that case, it still makes sense to look into the new code.

2. Check for common problems in the framework - when working with a framework or any other software it is a good idea to check if the problem is related to the framework and has been solved by other people already.

If both suggestions don't work out, it should be thought about where the program is the most complex as a developer is more likely to make a mistake in a complicated part than a simple one.

1.3.6 Isolate the infection in the chain

Isolating is done by creating a hypothesis and checking if it is true, then repeating the process until the problem is solved. A sample run would be:

1. Hypothesis - The application creates a faulty value.

2. Prediction - The faulty value is set to the particular variable in the expected line.

3. Experiment - Using the debugger the prediction is confirmed.

4. Observation - The faulty value is set to the variable.

5. Conclusion - The hypothesis is confirmed.

As the problem is not located yet a new hypothesis needs to be thought of. It is common to check if the failure is present at an earlier stage.

1. Hypothesis - The infection does not occur until function Y is called

2. Prediction - The variable should hold a sane value before Y is called

3. Experiment - Using the debugger the prediction is confirmed

4. Observation - The variable is already wrong

5. Conclusion - The hypothesis is rejected

As the infection is already present at the function call, the function should be examined

1. Hypothesis - Invocation of function Y with the faulty value causes the problem
2. Prediction - If the function is called with the correct value the program runs correctly
3. Experiment - Using the debugger the function call is called with a correct value
4. Observation - The program runs correctly
5. Conclusion - The hypothesis is confirmed

This method has to be repeated until the root of the problem is discovered. The advantage of this method is that the problem will be identified eventually.

1.3.7 Correct the defect

Once the defect has been found, it can be corrected. But solving the defect is not enough as it is also necessary to check if the problem is now solved. There are two reasons why this is necessary: enumerate

The defect was not responsible for the error - It is common to find other bugs while investigating a problem as the code is scrutinized. Another option is that the problem consists of two defects, not one. For both these reasons, it is necessary to check whether the defect was solved or not.

Is the cause an error? - Sometimes the deducted cause of the problem is in fact not the cause, and the correction is just another defect that fixes the problem in the particular problem case. To make sure that the cause really was the cause it is recommended to think about the correction and if it solves the problem for all cases without compromising all other working cases.

Once all of these steps are done the only thing left to do is to mark the problem as solved in the problem report and add it to the next patch of the software.

2 Related Work

The goal of this section is to give the necessary information to understand the rest of the thesis. Debugging is a complicated process that is made much harder by distributing the system to more than one physical location. To better understand the methods and principles, this chapter will use a simple example application to comprehend the differences in each method easily.

2.1 Flink Framework

The Flink Framework is a stream processing framework for the JVM. It is written in java and scala and can be used with a variety of languages and technologies. As is to be expected the most supported languages are java and scala, although there is also a python wrapper and a few more available. The following section will clarify how exactly a Flink application works and what kind of debugging and logging tools it already has.

2.1.1 Basics

Flink is a Framework for processing applications that are distributed across multiple computers. This part will lay out the foundations of the Flink Framework. Flink applications have a simple base structure that is used by the developer. Each application defines tasks which have a particular structure. Each task has an input and output stream. These are called source and sink. Because both the source and sink of these tasks are streams, they can be attached to each other thus creating a data flow from task to task until the wanted end state is reached.

Figure 2.1 shows a basic program flow. On the left side, it starts with a source. The source is then transferred over a data stream to the first task. The map operation is executed, and the result is sent over a data stream to the next task where the same procedure will start again. The resulting data in this example is the sink, meaning we reached the end of the program. The sink will typically be connected to a database or some other kind of Technology for preserving the data. The most basic option would be to write the sink onto the standard output on the console.

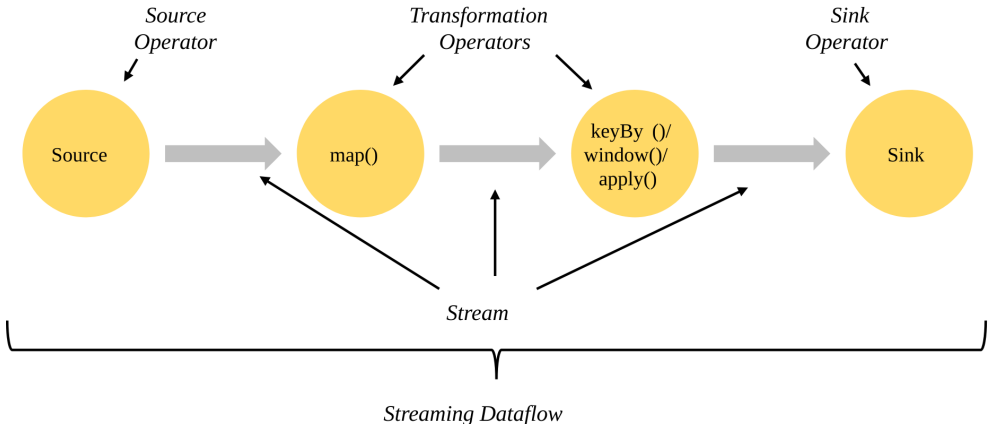

Figure 2.1: Simple Dataflow in Flink

Until now the system can only be distributed by having the different tasks on different physical computers. This distribution process will not suffice when the amount of input data gets too high. To achieve real distribution in the application, each task can be run multiple times as so called subtasks. These subtasks run as a single thread in the JVM and are managed by a task manager. Task Managers connect to a Job Manager which coordinates the distributed execution. The data flow from one subtask to another can either be one to one or as a redistributing data flow. A redistributing data flow is necessary to achieve an even distribution of data at the next subtasks.

Figure 2.2 presents the same example as figure 2.1 just with the subtasks shown as well. The map task is a vital step in each application as it connects the data from each subtask with each other. It is easily understood by a simple example. In an application that counts how often each word is in a text, it would only split the text at each space symbol. The redistributing data flow would then create an even distribution in the next node which specifies what to count by (id task) and applies a particular aggregation function to the "apply" operation. The result is then sent to the sink operator. This explanation is a simplification of the actual MapReduce model. For a complete overview see Dean und Ghemawat (2008).

2.1.2 Stream Processing and Batch Processing

Apache Flink is primarily a stream processing framework, although it can also be used for batch processing. As it makes a big difference in the infrastructure of the framework which process the primary one is, it is important to lay out the differences between the two.

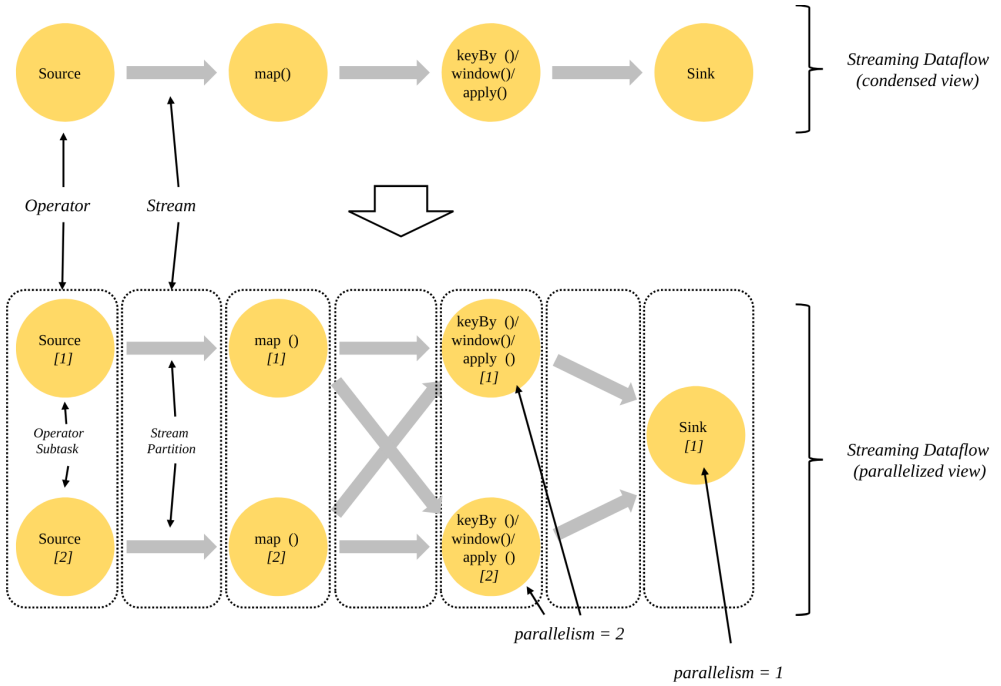

Figure 2.2: Distributed Dataflow in Flink

Stream Processing uses as the names suggest a stream to acquire the data. A stream is an endless sequence of data that can be utilised both as an input and output. As there is no end, it makes it tough to use some algorithms on it. For example, an algorithm that finds the highest number from the input. Flink uses "Windows" to solve this problem. These windows offer a frame for the operations to only use data that was received in each window. Windows can be programmed and provide various ways to define the frame. The simplest way would be just to give a timeframe, but more complex structures could also be built.

Batch Processing on the other hand has set boundaries, and it is evident how much data is sent and where it ends. To support batch processing as well in the Flink Framework the windows can be programmed to have the same size as the data in the batch. That way even though Flink uses streams it still supports batch processing.

2.1.3 Debbugging and Checkpoints

Flink comes with some tools that help the programmer debug his application and help prevent errors stopping the application. This section will highlight what Flink does different or on top of the regular java/scala debugging features.

Metrics

In a distributed application it can be hard to find out why something is not working properly or why some function performs worse than expected. To help the programmer understand these problems Flink provides metrics, these count or measure throughput on specific points in the application and send this information to a reporter. The reporter provides the information to external applications so that the metrics can be analysed as needed. There are four different metric classes available, a counter that can be in- or decremented, a gauge that can provide the value of a particular variable, a histogram which measures the distribution of long values and lastly a meter that measures the average throughput.

As applications could have a huge number of metrics which would make it very hard to find anything, it is important to have some grouping mechanism. Flink provides scopes to solve this. There are two types of scopes, user-scopes, defined by the user, and system-scopes that hold current information about the system state like the task in which the metric was saved. When a metric is registered, an identifier and a system scope have to be specified, a user-scope can optionally be added.

In addition to the metrics above, Flink automatically collects system information like ram usage, the number of threads, network usage and much more.

Checkpoints

In distributed systems it is quite common that parts of the system crash, this can have many reasons, the incoming data could be formatted wrongly, a data transmission could break away before it is finished, and so forth. Normally the application would log what went wrong and stop or restart the whole application. Depending on how large the application is this could cost a lot of time, flink creates checkpoints which it automatically falls back to if the application crashes. The way these checkpoints work is by periodically injecting barriers at the source. After a task receives a barrier at one of its inputs, it blocks that input until it received a barrier at all of its inputs. Once that happens it takes a snapshot of all the data it received since the last snapshot. This way it is guaranteed that every piece of information is part of a snapshot all the time. To keep these snapshots from taking up too much space on the hard drive flink only stores one snapshot for every task and overrides it with the next. For a more in-depth explanation of this algorithm see: Carbone u. a. (2015)

As well as these automated checkpoints Flink provides user-created checkpoints called savepoints. These can be set by the user and are not getting deleted everytime a new one is created. They are used to pause the application for example.

2.2 Similar work

Other people did similar work on another framework and distributed systems in general. This section will explore to what conclusions these people came and what can be applied to Flink as well.

2.2.1 BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark

Gulzar u. a. (2016). The University of California built a debugger for "Apache Spark" which focuses on similar points as this thesis. The motivating intention behind their debugger is to find the cause of the problem easily without having to go through millions of logs, and without stopping the system itself. BigDebug uses five different methods to solve this predicament. This section will not only lay these methods out but also show if they are already present in Flink.

Simulated Breakpoints are used to debug parts of the application without stopping the running system in every node. This is done by spawning a new process with the same beginning states as the remote one. After that the newly spawned process can be debugged without intervening in the running system.

On-Demand Watchpoints with Guard are user-defined methods in the system that can be set to inspect the value of a certain variable, check that value and store it in case the value of the variable fails the check. This method can be useful in certain situations for example when checking a variable that is supposed to hold a zip code to see all values that are malformed. This feature can easily be used in the Flink Framework with the use of metrics.

Crash Culprit and Remediation focuses on two things, first collecting interesting data in case of a program error so that the user can easily find out why the crash happened and secondly avoid rerunning the whole application when a crash happens. These two are grouped together as they are solved together as well. After a crash, the user will get the value that was responsible for the error, for example, the input could have been "23s" instead of 23. The rest of the application will continue to work and once the user corrects the value to "23" the program reruns this task. In Flink, this is partly implemented, as Flink creates automated checkpoints the application will not completely stop when it crashes as it would be in Spark. Flink offers no way to modify values that lead to a program error though.

Forward and Backwards Tracing is done to make it possible to track where a piece of data came from or where it will end up. This tracing is quite complicated as a piece of data in task X is a sum of all modifications that occurred before that task. The way this is implemented is by tagging each incoming piece of data with a unique identifier and adding all the identifiers that are used to create a new piece of data together. On the other hand, if data is split into multiple pieces, like splitting a sentence into words, every new piece gets a new id, and the relation between the old and the new id is saved. That way a programmer can easily backtrace to where a malfunctioning piece of data originated.

Fine-Grained Latency Alert is used to identify which records are causing delay. Baseline Spark can already measure the time between each task. BigDebug builds on this feature and extends it by making it available at each operator. This part is quite different in Flink as Flink offers no direct way to apply a metric to each task or operator.

Conclusion

BigDebug solves some challenges that a Flink developer has as well. It solves them only in the spark framework and suggests no methodology or procedure for debugging a spark application it just provides some tools for the developer to use.

2.2.2 Debugging Distributed Systems

Beschastnikh u. a. (2016): tries to lay out fundamental challenges developer face when building distributed systems and how to solve them.

They give an overview of seven approaches that can help build, validate and debug distributed systems. What follows is a summary of the seven methods.

Testing is crucial but can only help reveal some errors as testing every piece of the application is impossible.

Model Checking is a form of testing that automates the testing process somewhat by checking every possible input up to some upper bound in a predefined way. The way the model checker works depends completely on the model checker itself, as there are a lot of different options available. There are for example symbolic model checkers available that explore possible executions mathematically or black-box checkers that just run the application with various inputs. Model checking can be helpful as it can cover much more ground than

manually written tests, as well as check parts of the application developers, might not have thought of.

Theorem proving is a mathematical method of proving that the distributed system is free of defects even before writing a single line of code. It is mostly used for proving that the core of a new application is bug-free before spending lots of time building an application that might not work properly.

Record and Replay is a method for analysing a single execution of the application to gain insight into why an error occurs in a system that has non-deterministic events, as these change every time the program is run even with if the same input values are used.

Tracing tracing is the method of following data through a system. It has the same goal as Record and Replay have although it is easier to understand what is happening as only a subset of the data is shown.

Log analysis is a method for debugging black-box systems, but can be used on every application. It is done by applying algorithms to the logs to find bugs that are not easy to spot.

Visulization is used to make distributed systems more transparent. If a developer has a good visual representation of his work, it is much easier for him to find bad design choices that could lead to bugs.

Conclusion

The article provides some basic methods for debugging distributed systems, the analysis of which method is useful is short though.

3 Debugging Flink

This chapter is the centre part of the thesis. It explains in detail how the method works. It consists of two main sections, the first of which explores in detail what a developer should do while writing the program to minimise the work when debugging. This step is crucial as it is much harder to recreate problems locally as opposed to regular Java applications. The second part focuses on the debugging itself once a developer is informed of an error and has to figure out what is causing it.

3.1 Better Developing

Flink applications are run remotely and without an active user providing input as it is typical for a regular application. Not having a user makes it much harder to recreate the problem as we only have the log files and the stack trace as information. As such it is crucial to have all the information at hand when the program fails, or we notice discrepancies in the resulting data. The only way to make sure that the information is accessible once a problem is reported is to think about what data is necessary for the debugging developer while writing the program. Another issue is that some problems are unique to a distributed environment and won't happen when running on a local host. This section will explain what can be done while writing the program to make the debugging progress much easier.

3.1.1 Building Tasks

The Flink Framework promises a few features that try to set it apart from standard Java applications. These are referenced in 2.1. The one that sets these applications apart from standard java once while writing the application is the distribution. Applications should be built in a modular way to take full advantage of these features and also make the program easily debuggable. The development of these tasks should be done using a very similar set of rules as the Unix philosophy states. In fact, Flink modules are not too different to Unix command line tools, they both provide a service while taking an input/output in a predefined

way. For the Unix command line, this is the standard output, for Flink programs, these are input and output streams.

Unix Philosophy

The Unix philosophy is defined by Doug McIlroy, the inventor of the Unix pipe as follows: McIlroy u. a. (1978)

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Most of the points mentioned here are of some relevance for Flink programming as well. Each task should do one job and do it well. The next paragraph will look into why that is. The second point is necessary by default in Flink, each task has to use the provided streams to work. The third point is equally important if not more important in Flink applications as it is in Unix programs. Always test each Task individually to make sure it works as designed and has no flaws on its own, only then can the whole application work without any problems.

Dividing the program into various modules has a lot of advantages:

1. Easier to Develop - It is much simpler to develop a smaller application as it is much harder to lose track of what each piece of code should do. This reduced complexity in return reduces the likelihood of mistakes. Once the program is completed, it results in a more stable program that can be debugged easier.
2. Better Distribution - As every Flink task can run on a different computer the smaller the tasks are, the better the Flink Job Manager can distribute the load evenly between the available resources.

3. Checkpoints are easy to find - Checkpoints are a core piece of Flink technology. It allows the framework not only to jump back and repeat a failed run without having to restart the whole application but also provides information about which state the application is currently in. This is extremely helpful as a lot of Flink applications only end when the program is cancelled by the user.

The modulation of the program not only helps to achieve the advantages of the framework but also supports with debugging later as a lot of the information needed are gathered at the checkpoints.

Where to split the program

It should now be understandable that the programs should be divided into multiple tasks, the next question now is how to break the program to get a simple program where there are enough tasks but not too many as too many would lead to the opposite effect we want to achieve.

Why are too many tasks bad? When there are too many tasks, it gets even more complicated than when everything would be in its task as basically every line of code would be in a different place. Additionally, it wouldn't increase the performance as each task has some initialisation work that would diminish the performance gain achieved by distributing it perfectly.

It makes sense to use the Unix philosophy of having one task do one thing. In most cases there are some obvious choices as in most Flink programs data is modified or analysed each task could be one transformation of the data. It should also be stated that when ever possible the pre-existing transformation functions of the Flink framework should be used. Only in rare situations is it necessary to implement your own data transformation classes. The predefined classes have the advantage of being extensively tested in different conditions thus the risk of data going missing is extremely low.

3.1.2 Metrics

Now that the architecture of the program is done the next question is what metrics to use in which positions to achieve the optimal security.

Once the program architecture is finished, it makes sense to think about what kind of metrics can be used where. Metrics are used to monitor the program without having to debug it and are crucial in notifying the developer if something looks wrong. The first step is figuring out

where to use metrics. A good start is to look at the application in the worst possible way; what is the most likely area that an error will occur, after that it makes sense to surround the area with metrics that will catch and log the gathered data. Another great location for metrics is at positions where the through coming data is simple, and metrics can easily be implemented. This should be the case in between tasks. As optimally each task only does one thing it should be easy to check whether the starting and ending assertions are valid.

3.1.3 Logging

Logging in Flink is straightforward and can is used the same as in every other java program that uses log4j. As Flink already provides the necessary libraries to use log4j all a developer has to do is to write the logging config file. Although the logging process itself is the same as every other Java application, it should not be forgotten to use the different log levels that log4j provides. There is a lot of logging happening out of the box just by the Flink process itself. The six logging levels, from highest to lowest are:

1. FATAL - the highest logging level, should only be used when the application cannot continue to work because of an unexpected error.
2. ERROR - whenever an unexpected exception is thrown it should be logged.
3. WARN - warnings that could lead to errors later on. These are difficult to think of beforehand but if used correctly are very valuable for the debugging developer.
4. INFO - relevant information like successful database connections and other milestones in the application to let the reader of a log file understand at which point in an application the program currently is.
5. DEBUG - should be used to record relevant information along the way that could be useful to a programmer when debugging. This could, for example, contain values of variables like database connection strings.
6. TRACE - is used to let a developer searching for a bug understand the path the application took. Should be logged into a unique log file as it would flood every other one.

3.2 The Debugging Process

There are multiple reasons how an error might be discovered, the most common one being an exception in a log file. Another option is that the end user of the results finds that some of the

results are incorrect. Both of these cases require slightly different handling. An excellent way to start the debug process is by using a modified version of the Traffic approach introduced here: 1.3.

3.2.1 Track the Problem in the database

Tracking a problem is essential in every development cycle no matter in which language or with what framework and Flink is no exception. It is crucial for every developer to track the status of problems in Flink applications as it helps to minimise work. Along the already mentioned advantages in chapter 1.3.1, like having an easily accessible database of open problems and knowing which problems are more important than others, tracking the problems of flink applications offers some other advantages as well.

1. Having access to relevant log files.
2. Knowing how past problems were solved.
3. What relevant metrics were when the error occurred.
4. Which subtask of what task manager failed, allows seeing if problems only occur on one machine.

To achieve these advantages the tracking database needs additionally to the already mentioned fields in 1.3.1 a few additional columns. As soon as a problem is experienced the current log files should be saved so that it is easy for the developer to find the relevant lines in the log file even if he starts debugging months later. Secondly, for the same reason, all appropriate metrics should be saved as well.

The resulting columns now are:

1. Description
2. State
3. Resolution
4. Assigned Developer
5. Severity
6. Link to logs

7. Steps that were taken to resolve the problem
8. Relevant metrics
9. Task manager that was used

3.2.2 Reproduce the failure

Reproducing the problem is probably the most challenging part of debugging Flink applications. As there is no user to report the problem the only help the debugging developer has is information provided by the problem report from the last chapter.

There are two options how a problem is discovered, and both require different steps to reproduce the problem. The first and more difficult one is that an error in the resulting data is discovered without an exception being recorded in the log files. This means that the program is doing something different then what the developer expected when writing it. The second option to discover a problem is by having an exception showing up in the log file.

Faulty resulting data

This section will use the word count application as an example program to debug. Notice that the exact implementation of it is irrelevant at this point. The application is a black box as only the incoming, and resulting data are known. To always have the same expected result the following sentence will be used as an input each time: "Hello hello flink flink flink one two". The expected result would be:

```
1 hello - 2, flink - 3, one - 1, two - 1
```

The following text explains the process that is used in this case alongside a diagram: 3.1

The first question that has to be answered is "how much data is affected". Are only a few pieces incorrect or is everything faulty? Imagine the result of the application would be:

```
1 hello - 3, flink - 1, one - 1, two - 2
```

Each word has the count of the next word. So this would be considered as the second case "everything is faulty" even though the word "one" has the correct answer. This is important to note as sometimes a fault in a program can still result in a correct result. This first question just differentiates between a few faults and a majority of faults, so the debugging developer has to look at the whole picture and see if the majority of data is corrupt. An example for only a few faults would be:

Figure 3.1: Debbugging faulty data

```
1 Hello - 1, hello - 1, flink - 3, one - 1, two - 1
```

Here only one additional word was counted ("Hello").

If the first case is valid (all data is faulty) the next question that should be asked is why the problem is only now showing up. If the problem is observable for almost all the resulting data, surely it should have been noticed while testing the application. In most cases, the problem was either found during testing in which case the problem is already reproducible or was not

observable on the local test machine. That means that either the incoming data is different to the local one or that something is being executed differently on the remote network than on the local machine. As Flink is responsible for the distribution and everything is running in a JVM, it is implausible that flink is to blame. In most cases, the data on the server will differ from the local one. If that can be confirmed the only thing left to do is to update the local test data so that the problem can be reproduced locally.

The other option was that only some pieces of data were wrong. In that case, the process of reproducing the fault is entirely different. There are two options available. First, figure out what makes the faulty data unique in comparison to the other data. In the word count example, this would be the capital "H" at the beginning of the first "Hello". If this option is successful, the unique case can be added to the test cases, and the reproduction was successful. If on the other hand, the developer can't figure out why the one failing case is different to the others the tool that is written alongside this thesis can be used, it will be explained in detail later on. It can show which incoming data was leading to which result. In the example above it could show that the first "hello" was the result of the original sentence. As there is only one sentence in this example that is not very helpful, but in a more realistic use case, there could be millions of sentences where just a few have capital letters in them. Once the starting sentence is discovered, it can easily be reproduced.

The last step before moving on to the next section is to make sure that the reproduction works. There is little use in finding something in the code that supposedly is the problem only to find out later that the problem is something else.

The fault should now be reproducible on a local machine as the affecting test data was found. The only fault that remains are problems that only occur on the remote network and are not happening because of incoming data. Solving that problem is done by applying the scientific method explained in 1.3.6

Exception in log file

The most common way to discover a failure in a program is by discovering an exception. It makes no difference if this exception was found in a log file or directly in the developer's console. The steps that are necessary to reproduce these errors are often easier as well.

1. What kind of error was thrown? - The developer should always keep in mind what type of exception was thrown as it is much easier to find the origin of the failure when knowing what to look for.

2. Where was the failure thrown? - This information is the starting point for the search of how to reproduce the failure. It is included in the stack trace of the exception so it shouldn't be a problem finding it.

3. Isolate the conditions in the method - Knowing under which conditions the failure occurs is crucial. Without this information, it is almost impossible to reproduce the failure reliably as only a few errors occur under all circumstances. The developer should start by just focusing on the method the exception is thrown in and finding the conditions by looking at the "if statements" that precedent the line in question. Note in which state each variable has to be for the exception to be thrown. This can include the variable that is throwing the exception as well. In most cases, the developer should now already have a pretty good idea as to when the error occurs and if that is the case can skip the next step to save time.

4. Repeat the same step for the method - Once it is understood in which conditions the failure occurs in the given method, the developer has to find out in which cases the method is being called with the conditions that were deducted in the last step. This can be done by repeating the step above only for the line where the method was called, which is included in the stack trace of the exception as well. This step has to be repeated until the start point of the application has been reached. At this point, the developer should have a good understanding when the exception is thrown and should be able to reproduce it.

Conclusion

In most cases, the developer should have no problem reproducing the error easily without going to much in depth into the steps presented in this chapter. Although these steps help to guide a developer that might not even be deeply familiar with the code through the process with ease sometimes problems occur that can not be found using these steps as they are too unique and wouldn't appear for anybody else. If a developer stumbles over such a problem, it is suggested to apply the same method that is explained in 1.3.6 by creating a hypothesis checking the outcome and repeating the process until the failure can be reproduced.

3.2.3 Automate and simplify the test case

In comparison to typical Java applications where automated testing can be quite tricky as a graphical user interface, and user interaction is involved, automating the test cases for Flink applications is quite easy. All that the developer has to do is take the result of the last section and write a unit test with the specific start parameters. The "Traffic Approach" also suggests simplifying the test case to be as easy to implement as possible so that only the error and nothing else gets checked. This can be useful to make it easier to help spot the problem in the code later on but is not needed for Flink applications. For example, if the word count application were to crash on "-" characters and the input test case would be an entire book page of words it makes it harder later on to find the actual character that is causing the fault but makes it easier to create the test case. As understanding the problem and simplifying the test case is mostly the same it makes sense to leave the test case long and rebuild it once the failure is understood. This way if the same mistake gets built into the code again it is easily spottable as the test case already has a descriptive name, description as well as a ticket number associated with it. Another developer could then easily reproduce the steps that were taken to solve the problem the last time.

3.2.4 Find possible infection origins

Finding the possible infection origins is the same as the standard "Traffic Approach" **??** with a few small changes. Firstly if there is no failure and the resulting data is wrong, the developer can use the line where the false data got stored as the "exception throwing line". At that moment the failure is already present in the application so the line can serve the same way an exception would. If the program is too complex to overview how it got to that point a stack trace can be printed to the console by adding the following line (remove this afterwards):

```
1 System.out.println(new Throwable().getStackTrace())
```

Once the start point has been defined the process is the same as it is for standard Java applications.

3.2.5 Focus on the most likely origins

3.2.6 Isolate the infection in the chain

3.2.7 Correct the defect

Once the defect has been found, it can be corrected. Although fixing the defect is the reason the whole debugging process was started it is not the end. As explained in the **??** other questions need to be answered before the developer can move on to something else.

Alongside the already mentioned tasks a Flink developer should also think about the following steps:

1. If there was no exception thrown and the failure was purely faulty data, it is not enough to have a test case that makes sure that the same error gets reported again, but the developer should also add a check into the program that throws an exception. This is important as it is crucial that faulty data can't be generated by the program.

2. If the test case was not simplified now is the time to fix that. The developer should shorten the input to the bare minimum that would produce the failure, rename the test case accordingly and add the ticket number as a comment to the test case.

Bibliography

[Beschastnikh u. a. 2016] Beschastnikh, Ivan ; Wang, Patty ; Brun, Yuriy ; Ernst, Michael D.: Debugging Distributed Systems. In: *Queue* 14 (2016), März, Nr. 2, S. 50:91–50:110. – URL http://doi.acm.org/10.1145/2927299.2940294. – ISSN 1542-7730

[Carbone u. a. 2015] Carbone, Paris ; Fóra, Gyula ; Ewen, Stephan ; Haridi, Seif ; Tzoumas, Kostas: Lightweight Asynchronous Snapshots for Distributed Dataflows. In: *CoRR* abs/1506.08603 (2015). – URL http://arxiv.org/abs/1506.08603

[Dean und Ghemawat 2008] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *Communications of the ACM* 51 (2008), Nr. 1, S. 107–113

[Gulzar u. a. 2016] Gulzar, Muhammad A. ; Interlandi, Matteo ; Yoo, Seunghyun ; Tetali, Sai D. ; Condie, Tyson ; Millstein, Todd ; Kim, Miryung: BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA : ACM, 2016 (ICSE '16), S. 784–795. – URL http://doi.acm.org/10.1145/2884781.2884813. – ISBN 978-1-4503-3900-1

[McIlroy u. a. 1978] McIlroy, M.D. ; Pinson, E. N. ; Tague, B. A.: The Bell System Technical Journal. Bell Laboratories. Unix Time-Sharing System Forward. 57 (6, part 2). (1978), S. 1902

[Zeller 2009] Zeller, Andreas: *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2009. – ISBN 0123745152, 9780123745156

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstÃ¤ndig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Januar 2345 Jakob Joachim