Debugging Tool for a Distributed Streaming Application

Jesper Hansson Falkenby
Abstract

Debugging Tool for a Distributed Streaming Application

Jesper Hansson Falkenby

Debugging is a fundamental tool for software developers, it is therefore included in the majority of modern integrated development environments available today. When the capabilities of such debuggers are to be implemented in a debugging tool for a clustered, real-time stream processing system, the task becomes very complex. Knowledge on how to build debuggers for such systems is missing, and there is a need for these types of tools in the emerging real-time data stream processing industry.

This thesis explores ways to implement and find usages of a debugging tool in a clustered, real-time processing environment. In previous research two papers were of particular interest as to form a basis for this thesis. However, these papers do not target real-time stream processing systems, so the knowledge gap still persists. It was found that these types of systems have extended requirements in addition to the functionality debuggers of today that has to be fulfilled.

The first requirement found was that shared states has to be represented in some way in the tool, and multiple solutions were proposed. The second problem found was that the real-time data stream had to be integrated into the tool in one or more ways. This thesis has brought up two ways to solve it; stream generation and stream sampling. The last problem brought up came from the need for users to be able to debug fatal bugs that has crashed the system. Three solutions were proposed, and the concluded Monte Carlo solution builds upon the previous two problems.

The thesis presents a special debugging protocol and a blueprint of a debugging tool, with implementation details included, for others to take inspiration from. Thus it fills in the missing knowledge gap and successfully achieves the goal of answering the question of how to implement a debugging tool for a clustered, real-time data stream processing system.
Dedicated to my mother and the rBEA folks at King
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Evaluation</td>
<td>26</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Conclusion</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Implementation</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Protocol</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Shared state</td>
<td>30</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Implementation</td>
<td>30</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Event data stream</td>
<td>31</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Event generation</td>
<td>31</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Stream sampling</td>
<td>32</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Evaluation</td>
<td>32</td>
</tr>
<tr>
<td>5.4</td>
<td>Crash analysis</td>
<td>33</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Monte Carlo analysis</td>
<td>33</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Evaluation</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Discussion and Future work</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Missing knowledge</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Reflection</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Future work</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>37</td>
</tr>
</tbody>
</table>
1. Introduction

Debugging is a crucial component for any system to be able to find errors or bugs in the program due to involuntary programming mistakes. When combining the capabilities of a traditional debugger with a tool that enables the developer to debug a real-time system that processes data in a cluster, the task becomes less trivial than one might think. The goal of this thesis is to provide the reader with different theories of features for a tool that allows developers to debug scripts in a clustered real-time system used for streaming analytics. These theories are then evaluated and implemented into a prototype to be able to verify their real-world applicability.

Providing such a tool would enable developers to find bugs in code that is the cause of unexpected behavior or even system crashes. It also allows the developer to analyze the code in the event of any irregularities that might happen due to non-determinism.

1.1 Streaming analytics at King and the rBEA system

At King, streaming analytics is growing as a platform for analytics applications. Several frameworks for streaming analytics have been tried out and evaluated. A common problem with existing frameworks has been that they are inaccessible to the main user group; analysts. Existing frameworks have instead mostly been targeted towards experts in distributed computing, with complex feature sets, and complicated development cycles.

For this reason, a tool called rule-Based Event Aggregator (rBEA) has been developed at King. The aim is to provide a subset of streaming analytics, with analysts as the target users. rBEA provides easy deployment and monitoring of analytics processing jobs, connections to Application Programming Interfaces (APIs) for state handling, output to databases, graphical plots and more. The user writes processing scripts in the Groovy language [12], and then deploys and manages scripts through a graphical user interface (GUI). The scripts are deployed into a cluster of analytics processing nodes, where they can be active for a period of time, or indefinitely, depending on the use-case.
1.2 Problems with real-time streaming and debugging

For systems that are processing large amounts of data in real-time, like rBEA, debugging in the traditional sense is not feasible. The first technical challenge found is that it is not possible to simply halt the execution of the processing cluster. It cannot be determined on which node in the cluster to halt the execution on. Even if it could be determined, other nodes could depend on this node because of the possibility that these are currently working on the same state.

Since there also is a constant influx of data, halting the cluster means that the stream also has to be halted. The system would then lose its purpose with real-time processing.

Typically when a system crashes the integrated development environment (IDE) provides a stack trace. What this means is that it is possible for the developer to trace back to the point in code that caused the crash. Due to the large amount of data in the form of data streams, there is an uncertainty of what actually caused the crash. It is possible that a series of events caused it, which would require logging of each event that was processed, in their exact order, and a cache where all the events are stored. In the most trivial of cases these series of events lead to a node performing a division by zero in some script which would even in that case be hard to track down without some kind of debugging tool.

To briefly recap on why it is not possible to simply run a debugging tool directly in the live rBEA system:

- The cluster is processing the event stream in real-time and halting the cluster would require stopping the whole event stream, which would in turn render the whole cluster useless.
- Running a cluster means that there is an uncertainty as to where the script gets deployed. Thus, it is not easily determined on which node to deploy the debugger on.
- The event stream is distributed to multiple nodes which means that the behavior of the script is unpredictable since nodes asynchronously affect the shared states. In other words, this makes it near impossible to implement a debugger on the current system.

1.3 The need for developer tools in rBEA

It is a challenge to verify, or at least get a reliable indication that scripts in rBEA behaves as intended. Groovy code can be compiled, which enables some basic correctness checking, but there is still a need to test the written program more thoroughly before deploying it.

What is common in most IDEs of today is features that enables the developer to step through a program, break the execution at certain point in the code and view the state of stack and heap symbols. A tool that provides such features for rBEA would be very valuable to the analysts.
1.4 Related work

This thesis is based on the previous work by Feng Wang et al [4]. The paper was written in 1999 particularly for a debugging system using parallel computing. This paper presented a debugger that is distributed on several nodes which can be compared to as a cluster. The proposed solution consists of a client-server debugging system that contains a cluster of slave debugger nodes. For each new debugging task, a separate slave server is spawned and the GNU Project debugger (GDB) is used to debug the program. The client is implemented as a thin client; sending and receiving status from remote debugging servers is its core functionality.

Figure 1.1. Parallel debugger by Feng Wang et al

This paper provides a solid architectural basis for the tool that is proposed in this thesis. The debugger by Feng Wang et al is implemented in Java, and the motivation for using Java, is the following, cited from the paper:

*On the High Performance Debugging Forum of 1996[4], people arrive at an agreement about the advanced goals of parallel/distributed debuggers below: (1) Usability of User Interface; (2) Good Portability; (3) Scalability; (4) Support to popular parallel programming languages. Considering these goals and Java’s characters, we choose Java as our Programming language.*

A paper by Zhang Zan-jun et al written in 2009 proposes a practical implementation of a debugger in a clustered operating system environment [14].

The debugger in this paper is developed with the client-server approach in mind using the Java 2 Enterprise Edition framework. The front-end web server features monitors for every running node in the cluster displaying debugging data given by the GNU Debugger that is sent back and forth using a special protocol over the network. The debugger features step by step debugging, breakpoints, watchpoints and stack and heap inspection. In other
words, it provides all features expected from a traditional debugger explained in section 2.1. This remote cluster debugger implementation, complete with features, is a very valuable asset for the tool proposed in this thesis. The decision to utilize remote debugging in a client-server environment is a step forward into fulfilling the goal of this thesis.

1.5 Research question

Based on the problems described earlier in this chapter makes it clear that the purpose of this thesis is to provide an answer to the following research question:

*How can a debugging tool for a clustered, real-time event processing system used by data analysts be implemented?*
2. Background

Debuggers have a long history, the term *debugging* have roots from as early as the 50’s [2]. For single-threaded executions, debugging is usually a trivial and straightforward process. When multiple threads or processes are introduced to the program, debugging no longer becomes a straightforward task. The program might have a lot of different states that can be very hard to keep track of. The threads or processes have introduced non-determinism to the program, given that there are no synchronization mechanics implemented.

### 2.1 Debugger specification

In a compiler design book written by Sanjeev Kumar Aggarwal and M. Sarath Kumar in 2003, a debugger is specified as having the following features: a query processor, a symbol resolver, an expression interpreter and a debug support interface at its top level [1]. The book also specifies more sophisticated debuggers to have additional features such as running a program step by step, setting breakpoints within the program that halts the execution at the specified location, and tracking values of variables. For a lot of existing programming languages, sophisticated debuggers are included in the IDE.

### 2.2 King and the mobile games industry

The mobile games industry is a relatively new field in the video game industry. It is not as well established as the console market having roots from the 70’s. With the release of the App Store and Android Market in 2008, the mobile games industry has grown exponentially ever since. In 2016, the industry managed to surpass the revenue of the console market for the very first time.

King is a leading company within the mobile games industry. Pioneering casual games such as *Candy Crush Saga* and *Bubble Witch Saga*, the company has grown rapidly since it was founded in 2003. The key to success can be attributed to the extensive streaming analytics done for all of their released games.
2.3 About streaming analytics

Streaming analytics is an emerging technology field where data is processed in real-time, as opposed to traditional batch-processed analytics. This gives the opportunity for developers and data analysts to build applications where decisions can be made, and reports generated, in real-time.

Sometimes there is a need to handle a very large amount of data, for this reason streaming applications are typically distributed on a processing cluster. This is known as big data; using advanced data analytics methods to extract values from large quantities of data [8].

It is with no doubt that big data has become a staple for a lot of companies today. The quantities of data has grown rapidly since the term big data emerged in the 90’s. Measures in 2012 by IBM has shown that globally, 2.5 billion gigabytes of data was produced each day [7]. With data of such magnitude, traditional data processing methods such as batch processing are deemed insufficient.

2.4 How rBEA deploys and processes user scripts

A user script is compiled into an rBEA event processing function. The processing function will consume all incoming analytics events. The result of the evaluation of a processing function is dependent on several factors:

- The input data (event).
- The state of local memory variables.
- The state of shared-database variables.

There are several issues that complicates the process of debugging:

- An rBEA script is compiled and deployed to several nodes simultaneously.
- An analytics event is sent to one of the several nodes, using the distribution algorithm explained in section 2.5.
- A processing function may read and write to the shared state at any time.
- A complex relationship between a node and the shared state exists. Reads and writes are cached in local memory for performance reasons, and flushed to the remote database using some schema.
- No synchronization is performed between nodes, one node may lag or process events more slowly than others. Because of this the execution of any job that shares state between nodes is non-deterministic.

The last issue brought up can be comparable to the problems that multi-threaded programs have; the order of execution for all threads is non-deterministic, meaning that it is possible for two or more executions of the same program to end up with different results under the assumption that no form of synchronization exists. One of the executions might even have lead to a crash, which would be difficult to debug because of the implied non-determinism.
2.5 Event queuing and distribution

Incoming events are automatically put in a global queue. The events in this queue are partitioned into groups of users before being inserted into it. This way the events are easily distributed to specific processing nodes in the cluster. It also guarantees that all events of a certain user will end up at the same processing node in the cluster. See figure 2.2 on the next page for a simplified illustration of the queuing and distribution system.

2.6 States in rBEA

A state in rBEA is represented by one or more fields in a database table that is used to track player patterns, game statistics, and more. A state could represent information about the current game session for some specific user. These states are then, depending on the script, displayed in an accessible way for the data analysts. It is up to the data analysts themselves to write what kind of data should be displayed and in what form, usually the data is plotted into one or more graphs.

A state is shared whenever there are multiple nodes that read and write values to it. Since the nodes are not synchronized with each other in any way, for performance reasons, the nodes introduce non-determinism. As seen in figure 2.3, the first two processing nodes happen to be asynchronously reading
and writing to the same state. The first node reads a value and then writes a new value to it, while the second node only reads a value from it. The non-determinism in this case is that it cannot be guaranteed in any way that the second node will read the new value written by the first node.

2.7 Checkpointing and sampling

Checkpoint functionality is implemented in rBEA so that in the event of a crash, the system is able to roll back to the most previous stable state, which is hereby referred to as a *snapshot*.

In figure 2.4, there is a snapshot $S$ for user $X$, and event 1 caused the system to create another snapshot of the user state. When event 2 is processed, the
Figure 2.4. Snapshots

Because of the behavior of rBEA, it is possible to sample only the events that affect the state of user X. Instead of rolling back every state in the system and then replay all of events regardless of their type, it is possible to narrow it down to one specific stream of events. This saves a lot of time and resources, but the challenging part is to determine whether or not it was actually the event stream from user X that caused the crash.

2.8 Specific requirements of a debugging tool for rBEA-like systems

In order to be of value, any tool that provides step-debugger like functionality must somehow take the issues that were previously discussed into consideration. A traditional debugger operates on a single process that lets the user inspect local memory states. A tool that lets the user step through code in rBEA would need to fulfill extended requirements:

- Represent and display user states in some way. Since this is one of the fundamental features of rBEA, this should be the first requirement to fulfill within the tool.
- Provide means to simulate or recreate events. Without the event stream, rBEA provides no functionality. The next requirement is therefore to implement the event stream in one or more ways.
- Take the parallel execution created by the distribution algorithm into account. One of the core features of rBEA is that the nodes in the cluster are asynchronously processing events in real-time. The problem is that this also causes non-determinism within the system. Therefore a
requirement would be to restrict the tool so that the non-determinism is eliminated and find debugging scenarios for this restricted environment.

- Take the complexity of shared states into account. In order to find usages of the tool for bugs caused by the shared state problem, the tool should have to implement support to simulate this problem.

A complete simulation or deterministic representation is probably not possible. However, it should be possible to make some assumptions and restrictions, and in this context have a deterministic execution of a script. One part of this thesis would therefore be to define in what context and under what restrictions the tool could behave deterministically.

A script is non-deterministic as long as there are one or more shared states involved because the order of nodes affecting some state can not be determined. What would then be required, with the requirements in mind, is to set up the script debugging environment in such a way that the script is executed in a deterministic manner, which would allow the user to debug or verify the scripts for the applicable scenarios.

For example, a script that only affects the state of a single user at a time is deterministic, the idea is to let part of the thesis be to explore the cases where the script deployment becomes deterministic and then find how our tool can help the developer to debug said script.

The main part of this thesis would therefore be to propose possible solutions on how to construct a tool that fulfills the previously mentioned requirements, which represents and simulates the relevant parts of the streaming application, and as a result will give the user some level of confidence that the program will run as intended. This can be seen as a detailed follow-up to the research question in section 1.5.
3. Methodology

Based on the requirements in section 2.8; to represent user states, integrate the event stream and implement the shared state problem, and the research question in section 1.5 on how a debugging tool for a clustered, real-time event processing system can be implemented, three distinct problems that are specific for the type of debugging tool proposed in this thesis are found:

- Provide hypotheses on how states can be represented in the debugging tool.
- Provide hypotheses on how to integrate events streams into the debugging environment. This also includes hypothesizing about different debugging scenarios and how the event stream can be used in that scenario.
- Provide hypotheses on how the tool can be used to trace bugs in scripts that causes a clustered, real-time event processing system to crash.

3.1 Research steps

The process of answering the research question is divided into three steps:

1. Provide hypotheses for each problem.
2. Evaluate the hypotheses by discussing their obvious benefits and drawbacks. Then measure them against the criteria specified in section 3.2. Consult experts within the field of real-time data processing and let them evaluate the fitness of each hypothesis by providing scores for each criteria.
3. Conclude based on the evaluation and the given scores.

Two King employees in the Tracking department with plenty of real-time data processing experience are consulted:

- Jonathan Beck: Senior Developer with a Master of Science in Electrical engineering.
- Magnus Ramstedt: Principal Engineer with a PhD in Particle Physics.

Jonathan and Magnus are both pioneers of tracking at King. They have been working closely together since tracking at King started and they also play a major role in this thesis by providing supervision.

After concluding the hypotheses, a prototype is implemented. The prototype is then evaluated for different debugging scenarios. It is then discussed whether or not the prototype turned out as well as in our hypotheses.
3.2 Evaluation criteria

Three fundamental criteria are specified that will provide a baseline for hypothesis evaluation:

- **Complexity**: Required modifications and expertise for rBEA or some third-party library used within the system to implement the hypothesis. Hypothesis coupling with rBEA and complexity are highly correlated.
- **Correctness**: How similar the produced result of the hypothesis is to the corresponding version of the live system.
- **Time**: Estimated time required to implement and execute the hypothesis.
4. Results

This chapter proposes design decisions that answers the research question declared in section 1.5.

The previous work by Feng Wang et al, and Zhang Zan-jun et al discussed in section 1.4, made it clear that a client-server debugging tool written in Java provides the most portable, usable and robust system. The tool would have to be adjusted to fit the problem with real-time analytics and specifically for rBEA. The first step would be to replace the slave server and dbx/dbg layer in figure 1.1 with custom rBEA script processing nodes that send relevant debugging information over the network by using a special protocol.

Since rBEA has no purpose without the stream of events, the second step would then have to implement this type of feature in our tool. By completing these steps, the need for a prototype of a debugging tool for rBEA should be fulfilled, and thus progress towards the goal of this thesis.

4.1 Shared state

Three hypotheses are brought up for the shared state problem, differing in how the state is represented and how the database is implemented. When reasoning about possible solutions, additional criteria has to be considered. Together with the criteria specified in section 3.2, the fact that the hypothesis must be feasible for a thesis is also considered. In other words, if further evaluation of the conclusion is made, implementation and evaluation should be possible within reasonable time. The hypotheses range from dedicating certain rBEA nodes in the cluster solely for debugging, to isolating the debugger completely from the rBEA cluster by running it on a local machine.

4.1.1 Actual state

The first hypothesis revolves around using an actual database in the live system but rewired to communicate with the debugging tool instead. See figure 4.1 for a simple, illustrated overview of this. The idea is that when the execution of the debugger tool is halted, a snapshot of the current state is taken, in this case the actual represented state. The incoming event stream is then redirected and stored in a temporary cache as long as the tool is halted. When the tool resumes, the cache is flushed by sending its content to the rewired database.
All events that is passing through the system while the debugging tool is running is saved in a log. This is useful for when a script caused a crash and there is a need trace back to the cause of the crash. Thus the user of the tool would know the series of event that caused the crash, and also on which line in the script.

Figure 4.1. Actual database state

4.1.2 Fake database state
The second hypothesis builds on the first one, but for this a fake "dummy" database is used while still retaining the same state representation as in the live system. The fake database is essentially an interpretation of the current system but implemented on the same machine as the debugger. The database could either be stored in memory or on disk. The solutions would not differ in the end result, only in performance. What is important here is that the operations performed in the live database are preserved. See figure 4.2 for an illustrated example.

Figure 4.2. Dummy database state

What this essentially means is that the live database structure is cloned into a local "dummy" database, the database is actually never used for anything other than mimicking the behavior of the live system and it is solely used for correct state representation.
4.1.3 Emulated database state

The third and last hypothesis revolves around the idea that the shared database state is purely an in-memory emulation, meaning that it runs on the same machine as the debugging tool. The shared state is implemented based on a loose interpretation of the live shared state representation. See figure 4.3 to get a brief overview of this idea.

![Figure 4.3. Emulated database state](image)

What this means is that it does not have to be an actual database, it could simply be implemented as a map of key and value pairs in the heap.

4.1.4 Evaluation

Out of all hypotheses provided, completely emulating states is the least complex solution since it is completely decoupled from the live system. A drawback of this is that any kind of correctness in the implementation cannot be guaranteed. Partly because of not having an actual database, and also that the hardware differs depending on the machine the debugging tool is deployed on.

If correctness is the most important and only criteria, implementing the debugging tool within the live rBEA system is the most fit solution. It is most correct since the hardware is identical to the live system and the same goes for the shared state representation, it is also the most complex solution, since it requires modification to the current live system.

The second hypothesis works as a middle ground of the other two; correctness is maintained as much as possible by imitating the database storage used in the live system, while it is still completely decoupled from it.

The evaluation is finalized by measuring the fitness of each hypothesis with a score from 1 to 5. Higher score means better fitness. The scores have been plotted to a graph to get a better overview, which can be seen in figure 4.4.
4.1.5 Conclusion

The evaluation makes it clear that the state emulation hypothesis in section 4.1.3 is the most fit for further evaluation. The only drawback of this hypothesis, as seen in the scores, is its limited correctness. The benefits do however outweigh the drawback. The only benefit of using a fake shared state is a slight improvement in correctness, and significant decrease in the other two, and using an actual representation implies too many drawbacks.

It is concluded that implementing a completely decoupled and emulated shared state is the better option. It has the perfect balance between implementation complexity, correctness and time efficiency.

4.2 Event data stream

Proceeding with the second problem stated in chapter 3. The focus now lies on the incoming stream of events and how it can be utilized in the debugging tool. Two hypotheses are brought up; data stream sampling and generation.

4.2.1 Generating event streams

Individual event objects are easily generated using existing modules in the rBEA framework, thus generating a stream of events is a straightforward process. When considering the stream partitioning and partition assignment to different nodes, the task becomes a challenge.
This hypothesis revolves around the idea that the user of the tool is able to generate a specific stream that is dramatically reduced in size, and then proceeds to use the tool as usual. The stream should be generated in a separate thread, with specific parameters that the developer has set up beforehand.

4.2.2 Sampling the event stream

Because of the size of the event stream, using the event stream as is within the debugging tool is not feasible. Stream sampling provides a solution to this.

There are certain cases within rBEA where smaller samples of the event stream is beneficial. For example, in the event of a crash, the user of the tool might be able to narrow down the cause of the crash by filtering the event stream for one or more users.

4.2.3 Evaluation

Generating the stream requires development of an algorithm that generates coherent data. It allows user of the tool to control the size of the event stream and to isolate the debugging process from the live system. It would be especially useful to generate a stream when you have a script that only processes some specific event. If an event generator is implemented, the user can specify one or more types of events that should be generated for the tool.

Sampling the event stream is great when it is possible to narrow down a system crash to a specific user or a group of users. Filtering the stream by sampling it will dramatically bring down its size. Sampling requires modification of the live system. For example, in the event of a crash, the system would
have to with the help of checkpoint functionality, or similar, deploy a new cluster that the debugging tool would run on, pipeline the chunk of events to the debugging cluster, and then roll back and recover the live system according to the usual routine.

Even though they are not directly comparable to each other, it is still valuable to let the experts score them according to the criteria regarding complexity, correctness and time, specified in section 3.2. See figure 4.6 for the scores.

4.2.4 Conclusion

It is not possible to directly compare these two, since they are both useful for different cases. Generating the stream is great whenever the user is sure of the usage of the script. For example, a script that consists of code that is executed only for a specific event will find it especially useful to have its input as this specific event only.

The event stream can be sampled whenever it is known which group of keys, such as users, should be verified. The stream is significantly brought down in size and as a result cut down on the time the tool will require to verify some script.
4.3 Crash analysis

For the last problem specified in chapter 3, three hypotheses are presented. This problem revolves around the need of tracing bugs that caused a system crash. The first one uses a Monte Carlo approach and the remaining solves the problem by using different types of logging.

4.3.1 Monte Carlo analysis

In the first hypothesis, the Monte Carlo method [3] is used to reproduce the exact order of events that caused a crash. The idea is that when the system crashes, it is rolled back to the most previous snapshot and replayed with the exact same events that led up to this crash. The snapshot is replayed in the debugging tool instead of the live system.

When replaying the snapshot in the tool, the order of events are randomized and processed until it is found that the same execution sequence that caused the crash is encountered. At this point, the debugging tool would then be used as a traditional debugger to analyze the behavior of the script. Figure 4.7 gives an illustrated explanation of the procedure.

Figure 4.7. Monte Carlo crash analysis

In the figure above, the *Debuggee* has crashed while executing a certain script. The live system has crashed and the chunk of events from the most recent snapshot is then sent to a newly deployed debugging cluster. This cluster, as you can see in the figure, randomizes the order of these events until the culprit sequence \(x, y, z\) has been found. *Debugger X* then redeploys itself so that the user of the tool is able to proceed the debugging process as usual.
4.3.2 Log analysis

Another feature that would help with analyzing the cause of a crash is to store logs within rBEA. Two hypotheses are proposed for this. The first one is based on the idea that every node in the cluster stores a log within its own cache. Every time a new event enters this node, the event is logged in the cache.

The second hypothesis builds upon the first one; every node logs the processing start of every new event, but there are also logs being kept within the databases that stores the states. What this means is that when some node in the cluster begins to write to a database, the operation in the database is logged which includes the node and event.

4.3.3 Evaluation

Keeping logs of node and database operations provides the debugging tool with a near complete understanding of the order of events that were processed within the system. As a result of this it is possible to replay these events in the exact same order and successfully reproduce the same bug. It cannot be guaranteed that the replay will be 100% correct, since there could for example be a suddenly lagging node somewhere in the cluster. Regardless, it is with a high probability that the same bug that caused the crash is reproduced.

Logging is a significantly more complex solution since further development of the live rBEA system is required, by implementing special logging functionality. An additional layer of complexity is added when the database operations are logged. It would require consulting experts of the database framework that is used in rBEA to be able to add this type of functionality.

The Monte Carlo approach is with no doubt the least complex solution. As it does not require further development for any additional functionality within live rBEA, the approach is completely decoupled. It is also very likely the most time efficient solution, since it is possible to narrow down the sequence of events even more and run only the most recent events before the crash happened and randomize from there. However, analyzing with Monte Carlo gives us the least correct approach with no guarantee that the bug will be reproduced.

As usual, experts have been consulted and scores based on the criteria have been given to each hypothesis and summarized into a graph in figure 4.8.
Figure 4.8. Crash analysis hypotheses scores

4.3.4 Conclusion

The evaluation previously made makes it obvious that the Monte Carlo approach explained in section 4.3.1 is with no doubt the most decoupled and time efficient solution, as proven by the given scores. The only obvious drawback, as explained before, is the limited correctness of this approach. Compared to the other two, this is the only approach that does not require any modification to the live rBEA system. Based on the evaluation made, it is concluded that using the somewhat naive Monte Carlo approach is the most fit.
5. Implementation

This chapter presents the implementation made to further evaluate the conclusions drawn in the previous chapter. The chapter is introduced by going into detail on the architecture of the prototyped tool.

It was decided to go for a client-server approach for the debugging tool prototype. Based on the success from the paper by Feng Wang et al, it was decided to go for this approach because it allow for a thin front-end client approach which means that most of the debugging functionality is retained on the back-end server.

A cluster has been set up as a master debugger and slave processing nodes that is similar to the architecture in the paper. See figure 5.1 for an illustrated overview.

Figure 5.1. Debugger implementation

5.1 Protocol

A special protocol has been implemented so that the client and server can communicate back and forth by using a header, message type identifier, and message content represented as a string. For example, an identifier reqEvent, short for request event, can be sent, and then a message which in this case would consist of a simple integer. The identifier tells the server what type of request this is, and the message tells what the request actually is. A reqEvent identifier with the message 1 means that the client has sent a step request to the server where the debugger is running.
The message type identifiers implemented can be seen in figure 5.2 below:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>compileAndStart</td>
<td>String</td>
</tr>
<tr>
<td>reqEvent</td>
<td>Integer</td>
</tr>
<tr>
<td>setBreakpoint</td>
<td>Integer list</td>
</tr>
<tr>
<td>setEventsFile</td>
<td>String</td>
</tr>
</tbody>
</table>

*compileAndStart* tells the server to compile the given script and start a new debugging machine.

*reqEvent* is the most important type of message that controls the behavior of the current debugging machine. The user can send multiple step requests such as *step into* and *step over*, abort the current machine, or run it normally until the exit point of the script or a breakpoint is encountered.

*setBreakpoint* lets the user send breakpoints to the current debugging machine.

*setEventsFile* loads a pre-generated file that contains events to be processed within the debugger.

The server maintains a remote debugging session with a *Debugger* process when the user sends a request to debug some script. This session is maintained until the user aborts the debugging process or until the script finishes execution.

To be able to retrieve debugging information from the running script, the script is compiled by the Groovy compiler with a special debug flag. This allows the tool to inspect the complete memory stack, run the compiled bytecode, and trace back to the specific line in the script where this piece of bytecode is generated from.

A cluster is simulated by spawning a set number of event processing slave nodes that the master server has distributed events to. The events are distributed to the nodes by using an algorithm that divides a given stream of events into even chunks, which are then sent to each respective node.

When a node processes an event, it reads and writes to the state that it affects, thus an accurate representation of the current system is achieved with the same problems due to asynchronous behavior and concurrent reads and writes.
5.2 Shared state

The shared state problem was concluded with hypothesis 4.1.3; to emulate states in memory. Details will now be given on how the conclusion turned into an implementation.

5.2.1 Implementation

A state for each user has to be kept, and a key-value pair mapping that contains all states is sufficient. That is why it was decided to store the states in a hash map. Each state contains a hash map of string keys and values. In rBEA, this is referred to as *fields*, and is supposed to represent the fields in an actual database. This can be viewed as a database stored in memory. It also contains another hash map, called *local state*, since it is also possible to store values in a state that is local to some specific script, thus an additional hash map is maintained. In simpler terms, this can be seen as storing a hash map of hash maps.

Putting it all together by combining the debugger implementation in figure 5.1 with the shared state implementation, a more extensive illustration can be made:

*Figure 5.3. Shared state implementation*

As seen in this figure, when *Node A* accesses the *States* object, it gains access to all available states in the system. It can read and write to the states 1 to X depending on which current user is processed in the script, asynchronously or synchronously. Asynchronous reading and writing brings up all the problems related to concurrency in the live rBEA system.

When a field is registered in the script, it is a local field, thus put in another hash map. When a common field is accessed and modified, if the current event that is processed in the script affects this field, the field might not exist when accessing it, so it is created when first modifying it and updated only when an event affects it. This is typically called *lazy updating*. 
5.2.2 Evaluation

This method offers great flexibility and performance since everything is run in memory, so there is no network delay. It is accurate in the way a typical database stores values in tables; essentially key and value pairs.

The shared state problem is implicitly reproduced when one or more processing nodes access the same state and reads and writes to it. Thus the implementation is simple enough to be implemented within reasonable time, but also complex enough to reproduce the shared state problem.

Simulating the cluster by running multiple nodes in parallel will introduce the same shared state problem encountered in the live system. Since a database typically stores values by key-value pairs, one can easily implement a hash map that will resemble this behavior.

5.3 Event data stream

For this problem the reader is first presented with implementation details of an event generator. For the second hypothesis, the reader is provided with a theoretical solution. Since this cannot be implemented within reasonable time, and then evaluated, a solution that will work in theory is provided instead.

5.3.1 Event generation

A user can write a script and then have the choice of generating specific events to execute this script on. When the user writes a script in the environment, he or she may select one or more event types that should be generated during execution. This request is then sent to the tool, by sending a request message to the server, this message passing system is explained in the beginning of this chapter.

The events are built as custom Java objects, that are then sent to the appropriate processing node in the fake cluster. Each event is typically identified by an ID, and the parameters for each of them vary in numbers.

To generate each event, a custom EventGenerator class has been implemented. What this class does is that when the user inputs one or more event types, the EventGenerator first randomizes which event type to generate, and then automatically fill in all event-specific parameters with dummy data.
5.3.2 Stream sampling

The most relevant scenario where stream sampling is applicable is the scenario where the system has crashed and the developer wants to narrow down the cause of the crash, as discussed in section 4.2.2.

When a system like rBEA crashes, the debugging tool is built in such a way that only a single connection establishment with the server is needed; the server will then take care of the lifespan of the debugging process. This means that whenever an event stream has been sampled, the server terminates its current debugging process and re-deploys itself but with the sampled stream as input. This way the lifespan of the debugger will be determined by the size of the event stream sample.

The responsibility of the debuggee in this scenario is to filter its data stream for only the relevant segments and send this to the server, assuming that the data format is compatible with the server. The debuggee then sends a signal to the server that the debugging process should be started.

5.3.3 Evaluation

The events that are generated resembles the ones that are processed in the live system because the event objects generated inherit an Event interface which is essentially a blueprint of the event structure.

The EventGenerator can easily be modified to be a running process in the background that will generate and distribute events to the processing nodes during run-time, although the main purpose of the EventGenerator is to pre-populate the nodes with events.

Decisions made for the debugging tool has enabled ways to implement a stream sampling algorithm that will work in theory. The tool is also extendable and modular in such a way that should it be used for a system other than rBEA, only the Debugger class that executes the program and is responsible for returning debugging data has to be detached and rewritten.

The stream sampling algorithm itself is outside the scope of this thesis and is purely the responsibility of the debuggee to implement.
5.4 Crash analysis

From the results in section 5.3.2 on stream sampling, it has been ensured that the process of inspecting a crash is as straightforward as possible on the debugging client side. Since the tool already supports input of stream samples, the responsibility of the debuggee is therefore to implement measurements to handle crashes. The debugging tool puts the following requirements on the debuggee:

- The system should be able to recover from a crash.
- When the system is recovering, it should roll back to the most recent saved state. In other words, the system should cache its state within a certain time interval.
- The data stream should be replayable. What this means is that the system has to be able to filter out the chunk of data that was processed from the most recently saved state up until the point where it crashed.
- The system should be easily extended to communicate with a remote address and either send data over the network or save data to a disk that is accessible to the debugging tool.

5.4.1 Monte Carlo analysis

In figure 4.7, it is assumed that the debuggee has already crashed and reverts the system to some point before the crash. The debuggee then establishes a connection with the debugging tool and sends relevant data that should be processed within it. The tool then randomizes this data in a Monte Carlo fashion and runs it until it encounters a crash. When a crash is encountered, the user is certain of in what order the events cause a crash and enables the user to further debug the script step by step to conclude the routine or line of the script caused the crash.

It is the responsibility of the user to verify that the same bug has been encountered when the tool stops at the crash. As this is the major drawback of using the Monte Carlo method, the user has to circumvent this in one or more ways.

The Monte Carlo method could be brute forced to run until all possible sequence combinations have been tried out. Although it is not very time effective and lacks in performance, the bug is guaranteed to be found.

Debugging with the Monte Carlo method could also look like the following:

1. The user starts the Monte Carlo debugging session.
2. The debugger crashed at sequence X; user proceeds with regular debugging on this sequence.
3. When the bug for sequence X is found, the user fixes it and restarts the debugging process from step 1. Now excluding sequence X.
5.4.2 Evaluation

As long as a connection is established and maintained with the debugging server, continuous re-deployment with varying input of events is possible. This allows the debuggee to deploy the tool by using trial and error, i.e. randomize event input until the crash occurs.

It also allows the debuggee to replay a set of events and execute them with the functionality the debugging tool provides, such as step-by-step debugging. This is very useful in the situation where a data analyst discovers some abnormal behavior in the script and want to trace the cause of it. For example, there could be a graph plotted on some data, say *levels completed*, and this data suddenly spiked, the tool then allows the analyst to monitor the behavior of the script in a sandboxed debugging environment.

Monte Carlo analysis can be utilized in different ways, and the responsibility of how to utilize it is in the hands of the user.
6. Discussion and Future work

This thesis provides a prototype of a debugroovyging tool that was surprisingly straightforward to implement. Further development such as reworking the debugging front-end, implementing support for other systems to be easily integrated, and improving the communication protocol is possible but does not lie within the scope of this thesis.

6.1 Missing knowledge

This thesis was heavily influenced by the previous work of Feng Wang et al [4] and Zhang Zan-jun et al [14] and filled in where knowledge was missing for debugging a real-time data stream processing system.

One reason for the lack of existing knowledge or debuggers for this type of system is simply that the field of real-time data stream processing is a relatively new one, and any academic work that touches this subject has yet to catch up with the growth of this field.

What this thesis has done, is that it filled in the gaps to eventually achieve a type of blueprint for others to take influence from. This is comparable to how this thesis took influence from the two papers previously mentioned.

One of the missing knowledge gaps was found while the thesis was undergoing; crash analysis. This type of debugging, together with the shared state and event stream generation or sampling implementation proposed in the thesis, provides missing knowledge on how to debug crash causing bugs within a real-time streaming analytics system.

6.2 Reflection

The shared state problem has been the most problematic to define usages of. It is essentially a concurrency problem and shares the same problems that other debuggers have for concurrent programs. The major difference is in the context. This thesis gave a definition to shared states and implemented it such that the states interact with the debugging tool in a proper manner, e.g. in a similar fashion as in the live rBEA system.

This problem essentially led to more drawbacks than benefits when trying to completely solve it, and other more useful usages of the debugging tool was found instead. Thus this thesis still lack a complete solution to the shared state
problem, but has shed light on the drawbacks of trying to solve it and why it does not make a lot of sense using a solution of it in a debugger. In other words, when using the debugger in a concurrent environment, the traditional features of the debugger will always only make sense to use on only one node, thread or process per time.

The shared state problem, together with the event stream generation and sampling proposed in this thesis, did however provide the fundamental building blocks needed for the crash analysis debugging method that was proposed. This method is one of the more useful debugging methods to use for real-time data stream processing systems.

6.3 Future work

To continue on what this thesis has started would mean to further investigate the shared state problem. The investigation should propose usages of shared states or something similar in a debugging tool like the one proposed in this thesis. It is possible to investigate the non-deterministic part, and how it can be turned into an advantage instead of a disadvantage, or possibly find the constraints that has to be made within similar debugging tools, such as running the tool in a manipulated debugging environment.

The prototype as is could easily be used for further investigation, as one of the goals of this thesis was to properly represent shared states within the tool, which the thesis ultimately succeeded in doing.
References