Linköping University | IDA
Bachelor thesis | Innovative Programming
Spring term 2016| LIU-IDA/LITH-EX-G--16/014—SE

Tracing user transactions through a complex, multi-tiered business application

Björn Kihlström
Simon Gustafsson

Tutor, Rita Kovordanyi
Examinator, Peter Dalenius

LINKÖPINGS UNIVERSITET

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Björn Kihlström, Simon Gustafsson

Tracing user transactions through a complex, multi-tiered business application

Björn Kihlström
Institute of Technology, Linkoping University
bjoki902@student.liu.se

Simon Gustafsson
Institute of Technology, Linkoping University
simgu002@student.liu.se

ABSTRACT
This paper details the design and implementation of a software tracing application, used to trace transactions across the different layers and parts of a distributed enterprise system called IFS Applications. The tracing system is evaluated based on performance and usability in order to define some general concepts regarding how to trace flow through a complex enterprise system composed of many different components and layers. The implemented system showed great potential in accomplishing the goal of adding next to no overhead, but was lacking in that it could not scale to support many clients over any amount of time because of the amount of data generated.

Keywords
software tracing; monitoring; enterprise systems; distributed systems; performance

1. INTRODUCTION
The development and design of complex business applications is a long process during the entire life-cycle of the application. When large amounts of business-critical data and functionality is handled by the application in a multitude of services divided into several layers of functionality, it becomes increasingly challenging to diagnose a performance issue associated with a particular transaction[1].

This is especially true when the application setup varies across different installations[2] of the software. Technical support staff face a challenge in determining which route a certain transaction takes through the system. A customer's complaint of the system being slow and non-responsive under certain conditions can lead to a lot of time being spent on diagnosing where in the system the problem resides.

Several existing monitoring tools are in place to monitor the general state[3] of the system. This generates an immense amount of data for any particular installation, and correlating these data with a particular issue is not a trivial task. A transaction in a distributed environment is not a simple entity that travels from point A to point B in a deterministic way. Rather, it travels along a complicated path from the client through the system towards the database through several nodes and is subjected to load-balancing functionality as well as traffic through third-party modules with unknown internal implementation specifics along the way before it travels back to the client with a response.

[1] By "transaction", we mean a call from a client to the system and all associated network traffic and code execution inside the system up to and including the response to the client.

[2] By "installation" we refer to an actual existing setup of the system in a production environment.

[3] By "state", we refer to all factors that may affect the performance of a particular node, including things such as network load and hardware state.

Therefore, it is necessary to be able to trace such a transaction through the system to be able to correlate the already existing information about system state with a particular performance issue. Which node in the system it is that actually stalls the performance is a key piece of information. This complements the information already obtained by monitoring tools already in place and makes the information these provide more useful by narrowing the scope to the node where the performance issue arises.

The problem of monitoring complex business applications distributed across several machines is a challenge in itself. It is also not new. Joyce et al. [7] studied the problem in 1987 and identified several problem areas that still hold as a main focus of work in the area today. The difficulty of reproducing a given error that results from a possible but improbable execution path was already outlined in their work.

Around the same time, Wybranietz et al. [14] developed a system monitoring application for distributed systems and managed to outline several practical problem-areas for such implementations. A solution to the inherent non-determinism and the resulting non-reproducibility of a transaction was implemented as a real-time surveillance of the system. The paper also outlined the need to minimize the performance hit resulting from such an implementation and laid down some ground-work for the principles that govern implementation of such monitoring.

The problem of the non-reproducibility of the state of a distributed system is not new and has been studied since at least the early 90's [6]. Many different monitoring tools have been developed since then (such as GLIMPSE [3]) and the issue is still a subject of research.

1.1 Objective
In this work we implement and evaluate a solution for tracing specific transactions through a distributed business application. The data gathered by this implementation complements data from other, already existing monitoring tools and should be independent of these tools as their exact nature varies across installations. We are concerned with discerning where a problem occurs so that other monitoring tools can be used to figure out why. The system must be able to perform a trace in real time (or as close to real time as possible) as well as after the transaction is completed or has failed.

A challenge associated with such an implementation in a business environment is that no significant further overhead may be introduced by such a monitoring service. Apart from that, the implementation of the service must be done in such a way that no security issues that were not already present in the original business application may be introduced.

1.2 Research questions
Through the implementation of the tracing system in the given context, we aim to examine the following questions:

1. What kind of data needs to be collected in order to trace a transaction through a distributed business application; and to do so with sufficient detail and accuracy to be able to identify points of interest for diagnosing a performance issue?
2. Given that the answer to the first question is known, is it possible to collect the required amount of data without introducing significant further performance degradation?

1.3 Context and limitations

To answer these questions, a tracing system as described above is implemented in the context of an already existing, large business application. The chosen application for the purposes of this study is developed by IFS[4], a developer of enterprise systems. The application is called IFS Applications[5], being their core product. The application is already monitored by a number of other tools, making its state known at any given time.

The tracing system implemented follows a transaction originating from the graphical client called IFS Enterprise Explorer as it makes its way through different parts of the system. It is limited to transactions originating from this client and it is also limited to modules that are developed by IFS and which are part of IFS Applications. Transactions originating from other clients or which take paths through third-party modules connected to IFS Applications are not taken into consideration.

Whatever overhead is introduced by other tools in place is not the focus of the study. How the data is to be correlated with the output of these other tools is also not the focus of the study: Rather, the focus is enabling the possibility of correlation. No attempt at automatically correlating the data is within the scope of this study. Rather, this has to be done manually, and the purpose of our implementation is only to provide useful data in order to be able to pinpoint the location of the failure or performance bottleneck.

Network propagation delays, caused by congested networks or physical distance, is known to be a possible cause for transaction latency in existing installations and can as such not be ignored. Our implementation has to function even when a network is congested or if different parts of the system are located at a great physical distance from one another.

The nature of the data processed by IFS Applications and the fact that access-control policies may need to be enforced on the gathered information makes it a priority to not keep restricted information on a central logging server unless it is guaranteed that it can enforce the same, or more restrictive, access policies as the actual node. This introduces the need to make sensitive data more anonymous, so that business-critical details do not leak into the trace.

2. THEORY

Due to the fact that the problem area is well known among developers of distributed systems, many different takes on the problem have been proposed and implemented in the past. We aim to take this into account when carrying out our study on the subject. Large commercial actors have developed general solutions for carrying out software monitoring, and academic studies have evaluated alternative design patterns.

A paper by Sambasivan et al. [10] compares several existing implementations (both those that are developed as research projects and purely industrial applications) and different algorithms for establishing transaction paths through systems. It also outlines specific use-cases for certain approaches. These observations are useful when considering what is a reasonable approach in our case.

2.1 Technical background

Many different tools exist to carry out the general task of collecting monitoring data in a single location, which is necessary for software tracing. Facebook once developed a framework called Scribe, which has since been discontinued from active development [15]. Scribe aimed to collect all log data from a distributed system in a single location, adding minimal amounts of overhead. The source code is now available under a FOSS[6] license, but is not maintained.

Apache Flume [16] is another FOSS implementation that aims to solve the same problem which is still actively developed and maintained. It is able to collect all logs in one place through minimal overhead. It is, however, not a complete monitoring application as much as it is a framework that can be used to build one. It is written in Java and exposes a Java API.

A third FOSS alternative is Fluentd [8]. It is largely written in C (with some Ruby parts) and aims to log everything it receives in JSON format. It is designed to be usable as a base for a logging layer in a distributed system. Fluentd does not support the Windows platform, which is a major problem because of the context of this study. Therefore, it is not an alternative to base our implementation on Fluentd.

An application that very closely mimics the behavior we want to accomplish is Zipkin, employed by Twitter [17]. It gathers traces from a distributed application and displays them via a web-based UI. It is more of a complete solution than one that can easily be customized according to individual needs.

2.2 Related work

An academic evaluation of a tracing system similar to what we set out to accomplish is the implementation and evaluation of Pinpoint [4]. Pinpoint was aimed at tracing transactions through a distributed system and correlating the traces and request information with error occurrences in order to establish patterns for what transactions are more likely to fail and in which nodes this happens. However, one of the assumptions made in the development of Pinpoint was that transactions fail irrespective of other transactions as a result of their own intrinsic characteristics. We do not make this assumption in our work, as IFS reports that very few of the performance issues associated with their application are caused by bugs. Rather, we make the assumption that a transaction fails as a result of the system's state.

In some ways, a paper by Aguilera et al. [1] falls more in line with our take on the subject. The purpose of their proposed implementation was not to automatically correlate a given transaction with the cause of failure. Since the nature of the nodes in question could not be known in the context of their work, the ambition was only to provide the path of ill-performing transactions and the context in which they performed badly in order to aid manual analysis. On the other hand, Aguilera et al. did not propose a solution for real-time tracing, instead relying on analysis after the fact. The work is also based on the

[4] See http://ifsworld.com for more information.

[5] See http://www.ifsworld.com/en/solutions/ifs-applications/ for more information.

[6] Acronym for “Free and Open Source Software”. See https://opensource.org/licenses for more information.

assumption that all parts of the service to be analyzed are part of a local network and that network propagation delays did not have to be taken into account.

Another attempt to determine the cause of failure in ill-performing transactions is Pip [9]. This implementation is different from ours in that it takes the approach of first having the developers and maintainers specify the expected behavior of the system and then look for transactions that do not correspond to this behavior. Since expected behavior may be very difficult to determine in the context of our study, we consider this approach out of scope. Other similar takes on the subject exist, such as the approach proposed by Textor et al. [13]. They are different from what we aim to implement for the same reasons as Pip.

A paper by Animashree et al. [2] describes the feasibility and algorithmic complexity of a tracing system which is based on the footprints generated in local log files. The analysis of such a tracing system presented in their work supports the notion that our take on the issue is possible, as the context of our work includes a system that does not correspond to the proposed worst case scenario presented. However, Animashree et al. did not implement such a system.

An actual implementation that is superficially similar to what we aim to implement is X-Trace [5]. Many concerns that have to be taken into account during our work were discussed when implementing X-Trace, such as maintaining the security policies of the application. However, this is a very generalized implementation that is meant to operate by attaching extra metadata to all network requests, which can not be done in the context of our work.

A work that is often referred to in an industrial context is a paper published by Google detailing the implementation and design of Dapper [11]. In this document, main design concerns are laid down and many other applications (for example, Zipkin [17]) boast that they are compliant with these design principles.

2.3 Theoretical implications

One thing all software tracing tools have in common is the presence of central logging by sending logs over the network. A key focus in our study is performance, and the most expensive operation in this respect is a network transfer. Therefore, it becomes natural for our study to minimize the amount of network transfers.

There are a few possible takes on this subject. A common solution is to send several entries at once in batch network transfers, perhaps even delaying the sending of logs in conditions of high network load. This approach is pivotal to reducing network load.

3. METHOD

The method is divided into four parts, each detailed in this section. The first part is a feasibility study where IFS Applications is examined and important points of interest[7] for generating a comprehensive trace are identified. Further, the identified points are categorized and the possibility of local logging is evaluated for each category along with the necessity and usefulness associated with logging the point at all.

The second step is the implementation of a prototype corresponding to the conclusions drawn in the feasibility study. The implementation consists of the tracing application itself and a simulated system that mimics the behavior of a distributed business application that can run on virtual machines to allow for easy testing in a controlled environment.

The third step is a performance evaluation. The simulator is subjected to a series of stress-tests both when tracing is enabled and when it is disabled. This allows for accurate measurement of the imposed overhead in a controlled environment.

The fourth step is evaluating the usefulness of the produced traces. This is done by attaching the tracing system to a sandbox installation of IFS Applications and making observations from the gathered trace data.

3.1 Feasibility study

The feasibility study is conducted along with IFS staff by reviewing the system and identifying points of interest. Each point of interest is written down on a card along with a general description of the event. The purpose of this is to provide a basis which can be used to answer RQ1. Points of interest are candidates for logging and can be used to identify the amount of data that needs to be collected to produce a comprehensive trace.

The second part of this step is to use this information to guide the strategy for the implementation. To do this, the cards are categorized by any emergent properties which they have in common, a process called open card sorting[8]. Once the points of interest have been categorized, which ones are good candidates for logging can be evaluated.

When examining a point of interest, it becomes necessary to have a certain set of criteria for defining which are to be taken into account. A first criteria to qualify as a point of interest is generality, here defined as a point traversed by all transactions regardless of the implementation of the business logic. Potential points where the only way to produce trace logs would be to manually hard-code the logging calls inside business logic are immediately discarded and does not make it past system review.

Evaluation of the usefulness of any categories of points of interest for a trace is done based mainly on two further criteria. The first of these is redundancy. Redundant information is defined as information that can be inferred from other points of interest. This makes the point irrelevant to the trace and thus non-useful.

The second criteria is ambiguity. Non-ambiguous information is defined as information that can only produce a single trace where all continuous steps taken can be inferred, whereas ambiguous information can either produce several traces, or a single trace that can not be used to determine the series of events accurately.

Redundant information is considered a more relevant reason to exclude a point from the solution than ambiguous information (as long as it is not also redundant), because ambiguous information could still be useful if interpreted manually by someone with an understanding of how the system works.

3.2 Implementation

In this part, the system prototype is implemented as two components. The first component is a simulator made to behave in a way similar to an installation of IFS Applications. The actual simulator is made up of nothing but logging events, sleep calls, and communication over the network (with the possibility of disabling the logging events). The simulator must be able to simulate different scenarios.

The second component is the tracing service itself. It is connected to the simulator and receives the logs which are used to create traces. The development of this service is based on the

[7] A point of interest is defined as a system event that potentially can be used to identify a performance issue in the system.

[8] Cards are sorted by the authors based on perceived similarities.

conclusions drawn in the feasibility study and are refined during development as part of an agile process in coordination with IFS.

3.3 Performance evaluation

The third step is evaluating the prototype with respect to RQ2. In the context of the simulated system, it is possible to draw the conclusion that the difference in execution time of a simulation with tracing enabled compared to the same simulation with tracing disabled is exactly equal to the performance overhead introduced by the tracing service.

In order to evaluate if the implementation meets the performance requirements, several simulations are executed both with and without tracing enabled. The execution times of the simulations are then compared. Each scenario is executed several times in order to produce an average execution time. The difference in average execution time is then used to answer if a solution providing an answer to RQ2 has been implemented. The solution is considered to fit the requirements if a realistic simulation scenario performs at 1 millisecond per log event or better. 1 millisecond is chosen because it is the smallest unit of time that can be measured using the system clock of a Windows 7 computer.

3.4 Usability evaluation

The usability evaluation is aimed at finding support for the fact that RQ1 was answered correctly and ascertain that there are no serious performance issues in a real production environment that were not discovered during the simulation. This step is based on ascertaining that the tracing system can in fact produce the desired information when attached to the real application, and that this information is structured in such a way that it is actually possible to use the system for its intended purpose.

This step is based on general observations and an open categorization of problems discovered, outlining problematic areas that need to be addressed before deploying this application in any kind of production environment. These tests are done by collecting data from IFS Applications and attempting to examine and filter traces in a way consistent with common use-cases. Any problems that occur during the use of the tracing system are noted. Results are displayed in the form of general observations.

4. FEASIBILITY STUDY

The feasibility study was conducted during a period of one week in cooperation with IFS staff. The general structure of the system was studied in order to identify points of interest which were then categorized and evaluated. The results of the feasibility study is divided into several sections, each corresponding to a component of the study.

4.1 System review results

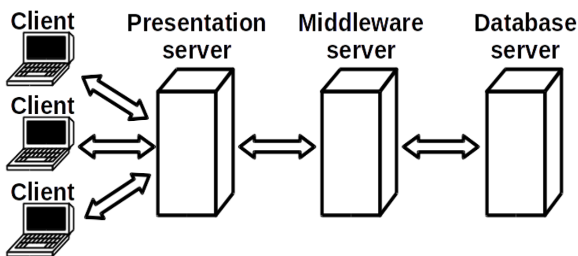

Figure 1: General overview of the system. Every server in this figure could represent several servers subjected to load balancing.

All transactions follow a certain path through the system components that can be seen as common ground between all transactions. This generalized path was examined in order to identify points of interest. Figure 1 illustrates an overview of the system design.

Every transaction is sent from the client to a server which presents an interface to the client (henceforth called a “presentation server”). The presentation server is largely disconnected from the rest of the system and contains no points of interest which are realistic to include in the trace. Any attempt to do so would require hard-coding the logging calls into the presentation logic, which excludes them immediately.

The presentation server, in turn, sends the request forward to a server handling business logic (henceforth called “middleware server”). This server is part of IFS Applications and has a well-defined internal structure. Every transaction targets a subset of functionality defined as an activity, which contains a number of methods. Methods can then invoke other methods or send a request to a server running a database (henceforth called a “database server”). There are some realistic points of interest here.

1. The arrival of a request to the middleware server from the presentation server.
2. Before invocation of a specific method inside an activity.
3. After a method inside an activity returns.
4. Before a method makes a request to the database server.
5. After a request to the database server returns.
6. Before a reply is sent to the presentation server after the request is completed.

Regarding the database server, some requests sent here do not lead to the execution of much code, but rather a database query which returns immediately upon completion. In other cases, programmatic database procedures are invoked for more complex tasks. These present one possible point of interest.

7. When a programmatic procedure is invoked.

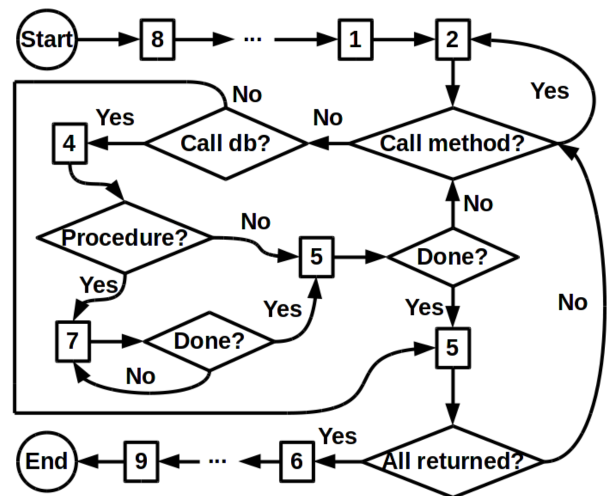

Figure 2: Flowchart illustrating the flow of a standard transaction through the system with the points of interest included.

Unfortunately, there is no way of knowing when such a procedure has returned aside from logging manually in each programmatic procedure as soon as any further calls return. Otherwise, this would also have been a point of interest.

Two additional points of interest lie in the client software, where logging is possible.

8. Before sending a request to the presentation server.
9. After receiving a response from the presentation server.

Aside from this, no further points of interest that fit our criteria could be identified during the review. Figure 2 illustrates how a transaction moves through the system and where the identified points of interest occur.

4.2 Open card sort results

The system review yielded 9 points of interest. One natural first step in the sorting process was deemed to be a division based on where in the system the point of interest resided. As such, the first three emergent categories were the following:

- Client points, containing the points 8 and 9.
- Middleware points, containing points 1 through 6.
- Database points, containing only point 7.

Regarding the middleware points, two stood out as different from the rest. Points 2 through 5 could happen any number of times in a transaction depending on what actions are taken by the invoked method. Points 1 and 6 happen only exactly once during a successful transaction. Point 1 always happens first and point 6 always happens last. As such, they are the only points of interest within this category which occupy completely deterministic locations in any generated trace.

Furthermore, point 1 is always followed by a method invocation, causing point 2 to be triggered. Similarly, point 6 is always preceded by a return from a method, triggering point 3. This makes points 1 and 6 stand out as possibly more redundant for tracing purposes than the other middleware points. Therefore, the final result of the open card sort is as follows:

- Client points, containing points 8 and 9.
- Middleware points, containing points 2 through 5.
- Middleware points (possibly redundant), containing points 1 and 6.
- Database points, containing only point 7.

4.3 Implications for implementation

Regarding the client points, both happen exactly once during a successful transaction. They also occupy completely deterministic positions in any trace (8 always being first in the trace and 9 always being last). This makes for a certain amount of redundancy as their position can always be inferred. In the cases where this redundancy does not present itself, such as when a transaction fails before it reaches the middleware server or after it returns to the presentation server, the information is instead ambiguous.

Because the presentation server cannot support tracing, any error between the client and the middleware server would result in one of two possible traces (either the request never reaches point 1 or it never reaches point 9 after it has logged point 6). From any of these traces, it can not be determined where the point of failure resides.

A time stamp from the client may be useful to the trace in some cases, but this information is also ambiguous because the clock on the client may or may not be synchronized with anything else in the system. In addition, the client is disconnected from IFS Applications even more so than the presentation server; the request may even travel through the world wide web and thus making any diagnosis of delays incurred impossible. Because of all this, our results indicate that client-side logging will not be implemented for the sake of the study.

Regarding the middleware points, the first middleware category produces non-redundant and non-ambiguous information about non-deterministic execution paths. According to our criteria this makes these points essential for tracing a transaction through the middleware server.

The possibly redundant middleware points produce information that can possibly be inferred by the points of the previous category. All transactions trigger point 2 directly after point 1 in the trace and also trigger point 3 right before point 6. As such, points 1 and 6 may be redundant, depending on the amount of information available at this stage compared to points 2 and 3. The redundancy also depends on the amount of pre-processing done between point 1 and the first occurrence of point 2 as well as the amount of post-processing done between the last occurrence of point 3 and point 6. Since these are done on the same server the time stamps of the points can be expected to provide useful information about any such processing.

The amount of traffic that could be saved by not logging this information is also questionable since the other middleware points happen more often, particularly if the sending of trace logs is performed using batch operations. With this motivation, the logging of these events is implemented for evaluation in the test environment.

The only point in the last category, point 7, happens with requests to the database server which cause the execution of programmatic procedures. When it does happen it produces non-redundant but somewhat ambiguous trace information. The reason for this is that it is impossible to trace when such a procedure returns.

Point 7 may although be useful anyway as it contains non-redundant information. Access to and understanding of the procedure source code also greatly alleviates this ambiguity. For this reason it is implemented in order to evaluate it during the performance and usability evaluations.

5. IMPLEMENTATION

The chosen approach, consistent with the functional demands, is to base the implementation on Apache Flume [16], mentioned earlier, to move the logs as quickly and efficiently as possible to a log server where the logs are stored in a NoSQL database called OrientDB[9].

An API for generating traces to be consumed by Flume was also developed to be used from within the monitored application. The main components of the system are, however, Apache Flume and OrientDB (via the custom sink). All custom components aside from stored functions in OrientDB are written in Java.

Aside from this, a simulator was developed to use the API in order to generate static traces for the purpose of measuring performance. This was also written in Java.

5.1 Apache Flume

Apache Flume is a framework designed to move large amounts of log data as efficiently as possible. Its primary purpose is to move logs to Apache HDFS which is part of a project called Hadoop[10] but it is designed with the inherent possibility of being extended and used for other purposes. It has the inherent advantage of providing reliability even in conditions of high network load, which solves one of the key issues associated with the

[9] See http://orientdb.com/orientdb/ for more information.

[10] See http://hadoop.apache.org/ for more information.

implementation. In such conditions, trace logs are sent at a slower rate but are guaranteed to ultimately arrive at the central logging server.

The ability to transfer large amounts of log data using Flume narrowed the scope of the project to the endpoints of a series of chained Flume processes running on different hosts. A Flume process is called an agent and is made up of several components. Each component is an implementation of a well-defined Java interface. The most important ones are the source, the channel and the sink. An agent consists of one or more of each component.

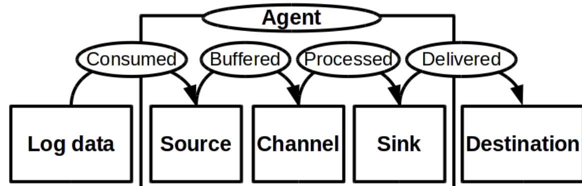

Figure 3: Simplified illustration of a Flume agent and its components.

The source is an implementation of a way to consume log data from an application. A channel is an implementation of a buffer where events are queued between being consumed by the source and being sent to the sink. A sink is an implementation of an endpoint where events are sent after being buffered. The sink may be some form of storage or be chained to the source of another Flume agent. Figure 3 illustrates the structure of a Flume agent.

The implementation of the tracing system consists of three custom Flume components. One custom source which consumes logs from an Oracle database table, one custom event deserializer (which is an interface used by a standard source) which deserializes events from a JSON format and one custom sink which persists the traces to an OrientDB database.

5.2 OrientDB

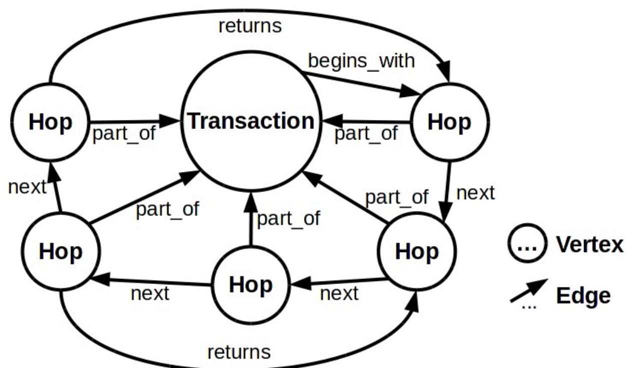

Figure 4: Simplified illustration of the way a trace is stored in the database.

OrientDB is a NoSQL database which is a hybrid between a graph database (like Neo4J[11]) and a document database (like MongoDB[12]). The trace logs and any collected statistics gathered from the tracing system follows the pattern of being mostly semi-structured data with many relations. Since OrientDB combines the schema-less nature of a document database with the support for graph database relations that are traversed in constant time without the use of costly JOIN-operations, it was deemed a good fit for representing trace information.

The custom Flume sink implemented as part of the tracing system communicates with OrientDB and is responsible for constructing comprehensive traces natively in the database. As an event arrives to the sink, it is first examined to determine which point of interest it is associated with. This information is then used to correlate the event with any sequence already constructed for a certain transaction. An event is stored as an entry, known as a “vertex” in OrientDB, and correlation is done by creating relations between these vertices, called “edges” in OrientDB.

Edges have a name and are directional in the sense that they store a direction as part of their logic. They can, however, be traversed in either direction. These relations are used to efficiently traverse a trace and correlate it with other associated information. The transaction itself is associated with a vertex, and all events, called “hops” in this implementation contain an edge pointing to such a transaction vertex. Each hop is also connected to the next hop in the transaction. Furthermore, a transaction has an edge pointing to the first hop in the trace. Hops contain properties specifying the point of interest, a time stamp and sometimes additional information.

Hops corresponding to a point of interest which is a return of something else (a method return, a database return or a return to the presentation server) also have an edge pointing to the hop corresponding to the associated initialization (a method invocation, a database call or a received request from the presentation server). These build a logical structure that separates code executed in different stack depths and layers. Figure 4 illustrates how a transaction is stored in the database.

Other vertex types exist for utility reasons. A transaction vertex is pointed to by one or more user vertices involved, enabling sorting transactions by user. Hops also have edges pointing to vertices describing known parts of the system, enabling sorting hops and transactions based on the methods involved, the server executing the method and other unifying properties. The purpose of this is to be able to follow a trace along the path it takes through the system at a topological level and to also be able to visualize the path on this level.

A bonus feature achieved from these additional vertices created to enable easy diagnosis of ill-performing transactions is indirectly constructing an entire topological map of the monitored system as it is used. It is possible that this enables other uses of the tracing system, but evaluating those uses is not the focus of the study. They are touched upon later in this paper, but not in-depth.

The most practical way to navigate through the data stored in OrientDB, using OrientDB Studio[13], is to write stored functions. Functions like this have been written during development in order to efficiently retrieve desired information, such as unfinished transactions or unusually slow execution times. There are three different languages available for native functions in the database (SQL, JavaScript and Groovy), but most of the functions in this implementation are written in JavaScript. However, these would not be used if a graphical client application existed.

[11] See http://neo4j.com/ for more information.

[12] See https://www.mongodb.com/ for more information.

[13] OrientDB Studio is a web-based graphical database administration tool with a visualizer that can be used to view OrientDB graphs. Since no graphical user interface was developed for the sake of this study, OrientDB Studio was used to visualize the generated trace data.

5.3 The simulator

The simulator used during performance evaluation is implemented as a middleware simulator. Because no points of interest exist on the client or presentation server, these can easily be ignored for overhead measurements. The middleware simulator therefore acts as if one or more clients were making requests to it via a presentation server, but in reality the simulator starts transactions on its own.

The simulator is a simple Java program that accepts instructions in the form of command line arguments. These are used to determine whether or not tracing should be enabled, how many transactions should be executed as well as how many points of interest of which type to simulate during a transaction.

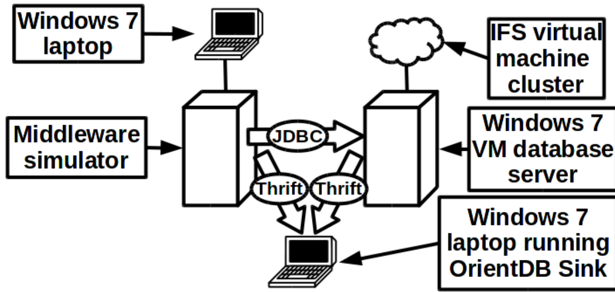

Figure 5: Illustration showing the setup of the simulator in a lab environment.

The simulator uses busy waits in order to simulate processing rather than sleep calls, as a sleep call would cause the running thread to become idle which is inconsistent with the way processing is done in a real environment. This way, measuring overhead becomes more realistic as Flume has to compete with other processing when consuming trace data.

The simulator uses the Java tracing API to produce trace logs, in the same way as IFS Applications. The Flume agent is set up using a spooling directory source (a standard Flume source reading files in creation order from a specified directory) with a custom deserializer, parsing one JSON object into a Flume event per line. A memory channel (a standard Flume channel which buffers events in memory) is used for buffering. The sink on the simulator is forwarding the events to another Flume agent using a Thrift sink (a standard Flume sink sending events over the network using the Thrift [12] RPC protocol to be consumed by a corresponding standard Thrift source). The simulator runs on a Windows 7 laptop.

The simulator also uses Oracle's JDBC[14] driver to simulate database requests. The database procedures executed simply insert log events into a table on the database server. The database server resides in a virtual machine cluster provided by IFS and provides an Oracle Database[15] This server runs a Flume agent which uses a custom source to consume events from the table using the same JDBC driver. In all other ways except for the source, it is set up the same way as the Flume agent on the middleware simulator laptop.

[14] JDBC is short for Java Database Connectivity, and it is a driver to connect to and communicate with an Oracle database from Java code. More information can be found in this FAQ: http://www.oracle.com/technetwork/topics/jdbc-faq-090281.html

[15] See http://www.oracle.com/us/corporate/features/database-12c/ for more information.

The log server is running on a Windows 7 laptop with a Flume agent accepting events from a Thrift source (a standard Flume source that can be connected to a Thrift sink). It uses a larger memory channel than the other agents, as to not easily become congested when accepting events from more than one agent. The log events are then persisted to the database by the custom OrientDB sink. Figure 5 illustrates the entire setup of the lab environment.

6. PERFORMANCE EVALUATION

Four scenarios were simulated for the purpose of evaluating performance through stress tests. All scenarios were simulated with and without tracing enabled. The system clock was used for measuring time during the simulations. All scenarios conform to the flow described in figure 2, but not all correspond to realistic transactions. Table 1 summarizes the executed scenarios.

Scenario	Transactions	Database trace logs	Middleware trace logs
1	10	1000	520
2	10	10000	4220
3	10	0	2220
4	10	10000	60

Table 1: Description of the simulated scenarios.

The purpose of the simulation was to carry out a stress test; and transactions generating more than 1000 trace logs, like the simulated ones, do not occur in reality. Scenarios 3 and 4 in particular do not follow the distribution of logging event found during normal operation (see Usability Evaluation for more information). They were designed to isolate one single type of trace log to as large a degree as possible in order to evaluate the Java API and the database logging separately. Table 2 summarizes the results of the simulations. Each simulation was executed 10 times with logging enabled and 10 times with logging disabled.

1 (en)	1 (dis)	2 (en)	2 (dis)	3 (en)	3 (dis)	4 (en)	4 (dis)
8330	6849	68173	57250	54132	53913	41122	26567
7753	6864	67736	56922	54007	54007	44523	15881
8393	6755	68567	56408	54023	54040	44351	42604
8471	6755	64039	56360	54007	53977	40374	11403
7628	6739	65833	56127	54007	53961	45615	11154
7550	6739	63383	55987	54039	54070	43914	11341
7519	6771	63664	56721	54008	54055	40747	11888
7269	6771	64272	56004	53960	53977	46567	13322
7535	6739	63415	56051	53977	54055	45210	12917
8623	6770	64238	56378	53973	54085	43649	12418

Table 2: Total execution times in milliseconds of all executed simulations, both with tracing enabled (en) and disabled (dis). Values that deviate from the trend of the result set are marked with gray.

From these measurements an average overhead can be calculated as the difference between the average execution times with tracing enabled and disabled for each scenario. The most interesting statistic is the overhead per traced point of interest in each scenario, which is calculated by simply dividing the average overhead with the amount of trace logs generated. Table 3 summarizes the results of these calculations.

Scenario 4 produced 2 highly deviating values during measurement with tracing disabled. The result set with these values included has a standard deviation of about 10,000 milliseconds, which is 5 times higher than the corresponding standard deviation for the result set with tracing enabled (at about 2000 milliseconds). Removing these deviating values produces a smaller result set with a standard deviation of around 1500 milliseconds, which is more similar in scale. This smaller result set is also displayed in table 3.

Scenario	Average time (en)	Average time (dis)	Average overhead	Average OH per log
1	7907.1	6775.2	1131.9	0.745
2	65332	56420.8	8911.2	0.627
3	54013.3	54014	~0	~0
4	43607.2	16949.5	26657.7	2.650
4 (adj)	43607.2	12540.5	31066.7	3.125

Table 3: Average execution time, overhead and overhead per trace log for each executed scenario. The bottom row is scenario 4 with the deviating values removed from the result.

The difference in execution time in scenario 3 indicated that overhead was slightly negative. Those result sets had a standard deviation of around 50 milliseconds however, which is 100 times larger in magnitude than the negative difference. The overhead is therefore equivalent to 0 because it is too small to be measured.

Interpreting the data directly indicates that scenarios 1,2 and 3 are within the limits of what has been considered acceptable overhead while scenario 4 is not.

7. USABILITY EVALUATION

A virtual machine running an installation of IFS Applications in a sandbox environment was used to test the tracing application on a real system. Data was collected by starting IFS Enterprise Explorer and pressing random buttons for between five and ten minutes. Earlier tests using the same method had been done during implementation as a sanity check in order to ensure that the tracing application was correctly implemented and able to handle actual trace data. The usability was not evaluated during these earlier tests, except for one observation made.

- The data generated was very large in volume, which made the database very large in size. During an early test, the tracing system ran for three minutes on a system with one mostly idle client active. This generated a database of over 2000 vertices and over 19,000 edges. The database reached 77MB in these three minutes. If that is an indication of how fast data accumulates, one mostly idle client would generate 42GB of data in one day. This exposes a need to filter or reduce data.

During the actual usability evaluation further observations were made, focusing on the usability of the system by examining generated trace data using OrientDB Studio.

- IFS Applications itself generates a lot of trace data behind the scenes requiring no user input. Active clients execute background jobs even when idle which generate even more trace data. This means that most traces are not interesting for diagnostics purposes.
- It is very hard to find a way to programmatically and generally identify which traces are interesting for diagnostics purposes. Some obvious non-interesting traces could be identified, but many traces look very similar; no matter if they are a result of user interaction or a periodic background job.
- Not all transactions follow the model described in figure 2. The tracing system handles these correctly as well, but it is worth noting that this can not be taken for granted.
- Given that an interesting transaction can be identified, it is easy to follow a trace and locate possible causes of bad performance.
- It is very easy to find transactions that have frozen before they have finished. A simple database query can return a list of those.
- The data is structured in such a way so that it allows for a multitude of use-cases beyond diagnostics. Easy access to statistical and topological information generated during tracing allows for unintended use-cases as a side effect.
- The tracing application is fast enough to follow a transaction in real-time.

One thing that could be noted was that the bonus feature of generating a topological map of the system as it is monitored allows for many other uses beyond performance diagnostics. One of these is simple visualization of the monitored system. When combined with other trace data, it could not only increase the readability of a trace by allowing for it to be visualized on a topological level; it could also be used for such purposes as generating statistics regarding system use, finding common performance bottlenecks and finding points in the system where transactions are likely to intersect.

The tracing application can be considered to be mostly useful for its intended purpose, as well as potentially useful for some other unintended use-cases; but a graphical client would have to be developed to put it to use.

Only one known limitation exists which causes the tracing application to sometimes generate faulty trace data. In some rare cases, traces where many different points of interest in both the middleware and database layers occurred within such a short period of time that the system clocks of the machine running the monitored application did not have the granularity to differentiate their time stamps, were encountered. Trace data from different layers arrive at the sink asynchronously. This is not a problem if they have different time stamps, and most of the time not even if they have the same time stamp.

The exception to this rule is were these sequences of trace logs with the same time stamp contain multiple database calls as well as programmatic database procedure events. In these cases, it is not possible to correlate the procedures with the correct database call using any available data. As such, the tracing application generates the wrong sequence in these rare cases.

8. DISCUSSION

The implemented system mostly conforms to given demands. The single largest problem is the amount of data persisted to the

database. One idle user would generate 42GB of data per day. A thousand idle users would generate 42TB instead. If those users were not idle, but instead actively used the system, this would amount to hundreds of terabytes per day which is unmanageable.

Aside from the issue of storage, the amount of worthless data persisted to the database drowns out the interesting data and makes it hard to find. It is also hard to find a reliable way of determining which traces are interesting, which further adds to this problem. A possible solution is to partially take the approach of specifying in advance how the system works, used for instance by Pip [9], and base some kind of filtering mechanism on this.

One possible take on the subject of reducing the amount of persisted data would be to reduce the points of interest taken into account. This would have to be done in line with the priorities laid down during the open card sort in such a case. However, the amount of storage space saved in contrast to the loss of usefulness makes such a solution unfavorable.

Which point of interest a trace log is tied to has no relation to the usefulness of the trace. Eliminating points of interest across all transactions would cause important data to be lost, and the more data that could be saved by doing so, the more useful information would also be lost. Therefore, filtering out entire useless transactions is a more fitting solution in order to save storage space.

8.1 Evaluating performance results

The data from the simulations indicate a case where tracing in the database layer is more expensive than tracing in the middleware. This result, combined with the observations when testing the tracing application with a real installation of IFS Applications puts into question the correctness of the simulator.

During testing with a real system, cases where entire procedure chains of more than five procedures were logged within a millisecond were observed in trace data. Examining the simulation data, this would be impossible if an overhead per trace log of 2 or 3 milliseconds was a correct representation of trace logging in the database layer. A procedure in the simulator was represented as a consistent SELECT query[16], either followed by a call to the tracing API or not. The exact implementation of the calling of the tracing API in IFS Applications may not be accurately represented by this simulation.

Inside stored database procedures, the logging API could be called asynchronously (similar to how tracing in the middleware is implemented). This was not possible in the simulation, and therefore the simulated tracing in the database layer is synchronous. If database tracing is asynchronous, a result more in line with scenario 3 (where overhead could not be detected at all) would not be improbable. This is also consistent with the procedure chains that both execute their code and call the tracing API within a single millisecond that have been observed.

8.2 Necessary points of interest

The results of the open card sort proved to be mostly correct. The database points had some ambiguity associated with them, but their usefulness in a diagnostics situation proved to be significant enough to overcome the limitations of the ambiguity. Very rarely did traces become truly ambiguous or wrong, and some observed traces (particularly ones with more than 30 database points) would be very hard to make sense of without the presence of procedure call chains.

Regarding the middleware points dubbed as possibly redundant, they proved to have great usefulness for the purposes of representing and visualizing data. The presence of a known starting point as well as a known endpoint enabled non-ambiguous representation of transaction completion.

Therefore, we consider this implementation to largely support the validity of the open card sort, with the reservation that the middleware points dubbed possibly redundant are in fact very useful to the tracing application.

Also, observed traces which did not follow the predefined flow still generated some kind of trace data that could be analyzed, which supports that all examined cases were handled by logging the points of interest suggested by the open card sort but that all points are not always needed.

8.3 Threats to validity

Aside from the uncertain accuracy of the simulator, discussed earlier, one of the biggest threats to the validity of the study is the use of OrientDB Studio during testing and usability evaluations. The lack of a graphical client to visualize the trace data in a more appropriate manner caused us to use the database administration tool that ships with OrientDB, capable of visualizing graphs on an HTML canvas.

Aside from having a multitude of bugs, the way in which the data is visualized is quite generalized and not adapted for the purpose of visualizing application traces. The fact that the usability turned out to be decent while using such an ill-fitting tool as OrientDB Studio may indicate that the usability had been even better with an actual graphical client developed for this purpose. However, it may also be a sign that the implementation has been biased towards producing data in a format that can be made comprehensible in OrientDB Studio. As such, the representation may be less fitting if an actual client had been used.

Another threat to validity comes from the sheer amount of traces generated and the fact that only a small subset of these have been examined during the usability evaluation. Even if no completely incomprehensible traces have been found, it is still possible that they exist.

9. CONCLUSIONS

The tracing system mostly performed in a satisfactory way with regards to both RQ1 and RQ2. There were some unanticipated problem areas, discovered during usability evaluation. This means that if this tracing application was to be taken into a production environment and used for diagnostics, something would have to be done to alleviate the issues regarding storage in order for the tracing system to be of any kind of use.

Aside from these problems the system performed well and exposed some unexpected use-cases for the gathered trace data. The results indicate that the system has potential if it is refined and some of the worst problems are addressed.

The answer to RQ1 must be dependent on the purpose of the tracing. The traces where at least most of the predefined points of interest are present are the most useful, but even a trace lacking most of them is possible to follow in some way.

Generally, only one coherent trend could be observed. The kind of data needed for a trace to be completely accurate is for all events which can be nested to log both when they begin and when they end. Since programmatic database procedures cannot log when they return, any trace involving these is somewhat ambiguous even if it can still be followed and analyzed.

[16] The exact statement was “SELECT 'hello world' FROM DUAL”, thought to execute in close to constant time.

Observations do however support the notion that the current implementation allows for all examined cases to produce coherent trace data.

Furthermore, we consider this implementation to meet the given performance demands; and that this implementation as such satisfies the requirements consistent with a valid answer to RQ2. Also supporting this notion is that the more realistic traces generated during the simulation, being scenarios 1 and 2, did in fact meet the requirements. Scenario 4 simulated an unrealistic case of 10 minimal middleware transactions executing 1000 database procedures each. Nothing like that has been observed with real trace data.

9.1 Future work

Several areas of future work remain unexplored in the context of this research. One of the main areas discovered to be of interest is how to differentiate which transactions are interesting with the amount of information available in the trace data. This is a main usability concern, and ways to efficiently sort out traces generated by background jobs is pivotal to the usability of a tracing application.

If this sorting is done before persisting trace data to the database, it would greatly reduce the amount of storage needed and thus alleviate the problem of the database growing too fast at least to some degree.

Evaluation of trace representation using a graphical client might be warranted in order to correct any bias introduced by OrientDB Studio. The main purpose of this work was to collect the required data and to represent it in some way. While this representation most likely works using a graphical client, it may not be an optimal representation if the visualization tool is implemented along with it.

10. REFERENCES

1. Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ACM, 74–89.

2. Animashree Anandkumar, Chatschik Bisdikian, and Dakshi Agrawal. 2008. Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints. *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM, 133–144.

3. Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, and Antonino Sabetta. 2011. GLIMPSE: A Generic and Flexible Monitoring Infrastructure. *Proceedings of the 13th European Workshop on Dependable Computing*, ACM, 73–78.

4. Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, IEEE, 595–604.

5. Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A Pervasive Network Tracing Framework. *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, USENIX Association, 20–20.

6. Arthur P. Goldberg, Ajei Gopal, Andy Lowry, and Rob Strom. 1991. Restoring Consistent Global States of Distributed Computations. *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM, 144–154.

7. Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. 1987. Monitoring Distributed Systems. *ACM Trans. Comput. Syst.* 5, 2: 121–150.

8. Fluentd Project. Fluentd | Open Source Data Collector. Retrieved February 29, 2016 from http://www.fluentd.org/

9. Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems. *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, USENIX Association, 9–9.

10. Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, et al. 2011. Diagnosing Performance Changes by Comparing Request Flows. *NSDI*.

11. Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, et al. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical report, Google.

12. Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8.

13. Andreas Textor, Markus Schmid, Jan Schaefer, and Reinhold Kroeger. 2009. SOA monitoring based on a formal workflow model with constraints. *Proceedings of the 1st international workshop on Quality of service-oriented software systems*, ACM, 47–54.

14. D. Wybranietz and D. Haban. 1988. Monitoring and Performance Measuring Distributed Systems During Operation. *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM, 197–206.

15. facebookarchive/scribe. *GitHub*. Retrieved February 29, 2016 from https://github.com/facebookarchive/scribe

16. Welcome to Apache Flume — Apache Flume. Retrieved February 29, 2016 from https://flume.apache.org/

17. openzipkin/zipkin. *GitHub*. Retrieved February 29, 2016 from https://github.com/openzipkin/zipkin

Tracing user transactions through a complex, multi-tiered business application: Appendixes

Björn Kihlström
Institute of Technology, Linkoping University
bjoki902@student.liu.se

Simon Gustafsson
Institute of Technology, Linkoping University
simgu002@student.liu.se

1. COMPONENTS OF THIS PROJECT

The implementation of the tracing system was composed of multiple parts of varying size and complexity. These are listed and described below.

1.1 Java components

Most of the work during this project is centered around the Java components, making up easily 90% of the body of work. The Java components are:

1.1.1 The OrientDB sink

This component is easily the largest and most complex Java component. It is responsible for persisting the trace data to OrientDB and building all logical connections. The final version is 885 lines of Java source code and it has been rewritten from scratch twice. During the time when this was worked on, all other Java components aside from the simulator were completed in parallel.

1.1.2 The JDBC table source

This component is responsible for periodically consuming the entire table of database logs and sending them into Apache Flume as Flume events. It contains a queue where all consumed events are stored. This is because Flume consumes a single event at a time and it is undesirable to query the database too often. It is made up of 266 lines of code. It was implemented in less than a week.

1.1.3 The JSON line event deserializer

This component is responsible for deserializing JSON events from a file storing one JSON object per line. This is to allow a standard Spooling directory source to consume events generated by the tracer module and deserialize them in a format that makes sense. It is made up of 109 lines of code in its final version and has been rewritten from scratch once (prior to that it was larger). It took three days to implement the first time and another two to rewrite.

1.1.4 The tracer API module

This component is responsible for exposing an API that produces files that can be read using a spooling directory source with the JSON line event deserializer. It is called from within IFS Applications and contains some JavaDoc strings for clarity and ease of use. It is made op of around 490 lines of source code and took around two weeks to fully implement.

1.1.5 The simulator

The simulator is a java program used during the performance evaluation. The final version is made up of 405 lines of source code, but code quality was not a concern during implementation. The last version of the simulator was implemented in less than a day.

1.2 Trashed components

Some java components were explored and partially implemented but were abandoned at some point. These did not make it into the final version of the project.

1.2.1 Support for non-synchronized system clocks in the OrientDB sink

This was implemented in a non-functioning state in earlier versions of the sink. Three weeks of work were put into this before it was abandoned because the amount of available log data did not allow for such a solution to consistently function. This was due to the problem of log events arriving asynchronously at the sink, and the only way to properly sort them was using time stamps. The support for non-synchronized clocks was based on the presence of a cache that is not part of the final solution as it did not work at all with given data.

1.2.2 An alternative source for middleware logs and associated tracer API module

Writing serialized JSON to disk may not be the most efficient method to send logs to Flume. A variety of IPC-based solutions were tried and scrapped because they had unsolvable problems associated with them. One week was spent on this.

1.2.3 The old simulator

Before the simulator that was actually used was developed, a simulator generating random trace data was being developed. Not a lot of time was spent on this, and a few parts of the solution made it into the final simulator which executed static scenarios. There was also a non-Java component to the simulator initially, in the form of an Arch Linux virtual machine that would have been used to create a controlled lab environment. This proved unnecessary and added too much difficulty to the implementation. Under a day was spent on configuring it.

1.3 Other components

A fem other components were part of the project. Building and configuring Flume to execute on Windows took around 3 days. Aside from this, a few stored database procedures were written for demonstration purposes. These are small in code volume and the time taken to implement them was no major issue.

2. DIVISION OF LABOR

During this thesis work two different ways of dividing labor have been used at different stages of development. The first is divided effort based on the natural division of the system and the second is collaborative effort through pair programming.

2.1 Divided effort

Throughout most of the project there has been a natural division between different tasks based on how the system has been structured. The main body of work has been located in both ends of the system; either where trace logs are generated and consumed by Flume for transportation or at the central logging server where trace logs are persisted to OrientDB.

During the phases of the project where such a division of active tasks was possible, the tasks were split in such a way that they could be done in parallel. The following procedure describes the work flow used:

1. First thing in the morning, recap the day before.
2. Do you have a task? If so, do it.
3. Have you completed your task? Claim another if possible. Inform your partner.
4. Are you unsure about how to proceed? Talk to your partner.
5. Are you both still unsure about how to proceed? Talk to IFS.
6. Have you or your partner ran into a difficult problem? Discuss it on a conceptual level using pen and paper, maybe use pair-programming to solve it if it is very important to be on the same page regarding implementation.
7. Do you need your partner to complete his task before proceeding? Finish that task using pair-programming.
8. Is there only one general task at hand, or several that cannot be done in parallel? Use pair-programming.
9. Discuss the state of the system and a strategy for how to proceed at the end of the day.

Most of the project has followed this procedure.

2.2 Pair-programming and collaborative effort

When it was impossible or impractical to divide the effort during a stage of the project, pair-programming or some similar kind of collaborative effort was chosen as a strategy.

When this was done it was fundamentally due to one or more of these reasons:

1. It was important to be on the same page (common near the start of the project, also applies to writing the report).
2. There were no other tasks at hand (common towards the end of the project, also applies to writing the report).
3. The task is complex enough to demand the attention of both parties (happened sporadically).
4. Waiting for the other party to complete a task would otherwise force one party to be idle (happened rarely and mostly near the start of the project).

The application of this technique to the report can not be called “pair-programming” because it does not involve programming, but the collaborative effort (as we choose to call it in this case) was structured in the same way.

2.3 Concrete division of tasks

Divided effort by both participants in different stages of development where it is hard to separate what has been done by what person is referred to as “mixed effort” below.

- The OrientDB sink: Programmed during divided labor completely by Simon. Refactored for stability and style towards the end using pair-programming.
- The JDBC table source: Programmed during divided labor completely by Björn.
- The JSON line event deserializer: Programmed during divided labor completely by Björn.
- The tracer API module: Programmed during divided labor completely by Björn.
- The trashed non-functioning IPC sources and associated tracer module: Programmed during divided labor completely by Björn.
- Stored OrientDB functions: Mixed effort (hard to say who wrote what).
- Setup of Apache Flume on Windows: Mixed effort with some collaborative effort in the beginning. Mostly Simon.
- Setup of simulator, including trashed ideas: Mixed effort. Mostly Simon on the setup of the Arch Linux virtual machines that were not used. Mostly Björn on the random simulator that wasn't used. The final simulator was developed from small scraps of the trashed random simulator using pair-programming.
- The report. Fully written together by collaborative effort. No pieces written individually.

3. WORK FLOW MOTIVATION

The method was chosen mostly based on practicality. Pair-programming has many advantages, but was mostly unsuitable for this project. Two of the main advantages of pair-programming are [2]:

1. The improved quality and readability of the code.
2. The improved overview and understanding of the system for all parties involved.

The main problem with pair-programming in this project which caused us not to adopt it as a main method was that the different components were interdependent. To be able to test one of them in terms of functionality, a working version of at least one other component was usually needed.

This (combined with the fact that only two people were working on the project) made it very impractical to use the method as it would result in one out of two cases:

1. The entire team implemented a large portion of functionality without an ability to test it.
2. The entire team would constantly be switching tasks, making it difficult to wrap their heads around more complex problems.

The method mainly used instead was the divided labor described above. When one component was ready for some functionality tests, the other one would also be ready for the same tests. This allowed for much quicker iterative testing, which would have been impossible using pair-programming.

Regarding the overview and understanding of the system, the interface-based programming paradigm which was enforced by the framework associated with Apache Flume caused implementation details from other components to be fairly unnecessary during the implementation of any one component. The fact that the interfaces implemented were also specified by Apache Flume and as such could not be changed during implementation also made it more relevant to discuss the system in very general terms.

How to send and receive data was never an issue as the interfaces already specified that. It was always rather a question of what data to send and how to format it. This made it more

useful to discuss the system at a much higher level, especially as the implementation details changed often and without notice.

As such, implementation was left at a component level which only made it necessary to understand the structure and high-level functionality of the system during development. Knowing what the JDBC source does is more relevant than knowing how it does it when working on the OrientDB sink, for example (the other way around is also true).

3.1 The agile process during development

Early in the project, development was done without much in-depth understanding of the functional demands posed by IFS Applications on the tracing system. As such, the need for agility was emphasized.

It was not only a matter of understanding the high-level structure of the system, but also a matter of being able to change it quickly.

No scrum board has been used for task management, because implementation demands have changed too rapidly. The process has been in line with very rapid Extreme Programming aside from the lack of pair-programming in most cases [1]. Functional demands were very prone to change as problems were discovered during acceptance tests, and sprints were a single week in length.

As such, most of the time there were only very few active tasks at any given time, which made actual SCRUM [3] using a board redundant. Priorities also had a tendency to change between task completions and sometimes during the course of a single day between daily standups, which made the use of a board in order to prioritize several tasks into nothing but unnecessary administration.

Two daily standups were done every day, one in the morning in order to recap and one at the end of the day in order to synchronize priorities and tasks. One meeting with IFS was held every Monday in order to demonstrate the current state of the project and to set goals until next meeting. This made sprint length into 1 week, effectively.

This process made all development into a series of small, low-risk gambles where losses could be cut short quickly. Iteratively, the gambles that paid off shaped themselves into a prototype.

3.1.1 Why do it this way?

This rapid development process is consistent with a project of an experimental nature. Quickly exploring possible solutions in this way where a losing gamble can be discarded quickly provided the project with many different explored avenues, as seen in the discarded parts of the project.

The ability to quickly discard a dysfunctional solution was intentional as knowledge of the problem and associated demands was known to be very meager in the beginning of the project. Flume was given three days to build and run on Windows before being discarded, and this was close to happening. Had this happened, the project would have taken a drastically different turn. In fact, not a single line of code written would have been the same.

The same applies for every chosen component. Only the ones that worked and met at least a sizable portion the demands within a relatively short period of time (measured in days) made it into the solution. Gradually, the method improved and refined our understanding of the problem enough to make more informed decisions. It also limited the number of options available, as increasingly large portions of the implementation had already been decided upon. This gradually reduced the risks taken during the small gambles.

Parallel work during these experimental phases also led to being able to try more options in the same amount of time. This is due to the nature of parallel trial and error in different domains. Even if pair-programming produces better code in less time, the idea of these gambles was to ditch bad solutions quickly. Improved code quality in these cases may have upgraded terrible solutions to just being bad or mediocre (bordering on bad), which in turn is contradictory to the purpose as it may not have been abandoned as quickly (or at all) if this was indeed the case.

What made this a fitting method was, in short, that this project was a completely blank slate in the beginning. And the fact that many different approaches had to be tried in a short time.

3.2 Use of pair-programming

Pair-programming had a purpose in some situations during the project. The main portion that would have benefited the most from pair-programming would have been the OrientDB sink. This was not the case from the beginning, however, as too little was known about the system during this time to build a sink that functioned according to specifications. As such, experiments had to be undertaken even at the sink side of the implementation.

This caused the sink to be completely rewritten twice. The first few gambles did not pay off, so the sink was rewritten once early in the project. The second time, a sound logic had started to take shape as a result of experimentation and it was known how the sink could work properly. The problem was that the code was so convoluted at this stage that the sink would have to be rewritten again for the emergent sound solution to be readable and of good quality.

For this purpose, pair-programming was suitable. The second time the sink was rewritten, quality and readability were prioritized and the risk of change was significantly lower than earlier in the project. This was done through pair-programming to produce the final version of the sink.

3.3 Planning the work flow

Implementing the system in this way was not planned in great detail from the start, but like the system itself it came into form through a series of small, pragmatic decisions during the project. The most important thing from the start, once the open card sort had been carried out, was to begin implementation. The work flow formed through the same iterative process that shaped the system.

Bad work methods were quickly discarded and those that produced results were retained. This formed the method described above, reminiscent of Extreme Programming. Although, the phrase “Extreme Programming” was never used until the similarities were noted when the project was practically finished.

The main purpose of the work flow was exploratory; to minimize the risk of change, failed attempts and wrongful assumptions. To plan work flow ahead of time was counter-productive with regards to this purpose.

3.4 About the possibility of test-driven development

Test-driven development was not even attempted at any stage of the project. The inherent impossibility of predicting enough details about what the result of an operation should be made it very difficult to write any kind of unit-tests ahead of time.

The other major problem with unit-testing was the fact that the implemented parts of the system did not do much at all on their own but required the rest of the system to be present in order to do anything at all. The OrientDB sink could not be tested, aside

from some minor implementation details, outside of Flume. The same is true for most other components in one way or another. The fact that almost all parts of the interface do not return anything also did not help in this respect.

This exposes several limitations with regards to unit-testing. The contract is very small and the desired result is only consistent based on earlier events. Testing the implementation goes against the paradigm of interface-based programming. This, combined with the uncertainty regarding any made assumptions about correct behavior made test-driven development impossible or at least very hard in the context of this project.

Another aspect that made the test-driven development paradigm undesirable in this case was the fact that the project was exploratory in nature. As many aspects as possible should be covered in an exploratory project, and one of the main purposes is to discover emergent problems and approaches during development. Test-driven development does not further this purpose.

All in all, tests performed by manual examination of generated traces proved more in line with the purposes and limitations of the project.

4. WORK FLOW EVALUATION

Generally, the work flow has been satisfactory with regards to the purposes of the project. The purposes, however, are key to the efficiency of a work flow that works as described. This work flow would not be fitting for all kinds of projects, or even most kinds.

4.1 When it works

Generally, the projects where the properties coincide in a very specific way are the only ones that would benefit greatly from this kind of work flow. This project fit all the criteria, and as such it was appropriate in this case. Most projects would not meet these criteria.

4.1.1 Small team

The first prerequisite for the work flow to be sound for a project is for the team to be very small, as in two or three people. Larger teams would find more benefit in using pair-programming to greater extent. The main benefit of not using pair-programming very much during this project was the possibility of rapid integration. With a larger team, this could be achieved anyway. In a larger team, gaining an overview and synchronizing work at a higher level would also have been much harder if everybody worked individually.

4.1.2 Short project with short sprints

As the project was short, there was only a limited amount of parts to implement. This kept the project from ever becoming too difficult for any one person to grasp at a higher level. For a longer project, this would become much harder and require a different work flow, for example a more formal agile method using a SCRUM board. Short projects also diminish the availability of time that can be used for proper planning and forces rapid development.

4.1.3 Low level of previous knowledge

If a known solution to the problem can be easily implemented by the team based on familiarity with the subject, most of the advantages with this work flow would be lost as the need for blind experimentation diminishes drastically.

4.1.4 Blank slate project

A prerequisite for this work flow to be useful is for very few parts of the solution to be finished or even decided upon. This makes for a situation where many different possible approaches present themselves to the team. Thus the need to be able to change approach quickly when these approaches do not perform according to expectations upon further investigation.

4.2 Conclusions drawn

There is a kind of project that fits the above description, and that is a low-resource, short-term, exploratory prototype implementation done for the purposes of trying something out without investing too much into it. Putting one or two people on a short exploratory project is not unheard of. In these cases, where the purpose is purely exploratory, this work flow is very fitting.

The exploratory nature of such a project demands that as many different angles as possible are explored. Administration can be kept to a bare minimum during these projects as all time spent on administrative tasks is time that is not spent exploring and trying things out.

For any other type of project, this method should be more formal and administered in order to ensure correctness and quality as well.

5. REFERENCES

1. Kent Beck. 1999. Embracing change with extreme programming. *Computer* 32, 10: 70–77.

2. Frank Padberg and Matthias M. Müller. 2003. Analyzing the cost and benefit of pair programming. *Software Metrics Symposium, 2003. Proceedings. Ninth International*, IEEE, 166–177.

3. Ken Schwaber. 1997. Scrum development process. In *Business Object Design and Implementation*. Springer, 117–134.