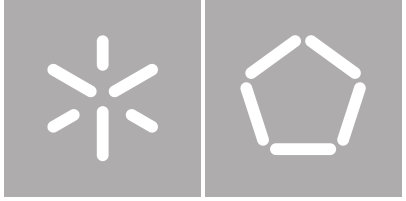
Universidade do Minho
Escola de Engenharia

Pedro Miguel Pimentel Guimarães

Monitoring and analysis of queries in distributed databases

Outubro de 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Miguel Pimentel Guimarães

Monitoring and analysis of queries in distributed databases

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor José Orlando Pereira

Outubro de 2014

Anexo 3

DECLARAÇÃO

Nome

Pedro Miguel Pimentel Guimarães

Endereço electrónico: pg22834@alunos.uminho.pt Telefone: 253 213 230 / _____

Número do Bilhete de Identidade: 13578590

Título dissertação □/tese □

Monitoring and analysis of queries in distributed databases.

Orientador(es):

José Orlando Rangel Pereira

Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

MESTRADO EM ENGENHARIA INFORMÁTICA

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, 31/10/2014

Assinatura: Pedro Miguel Pimentel Guimarães

Agradecimentos

Antes de mais queria agradecer ao meu orientador, Prof. Doutor José Orlando Pereira por toda a atenção, preocupação e tempo despendido bem como o grande auxílio prestado e que possibilitou o sucesso desta dissertação.

Queria dar os meus agradecimentos a todos do laboratório, em particular ao João Paulo por me ajudar a instalar e disponibilizar os recursos necessários, ao Miguel Matos pela ajuda com o Derby e com o resto da plataforma CoherentPaaS e ao Ricardo Vilaça, pela ajuda na optimização das escritas no HBase.

Agradeço a todos aqueles com quem tive conversas um pouco aleatórias, por vezes inspiradoras, filosóficas ou simplesmente tolas nos tempos mais livres e a todos os outros meus companheiros de luta, que me foram motivando.

Finalmente, obrigado à minha família, especialmente ao meu Pai, Mãe e Irmã, por todo o apoio, amor e carinho que sempre me deram.

Abstract

“ ***Monitoring and analysis of queries in distributed databases*** ”

Scalable database services combining multiple technologies, including SQL and NoSQL, are increasingly in vogue. In this context, the CoherentPaaS research project aims at providing an integrated platform with multiple data management technologies, united by a common query language and global transactional coherence.

For this integration to succeed, it must provide the same monitoring capabilities of traditional relational databases, namely, for database administrators to optimise its operation. However, achieving this in a distributed and heterogeneous system is in itself a challenge.

This work proposes a solution to this problem with X-Ray, that allows monitoring code to be added to a Java-based distributed system by manipulating its bytecode at runtime. The resulting information is collected in a NoSQL database and then processed and visualised graphically. This system is evaluated experimentally by adding monitoring to Apache Derby and tested with the standard TPC-C benchmark workload.

Resumo

“ ***Monitorização e análise de interrogações em bases de dados distribuídas*** ”

Os serviços escaláveis de base de dados combinando diversas tecnologias, incluindo SQL e NoSQL, estão cada vez mais em voga. Neste contexto, o projeto de investigação CoherentPaaS tem como objetivo oferecer uma plataforma integradora de múltiplas tecnologias de gestão de dados, unidas por uma linguagem de interrogação comum e por mecanismos de coerência transacional global.

Para que esta integração seja utilizável na prática, é necessário que ofereça as capacidades de monitorização que são comuns em bases de dados relacionais, por exemplo, para que o administrador de bases de dados seja capaz de otimizar a sua operação. No entanto, a concretização desta funcionalidade num sistema distribuído e heterogéneo é em si um desafio.

Este trabalho propõe uma solução para este problema com o sistema X-Ray, que permite adicionar a capacidade de monitorização a um sistema distribuído em Java através da ma-

nipulação do código-objeto em tempo de execução. A informação resultante é recolhida numa base de dados NoSQL e depois processada e visualizada graficamente. Este sistema é avaliado experimentalmente aplicando-o ao Apache Derby e utilizando o padrão de testes TPC-C.

Contents

Glossary

bytecode

or Java bytecode, it's a portable low-level instruction set, resulting from compilation of Java programs and designed to run in the JVM.

CCT

Context Calling Tree.

CEP

Complex Event Processing.

CQE

Common Query Engine.

DBMS

Database Management System.

JMX

Java Management E**x**tensions.

JRE

Java Runtime Environment.

JVM

Java Virtual Machine.

logback

a flexible and extensible logging framework that implements the Simple Logging Facade for Java (slf4j) API.

MDC

Mapped Diagnostic Context.

NoSQL

used to refer to databases that don't typically fit in the traditional relational paradigm. These generally favour availability, speed and efficient storage above consistency and ACID guaranties.

opcode

operation code; the part of the instruction that specifies the operation to perform. It's normally followed by its parameters if required.

slf4j

Simple Logging Facade for Java.

SQL

Structured Query Language.

type

a particular kind of data. Normally it denotes a reference (non-primitive) type, like interfaces or classes.

List of Figures

Listings

1 Introduction

1.1 Context

The use of multiple database management technologies, such as scalable SQL databases and key-value or document-oriented data stores, is increasingly in vogue. This is due to the increase of the amount of managed data that demands more efficient storage with high-availability. Other solutions were developed to satisfy requirements from recently growing domains as Machine Learning and Data Mining. NoSQL solutions are diverse and generally try to provide availability and partition tolerance guarantees at the cost of a reduced level of data consistency. Each of them is adequate to different situations and product requirements. One size does not fit all.

CoherentPaaS or "A Coherent and Rich PaaS with a Common Programming Model" is an European Union project [2] aiming to design and implement a PaaS (Platform as a Service) that allows access to a wide range of services and data storage technologies in the cloud. These include NoSQL-data stores (key-value data stores, graphDB data stores, document-oriented data stores), SQL-like scalable data stores (column-oriented data stores, in-memory databases, SQL databases) as well as Complex Event Processing (CEP) systems. Data will be available in a unified way through a common query language supporting global transactional coherence.

Figure 1.1 presents CoherentPaaS's general structure and main components. At the higher abstraction layer, the CoherentPaaS system includes the Common Query Engine, supporting the various datastores mentioned (on the bottom left of the figure). Cloud applications (on the top left) can be deployed and use those Database Management Systems (DBMSs) through the Query Engine. Other components necessary for the function of the

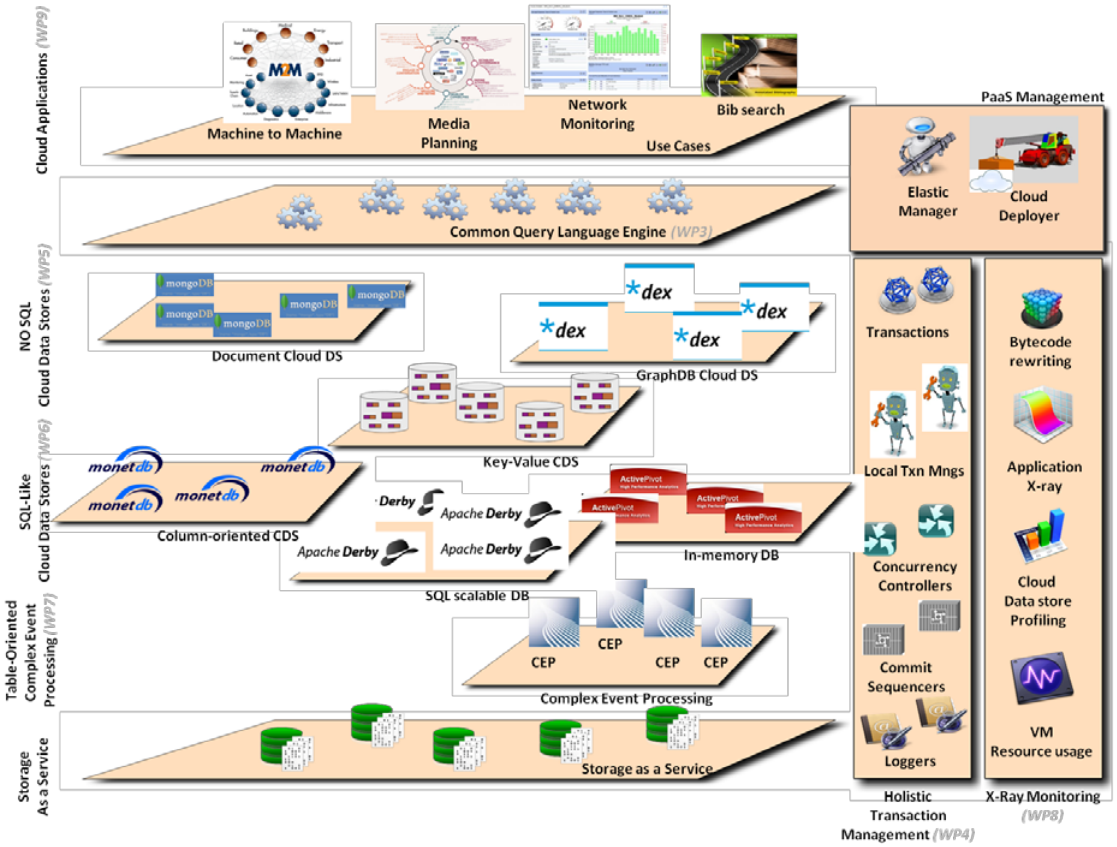


Figure 1.1: CoherentPaaS architecture.

PaaS are represented on the right side of the figure. They include transaction management and X-Ray monitoring, responsible for the monitoring, debugging and profiling of the cloud applications and databases (Figure 1.1, on the right). These facilities are used by PaaS management, responsible for deploying applications. Work developed for this thesis corresponds to the X-Ray monitoring module.

There are many scalable enterprise DBMSs solutions implemented in the Java language and among them are the most popular and used NoSQL ones. Examples of datastores in Java include HBase, Accumulo, Cassandra, Voldemort, Neo4j, Hazelcast, Apache Derby, Terrastore, InfiniteGraph, and OrientDB. The reasons for this are: a mixture of historical market share in commercial applications and middleware, the execution model in a virtual machine that enables its easy deployment independently of machine architecture or operating system[1] with increased security and a sizable ecosystem of open source, permissive licensed[2]

[1]This idea was codified in the slogan "Write once, run everywhere".
[2]See http://www.softwarelicenses.org/p1_open_source_glossary_permissive_licenses.php for a

frameworks and libraries used in conjunction or replacing various parts of the JavaEE platform. It is thus reasonable to choose Java to implement X-Ray. For other datastore systems not implemented in Java, a bridge or converter can easily be coded.

1.2 Challenges

Detailed execution plans, similar to the PostgreSQL's `EXPLAIN` and `EXPLAIN ANALYZE` commands – do not normally exist or are hard to gather in cloud platforms. This difficulty is due to:

- Obtaining the data requires the integration of several components;
- Some of these components do not offer the necessary features;
- Components are heterogeneous and distributed, so it is necessary to cross software and hardware frontiers to obtain global information.

These problems can be solved by monitoring each component in a distributed system. Then we can directly obtain information from the components and coordinate them, even if they were not designed with integration in mind. But this is not enough because when some component lacks a certain feature it is necessary to add it, and it may be onerous or even impossible to add it in the monitoring layer; maybe a more direct interaction with the component is necessary. Likewise, to support advanced debugging techniques, changes in the component can be necessary.

Another concern is how to change the components. Source code of some elements might be unavailable, as it can happen when using commercial software. Even if accessible, it can be infeasible to change it for some reason or being just plain burdensome to do so.

It also must be powerful. Consider a typical use case to epitomise this. Suppose one wants to alter a datastore to monitor some parts of its operation, typically how queries are processed. X-Ray must follow the execution flow of important events from start to end and filter irrelevant information. What matters here is to follow the execution when crossing

definition of permissive license.

from one task to another and be able to detect communication with other resources in the same or in a different computer. Task is used on this context in a generic sense, as a quantum of related work that corresponds to a singular operation in the optic of the system examiner. After collecting the data, it must process and present it in a useful format.

1.3 Goal and Contributions

The aim of this work is to equip CoherentPaaS with mechanisms to debug and optimise queries, comparable to those of traditional databases. These mechanisms are sufficiently powerful and generic to work in the several types of SQL and NoSQL supported by the project. They are designed to capture as well as possible the behaviour and inner workings of components. They are easy to use and expressive, allowing an user to only alter some parameters and obtain the expected results. Finally, these mechanisms are easily extensible, configurable, and customisable.

This work provides the design and implementation of a tool that:

- Adds the ability to report information to each component in a heterogeneous distributed database while:
  - Allowing users to choose what information to collect;
  - Working seamlessly across component and distribution boundaries.
- Enriches each event with additional information, as needed [24];
- Does not require source code to be available;
- Is applicable in the context of distributed databases.

1.4 Document Structure

The rest of this dissertation is organised as follows: Chapter 2 summarises the state of the art and background concepts; Chapter 3 presents the architecture of the developed solution, starting by a high-level overview, followed by a explanation of each feature and usage method,

and finally the configuration options available; Chapter 4 details how that architecture was implemented, referencing used technologies and important information about components or instrumentation aspects; in Chapter 5 the solution is evaluated experimentally with a case study. Finally, Chapter 6 discusses the conclusions and presents a brief description of the directions this work can evolve in the future.

2 State of the Art

There are numerous debugging and logging techniques, in multiple platforms, and for multiple purposes. In particular, it is possible to make a distinction between debugging technologies that work and are designed for distributed databases and general solutions for debugging, monitoring, and event logging. The topics of instrumentation and program analysis are analysed as well as core concepts necessary to better understand challenges and questions associated with each solution.

2.1 Monitoring and Distributed Databases

Suppose a Structured Query Language (SQL) query is executed in a database server. Tools like *pgAdmin*[1] easily allow the generation of a graphical representation of the query's execution plan, detailing each computation step, the elements involved, and the operations applied. Figure 2.1 is an example of such graphical output.

Adequate debugging tools must allow the generation of similar visual representations for queries running on CoherentPaaS. The difficulties arise with both the heterogeneous nature of the databases and the distributed execution context.

Data storage monitoring solutions are for the most part database-specific, for example, SQL, key-value store, or document-store specific. Moreover those solutions do not work for all data stores of that kind, but only on specific engines. For example, we could have a solution available to MySQL but incompatible with PostegreSQL databases.

Other tools have some limitations that would hinder their use in X-Ray, as they are implemented as an extension to some datastores or were conceived to solve very specific

[1] See http://pgadmin.org/ for more details.

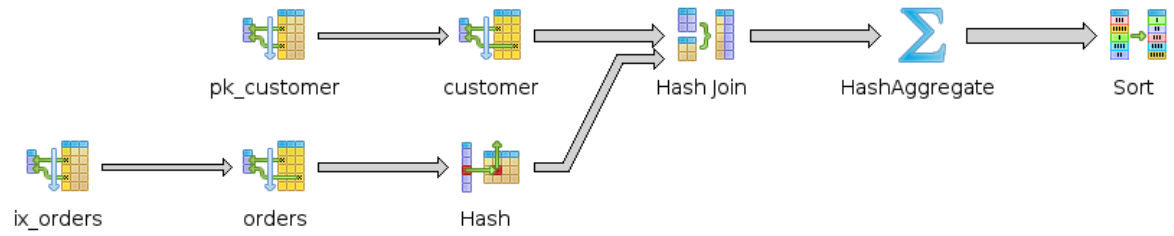


Figure 2.1: Representation of an execution plan generated by *pgAdmin*.

use-cases.

For example, the system described in [20] to extend PostegreSQL adds a message-passing system to a normal distributed database; the resulting system can send and receive messages besides upholding the normal database operations. Although interesting, the system only works on PostegreSQL databases and the message-passing model offers some drawbacks.

2.2 Distributed Application Monitoring

Another approach is to use existing general monitoring frameworks already capable of operating in a distributed context. This path, by its very nature, avoids the problem with the different datastore types by being agnostic to them.

An attractive system that follows this model is presented in [31]: considering that information about the components or middleware of a distributed service may be minimal and the source code unavailable, the proposed solution is a request tracing tool that worked despite this fact. To that end, each component is considered a black box, a device that receives input, processes it in an unknown fashion and returns an output. As the processing is opaque to the rest of the system, the tracing tool is used to follow requests, as they are passed between components until the computation associated to the request is produced (and optionally sent to the entity that made the request, if it required some kind of response). A per-request data structure capable of representing this information is defined, the component activity graph (CAG). This is a graph that represents causal paths between activities of

the components [31]. A means to recognise the most frequent paths is also provided.

This analysis is made on-line as the system and the logger nodes are running, without the need to stop them. Information can be collected on demand, meaning that it can enabled or disabled, or done intermittently, using sampling. The detection of communication from one component to the others is made within the kernel when a send or receive system call is used. SystemTap [6] is used to enable this detection and transmit this information to the appropriate nodes. So this solution cannot be used for CoherentPaaS as it is not OS independent and has no JVM support implemented. Besides, although the principle that no information exists about the components is a valid one and the results obtained are useful, this project makes different assumptions. Even if the source code of a component cannot be altered and deployed, it is still available or its general API is, and so more information can be extracted, if it is of interest to the user.

2.3 Debugging Applications

An alternative is to consider debug methods used for normal applications – i.e. those running in sequential or concurrent systems. They are not necessarily applicable to a distributed system, but while those techniques may be insufficient by themselves, they can be used to introduce some important concepts. Besides, analysis methods for distributed programs are inspired or built on top of these.

2.3.1 Metrics

The first approach is to make use of performance statistics and collect **machine/architecture-specific metrics**. If the same setup is used under different conditions and a sufficient number of times, statistically derived-data is found and validated. By running tests on other machines or simply considering the similarities between most modern computer architectures, it is possible to obtain global valid conclusions about an algorithm or program. Execution times, cache miss rate, percentage of failed predicted branch jumps are platform-dependent metrics, as they directly depend on the underlying platform (and so very different results for the same benchmarks can be obtained on different machines).

Other possibility is focusing in capturing **platform and/or architecture-agnostic metrics**, valid in any execution environment. Leaving out details that depend on the running platform reduces some utility of the metrics, as information is lost. In contrast, the analysis of the data or at least its validity to all systems is facilitated. For some cases, the missing information is not relevant nor needed; in a sense, collecting platform-agnostic data can be equivalent to filtering the noise. Examples of platform-independent metrics include the calls made in the execution of a program – number and relation. In the Section 2.3.2, this is expanded upon.

One example where platform-independent metrics can be very important is when developing applications for deploying in the Java virtual machine, where analysis should target the developed application, not the underlying system nor the JVM implementation (or even Java processor [33]) used. This does not mean platform-specific information cannot be measured and information general enough cannot be derived from there, through analysis or by mere empirical observation. It also does not mean that these metrics are in any way incompatible with each other or redundant. They are distinct only in the formal sense and can be freely combined to obtain a better understanding of applications. And sometimes, the base platform or JVM is indeed part of what is being investigated.

2.3.2 Call context and tracing

The relation between methods or actions is very important when examining data management applications. It should be possible to visualise how an action is composed and what sub-actions compose it. Actions could be methods or something more abstract. This is important because it allows the visualisation of a produced query execution plan, similarly to how some database management tools support it (cf. Section 2.1).

Visualisations might be obtained by a direct use of the same algorithm used in those solutions or maybe by applying the same concepts in a slightly different way [8]. This is related to the concept of software tracing: the act of obtaining data about an execution and its logical ordering, for an intra-program view, normally associated with a lower level analysis than event-logging.

The representation of the methods called when executing a program is named a call tree.

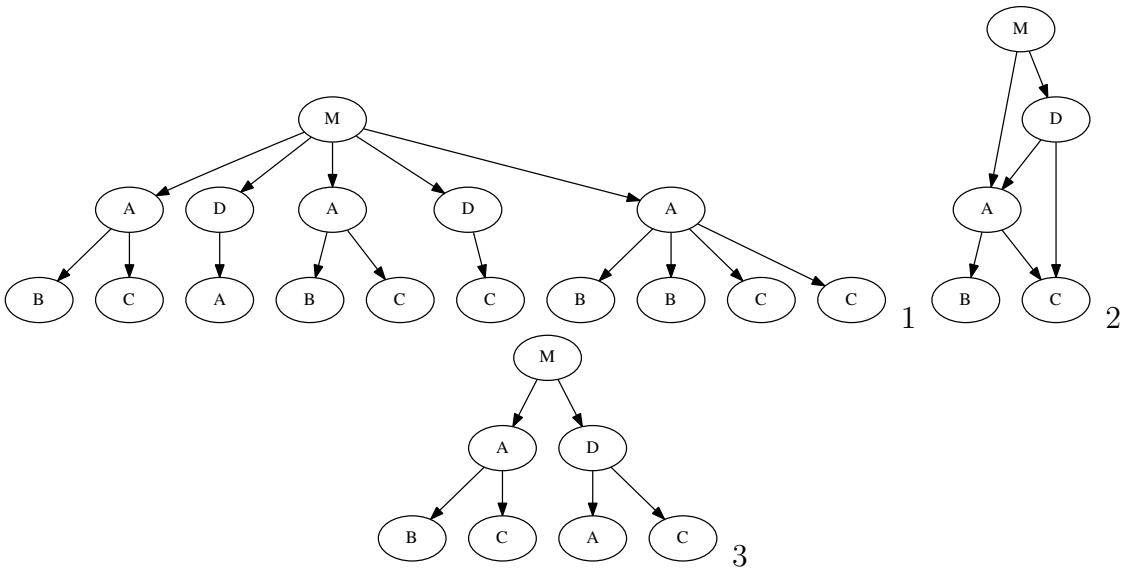


Figure 2.2: Different representations of calling relationships: 1) call tree, 2) call graph and 3) context calling tree.

Each node, besides the information of how a method relates to other method, can also store information such as execution times [9]. The call tree format allows the representation of all the original run information and includes as separated nodes different calls to the same function. Because of this, its size in memory can grown quickly, especially if one considers cases of call recursion. The detailed context present can also be unnecessary if the distinction of call sites can be inferred or is not important.

In the other extreme is the (dynamic) call graph. This structure uses only one node to represent a function, independently of the number of times or different contexts it was called. So although compact, it is difficult to extract meaningful information from the nodes, as context is lost and metrics associated with each function call are lumped in a node [8].

The Context Calling Tree (CCT) is a compromise between these two formats. It groups together several method invocations in the same node if and only if they share the same context – meaning that the invoked method is the same, and their parent trees are equivalent.

These trees can also be called dynamic context calling trees when they are constructed from a run, as it normally happens, due to the fact that static calculation of all possible call relations is a computationally intensive problem.

A distinction can be made between complete and a partial CCT. Complete CCTs con-

tain information about all calls and are generally built using complete code tracing, whereas partial CCTs are built with incomplete call and context information, normally because periodic sampling was used to obtain the call stacks. This technique fetches information and updates the CCT in intervals defined in some way and has less overhead than constant tracing. The consequent loss of accuracy can be reduced by using a shorter time-span between measurements or a more advanced sampling algorithm, such as adaptive bursting [37].

2.3.3 Data collection

Independently of the data being collected, there is the question of how to obtain that data. Some forms of collection may be more suitable for obtaining some metrics while inefficient for others. Besides the ease of use and the quantity and quality of the obtained information, we should consider the inconvenience and overhead caused by data collection. Here we can distinguish between application runtime overhead, application modification overhead, and cognitive overhead (if the changes to implement are not straightforward or require us to change how we rationalise the problem).

Generally speaking, three approaches to data collection can be employed: using hardware aid; using existing software to assist the debugging (usually leveraging on the existing functionality of the host JVM or OS, at other times by using an external program) or altering the program to collect them, that is, using instrumentation.

Instrumentation

Instrumentation is the act of adding to programs behaviour that allows the ability to measure or know some information when they are run. Though source code can be manually altered, this approach often involves repetitive work and mixes business logic and debugging logic, which should be separated. Normally, it is done automatically and what is changed is the program low-level representation, if dealing with a compiled language. When using a language that compiles to native code, the generated assembly is modified but when using languages that target the JVM or any virtual machine then the created bytecode is altered. This alteration can be made statically, in one occasion, or dynamically, every time the program is loaded and instrumentation is required.

Generally, the source code is not necessary, but it can be needed to aid the instrumentation effort, depending on the complexity of the application and the intended analysis. The overhead, while acceptable, is not negligible and should be considered [37]. Altering a program's representation is a means for instrumenting a program, but not a method and in fact one can use instruction rewriting to monitor various things in very different manners. Or we can apply it for completely different uses, such as generating boilerplate code [5], processing server annotations [7], and aspect-oriented programming [1].

The architecture of the instrumenter can also be very diverse. It can be a single, simple program that alters an application or it can be part of a modular system capable of being extensible with a wide range of plug-ins or similar tools. Examples of monitoring or tracing software that use instrumentation include the venerable gprof, valgrind [26] and its tools (memcache, callgrind, Google's Flayer [18], etc.), Intel's Pin [4] and tools. We can find various solutions specific to the JVM (Java Virtual Machine), maybe because the JVM abstracts away many platform and hardware details and provides some features that enable it, like Java Agents, instrumentation interfaces, custom classloaders and JVM TI. Bytecode, the instrumentation target, is also a relatively simple and high level target when compared to assembly. Examples of application debuggers include jp2 [32], JBInsTrace [12], and iPath [10].

Hardware counters

Hardware performance counters, or simply hardware counters, are special-purpose registers built into modern microprocessors to store the counts of hardware-related activities, such as cache misses, branch mispredictions or number of cycles executed. These counters can be used to conduct low-level performance analysis or tuning with very low overhead, especially compared to software instrumentation [8, 34]. The number of available hardware counters in a processor is limited and not every useful information can be kept. So it is necessary to conduct multiple measurements to collect the desired metrics or to continuously process or save this data. The types and meanings of hardware counters vary from one kind of architecture to another due to the variation in hardware organisations so they cannot be relied upon as an universal solution. Lastly, it can be difficult to correlate low

level performance metrics to the source code or application level [8].

2.3.4 Tracing tools

Native tracing solutions, that address programs compiled to native code, are not useful for Java programs, as they would end up tracing the JVM itself and not only the target program. This is because a Java program is not a self-contained executable but rather a piece of bytecode that is run by a JVM. So debugging a given program in that execution environment would be difficult. Some sort of support to explicitly separate the program from the virtual machine could be made but it would not be trivial.

Even frameworks that might offer some kind of support to programs in JVM still they go against the Java philosophy of compile once, run everywhere. Normally these solutions are only available in certain operating systems and as most use instrumentation of machine code or depend on some kernel or OS functionality, adding support to a new architecture may not be trivial. Native solutions neither can easily access JAVA code, which could be a valuable ability nor are they generally enough high-level. So in general a specialised tool will be a better fit.

General debugging frameworks like valgrind (and its tools like callgrind, Flayer, etc.) or Pin are powerful and extensible with the goal to easily allow building new tools by taking advantage of the provided features and the plugin architecture model. But they are native solutions and so are not compatible with CoherentPaaS.

iPath [10] is a dynamic instrumentation tool with some interesting properties for a distributed setting. It allows a more focused analysis as the methods to analyse can be chosen and this selection can be altered at run-time. Best of all is that it works on distributed systems. Yet it is a native solution and although one can choose what methods to instrument, when the call stack is walked to update the calling context information, all methods including those that were not declared for observation will be recorded in the calls information structure.

Solutions involving bursting or sampling are not generally interesting, as they seek to solve almost the opposite problem of this work's: they need to analyse the *whole* program and to compensate for this computational weight a lighter collecting process is used. This

contrasts with the need to analyse *some parts* of the program in a way that does not lose a invocation, which is lighter than instrumenting the whole code and is executed less times, so it can be heavier.

So it can be concluded that those solutions, by the intention they were designed, are not adequate in this instance. Still, it should be noted that some advanced algorithms can have a decent accuracy, matching about 84 % of the full context tree [37] [35].

Next are the JVM-specific tools, more viable candidates for possible solutions. Most of these tools focus on complete knowledge of the whole execution instead of finding when something interesting is happening. They normally run as an agent specified at run-time with a parameter at the command line or programmatically. That way any standard production JVM, is capable of executing the instrumenting code. They instrument a program's methods and some also can instrument native (JNI) and JRE classes' methods. Some allow for fine tracing, at the block level. Examples of JVM tracers include JBInsTrace and jp2, among others.

jp2 [32] is an interesting example, as it allows one to query the trace results, previously saved to an XML file, using the xQuery language. But the focus is still the gathering of excessive, and lower-level information than what matters to CoherentPaaS: it is done logging on all the methods (it is possible to disable tracing only per thread) and there is the ability to distinguish different calls to the same target method, even if from different positions in the same source method. These are excessive details. None of the presented technologies is a distributed and capable tracing solution for the JVM capable of fulfilling the project requirements.

2.4 Event Logging

Event logging is an action to obtain high-level information from a running program. According to Chandy, an event is a significant change in the state of the universe[13]. It follows that only changes interesting to an application are significant changes, and so, are considered events. Events must be observed to be reported and processed. They can be processed where they were first reported or transported to an event consumer. Normally

an event includes a timestamp and other descriptive parameters. The distinction between logging and tracing is that tracing is more of a low level method. Events logged normally have a specific structure/format that eases their processing and are normally more important to a developer or administrator, not to a developer. Events are normally used to change the behaviour of a system depending on some conditions, to monitor performance or to detect distributed/concurrency problems or security breaches [24].

Event logging is related to message-passing systems, as a possible mean to distribute the events to consumers and to continuous queries.

An example of a logging structure is EV-Path [19], a middleware infrastructure that extends a publish-subscribe system. It offers the flexibility needed to support the varied data flow and control needs of alternative higher-level streaming models. Others include Netlogger [22], WebLogic Event Server [36], and PreciseTracer [31].

These tools emphasise the logging infrastructure, the log producers and consumers, and how events move between nodes, but do not focus on how they are produced or assume access to the applications' original source code. But it might not be available or be feasibly to change it and re-deploy the application. Besides, the structure of a event's log still has to be decided having in consideration the domain-specific peculiarities of distributed databases and user requests.

2.5 Discussion

Succinctly, many of those solutions are not available in a distributed context or are not usable in a way that addresses the target problem. Data store monitoring solutions are for the most part database-specific. General debugging and monitoring solutions have got some problems. As the name implies, these solutions are general, thus they do not take in account the particularities of databases or distributed systems. This is a surpassable limitation; another problem is that the majority of the existing solutions do not allow users to explicitly choose what to log. Thus excessive information that amounts to noise is produced and collected. Some solutions allow that choice, but not in a simple way.

Based on the analysis of previous work, a complete and adequate solution should:

Be based in JVM/Java: Even if it was possible to use native solutions instead, they would still have disadvantages when compared with a solution in Java;

Be dynamic: As long as performance does not take a significant hit, instrumentation should be dynamic, done each time it is needed. One time instrumentation adds a new step in the compilation process and is not that useful because between runs it is likely that the original program or the configuration options will be altered;

Be automatic: It should do as much as it can automatically, including everything that must be done to alter and get the analysed program running. After that, it should let users specify what can not be guessed and allow them to change the default configurations if they want;[2]

Focus on important details: When reviewing existing applications under the prism of query monitoring and debugging, it becomes evident how excessive detail can hurt more than help;

Be extensible: Examples of good tools are Pin, valgrind, etc. These tools are popular because they can be extended to satisfy particular needs and requisites, because they allow different people to use its code base as a foundation to build upon and avoid dealing with (some) low level details and needing to reinvent the wheel.

We also can infer technological dependencies and required components. For example, based in the first two conclusions we know that we need something capable of altering a Java program at runtime to use for instrumenting programs.

[2] An example of this is valgrind. Users can run it without any options and it produces useful information, but using some options it can do more operations; it might surprise some people that it can do more than check for memory leaks!

3 Architecture

3.1 Overview

Figure 3.1 presents the X-Ray architecture as conceived for CoherentPaaS, with the components designed and implemented in this thesis, completely or partially, coloured in red.

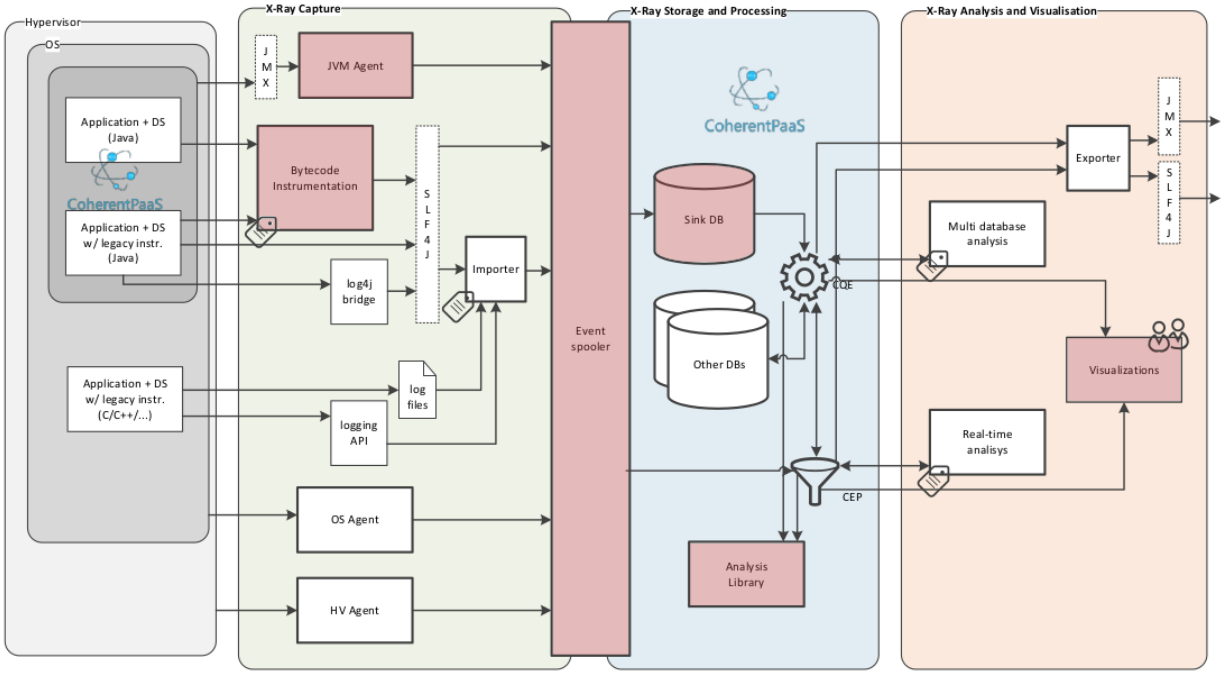


Figure 3.1: X-Ray sub-module architecture.

There is the assumption of the existence of distributed applications with software components in multiple servers, virtual hosts, and Java virtual machines. These applications generate monitoring events through the X-Ray Capture layer to the X-Ray Storage and Processing layer, to be used in the X-Ray Analysis and Visualisation layer. Label icons identify

the main configuration points for the system.

For Java applications, the X-Ray Capture module includes **bytecode instrumentation**, a mechanism for modifying compiled programs, by inserting into them code to generate logging events, maintain context, and allowing software probes. This is the main configuration point for Java programs. Generated events pass through slf4j[1] to an **event spooler**.

This **event spooler** enriches them with additional information and delivers them to the X-Ray Storage and Processing module. Bridges or file processing can be used to obtain information about non X-Ray-ready programs. Several **agents** responsible for monitoring and collecting metrics at different abstraction levels also supply the event spooler, namely, from Java Management Extensions (JMX) [15].

In the Storage and Processing module, events can be stored in a NoSQL database with high-throughput capacity (the **sink database**) or be fed to the **Complex Event Processing (CEP) engine** for real-time processing. A simple prototype of this database was implemented in HBase.[2]

The **Common Query Engine (CQE)** is CoherentPaaS's own database engine, capable of executing queries on several heterogeneous databases.

The **Analysis Library** contains analysis procedures applicable to the monitoring data, specially concerned with request tracking across software modules and components.

The data can leave for the analysis and visualisation module in a couple of ways: it can be **exported** to enable interaction with external systems, be **processed** (potentially in real-time) or be used to construct **visual representations** available to the end-user.

3.2 Main features

3.2.1 Probes

Probes implemented in X-Ray allow code execution on entry and exit(s) of selected methods. The target data – name and reference of executing class, name and signature

[1] See http://www.slf4j.org for more information.
[2] An Apache NoSQL database inspired by Google's Bigtable [14]. For more information about HBase, see https://hbase.apache.org.

of the method, the current thread and the parameters/return value – are accessible to all defined probes. Probes can execute any custom code, as long as it implements the required interface. That makes probes powerful and in fact, some of the provided functionality in X-Ray is implemented using them. For instance, they can be use to save information in a datastore or to do a simple local analysis.

Besides analysis, another use case for a probe, is to alter some parameters before passing them to the original method, or alter return values before the function end. Conditional actions can be implemented using probes: for instance, only do something if the return value is > 4 or if an exception was thrown.

Internally collected data is routed to probes through slf4j and logback, but this is only an implementation detail.

3.2.2 Tags

Instrumentation also provides the ability to tag entities being observed. A tag is an identifier that increases the data that can be collected by X-Ray, motivated by the recognition that certain computations happen in distinct contexts (even if the executed code is the same). A tag can be associated with an object or thread and through configuration instructions it is possible to generate a new one, remove it, move it, or copy it to a thread or object. Adding the choice between doing these operations on method entry or exit points, the result is a flexible mechanism enabling one to follow a request execution even when threads or objects are reused (e.g., the case when thread or object pools are involved). This also works in other similar situations in which long-lived applications need a low latency and the ability to answer a great number of clients, frequently recycling resources.

Unlike other configuration options such as the log level, tag indications are only applicable to methods, being optionally inherited. Indeed, as a fine-tuning option used to differentiate similar contexts, they would become meaningless and less useful if they were mass-assigned.

A tag is used as an identifier, so each tag value is a unique in a given JVM run, being unspecified how the values are formed or the relation between tags except equality. In practice values correspond to numbers obtained from an incrementing counter.

As a way to better illustrate how tags work in practice consider the following example of a simple producer/consumer application.

Assume a queue of objects, instances of the class SomeObject. Several producers create objects of this type, initialise them, and push them into the queue. In another thread a consumer obtains objects from the queue and executes a series of methods after inspecting the object. As the queue is being shared, elements are being inserted by several threads and each element is processed in one of several consumer threads. Without tagging the elements, it would be difficult to follow program execution.

Using tags it is easy to follow the object as it goes from the producer to the consumer and it is likewise easy to distinguish objects with different origins. See Section 3.4 for a reference of how the shown code could use them.

```
class Consumer{
    void consume(){
        SomeObject so = queue.take();
        SomeValue sv = s.getValue();
        doSomething(sv);
        int i = doAnotherThing(sv);
        doIt();
        // (...)
    }
}
class Producer{
    void produce(){
        SomeObject so = new SomeValue();
        queue.put(so);
    }
}
class SomeObject{
    SomeValue getValue(){...}
}
```

With tags it is possible to follow a logical work unit, starting from the consumer and going to the producer, passing by the queue. A simple advantage of employing them is that, when logging X-Ray information to the standard output or to a log file, searching through the generated text for any tag value will return all methods and events associated with that tag – for example, a `grep` command using some tag value might return information representing an work unit or associated with a single response to an external client.

3.2.3 Resources

Information about the usage of physical resources (CPU, RAM, etc.) and Java Virtual Machine (JVM) resources (garbage collection, loaded classes, etc.) is is also recorded, such that it can later be correlated with information from other events.

To accomplish this, it is not be possible to reuse other existing mechanisms from X-Ray. Method entry, exit, and communication events are only triggered in well defined but often not entirely predictable times. The collected data, although extensible with probes, is limited. But what is needed for metrics to be updated at regular intervals is a constant flux of data.

The JVM statistics module is responsible for monitoring the running system and collecting several important data about its operation. Measurements are collected periodically until the module execution is halted. The measurements are not only possible but easy to do, thanks to the JMX and JRE platform beans. Those MXBeans represent information about some parts of the runtime: the garbage collector, threads' state, or the used memory [15] and can be accessed with standard, well-defined methods.

The obtained information is dispersed using the Mapped Diagnostic Context (MDC) [30, 23], a per-thread[3] key-value map, available at runtime in several code locations. To avoid polluting the keyspace, all statistic keys share a prefix; that allows a simple handling of entries (either reading or deleting) according to its type so writing a balancer or other JVM-statistics interacting entity is easy and unconstrained by hard design choices.

[3]Normally implemented using Thread-Local Storage.

3.2.4 HBase storage and retrieval

It is possible to save structured event information to a sink database, implemented as an HBase cluster – materialising information to a non-volatile context adequate to processing of large amounts of information present in distributed applications. This process takes advantage of the high scalability and throughput of HBase and can be used to feed real time systems as well as CEP systems.

This prototype hbase-appender is based on logback's DBAppender.[4] In detail, each event is inserted as a line on an HTable with its key being the event unique identifier. Each event has a type (like call logging, remote communication or performance metric) and a timestamp. Depending on the event type, relevant attributes are saved using column families (CFs) as a way to group similar ones. The time-stamp, event type, and other attributes common to all events are saved in a CF called *common*. The other column families are *call* (related to methods entries and exits), *remote* (communications made between different hosts), *jvm* (to store statistics about JVMs) and finally *prop* is used to store all other miscellaneous information. This is possible because HBase is a NoSQL database that deals well with variable data, including the addition of new columns.

The provided `xray-merger` regenerates logs saved in HBase and exports them to various formats, including graphical ones. It can also join logs originated from different machines and produces a global coherent representation, interpreting remote communication events and pseudo-nodes labelled with the socket address used for the communication and connecting them in the right place on the graphs.

3.2.5 Visualisation

The proposed X-Ray framework includes components for visualising the collected information. Such visualisation is useful in itself, as a program analysis tool, as well as a demonstration of what X-Ray enables in the context of the CoherentPaaS framework. The first and simplest way the framework can be used is as a specialised logging tool and so it is possible to export information to files, console, a remote socket address or a database.

[4] See http://logback.qos.ch/manual/appenders.html#DBAppender for more information.

The exported data can be manipulated and visualised using the tools appropriated for the chosen export medium.

For more advanced visualisations, there is the graph-appender, one of the features built upon the core X-Ray functionality. This is a driver that enables the construction of graphics representing a program run. It has a two-fold purpose: on one hand it enables the construction of structures like the CCTs described in Section 2.3.2 and other useful data, and on the other hand, it aims to present this data in a visually attractive fashion which is in itself as big or even bigger challenge than knowing what information to collect and how.

A useful feature when analysing a program is obtaining its calls' graphic representation. Comprehending how code relates to each other, where some method is called, or discovering execution patterns, are all useful when trying to understand an application.

Data is collected normally by the framework, including method entries and exits. From the saved information it is possible to produce a representation of the invocation relationships between (specific or *selected*) methods. Internally the graph is maintained in memory using *JGraphT* [3].

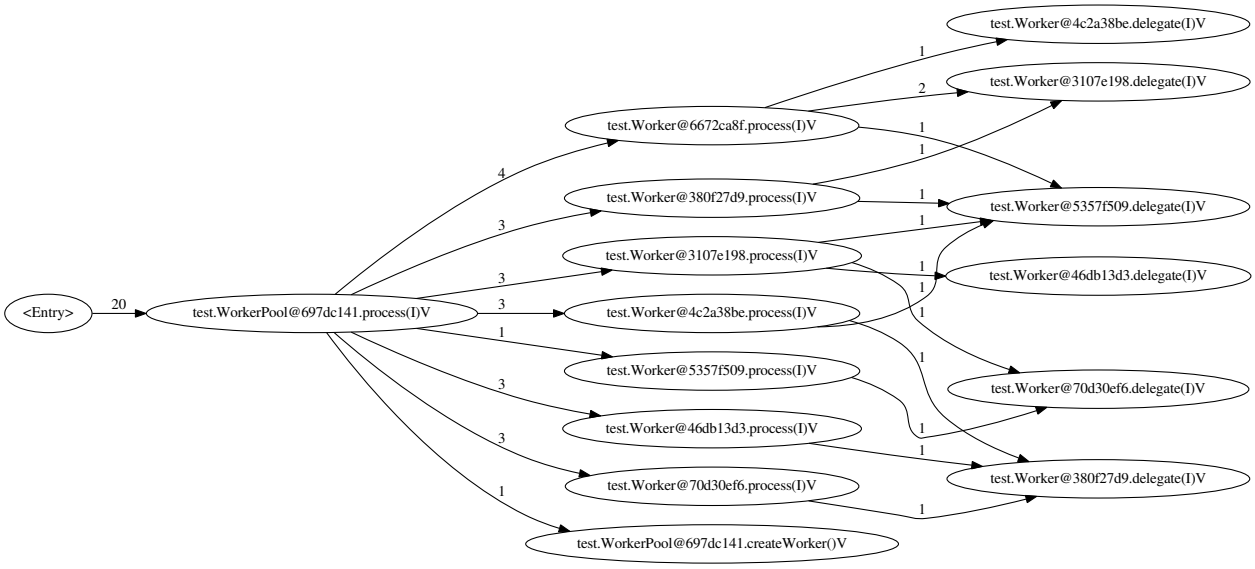


Figure 3.2: Example Graphviz's representation of the call relationships, resulting of calling `dot` on the X-Ray generated `.dot` file.

As for output formats, it is possible to export to graphviz's[5] `.dot` format to generate

[5]The project website is located at http://graphviz.org/.

afterwards a graphic with graphviz or feed it to other tools that can read that format. Another available way to see this information is with a browser through the use of an embedded server. This works with Jetty[6] + d3js[7] to obtain a browser accessible, graphical representation. Figure 3.2 show an example of a graph produced by X-Ray + graphviz and Figure 3.3 was produced by X-Ray using d3js.

In Figure 3.2, it is possible to see that the exporter generated an entry node, corresponding to the program's entry point. The graph also displays the length of each edge as a label; it corresponds to the number of times each pair *object+method* was invoked. Note that different objects from the same class are kept distinct in the picture. Figure 3.3 is a screen-shot of the web-page served with Jetty, with the same information represented using d3js. Each class+method pair uses a different colour. The width of each bar represents how many times the method was called. Finally, it should be noted that both graphics are generated and updated on-line, i.e., in real-time as the program runs.

3.3 Usage Methods

These usage methods were developed having in consideration situations where it might be interesting to instrument code, drawing experience from concrete use cases. For each of them the following factors were considered: the instrumentation cost, ease of use, number of times the solution would be used generally, its power, and expressiveness.

3.3.1 Custom Class Loader

The first alternative is to use a custom class loader. It is configured to read configuration files and react accordingly to classes to be loaded, selectively altering them or returning the original class unchanged, as appropriate. Figure 3.4 illustrates how execution is affected. For a better understanding, contrast it with the default, uninstrumented environment presented on Figure 4.1.

For those who already used this mechanism, it might seem somewhat onerous, as all in-

[6]See http://www.eclipse.org/jetty/ for more details.
[7]http://d3js.org/ has got more information about what d3js is and how it can be used.

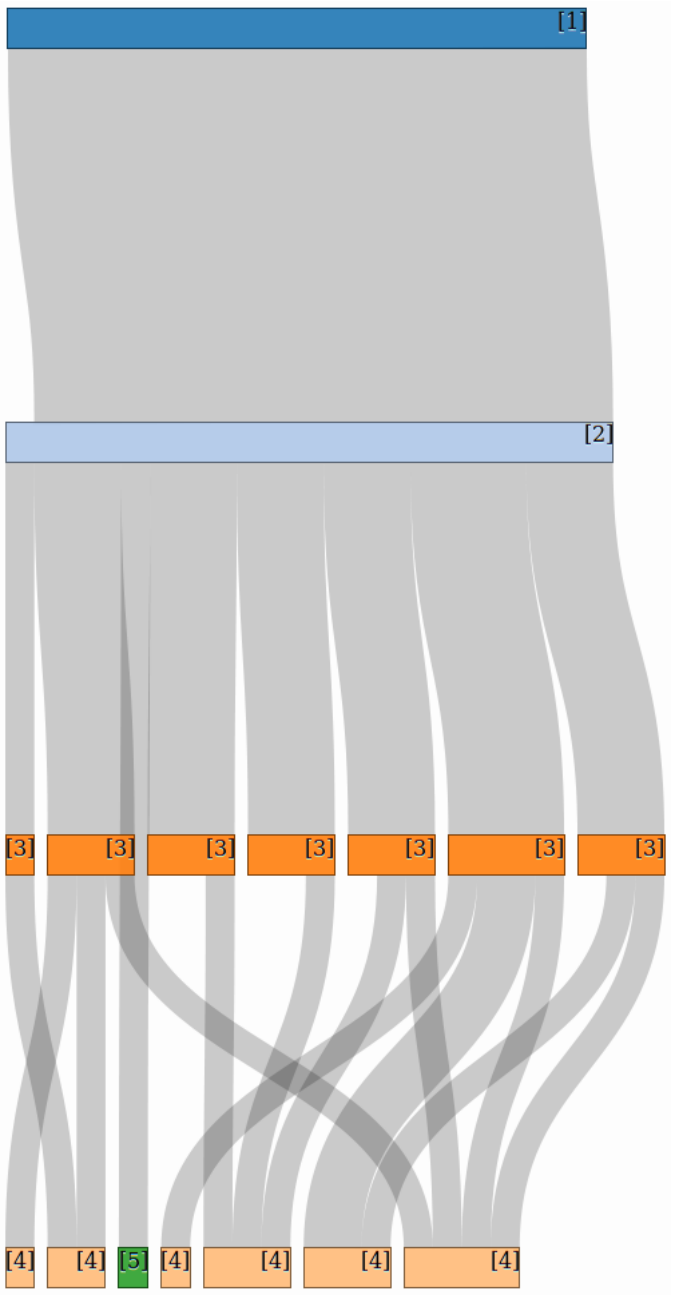


Figure 3.3: Example flowgraph representing the call relationships, produced by X-Ray and d3js: 1) *Entry*, 2) test.WorkerPool.process(I)V, 3) test.Worker.process(I)V, 4) test.Worker.delegate(I)V and 5) test.WorkerPool.createWorker()V.

teraction with the affected classes must be made through the class loader, returned `Class<?>` instances and using methods from the package `java.reflect`. This is more verbose and the compiler cannot provide much assistance (give type information, check the number of parameters, etc.). The altered methods have to be invoked differently and the types of their

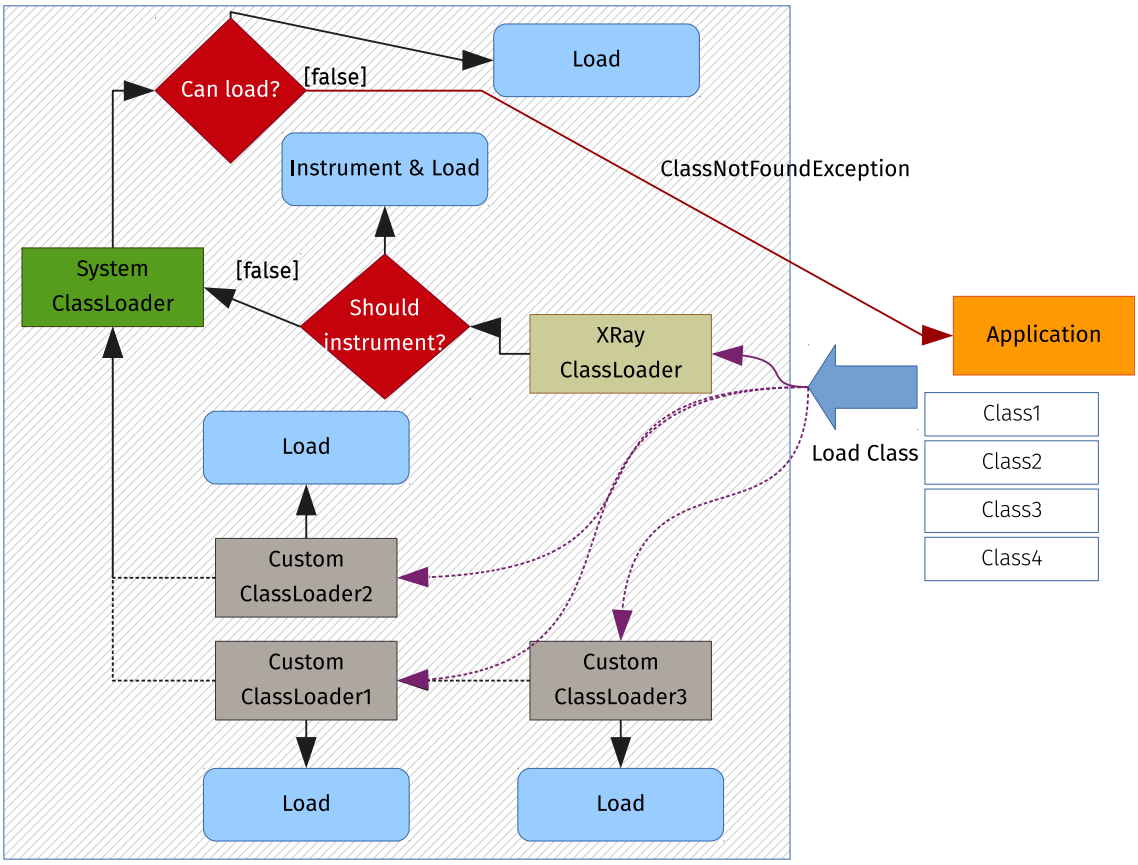


Figure 3.4: Program execution when using the X-Ray class loader.

parameters and return values also change – because we are using different class loaders, the system's and X-Ray's – a returned object sometimes cannot be casted to the normal, uninstrumented class and used normally.[8]

Yet this does not pose a problem when using X-Ray. First, the need to use different invocation strategies and reflection methods exists only where uninstrumented code interacts directly with instrumented code. But when using X-Ray the whole program is instrumented. The framework provides tools to specify or filter the classes or methods to instrument, avoiding a greater instrumentation boilerplate and the same control as if this choice was made at the source code level. Therefore a simple solution to ease the use of this modification strategy is to wrap the program to be executed in a single class and only worry about handling that class.

Because of security restrictions preventing deep and potential unsafe changes, it is not

[8]Classes defined by different class loader are considered different by the JVM, even if they have the same name or code [21, § 4.3.4], [25, § 5.3].

possible to alter methods in the `java.*` packages or native methods.

Also if X-Ray attempted to further alter the program representation, by changing multiple times the same class, the Java Runtime Environment (JRE) would give an error (a `java.lang.LinkageError`) about an attempted duplicated class definition, which is disallowed.

This makes it impossible to change instrumentation properties during the application run and seeing these changes take effect. The solution is to use the Java Agent (presented in Section 3.3.3) or modify the desired configurations and restart the program through the X-Ray class loader.

3.3.2 JAR Recompiler

Another solution is to statically modify the bytecode. Instead of modifying the program each time it is executed, it can be done just once. This is how `JarRecompiler` works: it alters all the necessary files from a JAR and saves them to a new file. This new JAR can then be normally used. Figure 3.5 details how this is done.

With this solution it is possible to develop code without the need to alter it to accommodate modified code. As the bytecode alteration is done just once, clients of the altered code do not need the `asm` library to run it. A disadvantage of this method is that it is less flexible - it is necessary to do a JAR recompilation every time a configuration is changed and one wants to see the effects of those changes. To mitigate it, a Maven plugin was developed for generating the altered JAR in the `package` phase.

It is also not possible to alter native methods. Methods in the `java.*` packages can be instrumented if the input JAR corresponds to the JRE classes, but this is not recommended.

3.3.3 Java Agent

The last option is a Java Agent [27, `java.lang.instrument` documentation], an auxiliary program distributed as a JAR and with special information in its `MANIFEST.MF` file. Depending on the support provided by the JVM, that might be different for each implementation, it can be initiated along the program by passing an argument to the command line or it can be attached to a running JVM instance after it has started. Similar to static

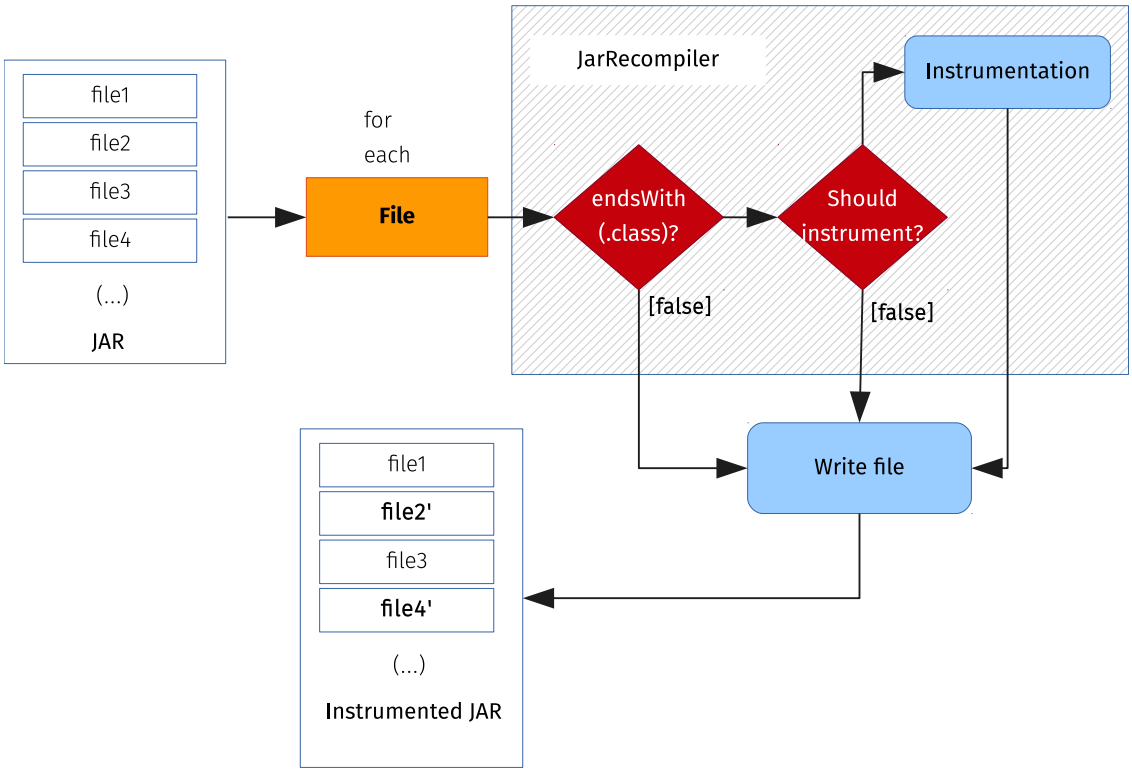


Figure 3.5: JAR instrumentation procedure.

recompilation, it is transparent for all normal code interactions and, as the class loader solution, to test some change, a simple program re-run is enough. Depending on how agents are configured, they have the ability to alter JRE classes and native methods and redefine classes already instrumented. A disadvantage of the use of agents is that not all JVMs support it. Among those who do, it does not exist a simple, universal way to do some things, specially initiating an agent after the virtual machine start-up [21, § 8.4.3.4].

Figure 3.6 shows how this method transforms the normal program execution. Compared to other runtime schemes (figure 4.1 and 3.4) this approach also funnels requests from custom class loaders.

3.4 Configuration

X-Ray can be configured in two ways: using annotations or configuration files. Both have the same expressive power, but the second approach is more flexible. If these two configurations strategies are used in parallel and conflict in some parameters, the value from

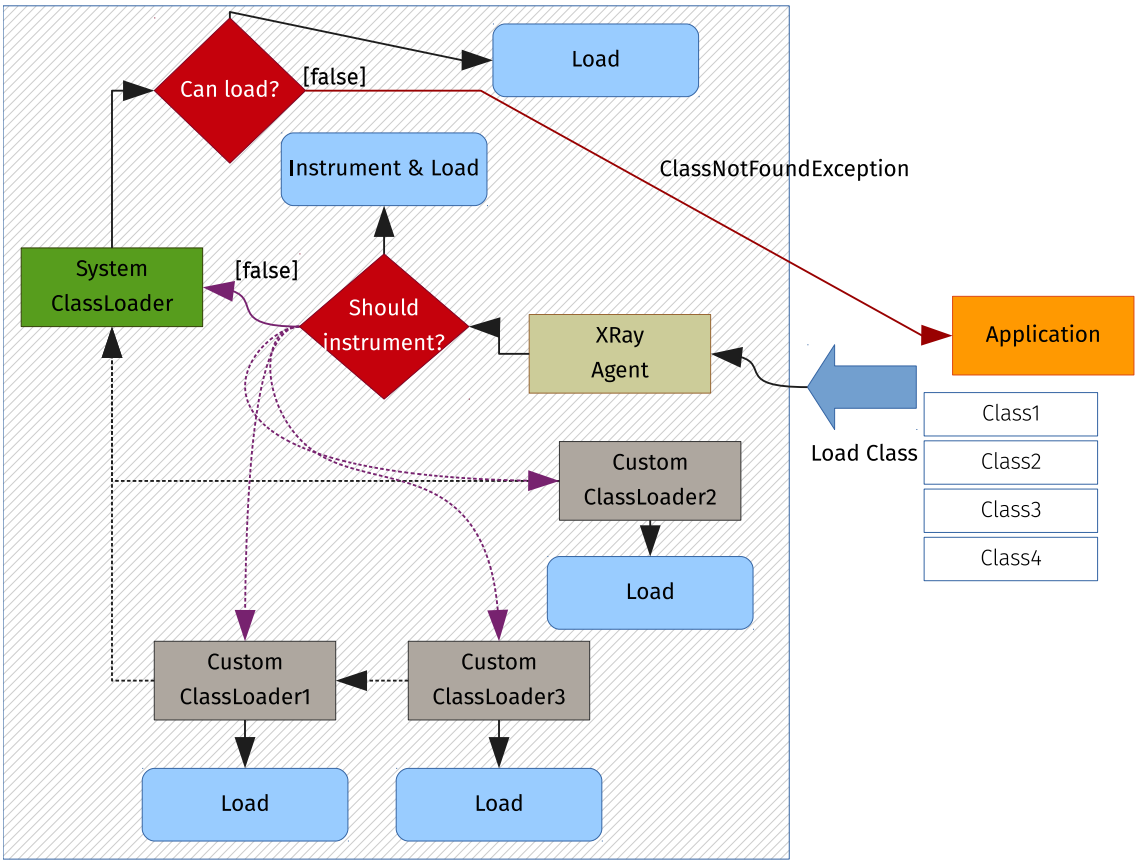


Figure 3.6: Agent instrumentation process.

the configuration file will override the annotations in the source file(s).

A typical configuration, using log levels, is as follows: uninteresting methods are not observed (are not specified to X-Ray), interesting but very common methods can have a low level (such as TRACE or DEBUG) and very important methods can have a higher level (e.g. WARN). This allows one to selectively choose what information to see and assign the adequate importance. When more details are needed, X-Ray or an exporter can have a low log level, displaying information about all the monitored methods. Otherwise, if the results have too much information or we want to see just the most important data, the global log level can be set to a higher level, displaying the methods assigned an equal or greater level than the one in use.

It is also possible to apply indications to a class or package and have all the methods contained be instrumented or apply these indications to a method and having them take effect in all the methods that override it.

But this framework is only useful in a distributed context if it can monitor or infer relations between simultaneous executions. So it is also possible to monitor executing threads and inter-method interactions, plus to gather other information about concurrency and communication events, using mechanisms such as tagging and monitoring of socket communication between different virtual machines. When and how to monitor these interactions is also configurable.

3.4.1 Annotations

The main annotation is `@Log`, applicable to methods, classes, and packages. It indicates that entries and exits of the affected methods should be logged. It possesses two optional elements: "inherited" controls inheritance and "level" specifies the messages log level. When not defined, default values are used for these elements.

```
@Log
@Log(inherited=true)
@Log(inherited=true, level=Level.WARN)
```

More annotations exist to deal with tag operations: `@SetTag`, `@CopyTag`, `@MoveTag`, `@RemoveTag` for local manipulations and `@SendTag` and `ReceiveTag` for remote manipulations involving sockets. They can only be applied to methods. All of them admit two optional elements: "target" and "time", indicating the target of the operation and when it should be executed. When values for these elements are not defined, the default value will be used and it is different according to the concrete operation.

```
@CopyTag
@RemoveTag(target=TagTarget.Thread)
@MoveTag(time=TagTime.AfterMethod)
@SetTag(target=TagTarget.Object, time=TagTime.BeforeMethod)
```

The annotations for remote operations are different because they need additional parameters that cannot be automatically inferred. The required parameters are the hostname and port, for the local and the remote sites. Because this information might be unknown at compile time or change throughout a run, a string starting with "?" indicates that the information

should be read from the field with given name. A string starting with ":", followed by a number n, signals information to be read from the n^{th} function parameter (counting from 0). An important note: some compiling options can erase metadata and optimise out other unused data that might in rare cases interfere with this process.

```
@ReceiveTag(hostname = "some.hostname", hostport = "?port",
            remotename = "another.hostname", remoteport = ":0")
void receive(int rport, InputStream is) throws IOException;

@SendTag(hostname = "another.hostname", hostport = ":0",
            remotename = "some.hostname", remoteport = "?port")
void send(int lport, OutputStream os) throws IOException;
@SendTag(hostname = "another.hostname", hostport = "8080",
            target=TagTarget.Thread, time=TagTime.AfterMethod,
            remotename = "?someHostname", remoteport = "8080")
void specialSend(OutputStream os) throws IOException;
```

Annotation use implies access to source code of the program to alter, a compile-time dependency on X-Ray and each change in the configuration requires a program recompilation to take effect. It also results in a spread of annotations to several files instead of a centralised place to read or alter everything. But this solution has certain advantages: it is simple and comes bundled with the code. Also, because it is applied directly on the entity to examine, it is not affected by refactoring, such as changing class names, method arguments, or package structure.

3.4.2 Configuration Files

The configurations presented here should be written in a specific `.properties` file for processing. This file type was selected because it is incredibly simple to visualise and to work with, by humans and computers, unlike XML. It can also be parsed by code from the JRE so no implementation needs to be bundled and distributed. Let an identifier be a string following one of these formats:

```
<packageName>.<className>.<methodDescription>
<packageName>.<className>
<packageName>.*
```

For example:

```
xray.example.SomeClass.aMetod()V
xray.example.SomeClass.aMethod(Ljava/lang/String;)[I
xray.example.SomeClass
xray.example.*
```

When the identifier appears in the configuration file, the instrumenter is instructed to alter the represented entity, as if it had an annotation in the source code.

`<identifier>.inherited` signals that the configurations for the given element should be inherited and `<identifier>.level` allows configuring the logger severity level of messages emitted by the element.

```
xray.example.SomeClass.method()V.inherited
#equals to
xray.example.SomeClass.method()V.inherited = true
#if the inherited property is not set, it is if it was written
xray.example.SomeClass.method()V.inherited = false
#this can used to explicitly limit the propagation of
#configurations inherited from an ancestor from reaching
#descendants of the designated entity

# only monitor method() from SomeClass if the defined log level
#is WARN or higer
xray.example.SomeClass.method()V.level = WARN
```

Finally, a local operation with tags is stated in the form `<methodIdentifier>.<operation>_<target>_tag_on_<time>`

```
xray.example.SomeClass.method()V.copy_object_tag_on_entry
```

```
xray.example.SomeClass.method()V.set_thread_tag_on_exit
xray.example.SomeClass.method()V.remove_object_tag_on_entry
xray.example.SomeClass.method()V.move_thread_tag_on_exit
```

and a remote operation is formulated in the form `<methodIdentifier>.on_<time>_<remoteOperation>_and_copy_tag_to_<target>` plus one or zero of the following strings, "-local_address" or "-remote_address". As when using annotations both the remote and local addresses must be defined. This is done using two keys, each with a different location key. Their values are in the form of `<hostname>|<port>`.

```
xray.example.SomeClass.method()V.on_exit_send_and_copy_tag
  _to_thread-local_address=localhost|6000
xray.example.SomeClass.method()V.on_exit_send_and_copy_tag
  _to_thread-remote_address=193.137.9.115|?port
xray.example.SomeClass.aMethod(I)V.on_entry_receive_and
  _copy_tag_to_object-local_address=?host|8080
xray.example.SomeClass.aMethod(I)V.on_entry_receive_and_copy
  _tag_to_object-remote_address=?host|:0

#the bellow statement signals that a remote operation should be
#applied to a method. If each location address is not specified,
#it will result in a runtime error
xray.example.Another Class.method()V.on_exit_send_and_copy_tag
  _to_thread
```

Unlike when annotations are used, access to source code it is not necessary. Likewise, it is not necessary to recompile the program for each change in the configurations – simply restarting the program is enough. Configurations are all grouped and separated from the code, which eases its reading or alteration and is architecturally cleaner. As for disadvantages, it is fragile in case of refactoring. The identifier is not exactly equal to the entity it represents so a search and replace may fail to modify it, and some IDEs do not normally alter resource files when refactoring.

Theoretically it is possible to update the bytecode automatically, depending on how the X-Ray software is used: the configuration file can be watched for changes and when they occur the classes bytecode can be reprocessed and swapped with the previously in use.

These two methods are available because both have advantages and disadvantages. In some occasions it is not possible to make a choice: if the source code is not available, a separate configuration file must be used. In other occasions, the choice between these strategies is not so important. When doing a static bytecode transformation of a whole program, some advantages `.properties` files have do not apply and the solutions are more similar to each other, in that particular case.

4 Implementation

4.1 Overview

When starting a program, the configuration file(s) to configure X-Ray, if any were given, are parsed and their commands are added to the framework's internal state. How these files are passed is dependent on how the framework was used. After this process the original program starts running.[1] In its course it needs to use classes or interfaces that are employed in the application but are not available, because they were not previously used and so need to be loaded. The JVM starts by locating and parsing the file with the class binary representation or bytecode and afterwards loads it and makes it available to the rest of the code, as it is responsible for coordinating the execution of Java programs.

It scans the file and for each method decides if it should be instrumented using these criteria:

1. Configuration from files.

2. Configurations acquired by reading annotations in the currently analysed class and other loaded classes.

This may require visiting the bytecode of superclasses or of the implemented interfaces if they were not already visited, as the decisions made about a class depends on decisions made about those other classes.

First of all, each class is tested against a filter to allow a quick rejection of classes to instrument, which reduces the examined code.

[1] See Section 3.3.2 for an alternative execution model.

The precise algorithm to decide if instrumentation should be made or not is defined in Section 4.4.

If any method should be altered, the new code for the method body is generated. After going through all the class code one of these situations will happen:

1. The original code of the class was altered at some point, and so this new code is returned to the JVM to be used by the program.

2. No original code was instrumented by lack of indications; if so the original code is simply returned.

Either way the program will have the class and can now resume. This process happens again each time a class is needed, until the end of the program. This approach is only possible because the binary representation of Java programs corresponds to an well-specified, platform-independent format possible to understand and programmatically manipulate.

If any method was changed for analysis, it is altered in at least two sites: its entry and exit(s). The exits can be normal – from `return` statements – or exceptional – from `throw` statements. At method entry and exit the method and class names are collected, as well as a reference to the current object and executing thread. At method entry passed parameters will be saved and at method exit, the return value is stored too.

Each time the execution flow passes through the method, indications of passage through its entry and an exit are given to X-Ray and optionally from there to other systems and all the collected information made available. Remote communication events are also listened for and reported.

Other information associated with methods, classes or packages is maintained by the framework. Part of it is only used internally at runtime and the other part is passed on and made visible. Managed data includes the inheritance information that controls how configurations are passed on to other entities and the level associated with each event. As with the traditional logging systems, the developed solution uses log levels to allow one to observe only more important information.

4.2 Understanding the JVM

Java and all other languages targeting the JVM produce `.class` files as a result of code compilation. These files contain instructions described in a machine and OS independent manner, the bytecode. They can also be compressed into a single JAR file, a ZIP file with an optional manifest file. The JAR may also be signed. Regardless of this, when a program is started, what happens is the same: a main class is specified (implicitly or explicitly), loaded, and the `main` method is invoked with the run arguments, normally given in the command line [25, § 5.2], [21, § 12.1].

Various data types used to represent different values and entities are used. These types are differentiated into two types: primitive and reference. Primitive types have explicit, well defined semantics and possible values, their own set of `bytecode` instructions, hard-coded support, and represent simple values. Reference types represent instances of classes, arrays, or interfaces. Values of those types are objects in the Object Oriented Paradigm sense and they are passed around by copying a *reference* or handle rather than by copies of values. Even at bytecode-level, almost all instructions or opcodes are typed in some way. For example there are several sum (*iadd*, *ladd*, *fadd*, *dadd*) and return instructions (as *ireturn* or *freturn*) to operate with integers, doubles, floats or reference types [25, § 6.5].

Of all types, the bulk of a program's logic is in classes and interfaces. They both define methods that reify segments of execution and classes allow the creation of various instances of that type, enabling complex programming logic. The other reference types, array types, are more data oriented and not as interesting to monitor using the chosen technique.

To call methods and create instances of the mentioned types one needs to load these types into the runtime and ready them for use. This is both true for the first loaded class at startup as well as all the later ones. This is done with a classloader. A default classloader is used to load the main class that initiates the program and if the programmer does not explicit employ another, all other types. It is called the bootstrap[2] or system classloader. User-defined classloaders can be used if one wants to have greater control over how classes are loaded and defined: it is possible to create a loader that defines types by reading bytecode

[2]It bootstraps the program, starting the VM, initialising all the required core classes and runs the appropriate main class passing its arguments.

from a datastore or a network address. These classloaders have an hierarchy that always ends in the system class loader.[3] They can delegate work to the parent classloader when they cannot or do not want to fulfil a request by themselves.

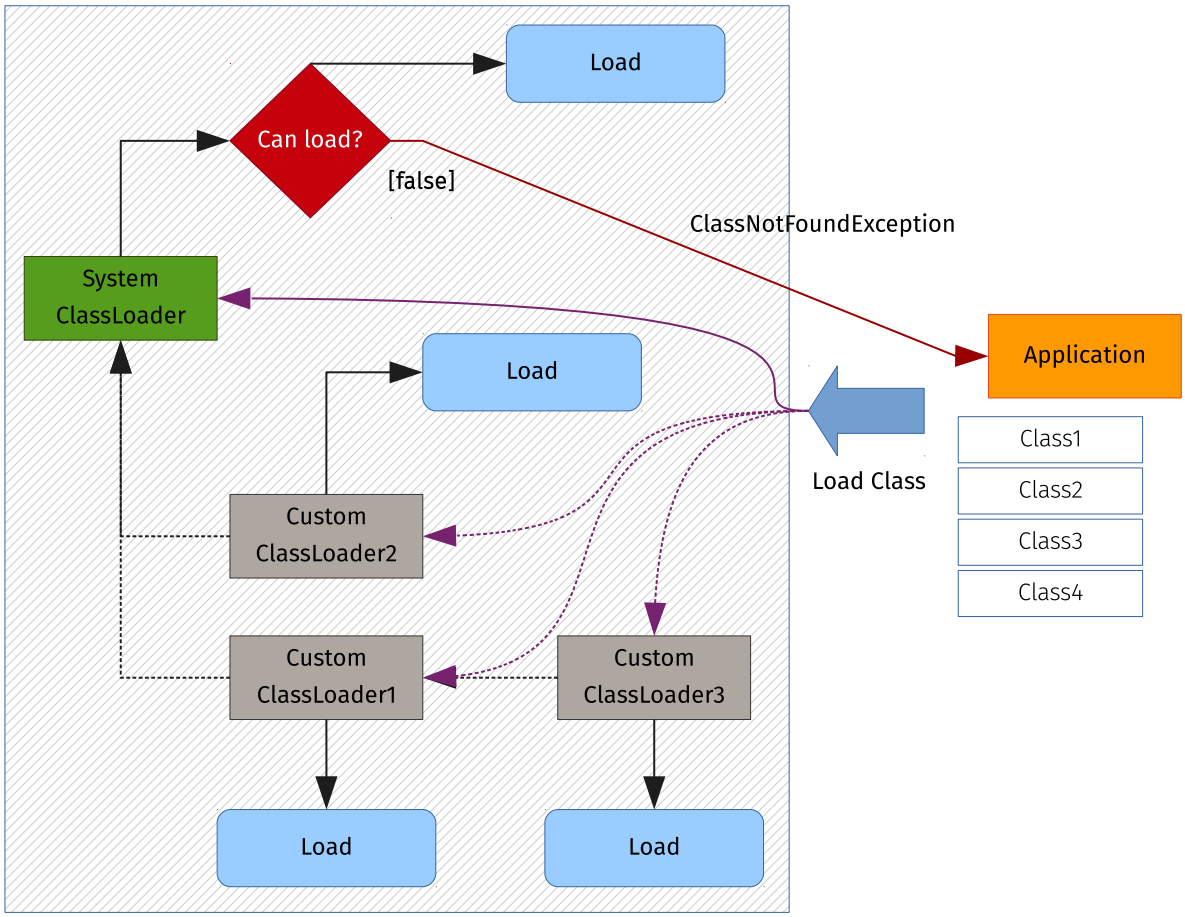


Figure 4.1: Normal JRE classloading procedure.

When the JVM needs to use a non-array type, it searches for that type in the loaded classes. If it is not there, it must be loaded by a class loader. Loading is the process of getting a binary representation (bytecode) of a type, normally by copying it from a file. Figure 4.1 shows the process of loading classes and interfaces (both the program's and others) normally happens. The purple lines represent the requests dispatch and the solid line represents the default dispatch. When not using a custom classloader, the system's will be used. Black dotted lines represent the classloader hierarchy through which request can be deferred. If the request arrives at the system classloader and it cannot be satisfied a error or exception

[3]Similar to how the class hierarchy ends in the Object class.

is thrown, normally a `ClassNotFoundException`.

After loading, the class must be linked or readied to be executed when requested. Linking implies verifying, preparing and (optionally) resolving phases [25, § 5]. First the code is verified to make sure it is valid – all opcodes used exist and are correctly used, `jumps` point to locations that make sense, there are no invalid method signatures or type operations.

A reference type is prepared by creating and setting `static` fields to their default values and initialising other structures used internally to represent the type in the JVM. Resolving symbolic references such as method calls or other classes' use is the final step, but there is some amount of flexibility about when this can happen. As long an error is thrown when a referenced class or interface cannot be processed, a certain VM can choose whether to resolve all references immediately (and their references), resolve a reference only when used, or something in between.

Finally, the linked class or interface is initiated by having its static fields initialisers run and other static initialisers (static blocks).[4] Superclasses are first initialised if they were not already. Implemented interfaces or superinterfaces are not pre-initialised. After this operation, static methods and fields can be used.

Instantiation of new objects of a given class is performed when a class constructor is invoked.[5] Memory is allocated for the new object and the constructor method in evaluated, not before running a constructor from each superclasse in descending order – this can be called constructor chaining [16].

From there on, the executed binary code is relatively similar to the original source code, with instructions corresponding to field reads and writes, method invocations, object and array creation, sums, `if`s, `switch`es and `try-catch` blocks, etc. and where applicable these instructions have specific types [25, § 3,§ 6.5].[6]

All these and other aspects related with Java program execution and class handling conducted on runtime are explained clearly and in greater detail in [25, § 5] and [21, § 12], including edges cases not considered here. [25, § 6.5 and § 6 in general] cover the existing opcodes.

[4] Internally a special static method `<clinit>` is invoked. All initialisers are merged to form this method accessible only by the JVM.
[5] It also happens implicitly in some special circumstances, for example, when concatenating strings.
[6] E.g. an *ireturn* allows one to see that an integer will be returned.

4.3 Technologies used

The framework being described here is implemented using some technologies as foundation. One is `asm` [11], a *bytecode*-manipulation library. It is small, fast, and powerful and so is used in projects such as AspectJ, Scala, Clojure, and Groovy. It provides two interaction interfaces, the *Core API*, focused on simplicity and speed, and the alternative *Tree API*, easier to use but less efficient. The *Core API* is used in X-Ray because performance is an important consideration when aiming to support runtime *bytecode* transformations. It is modelled after the *Visitor* design pattern, or to be more precise the *Hierarchical Visitor* pattern [17]. Other solutions such as javassist, BCEL or cglib do not offer the best performance or have not been updated for some time.

The other main component of the software is related to the logging of collected information. Logging in X-Ray is done with slf4j, a logging facade for Java [29], and `logback`, a logging framework implementing the slf4j API [28]. slf4j acts as a facade through which all logging actions are made, removing a direct dependency on a specific framework: only the JAR with the API must be distributed. If no binding is found, logging does not occur; if a binding is detected at runtime, it is used. Thus, bindings can be freely replaced between program executions. This isolates many implementation details, increases modularisation and decoupling.

`Logback` enables processing various actions for the same logging message and is easy to extend. Those actions are executed by entities called appenders, programmable in Java, and configurable with a simple XML file. The framework's core only depends on slf4j; some X-Ray extensions use directly `logback` because of its advanced features and ease of configuration.

4.4 Selection of instrumentation targets

On this section the answer to two essential questions will be presented. They are: "How it's decided if a method should be instrumented?" and "How are the various configuration options applied, specially when they conflict with each other?"

X-Ray operates every time it is called to resolve a class, meaning, to return the bytecode

associated with a class. Depending on how it is invoked, this could happen dynamically or statically. Regardless, the decision whether to instrument or not is made for each method, on a case-by-case basis. The available configurations can be made in a larger scope if wanted though, such as on a class or package.

Let m be a method from class C. If an indication to instrument exists in:

- the analysed method m;
- m's enclosing class C;
- the package where C belongs;
- the original method declaration or a super implementation of m, if that indication is inheritable;
- a supertype that C extends or implements, if that type's configurations are to be inherited;
- a package that contains a supertype C extends or implements, but only if that indication is inherited

then the method m will be altered by X-Ray. Otherwise, the original method code is returned and the rest of the class is visited. The given conditions are tested by the order they were presented. The following code listing presents this information in the form of pseudo-code.

```
def instrument?(m):
    if method_instrument_list.contains?(m):
        return true
    else if class_instrument_list.contains?(m.class):
        return true
    else if package_instrument_list.contains?(m.class.package):
        return true
    end
    for met in hierarchical_method_chain(m):
        if method_instrument_list.contains?(met)
```

```
      ∧ is_inheritable?(met):
         return true
      end
   end
   for cla in hierarchical_class_chain(m.class):
      if class_instrument_list.contains?(cla)
      ∧ is_inheritable?(cla):
         return true
      end
   end
   for p in hierarchical_package_chain(m.class):
      if package_instrument_list.contains?(p)
      ∧ is_inheritable?(p):
         return true
      end
   end
   return false
end
```

The instrument lists are built when processing configurations and contain the classes, methods or packages to instrument. The hierarchical chains relate included elements with each other. The hierarchical method chain of m contain the methods from the supertypes of C that contain a method with the same name and signature as m.

In a hierarchical class chain, every supertype of C is included and all its supertypes are included recursively, up to the class hierarchy root: `java.lang.Object`. Types closer to C appear before more distant ones in the hierarchy chain. For example, in the class chain of `java.lang.InterruptedException`, `java.lang.Exception` will appear before `java.lang.Object`.

Finally, hierarchical package chains are defined for the class C as all the packages that contain any supertype of C. The class `java.lang.String` extends `Object`, implements two interfaces from `java.lang` and also the `java.io.Serializabe` interface. Its hierarchical

package chain contain the packages `java.lang` and `java.io`.

The rules used to determine what configurations should take effect are also identical to the ones presented to decide what to instrument. Similarly, the same places are searched for the configurations in the same order and if configurations are found they are followed. Otherwise, the default values are used.

Package settings are applied to all the classes inside that package. Class configurations are applied to all the class methods, declared or provided by supertypes. And each method can be individually configured.

Those configurations can optionally be inherited. If the package-level configurations are inheritable, all the classes that extend or implement classes defined in the original package will use the configurations defined, unless overridden.

Similarly, if a type-level configuration is inherited, all subtypes are affected by that configuration. In the case of configurations at the method level applied to m, for any type extending/implementing the enclosing type, the corresponding method m' will use those configurations, whether m was overridden (or implemented if abstract) or the original implementation used.

A difference the configuration algorithm has relative to the instrumentation decision one, is that instead of having instrumentation lists it uses <attribute>_maps (with an attribute being something like the log level) and for each element the value of that attributed is stored.

When a settings conflict arises the priority is given to the more specific indications, followed by the closest ones. For example if the class C of method m should log events with the TRACE level and exists an original implementation of m with log level of DEBUG, m will have the DEBUG level.

If C is the class of method m and C extends C' and m has not configuration whatsoever, logging properties will be inherited from C, unless all of these conditions are met:

1. C' has configurations defined

2. C' configurations are inheritable

3. C has no configurations defined

4.5 Modifications to targets

If the conditions analysed in Section 4.4 indicate that any method should be altered, it is necessary to take the class binary representation and alter the method(s) bytecode. But by changing the original bytecode it might be then necessary to update other internal class structures, such as the constant pool, responsible for holding string and numerical constants as well as class, method and field names [25, § 4.1, § 4.4]. Something as simple as referring to a new class requires this change. `asm` does this work automatically.

X-Ray also adds fields or other information needed for the framework's operation. A produced change is the addition of a reference to a slf4j logger object as a new static field. Tags also require the creation of a new field. These constructed fields are named in an unusual way to avoid colliding with existing code (their names start with the "$_xray_" prefix) and with a special marker to indicate they were not originally present but were generated (their `ACC_SYNTHETIC` flag is set). This might be useful to other class manipulation or reading tools to warn them it might not be necessary to process these constructs or to enable the use of all tools simultaneously.

If a method is to be analysed, it is changed at least in its entry and exit(s) as already mentioned. The entry point is easy to alter by inserting the relevant code before any of the original bytecode instructions. Exit points occur at `return` and `throw` statements. A `throw` statement results in an *athrow* instruction in the compiled code. On the other hand, a `return` statement results in one instruction of the *return* opcode family (*ireturn*, *lreturn*, *freturn*, *dreturn*, *areturn* and *return*), depending on the type of the return value. So all exit points are identifiable and it is trivial to add the necessary instrumentation code.

Before copying the original instruction to the new class representation, the needed logging instructions are inserted. These instructions capture all the relevant execution information and pass it through calls to methods on core X-Ray classes, needed at runtime. Finally the produced bytecode is returned. These X-Ray methods are responsible for producing logging events following a certain structure, sending them to the defined outputs and invoking any user-defined probes.

4.5.1 Eager loading and callbacks

When loading a class, its binary representation is first retrieved. Then referenced methods or classes are resolved in the linking phase. Although the JVM can resolve most references at any arbitrary time, classes in a superior hierarchical level must be resolved before the preparation phase, if they have not been resolved yet. This is important because when deciding what modifications to perform on a class, super classes and interfaces give indications to subtypes. These can happen both in configuration files – easy to solve as the file entries are read all at once – and in their binary code, using annotations.

So one might think that X-Ray can work by simply intercepting class definition requests when they are made: superclasses are visited before ending the visit of the current class and annotations could be read in time to be applied to subclasses. However, super types are only loaded between linking and initialisation, but it is in the loading phase that the binary representation of methods is defined and it cannot be altered in the next phases. Furthermore, interfaces (both implemented and superinterfaces) are not necessarily initialised before initialising a type. So it is necessary to force the loading of supertypes before ending the loading phase. This is accomplished by requiring each X-Ray instrumentation process to pass a callback to be invoked when reading the bytecode of a type T and encountering unloaded super types. Those types will be loaded using the callback and theirs supertypes too, recursively. After that we know whether T should be instrumented or not and we can return the original bytecode or modify and return it, as appropriate. A list of visited types and their hierarchical relations is maintained to avoid repeated visits.

4.5.2 Event Information

Each event has an unique identifier associated. The identifier is composed of a VMID (Virtual Machine Identifier) and a sequence number. The VMID is an unique identifier for each JVM, based on some of its unique properties and it is valid as long as its IP address remains unique and constant(cfg. [27, `java.rmi.dgc.VMID` documentation][27, `java.rmi.server.UID` documentation]). The sequence number is a local identifier that is incremented once after each logged event. Each event has a type (like call logging, remote

communication or performance metrics) and a timestamp.

At method entry each parameter value will be saved by X-Ray. For objects, a reference is saved and for primitive types, boxing of the original value is performed. At method exit, before the terminating statements, the return value or exception is also saved, as well as a flag indicating whether the method returned normally or not. Stored data includes also the name and reference of the current object (Java's `this`), the method name, signature and the running thread.

For the most part they are passed using slf4j parametrised messages,[7] if not directly obtainable from logback. Other information is copied using MDC[30, 23].

[7] See http://www.slf4j.org/faq.html#logging_performance for more details.

5 Evaluation

X-Ray is evaluated qualitatively, by being applied to Apache Derby.[1] Its performance is also evaluated quantitatively by observing the impact of instrumentation and of different kind of exporters while running a standard performance benchmark.

5.1 Qualitative analysis

Starting with a simple example to see if more complex case studies could succeed, a simple client-server configuration was picked. The client connects to the Derby database and executes a query that runs in that database. After the computation produces a result, it is returned to the client, which prints the results and exits. `SELECT t1.a,t3.e FROM T1 JOIN T3 ON t1.b=t3.b WHERE t1.c > (SELECT avg(t2.c) FROM T2)` was the executed query.

Figures 5.1 and 5.2 were obtained by saving the logs produced by client and database server to HBase, using hbase-appender, and by invoking xray-merger to read that information from HBase, reconstruct it, and add relevant remote communication event nodes between the datastore and the client. The resulting flowgraph is shown in Figure 5.1, as rendered by d3.js, and in Figure 5.2, by Graphviz.

The second experiment was to instrument and follow a request made by a client to a cluster of two Derby databases, that communicate with each other. That is a true distributed query. From this experiment with distributed databases, it was possible to obtain Figure 5.3, using HBase plus xray-merger.

For X-Ray to successfully perform its job and produce the graphics presented, it must work in the context of a distributed query, capture the communication between database

[1] See https://db.apache.org/derby/index.html for more details.

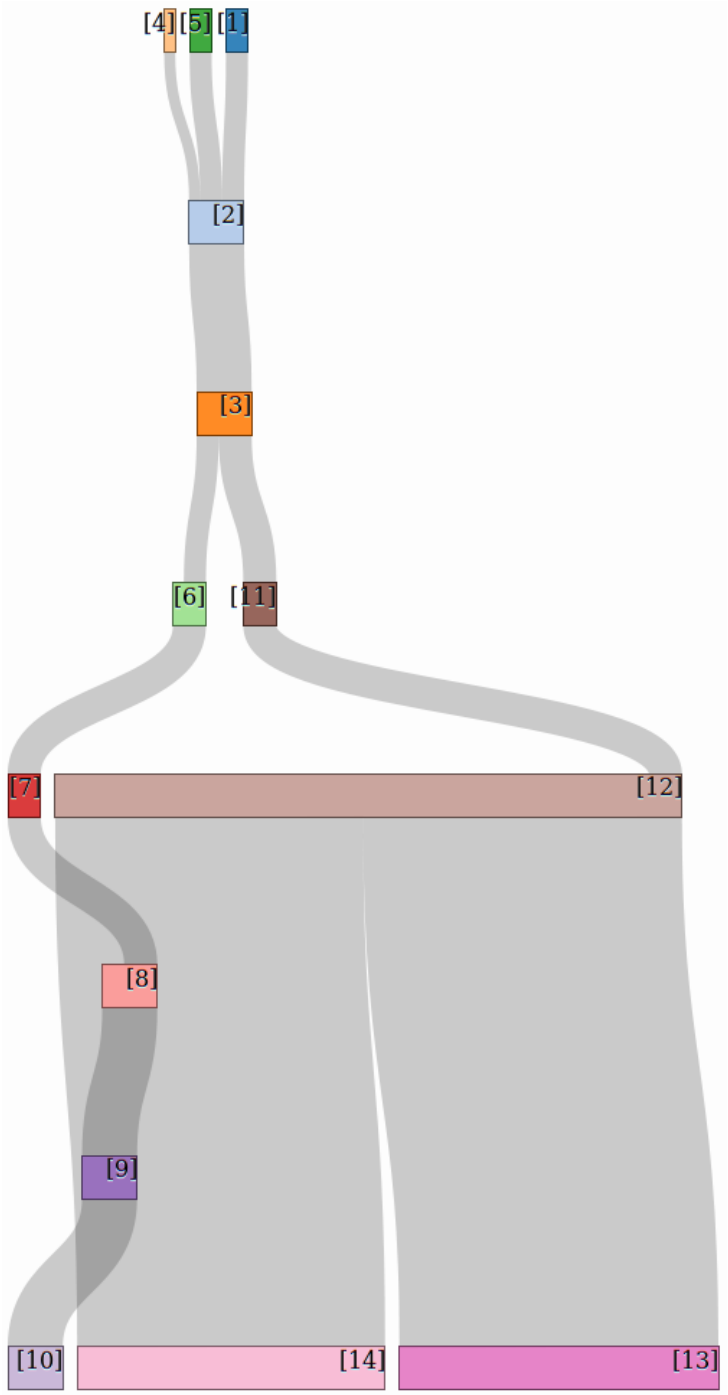


Figure 5.1: Flowgraph produced when sending a query to Derby. Each node represents an object. When more than one object of the same class is used an identifier is used to distinguish them.
1) NetConnection, 2) localhost:60059-localhost:1527, 3) DRDAConnThread, 4) ClientStatement, 5) NetResultSet, 6) OnceResultSet, 7) ProjectRestrictResultSet@1, 8) ScalarAggregateResultSet, 9) ProjectRestrictResultSet@2, 10) BulkTableScanResultSet@1, 11) ProjectRestrictResultSet@3, 12) HashJoinResultSet, 13) BulkTableScanResultSet@2, 14) HashScanResultSet

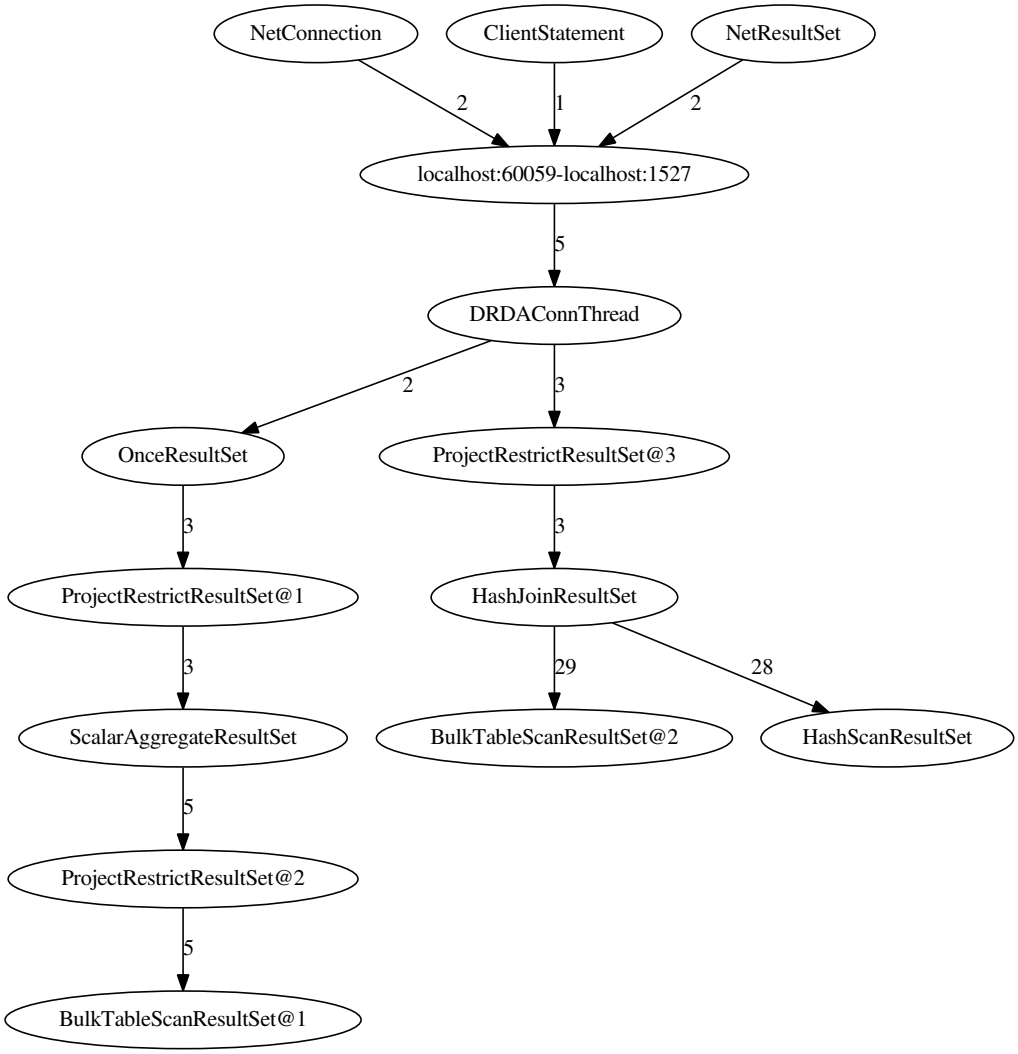


Figure 5.2: Graphviz's representation of the call relationships when sending a query to Derby.

nodes and/or client, track the request in the middle of unrelated computations, etc. This requires minor changes to Derby's source code, to expose the communication ports to X-Ray.

Also of note is that although the source code was modified, the instrumentation configurations did not take advantage of that access but were made using `.properties` files.

In fact, having only that problem when using X-Ray in such mature codebase as the Derby project has, with about 762 KSLOC (thousand source lines of code)[2] and a complex execution model, making use of custom classloaders and generating classes on the fly for each SQL Statement required the use of the X-Ray instrumentation Agent.

[2]According to https://www.openhub.net/p/derby/analyses/latest/languages_summary, accessed on October 30 2014. Although a rough estimate to evaluate a project, it reflects its size and complexity.

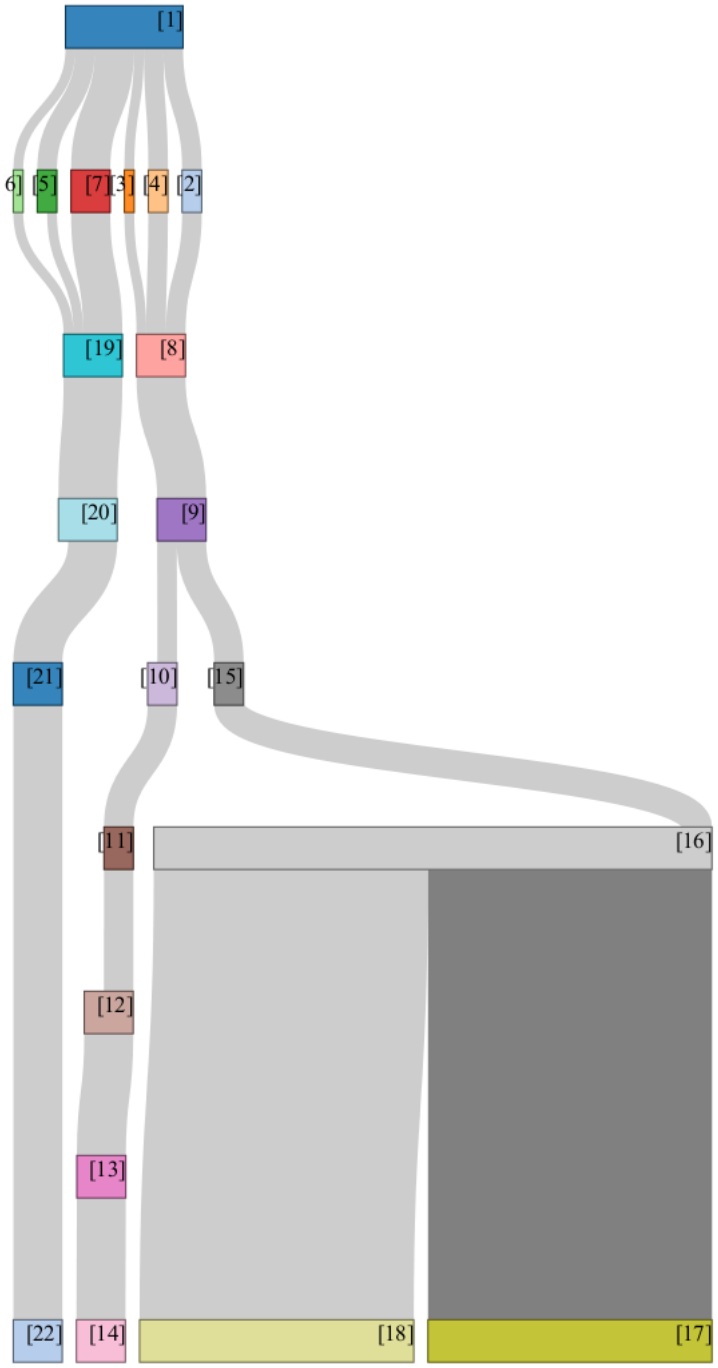


Figure 5.3: Flowgraph produced when sending a query to a Derby cluster composed by two databases. Each node represents an object. When more than one object of the same class is used an identifier is used to distinguish them.

1) MultiDatabaseQuery, 2) NetConnection@1, 3) ClientStatement@1, 4) NetResultSet@1, 5) NetConnection@2, 6) ClientStatement@2, 7) NetResultSet@2, 8) localhost:47903-localhost:1527, 9) DRDAConnThread@1, 10) OnceResultSet, 11) ProjectRestrictResultSet@1, 12) ScalarAggregateResultSet, 13) ProjectRestrictResultSet@2, 14) BulkTableScanResultSet@1, 15) ProjectRestrictResultSet@3, 16) HashJoinResultSet, 17) BulkTableScanResultSet@2, 18) HashScanResultSet, 19) localhost:47905-localhost:1527, 20) DRDAConnThread@2, 21) IndexRowToBaseRowResultSet, 22) MultiProbeTableScanResultSet

5.2 Measurements

The performance impact of X-Ray instrumentation and probes was measured by starting a Derby database and applying it to the TPC-C transaction processing benchmark.[3] The goal is to obtain significant statistics about the state of the database over the course of the benchmark, to see what was the overhead of using X-Ray and if it was even possible to instrument such a large codebase developed by a third-party and that potentially makes use of features that conflict with the framework.

The benchmark details are in Table 5.1 and the results can be seen in Table 5.2.

DB:	TPC-C 1WH, 200MB
WL:	1 client, no think time

Table 5.1: Benchmark details for TPC-C on Derby

The database where the TPC-C benchmark was run is a single warehouse with approximately 200MB of data and for work load, 1 (one) client making requests without delay between them (i.e. with no think time).

Baseline	29.0ms	923tpmC
Instrumented	29.5ms	909tpmC
Logging to File	43.6ms 1518144KB	621tpmC 6946245 lines
Logging to HBase	115.8ms 22844152KB	226 tpmC 3278000 lines

Table 5.2: Execution times of running TPC-C on Derby.

The columns in Table 5.1 explain how TPC-C ran and the obtained results, with times in milliseconds (ms) and transactions in transactions per minute (tpmC). The different experiments runs were:

Baseline No instrumentation was used.

Instrumented Run with the instrumentation turned on, but not using any appender.

Logging to File Using TPC-C with instrumentation plus enabling logs to stdout and redirecting them to a file.

[3] See the homepage, http://www.tpc.org/tpcc/default.asp, for more details.

Logging to HBase Having the HBase-appender save logging information to HBase. In the HBase-appender auto-flush was turned off.

For *Logging to File* and *Logging to HBase* tests, it is also shown the size occupied in disk at the end of the benchmark and the number of lines written. In the *Logging to File* test, the number of lines and size refer to the written file, and in the case of the *Logging to HBase* run, size is the size of the data written to HBase during the benchmark, and number of lines is the number of written entries on its tables.

It is possible to see that when using instrumentation (*Instrumented*) the benchmark ran at $98,3\%$ of the original speed; that is only a $1,7\%$ slowdown.

6 Conclusions

X-Ray's goal is to allow CoherentPaaS to support debugging, monitoring, and optimisation of its platform, composed of several databases, highly heterogeneous, distributed and accessible through a common query engine. To accomplish that, X-Ray uses a generic approach, working as a framework accepting configurations and altering bytecode to monitor events, tracking resource usage, maintaining context information and supporting the insertion and execution of arbitrary custom code. Besides that, mechanisms were added to save information, gather it in a global state, process it in real-time, and visually represent it.

During the qualitative analysis made when applying X-Ray to Apache Derby, its applicability and usefulness when instrumenting and analysing a large codebase was demonstrated. Note that Derby is particularly demanding, as it makes use of dynamic code generation, which was dealt with by using the JVM agent. On the other hand, minor changes to the source code were still required, to make communication ports available to instrumentation code. As for the quantitative measurements made, they confirmed the notion that the proposed solution is lightweight and its usage does not impose an expensive overhead.

The most important aspect shown by the experiments is that the framework can be used to fulfil the requirements of the CoherentPaaS's X-Ray monitoring module. One experiment focused on two datastores communicating with each other (using sockets), executing a distributed query and returning a result to the client. The experiment that tested the X-Ray performance used the TPC-C industry standard benchmark.

Finally, if on one hand, the developed work satisfies CoherentPaaS's requirements, on the other hand, the produced tool has a sufficiently general conceptional model and avoids making too many assumptions that tightly tie it to the platform. This enables it to be useful and have a purpose outside CoherentPaaS.

6.1 Future Work

The proposed framework can still be extended and improved in several ways. First, to solve the problems found in the configuration of socket communications, in Chapter 5, more ways to support reading values can be implemented, namely, the ability to invoke methods or accept `SocketAddress` objects directly to obtain those addresses.

The instrumentation of native methods by the JVM Agent is a goal needed to satisfy other less frequent use cases and so is testing the effects of applying instrumentation several times in the same run. This last feature of using an agent poses several challenges in two places: the internal X-Ray configurations and the JVM and program state. When a user changes some instrumentation options online the saved configurations must be invalidated or more smartly, be selectively changed having in mind what can or cannot be modified. On the JVM part, some changes to the defined classes can't be made and both it and the running program have to accommodate for the produced changes, resulting in a potentially inconsistent runtime state.

Further work is also needed to deal with legacy systems not implemented in Java where the X-Ray system can't be used. That is the bridge or converter mentioned in Section 1.1. It should be developed, preferably in the C language to allow an easy interoperability with C, C++ and if needed with other languages via bindings.

Another possible feature is to allow following inter-thread communications that use mechanisms such as message passing. Tags can already be used to this end, but a default provided instrument would be interesting, perhaps implemented using tags or complementing them.

The usefulness of X-Ray can be improved by further developing the graphical representation: adding more export formats as well as improving the existing ones by incorporating more of the collected information on them, e.g. execution times. Collecting and using more information is an useful thing, so a important future task is to associate the collected JVM statistics (cf. Section 3.2.3) with each execution instant and each method invocation. If more information is included by default and presented to the user in a visually clear and attractive fashion, their comprehension about the analysed program can only increase.

Bibliography

[1] The aspectj project. http://www.eclipse.org/aspectj/.

[2] Factsheet CoherentPaaS. https://ec.europa.eu/digital-agenda/sites/digital-agenda/files/130816%20Factsheet%20COHERENTPAAS.pdf.

[3] Jgrapht: a free java graph-theory library. http://jgrapht.org/.

[4] Pin - a dynamic binary instrumentation tool. http://software.intel.com/en-us/articles/pintool.

[5] Project lombok. http://projectlombok.org/.

[6] Systemtap. https://sourceware.org/systemtap/.

[7] Using Java EE Annotations and Dependency Injection. http://docs.oracle.com/cd/E11035_01/wls100/programming/annotate_dependency.html.

[8] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, May 1997.

[9] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[10] A. R. Bernat and B. P. Miller. Incremental call-path profiling: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(11):1533–1547, Aug. 2007.

[11] E. Bruneton, R. Lenglet, and T. Coupay. Asm: a code manipulation tool to implement adaptable systems. http://asm.ow2.org/current/asm-eng.pdf.

[12] P. Caserta and O. Zendra. Jbinstrace: A tracer of java and jre classes at basic-block granularity by dynamically instrumenting bytecode. *Sci. Comput. Program.*, 79:116–125, Jan. 2014.

[13] K. M. Chandy. Event-driven applications: Costs, benefits and design approaches.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[15] O. Corporation. Monitoring and Management of the Java Virtual Machine – Overview of the JMX Technology (The Java ™Tutorials). http://docs.oracle.com/javase/tutorial/jmx/overview/javavm.html.

[16] O. Corporation. Using the Keyword super – Interfaces and Inheritance (The Java ™Tutorials). http://docs.oracle.com/javase/tutorial/java/IandI/super.html.

[17] R. A. DiFalco. Hierarchical visitor pattern. Wiki Wiki Web http://c2.com/cgi/wiki?HierarchicalVisitorPattern, 2011.

[18] W. Drewry and T. Ormandy. Flayer: Exposing application internals. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, pages 1:1–1:9, Berkeley, CA, USA, 2007. USENIX Association.

[19] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan. Event-based systems: Opportunities and challenges at exascale. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM.

[20] D. M. Eyers, L. Vargas, J. Singh, K. Moody, and J. Bacon. Relational database support for event-based middleware functionality. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 160–171, New York, NY, USA, 2010. ACM.

[21] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[22] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '00, pages 267–, Washington, DC, USA, 2000. IEEE Computer Society.

[23] C. Gülcü, S. Pennec, and C. Harris. The logback manual, Chapter 8: Mapped Diagnostic Contexts. http://logback.qos.ch/manual/mdc.html.

[24] A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, New York, NY, USA, 2009. ACM.

[25] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[26] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[27] Oracle Corporation. *Java Platform, Standard Edition API Specification*, 8th edition.

[28] QOS.ch. Logback Project. http://logback.qos.ch.

[29] QOS.ch. Simple Logging Facade for Java (SLF4J). http://www.slf4j.org.

[30] QOS.ch. Slf4j user manual. http://www.slf4j.org/manual.html#mdc.

[31] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia. Precise, scalable, and online request tracing for multitier services of black boxes. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1159–1167, 2012.

[32] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 11–20, New York, NY, USA, 2011. ACM.

[33] M. Schoeberl. A java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, Jan. 2008.

[34] M. J. Serrano. Trace construction using enhanced performance monitoring. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 34:1–34:10, New York, NY, USA, 2013. ACM.

[35] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. ACM.

[36] S. White, A. Alves, and D. Rorke. Weblogic event server: A lightweight, modular application server for event processing. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 193–200, New York, NY, USA, 2008. ACM.

[37] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not.*, 41(6):263–271, June 2006.