X-Ray: Monitoring and Analysis of Distributed Database Queries

Pedro Guimarães and José Pereira(✉)

INESC TEC and U. Minho, Guimaraes, Portugal
pg22834@alunos.uminho.pt, jop@di.uminho.pt

Abstract. The integration of multiple database technologies, including both SQL and NoSQL, allows using the best tool for each aspect of a complex problem and is increasingly sought in practice. Unfortunately, this makes it difficult for database developers and administrators to obtain a clear view of the resulting composite data processing paths, as they combine operations chosen by different query optimisers, implemented by different software packages, and partitioned across distributed systems. This work addresses this challenge with the X-Ray framework, 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 to visualise data processing paths as required for optimising integrated database systems. This proposal is demonstrated with a distributed query over a federation of Apache Derby database servers and its performance evaluated with the standard TPC-C benchmark workload.

Keywords: Distributed databases · Monitoring · Java instrumentation

1 Introduction

The performance of data management systems depends on how operations are mapped to different hardware and software components. This mapping is driven by the developer, by query compilation and optimisation in the system itself, and finally by database administrators. Obtaining the best performance thus depends on monitoring and analysing such mapping. Relational database management systems have traditionally included tools to expose the execution plan for a query, identifying what implementation is used for each abstract relational operation, in what order, and including a detailed accounting of I/O operations, memory pages, and CPU time used. As an example, in PostgreSQL this is provided by `EXPLAIN ANALYZE` [27] and presented graphically with pgAdmin3 [14].

Recently, driven by novel applications, there has been a growing trend towards using different data management techniques and tools for different purposes, instead of always resorting to relational database management systems [26]. For instance, the CoherentPaaS platform-as-a-service integrates various SQL and NoSQL data stores in a common framework [3,20]. Moreover, the large scale

© IFIP International Federation for Information Processing 2015
A. Bessani and S. Bouchenak (Eds.): DAIS 2015, LNCS 9038, pp. 80–93, 2015.
DOI: 10.1007/978-3-319-19129-4_7

of current applications means using distributed data stores that scale out with data size and traffic, such as HBase, trading off in the process query processing capability and transactional ACID guaranties.

This poses several challenges to monitoring and analysis. First, some of the data stores now commonly used have only minimal support for data collection on individual operations, providing only aggregate resource measurements. In fact, the additional application code needed for integration and to overcome the limitations of NoSQL data stores is likely to have no monitoring capabilities at all. Second, even when monitoring tools are available for the required data stores, they provide partial views that cannot easily be reconciled and integrated into a coherent observation, namely, by tracking its relation to a common user request. Finally, when multiple instances of a specific data store are used for scale out, such as in replication and *sharding* configurations, distributed monitoring information has to be collected and organised according to its role in a global operation, for instance, to reason about load balancing and parallelism.

This work addresses these challenges with X-Ray, a framework for monitoring and analysis of distributed and heterogeneous data processing systems. First, it provides a way to add monitoring code to applications and data stores running in the Java platform. By using bytecode instrumentation, this does not rely on the availability of the source code and can be applied conditionally to avoid overhead in production systems. Second, it provides mechanisms for tracking the interaction of multiple threads, on synchronisation primitives, and of distributed processes communicating with sockets. Finally, it provides a tool to reconcile monitoring data from multiple software components in a distributed system taking into consideration their relation to actual user requests, thus providing a cross-cutting unified view of the system's operation.

The rest of this paper is structured as follows: Section 2 introduces the X-Ray approach and how it is applied to monitor data processing systems. Section 3 describes how it is implemented using bytecode instrumentation. Section 4 evaluates the proposal with a case study and a benchmark. Section 5 contrasts the proposed approach with related work. Finally, Section 6 concludes the paper.

2 Approach

Figure 1 presents an overview of the proposed X-Ray architecture. From left to right, X-Ray targets distributed applications and data stores 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.

The main component of X-Ray Capture uses bytecode instrumentation, a mechanism for modifying compiled programs. It uses `asm` [10], a stateless *bytecode* manipulation library modelled on the *hierarchical visitor* pattern [15]. This instrumentation inserts instructions to generate logging events and maintain context.

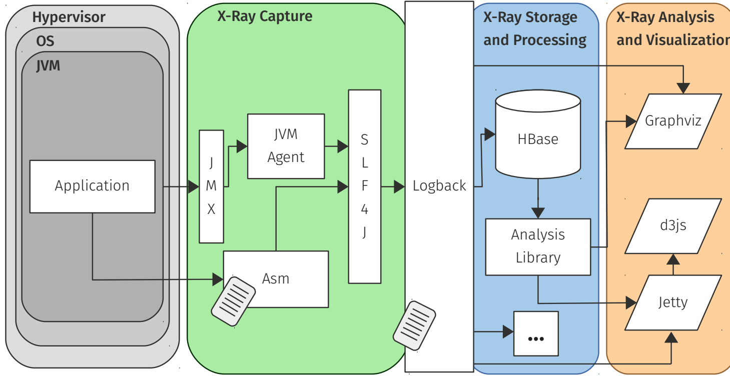


Fig. 1. Overview of the X-Ray architecture

This is the main configuration point for monitoring Java programs. Bridges or file processing can be used to obtain information about non-X-Ray-ready programs. Finally, an agent periodically monitors and collects metrics about the underlying Java runtime from Java Management Extensions(JMX) [12].

Bytecode instrumentation can be customised by choosing what methods to alter and what operations to perform. `Logback` enables processing various actions for the same message. Those actions are executed by entities called appenders, programmable in Java, and configurable with a simple XML or Groovy file.

The resulting events are routed through Simple Logging Facade for Java (slf4j), a logging facade for Java [23], and `Logback`, a logging framework implementing the (slf4j) API [22]. `Logback` works as an event spooler: it enriches events with additional information and delivers them asynchronously to the X-Ray Storage and Processing module. In the Storage and Processing module, events can be stored in HBase (the sink database), that can be configured to sustain high-throughput writes.

The Analysis Library contains analysis procedures applicable to the monitoring data, specially concerned with request tracking across software modules and components. It joins logs originating 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. A visual representations is then made available to the end-user, with Graphviz or D3.js, or even exported to enable further processing and interaction with external systems.

2.1 Request Tracking

The key feature of the X-Ray Capture layer is being able to chain operations performed on behalf of each end-user request, to highlight the decomposition of a data processing request in terms of software components and hardware resources. This is achieved by using tags and probes, as follows.

Instrumentation provides the ability to add tags to entities being observed. A tag is an automatically generated unique 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. In detail, a tag can be associated with an object or thread and through configuration instructions it is possible to generate, remove, move, or copy it to a thread or object when a method is executed. Thus, during execution tags can be associated with multiple threads and objects and flow through them.

It is also possible to associate a tag with message send and receive events in sockets thus supporting communication between different virtual machines. This takes advantage of FIFO order and the unique identifier of the socket (including both addresses and ports, as well as a timestamp) to establish a mapping between tags existing at both ends.

Probes implemented in X-Ray allow information to be collected on entry and exit(s) of selected methods. The target data – name and reference of executing class, name and signature of the method, the current thread and the parameters/return value – are accessible to all defined probes. Moreover, the probe also collects tags associated with the current object and thread.

This makes it is possible to follow a logical work unit, even if it is scattered across multiple threads and processes. The simplest usage just adds a tag to the thread on starting to execute the request and removes it on completion. This makes all work done by that thread, regardless of the software component invoked, to be associated with that request.

Consider a more complex example of an application that uses a background thread to periodically write multiple outstanding data items produced by different clients. This is harder to track as the work done by the background thread contributed to multiple requests. With X-Ray, one would copy the tag from the request thread to the object queued for the background writer. On reception, the tag would be copied from the object to the background writer thread. When the background thread uses the I/O resource, it would be tagged with the tags of all corresponding requests.

Finally, consider an example of a client/server system, where a request is partially executed at the client and at the server. In this case, one would tag the client thread upon starting the request, but also server threads whenever a remote invocation is received. Moreover, when a remote invocation is issued and received, socket tagging will map client and server side tags.

2.2 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 configuration files will override the annotations.

Annotation use implies access to source code of the program to alter, and each change in the configuration requires a program recompilation to take effect. It also results in configuration being spread over 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.

On the other hand, configurations written in files do not require access to source code. Likewise, it is not necessary to recompile the program after each change – 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. Entities (class, method and package) identifiers are not exactly equal to the entities they represent so a manual search and replace may fail to modify them, and likewise IDE-assisted refactoring can also be ineffective.

2.3 Usage Methods

The first alternative to apply X-Ray 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.

Because of security restrictions preventing deep and potential unsafe changes, it is not 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 or modify the desired configurations and restart the program through the X-Ray class loader.

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.

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. As the recompiler acts on JAR files, it is possible to modify methods in the `java.*` packages if the input JAR corresponds to one where JRE classes are, but this is not recommended as it would permanently alter them.

The last option is a Java Agent [21, `java.lang.instrument` documentation]. Depending on the support provided by the JVM, it can be initiated along with the program by passing an argument to the command line or it can be attached to a running instance after it has started. Similar to static 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 Java Virtual Machines (JVMs) support it. Among those that do, there is no standard way to do some things, specially initiating an agent after the virtual machine start-up [16, § 8.4.3.4].

3 Implementation

Instrumentation of Java programs to insert tags and probes starts by reading configuration files, if available, adding their commands to the framework's internal state. Classes are then loaded, either due to the program running or by statically traversing the JAR file, depending on the usage method chosen. For each of them, it scans the file and for each method decides if it should be instrumented using configuration from files and configurations acquired by reading annotations in the currently analysed class and other loaded classes.

This may require visiting the bytecode of super-classes or of the implemented interfaces if they were not already visited, as the decisions on a class may depend on information contained in other classes. 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.

This process happens again each time a class is needed, until all are loaded and transformed. This approach is only possible because the binary representation of Java programs corresponds to a well specified, platform independent format that can be manipulated.

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, 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.

3.1 Selection of Instrumentation Targets

X-Ray operates every time it is called to resolve a class, meaning, to return the bytecode associated with a class. The decision whether to instrument or not is made for each method, on a case-by-case basis. A method m in class C will be altered by X-Ray if there is configuration on: the analysed method m;

m's enclosing class C; the package of C; 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; or a package that contains a supertype C extends or implements, but only if that indication is inherited. 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.

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 there is an original implementation of m with log level of DEBUG, m will have the DEBUG level.

3.2 Modifications to Targets

X-Ray adds fields and other information needed for its operation. The first 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 (using the "$_xray_" prefix) and with a special marker to indicate they 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.

Before copying the original instruction to the new class representation, the necessary 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. 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.

Each event has a unique identifier associated with it. 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. [21, `java.rmi.dgc.VMID` documentation][21, `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 time stamp.

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, if not directly obtainable from logback. Other information is copied using Mapped Diagnostic Context (MDC), a per-thread key-value map, available at run-time in several code locations.

```
1  xray("org.apache.derby.iapi.sql.execute") {
2    instrument("NoPutResultSet") {
3      inherit = true
4      log "openCore()V"
5      log "getNextRowCore()Lorg/apache/derby/iapi/sql/execute/ExecRow;"
6    }
7  }
8  xray("org.apache.derby.impl")
9
10 xray("org.apache.derby.client.am") {
11   instrument("NetResultSet") {
12     tag "next()Z"
13     send("next()Z", "?laddr", "?raddr")
14   }
15 }
16 xray("org.apache.derby.impl.drda") {
17   instrument("DRDAConnThread") {
18     tag "processCommands()V"
19     receive("processCommands()V", "?laddr", "?raddr")
20   }
21 }
```

Fig. 2. X-Ray configuration for federated Apache Derby

4 Evaluation

4.1 Case Study

The first experiment is to monitor a distributed query. The query is made to a federated SQL database management system built with Apache Derby. That database is composed of two nodes, communicating with each other. The final element of the distributed query is the client, which initiates the computation by making a request to one the servers. To satisfy the request the server must execute a sub-query on the other server, join the partial results, and return the final values of the query to the client. The goal in this case study is to apply the X-Ray framework to assess if it is possible to monitor this process, see how to do it and observe the obtained results.

Most of the configuration needed for X-Ray to do this is shown in Figure 2. In detail, lines 1 to 7 set probes on the base class of relational operators. Then, in line 8, the package implementing such operators is declared as being instrumented. This means that all classes found extending such base class, *i.e.*, all operator implementations, get instrumented.

Moreover, lines 10 to 15 target the JDBC driver, which is the entry point into Apache Derby. It sets a tag on entry of the `next()` method that is used to retrieve data. Moreover, it links this tag to a message being sent on the socket connection to the server. Some other methods in the client driver (not shown) are also instrumented in the same manner. Finally, lines 16 to 21 target the server-side protocol handler, by setting a tag on each received message and then linking it to the context of the client socket, thus relating it to the corresponding client-side context. This requires minor changes to Derby's source code, to expose the communication ports to X-Ray.

As a result we obtain Figures 3(b) and 3(a). They were obtained by saving the logs produced by client and database servers to HBase, and by invoking the analysis component to read that information from HBase, reconstruct it, and add relevant remote communication event nodes between the datastore nodes and the client.

The resulting flow graph shown in Figure 3(b), was rendered by d3.js and served by Apache Jetty. It is accessible and continuously updated at runtime. Each bar represents a logged object and its width how many times it emitted logging events to X-Ray. Figure 3(a) was produced by Graphviz from a `dot` file also being continuously updated. It is similar to other graphs used to represent relationships between objects. X-Ray could be used to track the relations between methods, as presented in [7,8]. All these graphics, additionally, have the added feature of also representing remote connections.

Note that node labelled as "[8]" denotes a socket connecting two processes and that the relation between parts of the computation taking place in different processes is done automatically by the X-Ray system. Furthermore, most of the nodes represent classes whose name ends in ResultSet. Except the one labelled NetResultSet, all of them were obtained from the configuration in lines 1 to 8 of Figure 2: the inheritance of configuration makes this succinct. Finally, it is clear that the structure of the computation and the amount of data handled by each software component is exposed to developers and administrators.

4.2 Performance

The performance impact of X-Ray instrumentation and probes was measured by starting a Derby server and running the TPC-C transaction processing benchmark [13]. 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 code base developed by a third-party and that potentially makes use of features that conflict with the framework.

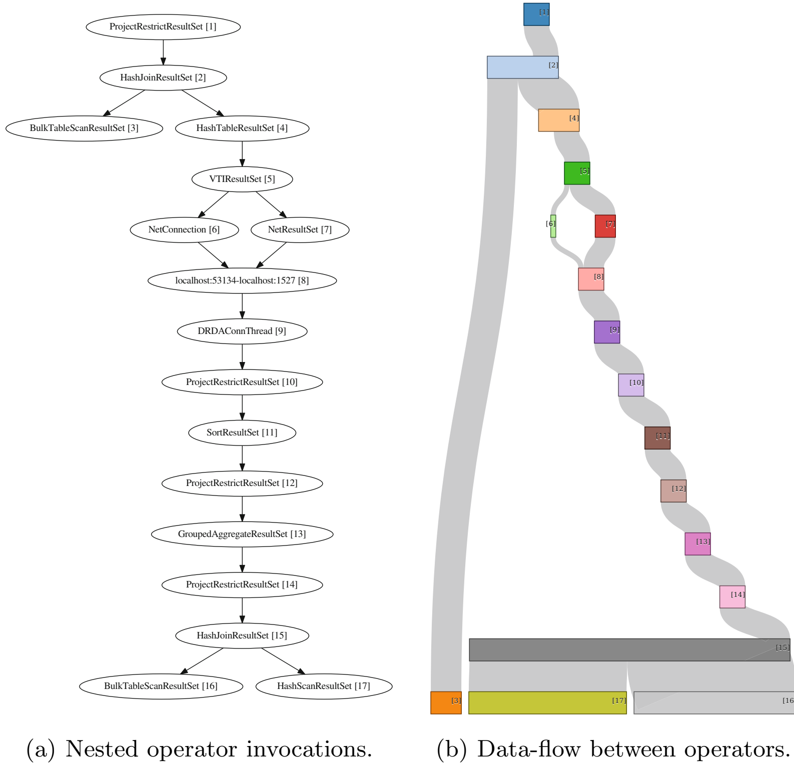
The database where the TPC-C benchmark was run is a single warehouse with approximately 200 MB of data and for the workload, 1 (one) client making requests without delay between them (i.e. with no think time). Two machines were used for these tests, both with 128 GiB of RAM, 24 cores and a disk with 7200 rpm. Their OS was Ubuntu 12.04.3 LTS (GNU/Linux `3.2.0-27-generic x86_64`) and their Java environment was the Oracle JVM version `1.7.0_60`. One of them was used to run the Apache Derby instance plus the benchmark client and the other was used to run HBase.

Table 1 shows results obtained with the following configurations:

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.



(a) Nested operator invocations. (b) Data-flow between operators.

Fig. 3. Visualisations of a distributed SQL query

Table 1. Execution times of running TPC-C on Derby

Configuration	Latency (ms)	Throughput (tpmC)
Baseline	50.2	526
Instrumented	57.3	478
Logging to text file	64.1	414
Logging to HBase	67.4	7
Asynchronous HBase	63.3	422
Lossy asynchronous to HBase	61.4	437

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

Asynchronous to HBase. Using the previous appender, but wrapped under an asynchronous appender [2,4] that performs logging in other threads, asynchronously.

Lossy synchronous to HBase. Again, the HBase-appender is employed, but this time wrapped under the asynchronous appender provided by Logback [18] that may drop messages at times of congestion.

Table 2. Space occupied and number of lines written during the benchmark

Configuration	Size (Bytes)	Size (lines)
Logging to File	317 024 499	2 040 975
Logging to HBase	436 240	260 760

For the *Logging to file* and *Logging to HBase* tests, Table 2 shows the size occupied in disk at the end of the benchmark and the number of lines written. In the case of the *Logging to file* test, the number of lines and size refer to the written file, and in the case of the *Logging 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 recognise that data written to HBase was much less than what was written in the file. But considering the total number of transactions, we have Hbase with $436240/877 \simeq 497.42$ and file with $317024499/6400 \simeq 49535.08$ bytes per transaction. This means that each set of events representing a transaction takes ≈ 99.6 times more space to represent in a file than in a table in HBase. The *Logging to file* run is portrayed to show an upper limit to logging load. In practice, better performance could be achieved by tweaking some parameters, such as changing Logback appender layout, delaying writes to disk or using several log files and removing old ones.

It is possible to see that when using instrumentation (*Instrumented*) the benchmark ran at 90.87% of the original speed; that is only a 9.13% slowdown. Although simply using logging events to HBase synchronously incurs an heavy penalty, when using an asynchronous approach, the result is far more acceptable – 88.28% of the simple instrumentation speed, or 19.77% of the original *Baseline* run. If one does not care about the possibility of dropping events, a better 16.92% can be achieved.

5 Related Work

There are already some monitoring or analysis solutions for distributed contexts. Their purposes range from assisting debugging, identifying system bottlenecks and network problems, discovering most frequent paths between nodes, to detecting potential intrusions.

NetLogger [17] has a architecture similar to X-Ray, but no support for automatic insertion of logging statements. To use it developers need to add explicit invocations to the framework to benefit from it, which forces access to source code. Pinpoint [11] does monitoring of components in a distributed system automatically, by tagging client requests with an identifier. Although useful, it only works for J2EE applications. Pip [24] allows users to configure its operation by using a domain-specific language. It with predicates about communications, times, and resource consumption can be made and at runtime those predicates can be validated or invalidated. Relations between components can be discerned by the construction of causal paths, with the help of the provided configurations and collected information. As with NetLogger, the source code is required,

as annotations are used to generate events. iPath [9] is a dynamic instrumentation tool with some interesting properties for a distributed setting. It was conceived and works on distributed systems. It also allows a more focused analysis as the methods to analyse can be chosen and this selection can be altered at runtime. 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. Aspect-Oriented Programming (AOP) [19] has also been used for monitoring. However, on one hand it provides too much freedom when altering programs, hence, additional complexity, and on the other hand does not provide the event spooling and processing components of X-Ray. Frameworks such as AspectJ [1] could also have been used instead of `asm` [10], although this would add some overhead to class loading in comparison to the simple processing done now.

All these proposals, like X-Ray, require some information about the software to examine. This enables the extraction of more targeted and meaningful information. On the other hand, treating target sysems as black boxes makes the tool applicable when no internal details are known.

A system that follows the latter model is presented in [25]: considering that information about the components or middleware of a distributed service may be minimal and the source code unavailable, the proposed request tracing tool considers each component as 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 follows 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). 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. That detection depends on SystemTap [5] hence this solution it is not OS independent and has no JVM support implemented. Another similar solution is present in [6]. It only traces network messages, without any knowledge of node internals or message semantics and infers the dominant causal paths through them. It uses timing information from RPC messages and signal-processing techniques to infer inter-call causality. But although the principle that no information exists about the components is a valid one and the results obtained are useful, for X-Ray different assumptions were made: even if the source code of a component cannot be altered and deployed, it is still available or its general API is, and so should be used.

6 Conclusions

In this paper, X-Ray, a framework for distributed systems analysis and monitoring was presented. Due to its flexible instrumentation mechanism, accepting

configurations and altering bytecode accordingly, it is applicable to any system on the Java platform. The main contribution is the ability to track requests across thread and process boundaries, and to expose distributed data processing operations. Unlike existing solutions, it tries to balance the need for application-specific information, that normally require source code, with the goal of working with highly heterogeneous components.

During the qualitative analysis made when applying X-Ray to Apache Derby, its applicability and usefulness when instrumenting and analysing a large code-base was demonstrated. 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, a certain amount of knowledge of the source code was still required, specially to make communication ports available to instrumentation code, but with mechanisms such as reflection and callbacks, access to the source at runtime is still not required. As for the quantitative measurements made, they confirmed that the proposed solution is lightweight and its usage does not impose an expensive overhead.

Acknowledgements. This work has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 611068, project CoherentPaaS – Coherent and Rich PaaS with a Common Programming Model (http://CoherentPaaS.eu).

References

1. The aspectj project, http://www.eclipse.org/aspectj/
2. Disruptor-based AsyncAppender for Logback, https://github.com/reactor/reactor/tree/master/reactor-logback
3. Factsheet CoherentPaaS, https://ec.europa.eu/digital-agenda/sites/digital-agenda/files/CoherentPaaS_Factsheet_v0.21_0.pdf
4. Project Reactor Homepage, http://projectreactor.io/
5. Systemtap, https://sourceware.org/systemtap/
6. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 74–89. ACM, New York (2003), http://doi.acm.org/10.1145/945445.945454
7. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. SIGPLAN Not 32(5), 85–96 (1997), http://doi.acm.org/10.1145/258916.258924
8. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 29, pp. 46–57. IEEE Computer Society, Washington, DC (1996), http://dl.acm.org/citation.cfm?id=243846.243857
9. Bernat, A.R., Miller, B.P.: Incremental call-path profiling: Research articles. Concurr. Comput.: Pract. Exper. 19(11), 1533–1547 (2007), http://dx.doi.org/10.1002/cpe.v19:11
10. Bruneton, E., Lenglet, R., Coupay, T.: Asm: A code manipulation tool to implement adaptable systems, http://asm.ow2.org/current/asm-eng.pdf

11. Chen, M., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of the International Conference on Dependable Systems and Networks, DSN 2002, pp. 595–604 (2002)
12. Corporation, O.: 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
13. Council, T.P.P.: TPC-C Homepage – Version 5.11, http://www.tpc.org/tpcc/
14. pgAdmin Development Team, T.: pgAdmin III 1.20.0 documentation, http://pgadmin.org/docs/1.20/query.html
15. DiFalco, R.A.: Hierarchical visitor pattern. Wiki Wiki Web (2011), http://c2.com/cgi/wiki?HierarchicalVisitorPattern
16. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional (2014)
17. Gunter, D., Tierney, B., Crowley, B., Holding, M., Lee, J.: 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 2000, p. 267. IEEE Computer Society, Washington, DC (2000), http://dl.acm.org/citation.cfm?id=580760.823762
18. Gülcü., Pennec, S., Harris, C.: The logback manual, Chapter 4: Appenders, http://logback.qos.ch/manual/appenders.html#AsyncAppender
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Marc Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Kolev, B., Valduriez, P., Jimenez-Peris, R., Martínes Bazan, N., Pereira, J.: Cloud-MdsQL: Querying heterogeneous cloud data stores with a common language. In: Gestion de Donnés – Principes Technologies et Applications (BDA) (2014)
21. Oracle Corporation: Java Platform, Standard Edition API Specification, 8th edn.
22. QOS.ch: Logback Project, http://logback.qos.ch
23. QOS.ch: Simple Logging Facade for Java (SLF4J), http://www.slf4j.org
24. Reynolds, P., Killian, C., Wiener, J.L., Mogul, J.C., Shah, M.A., Vahdat, A.: Pip: Detecting the unexpected in distributed systems. In: Proceedings of the 3rd Conference on Networked Systems Design & Implementation, NSDI 2006, vol. 3, p. 9. USENIX Association, Berkeley (2006), http://dl.acm.org/citation.cfm?id=1267680.1267689
25. Sang, B., Zhan, J., Lu, G., Wang, H., Xu, D., Wang, L., Zhang, Z., Jia, Z.: Precise, scalable, and online request tracing for multitier services of black boxes. IEEE Transactions on Parallel and Distributed Systems 23(6), 1159–1167 (2012)
26. Stonebraker, M.: Technical perspective: One size fits all: An idea whose time has come and gone. Commun. ACM 51(12), 76 (2008), http://doi.acm.org/10.1145/1409360.1409379
27. The PostgreSQL Global Development Group: PostgreSQL Documentation 9.4: EXPLAIN, http://www.postgresql.org/docs/9.4/static/sql-explain.html