CoherentPaaS

Coherent and Rich PaaS with a Common Programming Model

ICT FP7-611068

X-Ray Subsystem Design

D8.1

May, 2014

Document Information

Scheduled delivery	31.03.2014
Actual delivery	06.05.2014
Version	1.0
Responsible Partner	INESC

Dissemination Level:

PU	Public
PP	Restricted to other programme participants (including the Commission)
RE	Restricted to a group specified by the consortium (including the Commission)
CO	Confidential, only for members of the consortium (including the Commission)

Revision History

Date	Editor	Status	Version	Changes
26.03.2014	J. Pereira	Draft	0.1	Initial version
17.04.2014	J. Pereira	Draft	0.2	Revised version
06.05.2014	J. Pereira	Final	1.0	Final version

Contributors

J. Pereira, P. Guimarães (INESC)

Internal Reviewers

V. Spitadakis (NTUA), N. Martinez (Sparsity)

Acknowledgements

Research partially funded by EC 7th Framework Programme FP7/2007-2013 under grant agreement n° 611068.

More information

Additional information and public deliverables of CoherentPaaS can be found at: http://coherentpaas.eu

Glossary of Acronyms

Acronym	Definition
CEP	Complex Event Processing
CPU	Central Processing Unit
CQE	Common Query Engine
D	Deliverable
DQE	Distributed Query Engine (Apache Derby)
HV	Hypervisor
JMX	Java Management Extensions
JVM	Java Virtual Machine
CloudMdsQL	Cloud Multi-datastore Query Language
MDC	Mapped Diagnostic Context
OS	Operating System
PaaS	Platform as a Service
SLF4J	Simple Logging Framework for Java
SQL	Structured Query Language

Table of Contents

List of Figures

1. Executive Summary

The X-Ray subsystem aims at a fine-grained analysis and real-time visualization of performance and resource usage of cloud applications deployed on CoherentPaaS. This will enable the identification of each request, which software modules it uses, and associate resource usage to them. In particular, it will allow cloud data stores to provide detailed profile information about the executed queries such as their cost, selectivity, and so on.

This document describes the design of the X-Ray subsystem, focusing on the overall architecture and identifying the common data-model, inter-operability interfaces and protocols, and guidelines for data sources. It also describes how bytecode rewriting techniques play a key role in transforming Java cloud applications to analyse their performance and how monitoring information can be stored on any of the CoherentPaaS data stores to perform analytical queries over it.

2. Introduction

CoherentPaaS aims at designing and implementing a Platform-as-a-Service (PaaS) that allows access a wide range of services and data store technologies in the cloud. These include NoSQL-data stores (key-value data stores, graph 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. This data will be available in a unified way and with the ability to use a query language.

This platform is thus inherently heterogeneous, with components from multiple sources, and distributed, to scale out and take advantage of cloud infrastructure. Both these characteristics make it difficult to monitor its operation. Moreover, cloud applications and services tend to be themselves synthesized out of multiple components and libraries and typically service providers do not have a detailed understanding of all components.

On the other hand, CoherentPaaS is focused on storing and processing large amounts of data, thus providing the foundation for a more flexible monitoring system that can answer a wider range of questions over the system as a whole. The X-Ray subsystem addresses this challenge and exploits this opportunity, by providing the missing link between the monitored system and the data storage and processing functionality built into the CoherentPaaS platform.

The following sections discuss the main issues in system and application monitoring, existing proposals, providing the background for the X-Ray subsystem in CoherentPaaS.

2.1. Data sources (What to Monitor)

In this section we discuss what information is relevant in application and systems monitoring, regardless of how it is collected and how it is analysed or presented in a useful format to the end-user. These data include hardware and system resources usage, application structure and events, request tracking, as well as the relation between different sources.

2.1.1. Hardware and system resources

The main target for monitoring is usage of hardware resources such as CPUs, main memory, disk storage, and network. This information is often only statistically accurate or machine/architecture-specific. 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 different results on some metrics can be obtained for the same application.

The usage of system resources, such as open file descriptors, threads, processes, and virtual memory, is also important in understanding application behaviour for debugging and performance evaluation. The JVM in the Java platform adds an additional layer with resources that need to be monitored. In particular, the usage of a garbage collector adds substantial complexity. For instance, regarding the pauses introduced in normal application behaviour while unused memory is reclaimed, or how object life-cycles match the expectations of a generational garbage collection mechanism. Both have an impact in application performance and need to be monitored.

2.1.2. Application events and state

Application events are exposed mainly as text-based logs. Each event is thus formatted by application code as a text line that can be displayed in a console, recorded to a text file or shipped across the network for further processing and storage. The information in each event is thus application specific and more suited for humans than for automated processing. It is usually annotated with a detail level for coarse grained filtering, ranging from errors and warnings to debugging and tracing. More recently, the SLF4J and Logback frameworks (1) allow that log entries to be annotated with markers, to ease filtering, and with arbitrary key-value pairs that describe event context, to allow chaining related events. This makes these events more amenable to automated processing.

Applications can also expose the usage of their internal subsystems with considerable detail, although in an application specific format. An example of this is how database management systems, with a high level query languages such as SQL that allows alternative execution strategies, show how queries map to internal code modules. This allows users to better understand what the actual actions the database management system is performing. Examples of this are the EXPLAIN ANALYZE statement in PostgreSQL and MonetDB's Stethoscope (2).

2.1.3. Context and request identification

A key issue is the relation between code elements (methods, functions, or procedures) such that it is possible to visualize how an action is composed and what sub-actions it executes. This is important because it allows a query execution plan to be produced similarly to how a relational database builds it, even if not by a direct use of the same algorithm, by applying the same concepts in a slightly different way (3).

This information can be collected with different levels of detail. On one extreme, full information of the methods called when executing a program can be collected, obtaining a call tree. 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, it can grow to a large size, especially if one considers cases of call recursion. The detailed context present can also be unnecessary if the distinction of calls can be inferred or is not important.

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

The context calling tree (CCT) is a compromise between these two formats (3). 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 hard problem.

2.2. Data collection (How to Monitor)

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 the 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 rationalize 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 host JVM or OS functionalities) or altering the program to collect them. In distributed systems, there is the additional issue of consistent observation.

2.2.1. Hardware and platform assisted

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 when compared to software instrumentation (3) (4).

The number of available hardware counters in a processor is limited and not all useful information can be kept. So it is necessary to conduct multiple measurements to collect the desired metrics or to continuously collect, process, and save this data. The types and meanings of hardware counters vary from one kind of architecture to another due to the variation in hardware organizations so they cannot be relied upon as a universal solution. Lastly, it can be difficult to correlate low level performance metrics to source code or application level.

The operating system collects a large amount of useful information about applications, either for its own usage such as maintaining a fair allocation, or because it is in a privileged position to do so. Another key role of the operating system is to virtualize hardware counters such that multiple applications can exploit them concurrently. Information collected by the operating system can be exposed through a system-call interface, a device interface, or a file-system interface.

Likewise, the Java Virtual Machine (JVM) collects information about resource usage by Java applications. This includes the heap, with information on garbage collection, and threads. This information is in current versions made available mainly through managed beans (MBeans) according to the Java Management Extensions (JMX) specification. These objects represent resources to be managed providing both attributes that can be periodically queried, thus extracting a time series, or events source that can be listened upon.

2.2.2. Code instrumentation

Instrumentation is the act of altering program behaviour by adding the ability to measure or know some information when they are run. This can be done by developers in the source code and is eased by frameworks that reduce the effort involved while improving interoperability with different processing and analysis tools.

The form of instrumentation that is most used in the Java platform is logging of significant events. This is supported by frameworks such as Log4J that allows the end-user to filter events based on program module and level of detail, while configuring the log format and its final destination. More recently, there has been a growing interest in reducing the overhead of logging operations by avoiding text formatting for events that have been disabled at run-time (1).

The Java platform supports source code instrumentation also with the Java Management Extensions (JMX) specification by describing data exposed as managed beans (MBeans). These can be queried by the same tools that consume platform information through JMX.

Even though the programmer can insert instrumentation manually, it is more convenient to have it inserted automatically by tools. This can be done statically, in one occasion, or dynamically, every time the program is loaded or when 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 (4).

The architecture of the instrumenter can also be very diverse. It can be a simple program that alters an application or it can be part of a modular system capable of being extendible with a wide range of plug-ins or similar tools. Examples of monitoring or tracing software that use instrumentation include the venerable gprof, Intel's Pin (5), DTrace (6) and SystemTap (7), valgrind (8) and its tools, such as memcache, callgrind, and Google's Flayer (9).

Nowadays, most high-level programming languages provide reflection capabilities. Reflection allows a program or application to change its behaviour at runtime. In particular, for object oriented programming languages, reflection enables changes to the behaviour or data structures of objects at runtime. Python and Ruby are examples of languages that support reflection and can use it to insert instrumentation.

For the particular case of Java applications, there various solutions specific to the JVM (Java Virtual Machine), as the JVM abstracts away many platform and hardware details and provides some functionalities that enable it, like Java Agents, instrumentation interfaces, custom class loaders and JVM TI. Java bytecode, the instrumentation target, is also a relatively simple and high level target when compared to assembly. Examples of tools that take advantage of these features include jp2 (10), JBInsTrace (11) and the systems described in (12) and (13).

An extreme example of automated code instrumentation by bytecode manipulation is provided by Minha (14) (15). By intercepting all synchronization and communication primitives, it virtualizes multiple JVM instances in each JVM while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of a much larger distributed system. This enables fine grained tracking of requests in a distributed system and consistent global observation.

2.2.3. Distributed monitoring

Monitoring distributed systems raises the issue of consistent observation. Briefly, as communication is asynchronous and clocks on different nodes are not necessarily synchronized, a naïve monitor that builds global observations of the system, including data from multiple nodes, might observe paradoxical states (16). This can be solved by causally ordering monitoring messages such that observations correspond to consistent snapshots. However, observing transient states requires that the monitor reconstructs different sequential orders of events and might, in the end, not be able to definitely assess if some state actually existed in the system.

Another issue with distributed systems is handling the sheer amount of monitoring events generated, as any event can potentially be related to any other. This means that the infrastructure needed to aggregate and process events becomes a challenge. An example of a logging structure is EV-Path (17), 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 (18) and WebLogic Event Server (19). This is also addressed by tools such as Fluentd (20), which aggregate text-based logs and allow storage and processing with the Hadoop (21) tools.

Finally, request tracking across distributed systems presents additional challenges. This is addressed by PreciseTracer (21), a tool that regards distributed components as black boxes and follows requests as they are passed between components and save information in a per-request structure. This analysis is made on line and on demand as the logged system and the logger nodes are running without the need to stop neither.

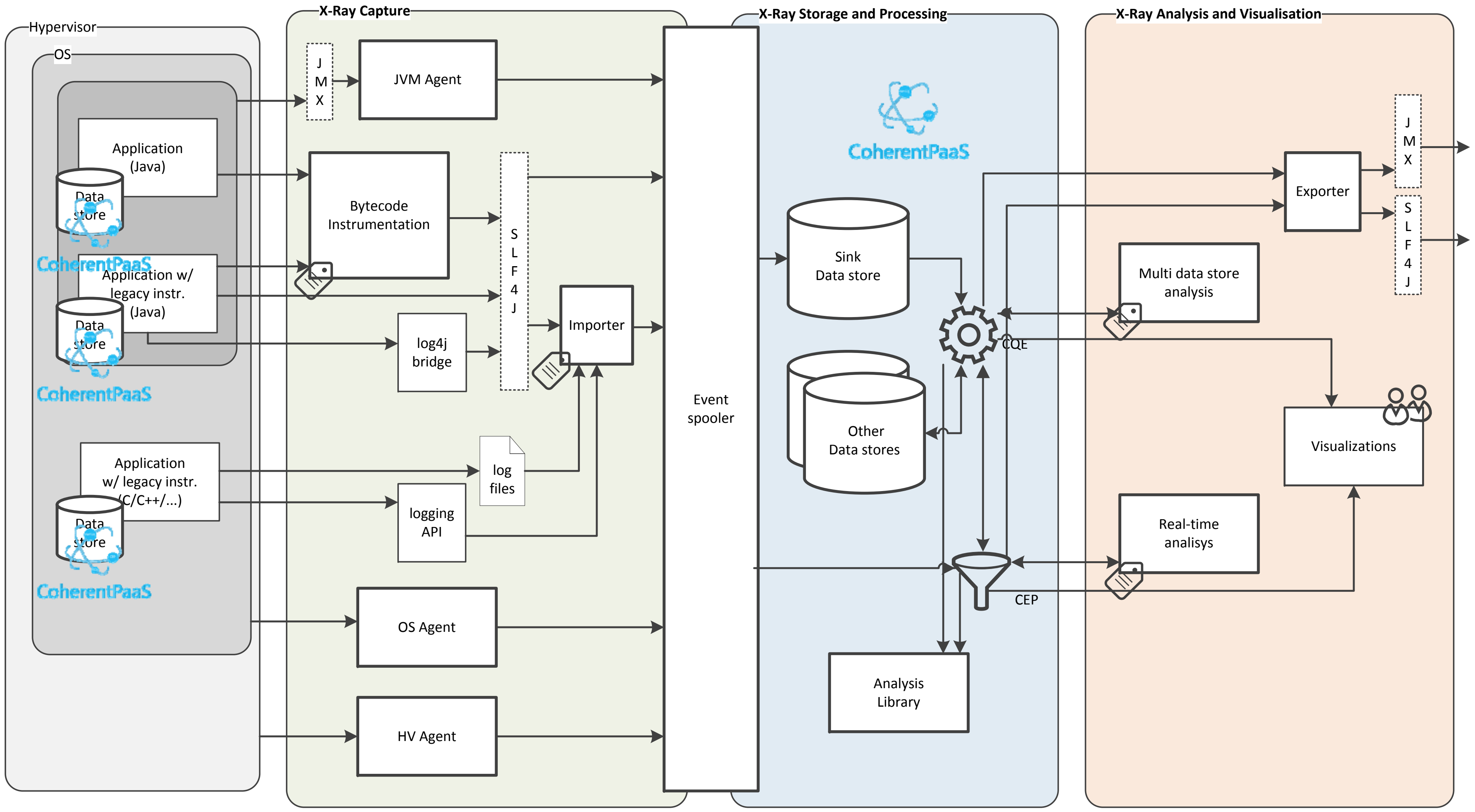

Figure 1 Architecture overview

3. Architecture

The X-Ray subsystem focus on improving the information that is generated using code instrumentation and using the CoherentPasS data stores and common query engine for processing monitoring data. The delivery of a fully self-contained prototype requires that a complete monitoring architecture is proposed and implemented.

Figure 1 shows an overview of the proposed X-Ray Subsystem architecture. It assumes distributed applications with 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 Visualization layer. Label icons identify the main configuration points for the system.

Note that the CoherentPaaS platform and the data stores have two different roles in this architecture. First, they are the mechanism used to store and process monitoring data. Second, they are the monitoring target while being used by applications, requiring that an adequate configuration of the Capture layer is provided and can easily be enabled by system operators.

The following sections briefly describe the components and interactions in each of them.

3.1. Capture

The objective of the capture module is to obtain information from application programs and resources used, through a variety of means and sources, and providing it to the next stage. This includes the following main components:

Bytecode instrumentation Tools for dynamically modifying compiled Java bytecode to insert request context maintenance and propagation and software monitoring probes. This can be complemented by manually inserted logging code in application components targeting SLF4J. This is one of the main configuration points, where software structure is described to the monitoring platform.

JVM, OS, and HV Agents Tools for monitoring resources used at different levels of abstraction.

Event spooler Middleware layer for enriching events with additional information, namely, that aids in correlating different event sources from the same component or from multiple components in different servers, and directing them to storage and processing stage.

Support for application components that already have relevant monitoring capabilities built on logging frameworks or JMX, or application components that are not based on the Java platform, is achieved with the following component:

Importer Extracts information from textual log messages, producing monitoring events. This is a main configuration point, where legacy software is described to the monitoring platform. These messages originate from multiple sources:

- From textual log files produced by application components using their native logging mechanism.
- From log messages produced by text-based Java logging frameworks (e.g., log4j) captured using a bridge.

- From log messages produced by text-based logging frameworks outside the Java platform.

The preferred scenario for the capture module is an application in the Java platform where monitoring is added using bytecode instrumentation to direct events through the SLF4J interface to the event spooler, while resource consumption is monitored by agents at different levels.

3.2. Storage and processing

This module leverages an instance of the CoherenPaaS stack as mechanism for storage and processing monitoring data as follows:

Sink Data store A database engine optimized for throughput and accommodating the variable information contained in events (NoSQL). Applications can make direct use of this data and native processing capabilities.

Other Data stores Database engine holding intermediate information resulting from event processing in the CQE and CEP engines, optimized for their processing operations. Applications can make direct use of this data and native processing capabilities.

CQ Engine The common query engine orchestrates and performs computations across data stored in multiple database engines.

CEP Engine The complex event processing performs real-time processing of monitoring events, making use of database engines through the CQE for historical data.

Analysis Library A set of general purpose analysis procedures that can be applied to monitoring data. Mainly, this is concerned with request tracking across software modules and distributed components.

The storage and processing component assumes that minimal pre-processing or filtering has been done, namely, by the Event Spooler. The entry points to a Sink Data Store or the CEP Engine are thus optimized for throughput. If manipulation and filtering on arrival are required, these are done in the CEP Engine, directing the results to any database engine.

3.3. Analysis and visualization

The analysis and visualization layer is directly visible to end-users, focusing on monitoring policy. It includes:

Real time analysis Defines data transformation in terms of complex event processing operations. This is one of the main configuration points, where extracted information is defined.

Multi-data store analysis Defines data transformation in terms of single or multi-database query languages. This is one of the main configuration points, where extracted information is defined.

Exporter Exposes data stored and computed using standard JMX and SLF4J interfaces to chain to existing monitoring and visualization systems.

Visualization applications Expose data stored and computed to end-users, thus providing a self-contained proof-of-concept of the monitoring system.

4. Design issues

This section discusses design issues, namely, the common data-model for monitoring data, inter-operability interfaces and protocols, and guidelines for data sources.

4.1. Common data-model

The X-Ray platform will use the CloudMdsQL data types as specified in Appendix A of D3.1, as these are already mapped to data stores considered in the project and are sufficiently generic for a wide range of applications. They include simple data types as well as arrays and dictionaries. Moreover, a mapping between these data types and those allowed in JMX (22) will be provided.

4.2. Interoperability interfaces

Regarding data capture, the main interoperability interfaces that allow taking advantage of existing instrumentation are:

- JMX is used mainly for reading property values that are monitored by existing systems, and also for capturing events where available.
- SLF4J is used for events, taking advantage of existing legacy bridges (e.g., log4j-over-slf4j) to capture log messages associated with them.

For data storage and processing, the main interoperability interface is CloudMdsQL, the CoherentPaaS common query language as defined in D3.1.

Resulting data is exported to third party tools mainly through the following interfaces:

- JMX is used for exposing values stored in any of the supported data stores as properties and results from complex event processing as event notifications.
- SLF4J is used for periodically logging values stored in any of the supported data stores or results from complex event processing as event notifications.

By relying on these interfaces, compatibility with a wide range of tools is achieved. First, most management tools already support JMX, even if they are not dedicated to the Java platform. With SLF4J, backends such as Log4J and Logback can be used to locally record data or to ship it to repositories over the network.

4.3. Data source definition

The main interface for definition of data sources is by specifying the instrumentation to be inserted in Java code, using three different formats:

- Annotations inserted in the Java code by the developer to delimit logical code units and their interactions.
- An external definition file that can be used in alternative to annotations to delimit logical code units and their interactions, when source code is not available or it should not be changed.
- External configuration files that select what annotations and definitions are actually enabled and translated into active instrumentation code, while directing the output to the event spooler.

Data imported from legacy data sources is specified in two different ways:

- Designating JMX attributes to be queried or events to be listened. These are converted to the internal data model using the JMX mapping and sent to the event spooler.
- Designating SLF4J loggers as legacy event sources, by associating them with an *appender* that parses them using a supplied regular expression and sending the resulting data to the event spooler.

5. Implementation issues

This section discusses implementation issues, namely, how existing software packages can be used as-is or with some modification as building blocks in the proposed architecture.

5.1. Bytecode instrumentation

Inserting instrumentation in Java bytecode is now widely used for multiple purposes and is supported by a variety of tools. This project will leverage ObjectWeb's ASM library (23) for bytecode manipulation given its low level and flexibility, in addition to the reduced overhead allowed by its stateless operation based on the visitor pattern.

There are also multiple third-party add-ons to ASM that provide useful functionality, namely, the native-intercept library (25) allows seamless handling of native methods implemented in C/C++ according to the JNI specification.

Inserted instrumentation collects information using SLF4J and the Mapped Diagnostic Context (MDC). However, in contrast to the simple instrumentation currently allowed by bare SLF4J that can only log method entry and exit, X-Ray will:

- Allow collections in context information.
- Allow context to be associated with objects and user-defined modules, besides threads.
- Allow a fine grained definition of context propagation between threads, objects, and modules.
- Allow recording data in method parameters, return values, and object fields.

The project will also make use of the *java.lang.Instrumentation* interface for attaching the monitoring tool to a JVM, allowing instrumentation to be dynamically inserted and removed in a running application.

5.2. Resource agents

Resource agents for the Java platform make use of the JMX interface, which exports all relevant information from the JVM in a portable way.

For the operating system, to cope with a diversity of platforms, resource agent uses the SIGAR library (24). This library collects all relevant information in all major operating system platforms and can already expose it through JMX.

5.3. Event handling

The event spooler is implemented as a physically distributed system, with independent instances attached to different applications and communicating directly with the data stores. Since it is to be implemented as an SLF4J backend, it can leverage existing backends. In particular, Logback fully implements Mapped Diagnostic Context (MDC) and allows target data stores to be easily integrated. Moreover, it provides the ability to extensively filter data collection based on detail level, software components, and abstract markers.

The implementation of the Importer requires that text in log messages is processed to extract structured information. This can be done by describing text structure with patterns and regular expressions with a tool like libgrok (25).

The Exporter to JMX is implemented using a configuration mechanism and Dynamic MBeans, to expose information resulting from processing. Likewise, exporting to SLF4J requires only a configuration mechanism to map events to formatting templates, parameters, markers, and context.

6. Example

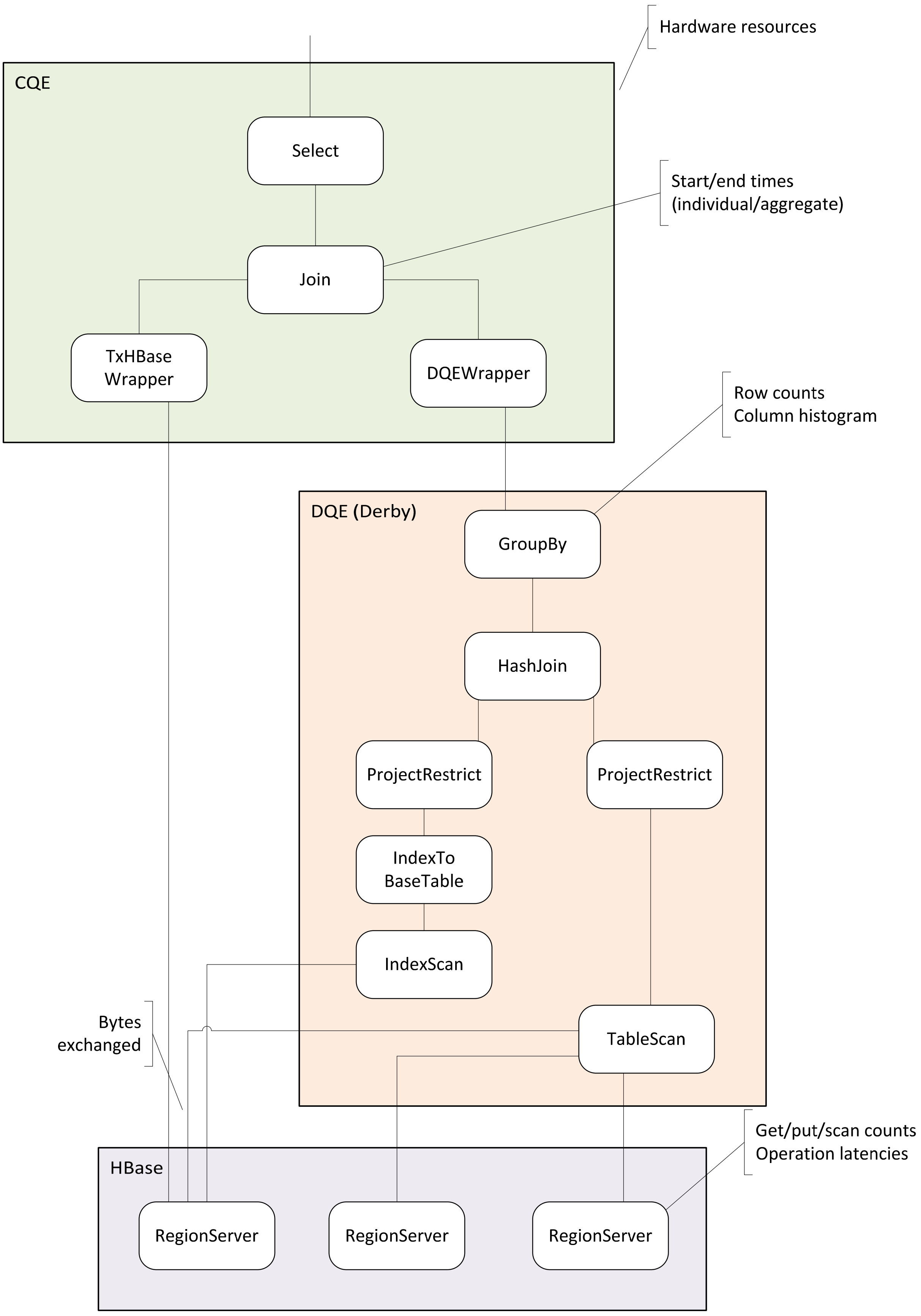

Figure 2 Example of target visualization.

As an example, consider an application that uses the Derby-based Distributed Query Engine (DQE) and native HBase through the Common Query Engine. Assume that the following instrumentation has been defined by developers of each component and enabled by the system operator:

- Logging of entry and exit to each operator in the C++ implementation of the CQE, done by manually modifying the code.
- Logging of client context and network connections in the DQE and HBase wrappers, as well as in the DQE HBase scan operators.
- Bytecode instrumentation of all operator classes in DQE, defined on their abstract interface at the root of the class hierarchy.
- Bytecode instrumentation of HBase servers, defined on the interface exposed to clients.

Figure 2 shows the what information is obtained when applying X-Ray to the execution of an CloudMdsQL query that joins the result of a SQL query, running on the Derby-based Distributed Query Engine (DQE) and a native HBase table scan. The information provided includes:

- Internal program structure of the CQE and DQE, depicting how execution units are assembled in an execution plan.
- Grouping according to larger program structure, that map to operation system processes and resource consumption on actual hardware components.
- Detailed information depending on the component being observed, such as rows for DQE operators and get/put/scan operations on HBase.
- When detailed logging of method results is possible, a histogram of actual values can be computed and displayed.
- All this information can be obtained for an individual request as well as aggregated over a period.

7. Road map

- 2014 Q2 Preliminary bytecode instrumentation.
- 2014 Q3 Capture components, including bytecode instrumentation, resource agents, and event spooler.
- 2015 Q1 Exporters and instrumentation of CoherentPaaS data-stores.
- 2015 Q3 Analysis components, including distributed request tracking.

8. References

1. Simple Logging Facade for Java (SLF4J). [Online] http://www.slf4j.org.
2. *Stethoscope: A platform for interactive visual analysis of query execution plans.* **Gawade, Mrunal e Kersten, Martin.** 2012, Vols. PVLDB 5(12): 1926-1929.
3. *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling.* **Ammons, Glenn, Ball, Thomas e Larus, James R.** 5, New York, NY, USA : ACM, may de 1997, Vol. 32, pp. 85-96.
4. *Trace Construction Using Enhanced Performance Monitoring.* **Serrano, Mauricio J.** New York, NY, USA : ACM, 2013. ConferenceProceedings of the ACM International Conference on Computing Frontiers. pp. 34:1-34:10.
5. Pin - A Dynamic Binary Instrumentation Tool.
6. *DTrace: dynamic tracing in oracle Solaris, Mac OS X, and free BSD.* **Gregg, Brendan e Mauro, Jim.** s.l. : ACM SIGSOFT Software Engineering Notes.
7. SystemTap. [Online] https://sourceware.org/systemtap/.
8. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.* **Nethercote, Nicholas e Seward, Julian.** New York, NY, USA : ACM, 2007. ConferenceProceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89-100.
9. *Flayer: Exposing Application Internals.* **Drewry, Will e Ormandy, Tavis.** Berkeley, CA, USA : USENIX Association, 2007. ConferenceProceedings of the First USENIX Workshop on Offensive Technologies. pp. 1:1-1:9.
10. *Portable and Accurate Collection of Calling-context-sensitive Bytecode Metrics for the Java Virtual Machine.* **Sarimbekov, Aibek, et al., et al.** New York, NY, USA : ACM, 2011. ConferenceProceedings of the 9th International Conference on Principles and Practice of Programming in Java. pp. 11-20.
11. *JBInsTrace: A Tracer of Java and JRE Classes at Basic-block Granularity by Dynamically Instrumenting Bytecode.* **Caserta, Pierre e Zendra, Olivier.** Amsterdam, The Netherlands, The Netherlands : Elsevier North-Holland, Inc., jan de 2014, Vol. 79, pp. 116-125.
12. *Accurate, Efficient, and Adaptive Calling Context Profiling.* **Zhuang, Xiaotong, et al., et al.** 6, New York, NY, USA : ACM, jun de 2006, Vol. 41, pp. 263-271.
13. *A Portable Sampling-based Profiler for Java Virtual Machines.* **Whaley, John.** New York, NY, USA : ACM, 2000. ConferenceProceedings of the ACM 2000 Conference on Java Grande. pp. 78-87.
14. Minha: Middleware Test Platform. [Online] http://www.minha.pt.
15. *Experimental Evaluation of Distributed Middleware with a Virtualized Java Environment.* **Carvalho, Nuno A, et al., et al.** s.l. : Proceedings of the 6th workshop on Middleware for service oriented computing, 2011.
16. **Babaoglu, Özalp e Marzullo, Keith.** Consistent global states of distributed systems: Fundamental concepts and mechanisms. 1993.
17. *Event-based Systems: Opportunities and Challenges at Exascale.* **Eisenhauer, Greg, et al., et al.** New York, NY, USA : ACM, 2009. ConferenceProceedings of the Third ACM International Conference on Distributed Event-Based Systems. pp. 2:1-2:10.
18. *NetLogger: A Toolkit for Distributed System Performance Analysis.* **Gunter, Dan, et al., et al.** Washington, DC, USA : IEEE Computer Society, 2000. ConferenceProceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. p. 267.
19. *WebLogic Event Server: A Lightweight, Modular Application Server for Event Processing.* **White, Seth, Alves, Alexandre e Rorke, David.** New York, NY, USA : ACM,

2008. ConferenceProceedings of the Second International Conference on Distributed Event-based Systems. pp. 193-200.
20. Fluentd: Open Source Log Management. [Online] http://fluentd.org.
21. *Apache Hadoop.* [Online] http://hadoop.apache.org/.
22. *Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes.* **Sang, Bo, et al., et al.** 6, Los Alamitos, CA, USA : IEEE Computer Society, 2012, Vol. 23, pp. 1159-1167.
23. Java Management Extension (JMX). [Online] Oracle. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html.
24. **Bruneton,Eric, Lenglet, Romain e Coupay,Thierry.** ASM: a code manipulation tool to implement adaptable systems.
25. SIGAR API (System Information Gatherer and Reporter). [Online] VMWare, Inc. http://www.hyperic.com/products/sigar.
26. grok. *logstash - Open Srouce Log Management.* [Online] http://logstash.net/docs/1.4.0/filters/grok.