Ph.D. Proposal:

Dynamic monitoring to find and diagnose software bugs in cloud applications

Martin Monperrus
Martin.Monperrus@univ-lille1.fr
Associate Professor
Université de Lille 1

Walter Rudametkin
Walter.Rudametkin@polytech-lille.fr
Associate Professor
Polytech'Lille

Romain Rouvoy
Romain.Rouvoy@univ-lille1.fr
Associate Professor (HDR)
Université de Lille 1

January 15, 2015

To apply to this position please contact all supervisors by email.

1 Resumé

Les applications Cloud sont composés d'une myriade de composants logiciels s'exécutant sur des systèmes physiques et virtuels répartis. Ces systèmes sont naturellement complexes à diagnostiquer et à réparer.

Cette thèse se focalise sur (i) le design d'une infrastructure logicielle pour surveiller et introspecter des applications réparties, et sur (ii) la création d'algorithmes pour détecter, diagnostiquer et réparer des bogues logiciels dans ces systèmes. L'apprentissage automatique sera utilisé pour traiter les quantités massives d'informations produites par le système de surveillance.

2 Short summary

Cloud applications are composed of a myriad of software components distributed across heterogeneous physical and virtual systems. These systems are inherently difficult to debug, diagnose and repair.

This Ph.D thesis will focus on (i) designing dynamic instrumentation techniques and developing a monitoring infrastructure to introspect distributed applications, and (ii) creating algorithms to detect, locate, diagnose and repair software bugs. Machine learning algorithms will be used to tackle the massive amounts of information generated from the monitoring systems.

3 Research team

Spirals Research Group — https://team.inria.fr/spirals/
Inria Lille - Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley - Bat. B, Park Plaza
59650 Villeneuve d'Ascq – FRANCE

4 Context

Modern software engineering approaches—such as Service Oriented Computing, Component-based Software Engineering and Cloud computing—allow us to build ever more complex software applications and to execute them

across heterogeneous physical resources. The increased level of decoupling among the components of such complex systems allows for them to be developed independently, and to be updated, patched, replaced or removed at runtime. In production, these applications are almost never stopped.

Nevertheless, distributed applications in general, and cloud applications in particular, are very complex to diagnose and to repair when problems arise. These can include software bugs, hardware failures, resource usage bugs and performance bugs. Moreover, it is often impossible to reproduce the production environment in which such issues occur because of data privacy issues, non-deterministic execution across multiple processors, the unavailability of the same resources for testing, etc. However, the popularity of distributed applications is growing due to the massive use of Cloud computing, bringing the issue of bug detection and diagnosis to a point. This raises many questions. How should developers go about detecting, diagnosing, and fixing software bugs and performance issues in their applications? What tools exist? What's missing? How do we automate this process?

5 Ph.D. Project

In the context of locating and diagnosing software, performance and resource usage bugs in cloud applications, this thesis will focus on two issues that go hand in hand: (i) creating a monitoring infrastructure that permits dynamic monitoring of cloud applications, and (ii) creating algorithms that allow locating, diagnosing and potentially repairing these bugs.

5.1 A monitoring infrastructure for cloud applications

A software monitoring system must be able to extract information from the application in order to perform diagnosis and repair. The objective of this monitoring system is to extract as much information as possible in order to properly diagnose the issue, all the while being as uninstrusive as possible to the application. Furthermore, monitoring is not always required—one way to reduce instrusiveness is to disable monitoring dynamically. Software engineering techniques allow us to create monitoring systems that are both external to the application and dynamic, that is, they can be activated and deactivated at runtime.

The list of requirements of a such a distributed monitoring infrastructure are the following:

- **Diverse instrumentation mechanisms.** The instrumentation techniques used to to insert monitoring probes depend on what information is to be extracted. A probe is a piece of code that collects specific information. Monitoring probes can be inserted at component interfaces using dynamic reconfiguration techniques, object proxies can be used for monitoring specific objects, reflection can extract fine grain information, and bytecode and source-code transformations using tools like ASM [1] and Spoon [4] can be used to introspect the entire application.

- **Fine-grain data collection.** Data collection needs vary from high-level interactions between software components to low-level values of variables and registers. This requires the monitoring system to be capable of instrumenting a wide-array of artifacts.

- **Dynamic instrumentation.** Instrumentation should be inserted in the application dynamically (at runtime). It is impossible to predict beforehand all the issues that might occur in an application; dynamic instrumentation allows probes to be added and removed while the application is still running. Multiple techniques should be devised depending on the instrumentation mechanism used.

- **Adaptive and configurable.** Monitoring levels should increase and decrease in order to refine the instrumentation probes and locate the issue at hand. Such a system should propose a Domain Specific Language or API in order to facilitate the targeting of specific issues. It should also be capable of adapting itself to the resources that it has at hand—the monitoring system should not exceed the limits on space and bandwidth available.

- **Zero overhead when monitoring is disabled.** Monitoring production applications requires minimal intrusiveness otherwise it is too costly to use. A monitoring system should ideally introduce zero overhead when disabled and only little overhead when enabled.

- **Aggregation, filtering and collaboration.** The monitoring information from all machines executing the same software application must finally be filtered and aggregated in order to identify the root causes of the bugs.

Today's complex software systems are composed of a myriad of software components running across many different physical and virtual systems. The first objective of this Ph.D. thesis is to design this futuristic monitoring infrastructure.

5.2 Detecting, interpreting and diagnosing bugs using machine learning

To build software that detects, locates and diagnoses software bugs and performance issues, we need sophisticated algorithms to parse the mountains of monitoring information and to guide the monitoring system to only inspect interesting parts of the application. Machine learning provides many useful tools for building such algorithms and for handling the large amounts of data.

These algorithms should focus on the following issues:

- **Bug detection.** High-level, low-intrusive monitoring should detect potential issues that have occurred or that will occur in the application. This is used as a starting point to begin inspecting the application more closely. The type of issue, for example a component crash or excessive memory consumption, is initially detected at this level.
- **Locating and refining the problem.** Once an issue has been identified, a process of refinement and location, through various heuristics, must be used to closely scrutinize the software components that are involved. This leads to the software component or components at cause being identified.
- **Controlling intrusiveness.** The algorithms should control the intrusiveness of the monitoring infrastructure by never exceeding the allotted bandwidth, space and CPU allotments they are given. However, they should allow for deep introspection of the buggy software components.
- **Analyzing aggregated information.** Thanks to multiple instances of the same components being executed, once a bug has been refined and the faulty components have been located, the machine learning algorithms can better determine the conditions under which the software bugs are found. This allows identifying the root cause of the issues. Furthermore, this can lead to the automatic repair of such bugs.

The second objective of this Ph.D. thesis is to design the algorithms that detect, locate and diagnose software bugs in cloud applications. The barrier to this is two fold: (i) today's software systems are composed of myriads of integrated components that result in huge amounts of heterogeneous monitoring data, making size an important issue, and (ii) the difficulty of properly identifying issues in such complex systems is akin to finding a needle in a haystack.

Machine learning's dimensionality reduction techniques on monitoring data will be used to tackle the size of the monitoring information.

6 Skills Summary

The Ph.D. candidate will develop her/his skills in Java SE (the main programming language used), Spoon and ASM for software transformations, JVM TI for Java Virtual Machine instrumentation, Python for scripting, Linux for executing and testing applications, Docker for packaging applications, Git for source-code versioning, OSGi and iPOJO for software modularity and dynamic updates, Hadoop [3] and Storm [5] for for distributed computing, among many others.

As is a common practice in the Spirals research team, all source code will be open sourced using either the GPL or the Apache License. It is expected that the student participate in related open-source communities. This should also assist in the technological transfer from academic prototype to industry-ready tools.

Experimentations to demonstrate the effectiveness of developed tools should be performed on large and complex software systems running on a cloud computing infrastructure, such as Microsoft Azure, Amazon EC2 or OpenStack. A broad range of empirical studies will test the correctness and the scalability of the solutions.

7 Background and state-of-the-art

We invite interested candidates to familiarize themselves with the following works.

Software engineering. Modern software engineering approaches enable the construction of large and complex software. Some approaches also allow the application to be adapted at runtime. We recommend studying components [18], component interfaces [9] and modern component frameworks like iPOJO [13][2], OSGi [6] and Kevoree [12]. The research challenges and approaches regarding software adaptation are well described in [17] and [11].

Dynamic software updating. There is also much work done in the area of dynamic software updating. These techniques are fine-grain and allow minute changes to be applied to an application at runtime. An interesting read is the works by Hicks [16]. These techniques can be combined with tools like [4] and [1] to insert monitoring probes into running software.

Dynamic tracing and application analysis. Application tracing and statistical analysis approaches are proposed in the Magpie project [8] and by Aguilera et al. [7]. They use information retrieved from production systems to infer statistical information to solve performance bugs. The DTrace [10] project, with is part of Solaris, FreeBSD, NetBSD and Mac OS, is likely the most well known project for dynamic instrumentation. DTrace provides thousands of probes that are activated by user-written scripts to extract, filter and aggregate applications and the operating system. Most notably, DTrace has zero-overhead when instrumentation is not activated.

Examples of instrumenting software to solve bugs are provided by Service Coroner [14], used to find stale-references (i.e. invalid pointers) that lead to memory leaks and are caused by programming erros, and Scapegoat [15], that monitors CPU, memory and I/O usage at a high-level then, when an issue is found, uses heuristics and invasive instrumentation to locate the faulty components. Other approaches analyze the logs by taking into account information obtained from source code analysis [19]. This improves the quality of the resulting diagnostic.

References

[1] `http://asm.ow2.org/`.

[2] `http://felix.apache.org/documentation/subprojects/apache-felix-ipojo.html`.

[3] `https://hadoop.apache.org/`.

[4] `http://spoon.gforge.inria.fr/`.

[5] `https://storm.apache.org/`.

[6] `http://www.osgi.org`.

[7] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[9] Antoine Beugnard, J Jezequel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[10] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[11] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer, 2009.

[12] Erwan Daubert, François Fouquet, Olivier Barais, Grégory Nain, Gerson Sunye, J-M Jezequel, J-L Pazat, and Brice Morin. A models@ runtime framework for designing and managing service-based applications. In *Software Services and Systems Research-Results and Challenges (S-Cube), 2012 Workshop on European*, pages 10–11. IEEE, 2012.

[13] Clément Escoffier, Richard S Hall, and Philippe Lalanda. ipojo: An extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481. IEEE, 2007.

[14] Kiev Gama and Didier Donsez. Service coroner: A diagnostic tool for locating osgi stale references. In *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*, pages 108–115. IEEE, 2008.

[15] Inti Gonzalez-Herrera, Johann Bourcier, Erwan Daubert, Walter Rudametkin, Olivier Barais, François Fouquet, and Jean-Marc Jézéquel. Scapegoat: an Adaptive monitoring framework for Component-based systems. In Antony Tang, editor, *Working IEEE/IFIP Conference on Software Architecture, WICSA 2014*, Syndey, Australie, 2014. IEEE/IFIP.

[16] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, 2005.

[17] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[18] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[19] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 143–154. ACM, 2010.