Tracking Behavioral Changes in Distributed Systems using Distalyzer[*]

Karthik Nagaraj[†], Charles Killian, Jennifer Neville
Purdue University
{*knagara, ckillian, neville*}@*cs.purdue.edu*

Distributed systems executions are complex and hard to understand. Understanding the behavior of distributed systems is useful in finding avenues for improvements, performance benefits, identification of bugs and unexpected operations. Large distributed systems are hard to study because of their complex and numerous interaction possibilities, stemming from variable network delays and node processing delays that are characteristic to heterogeneous environments. Currently, there is no technique for analyzing these executions to identify problems that affect global system metrics, thus severely hindering the maintainability of these systems. We present this work on Distalyzer which automatically analyzes large distributed systems at their repositories, and tracks updates to the software, together with monitoring of global system metrics. This will hugely help developers in understanding impacts of bug fixes, feature changes, trend of system growth and so on, thus assisting in development and maintenance.

Large projects and code-bases have many simultaneous contributors which keep the features (and behavior) changing at a rapid pace. For example, the HBase project (an open-source port of BigTable) has over 3000 commits over the past 4 years, averaging at over two commits each day. Similarly, other Hadoop projects and some of the other open source distributed systems (*e.g.* BitTorrent, file systems, etc.) develop very fast. The state-of-the-art in project management are unit tests and regression tests written by the developer to verify correctness and performance respectively of *specific* modules. In other words, there is no available mechanism to test the global effects of code changes on the system behavior. Unit and regression tests are unfortunately ineffective in capturing the complex network interactions of distributed systems, and hence incomplete in analyzing the system. Also, *problems* are not always self-detected by these systems and flagged as error or warning log messages. *e.g.* Performance degradation.

By monitoring the code-bases of these systems over time, system designers could analyze the long-term goals, benefits and deficiencies. A natural place for such an analysis is the code repository. System properties could change over a short-term (a few commits) or over a longer term such as a year, thus even keenest designers could miss it on manual inspection. For instance, suppose a code commit is meant to affect a single component of the system, this continuous analysis can provide the *actual* impact of that change. If the change mistakenly affected other components, it can be caught at the level of the repository itself. Another challenge with repository maintenance is monitoring the trend of system behavior over long periods of time such as a year. For example, small increments in slowdown or resource usage of the distributed system can be trapped as an undesirable effect, and analyzed for the cause.

Some of the challenges include an automated test infrastructure coupled with code management, extracting logs that describe the runtime behavior in a meaningful way, picking revisions to compare, automatically comparing these revisions of the distributed system, and finally bridging these with developer notions of the project. We aim to leverage the bulk of logging and tests that have already been put into the code base by the developers, to be easily applicable to real projects. In this regard, we target our techniques and analysis to extracting useful structure from large volumes of log data gathered from systematic tests. The state-of-the-art in the automated analysis of distributed systems consists of identifying root causes of failures [2], profiling and comparing request flows [1,3] and finding anomalous local execution patterns [4]. However, this work builds a framework for comparing distributed system execution logs automatically through the use of machine learning techniques, to bring out the most *misbehaving* components that affect global system metrics.

[*]This research was supported in part by the National Science Foundation (NSF) grant #1054567-CNS.

[†]PhD student, Presenter

The first challenge is the automatic collection of log data from the distributed system code base, which can be achieved through periodic experiments setup on a testbed. A set of experimental tests are run against a particular version of the system, and the resulting logs collected in a revision annotated log store. Some of the smaller challenges include test definitions, version compatability, test changes over time, etc. An important distinction between Distalyzer and other previous work is that it focuses on identifying non-traditional bugs (referred here loosely as *problems*). These issues do not commonly affect the correctness of code or failure of unit tests, but however address global metrics of the system. This includes performance, resource usage, CPU usage, system-specific metrics, etc. all of which are significant to these systems.

There are many logging libraries that allow for formatting logs in the system – *log4j*, *printf*, Python logging, etc. However, these formats are still highly variable and many developers choose to use free-form text as the actual content. On the other hand, some logs are completely structured with no loss of information from the program form, but they are still a minority. We work to compromise the situation between fully meaningful logs and free-text logs, and find a middle-ground that is easy to adopt and amenable to automated analysis. Our intuition is that logs provide one of two purposes as *state* and *event* logs, which represent snapshots of the system state and times of occurrences of events respectively.

We've identified that comparing two executions of a distributed system to be a crucial component powering different types of analysis. As described earlier, revisions from before and after a code commit can help understand the actual *impact* of the change, against the expectations of the committing developer. Similarly the divergence between the trunk and release branches of a repository can be studied to analyze an eventual merge. An important problem in this aspect is the ability to pick two *interesting* revisions to comparatively analyze, from the large number of revision possibilities. Once the logs from the two revisions have been classified into *event* and *state* logs, the statistical approaches take over to find interesting differences between the executions. Underlying experimental factors such as randomness and scheduling prevent executions from being identical in most cases, motivating the need for a more general interpretation. Comparisons can be deemed significant if there are significant divergences in component behavior between the two sets, further enhanced by those components that indicate deviations from expected behavior. To reduce false positives, thresholds can be used to flag only the most important behavioral changes.

Phase One Results In the first phase of this research, we targeted an important and hard challenge in distributed systems – performance debugging. We started with a generic case of applying Distalyzer to logs from different implementations of the same protocol and different requests from the same system, rather than revisions. Given two sets of logs, our techniques identify the root cause of the performance divergence using statistical t-tests and Dependency Networks, and present it visually to the developer. Our analysis on two different mature distributed systems show its generality, applicability and feasibility in comparing executions.

In one case, we took two implementations of BitTorrent – Transmission and Azureus and setup experiments to judge their download efficiency. We found Transmission to be 45% slower than Azureus without any apparent indication in the logs. Using Distalyzer, we identified a bug and a configuration problem with Transmission that accounted for all of the performance difference between the systems. In the second case, we found that read request latencies in HBase under a standard workload exhibited a heavy tail distribution, with the highest latencies about 3 orders of magnitude larger than the average. By classifying the requests into slow and fast buckets for comparison, we were able to identify a bug and a disk scheduling policy issue that influenced a large portion of the slow requests. These problems were neither identifiable as anomalous executions, nor recorded as errors in the logs making them hard to catch.

In conclusion, we motivate the need for an automated framework for studying the behavioral trend of distributed systems. Such a tool would be useful to developers looking for performance leaks, study effects of a code patch, improving manageability of a large code base. By not attempting to perform semantic analysis on raw system logs, we aim to avoid uncertainties and errors in understanding logs, and at the same time remain agnostic to the implementation language. With the availability of a baseline execution, it is easier for designers to understand changes as divergences in component behavior, and find non-standard problems.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *SOSP*, 2004.

[3] R.R. Sambasivan, A.X. Zheng, M. De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and G.R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[4] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.