BlackSheep: Inferring white-box application behavior using black-box techniques

Jiaqi Tan
Computer Science Senior Research Thesis
Extended Abstract

March 21, 2008

Abstract

We describe and evaluate a new technique for diagnosing performance problems in distributed systems in a scalable manner by exploiting and analyzing only local (i.e. intra-node) black-box system metrics, and inferring white-box application behavior. We study the novel method of correlating white-box application event logs with black-box system metrics to gain insight into the behavior of a distributed system, and validate our approach through experiments on the Hadoop open-source implementation of Google's Map/Reduce distributed programming model. We inject failures and real performance problems gathered from failure data recorded in Hadoop's bug database.

1 Introduction

Finding the location and root cause of a failure in a distributed system is an inherently difficult problemexecution paths span multiple machines and can be arbitrarily complex, so that a fault (a possibly latent defect in the system) may manifest itself as an error (an externally visible incorrect state of the system, such as erroneous user output) many execution modules down the execution path, before the error even manifests itself as a failure (an incident in which the system fails to deliver service). Tracing a system failure to its initial manifestation as an error requires either a characterization of externally observable correct system states, so that system states falling out of this set are indirectly detected and marked as erroneous, or a characterization of erroneous states, for direct detection. Tracing a system error to its fault, on the other hand, requires detecting when software behavior deviates from what the programmer intended it to dothis requires knowledge of the intended semantics of the program, which is not present in and is outside the realm of consciousness of the program.

There are two broad classes of techniques for analyzing systems and software. Black-box techniques treat the software system as an enclosed, unobservable entity that cannot be modified; we classify information sources that do not reveal the execution path inside software components as black-box, and techniques that do not require source code nor machine code modification as black-box techniques. White-box techniques provide views into the internals and execution path of the software system; we classify information sources that provide knowledge of the original source code or execution path structure of the software, such as knowledge of the order of function calls, as white-box, and we classify techniques that require some form of source code modification as white-box techniques. While white-box techniques to gather white-box information are much wealthier sources of information than black-box sources, there is an inherent trade-off between the richness of information that can be extracted from software, and the cost of gathering that information in terms of runtime overheads and ease of deployment. Black-box techniques are easy to use at existing software installations and typically involve setting up external software monitors that record general system state, but provide limited information; white-box techniques may involve significant initial programmer effort to insert source code such as assertions (which are only as good as the correctness of the assertions, creating a dual problem), and providing a fine granularity of information about control flow may have involve high overheads as large numbers of probe-and-record instructions will be needed.

It immediately appears that white-box techniques are necessary to trace a software error to the fault that is its root-cause, because a fault arises out of a deviation of software behavior from programmer intention, and programmer intentions are reflected in the execution path at the granularity of control flow through functions. Current techniques have danced this tightrope of the inherent tension between overhead and information to try to find a good leverage on the smallest possible information source, while providing diagnostic value on this information.

The difficulty of finding the location and root cause of a failure in distributed systems is further complicated by the fact that execution can take place on arbitrarily many systems, leading to a possible explosion in the volume of trace data gathered. Again, there is a trade-off between gleaning more information by combining trace data across systems to obtain a system-wide view, and incurring higher bandwidth and processing costs of transmitting large amounts of data across a network and processing it.

Major black-box techniques have included Pinpoint, which instrumented the J2EE middleware platform to trace message flows between software components, to associate particular groups of components with erroneous transactions, and to find anomalous control flow paths [2]. Cohen et. al.'s work has focused on using clustering on black-box system metrics, and building informative summaries of metrics to reduce the amount of information that must be exchanged among the nodes of a distributed system to minimize bandwidth use [3], but can only detect the location but not root-cause of anomalous behavior. Magpie correlated resource usage information operating system-provided resource accounting facilities and output from application event logs to build causal paths of applications on a single node using clustering (that is extensible to multiple nodes, albeit at possibly high cost) [1], but such accounting is at a syscall level which is expensive to instrument on Unix-based systems.

Major white-box techniques have included Pip, which relied on programmer-written expectations of correct behavior, and recorded alarms of anomalous behavior raised from within the software itself [5], but Pip is only as good as the programmer-written expectations. Also, Triage works on single-node (non-distributed) software to uncover the faulty source code behavior or system environment feature which caused a crash by using a re-execution framework combined with a trial-and-error automation of the intuitive human troubleshooting process [6], but this method is an after-the-fact technique that relies on the system being down to allow such root-cause discovery (rather than diagnosis).

Also, distributed systems, such as Hadoop and other Map/Reduce-type distributed parallel processing systems [7] [4], are designed for batch processing of large datasets. These distributed systems see much fewer user-initiated requests, so that there are much fewer runs on which systems such as Cohen's work, Magpie, and Pinpoint can perform clustering for learning the correct behavior of the system. Cohen et al.'s work, Magpie, Pinpoint, and Pip all assume the availability of large numbers of short-lived user-initiated requests, so that each of these requests can be used as a sample for clustering for determining which requests are anomalous. This model is well-suited to the vast majority of traditional multi-tier web-based applications, with common tiers being a web-server front-end, an application server tier, and a database back-end. Also, Hadoop has uninteresting execution paths through its components, with only one type of execution component (the TaskTracker), such that path-based techniques such as Pinpoint's Probabilistic Context-Free Grammars and Pip will not be able to get leverage from analyzing paths of execution flows.

Current techniques which allow for root-cause analysis, such as Pip and Triage, require too much programmer input, which precludes the discovery of bugs that programmers are unaware of, and do not allow for runtime prognostics to be made for detecting errors before they have resulted in failures. Also, both Pip and Triage require access to program source code, which is not a given, especially in commercial production sites. Even black-box techniques such as Pinpoint are not necessarily suited to production sites, because Pinpoint requires a modified middleware, which production sites may not allow due to various concerns such as security, while techniques such as Cohen et al.'s work do not allow for root-cause analysis although it is amenable to deployment at production sites.

The goal of this work is to develop techniques for problem diagnosis on software deployed in production environments. Production environments typically deploy commercial or otherwise third-party software packages for which source code is often not available, and production environments typically have strict limits on availability and quality of serviceproduction systems strive to achieve maximum throughput and minimum latencies on servicing requests at a minimum cost. Also, production environments will typically prohibit modifying even program binaries for security and privacy concerns. Hence, our techniques have minimal overhead, and do not require access to program source nor modifications to program binaries.

A failure is an instance of service unavailability in a software system, while an error is an incorrect state of the system, and the root-cause of the failure is the fault in the software its environment, which manifested as an error that led to the failure. In general, white-box information is necessary to diagnose the root-cause of a failure, because the notion of correctness, or the semantics of program behavior, is needed to detect incorrect behavior. However, white-box techniques for extracting white-box information are not amenable to production systems as they violate our requirement that our technique not require access to program source nor modifications to binaries. We propose a new technique for identifying the location and inferring the root-cause of a failure in an online fashion on a distributed system that

is amenable to use on production systems. We achieve this by using *a priori* knowledge of the deployed software to build inference models that allow for white-box information about the phase of execution of software to be inferred from black-box information. In addition, our technique requires only intra-node information within a given node, so that this technique is immediately scalable to distributed systems containing arbitrarily many nodes.

We demonstrate the efficacy of our root-cause diagnosis technique on Hadoop, the open-source implementation of the Map/Reduce distributed parallel programming runtime environment and distributed filesystem, and further demonstrate the applicability of our technique where current techniques are not immediately applicable, on Hadoop.

2 Approach

We describe the general framework of our approach and algorithm for the online diagnosis of root-causes of failures in distributed systems. Prior to the deployment of our algorithm, we conduct a simple study of the externally-observable behavior of the target distributed system to build models of externally-observable system behaviorsystem signatures. Specifically, we consider system behavior as represented by correlations of various system metrics, as described in Section 2.1. Signatures of both correct and incorrect system behavior are built, and signatures of incorrect system behavior, or problem signatures, are augmented with causal information to allow causal inferences to be made during diagnosis. Then, erroneous behavior is detected as instances in which the signature of observed system behavior deviates from signatures of correct system behavior. The erroneous signatures are then matched against the store of problem signatures, from which causal information can be extracted to allow root-cause diagnosis.

2.1 Correlated metrics

We consider black-box metrics that describe (i) aggregate activity of system-level components, such as the disk, virtual memory, and networking subsystems, and aggregate CPU utilization in user-mode and kernel-mode, and (ii) per-process CPU utilization in user-mode and kernel-mode, and per-process memory utilization. We consider each of the CPU, physical memory, virtual memory (paging), disk, and network, as separate subsystems of each node in the distributed system, and consider, each of these subsystems as resources that can be used. We also take into account, where possible for us to record suitable metrics (specifically CPU and memory utilizations), the per-process utilization of these resources as well. Then, from the correlations among groups of metrics, we can identify, on a node, the particular subsystems that are in use at given points in time; where per-process utilizations are available, we can also infer the subsystems of the node that processes are engaging based on correlations between metrics for the process and metrics for the subsystem. Further, with white-box information about the phase of execution that the software of the node is in, we can characterize the workload of the software based on a priori knowledge of the application, in terms of the resource usage patterns of the particular execution phase. For instance, when there is high correlation between context switches and application CPU utilization, we can infer that the application is spawning new worker processes to handle incoming job requests.

2.2 Constructing Signatures

Prior to the deployment of our problem diagnosis algorithm, a pre-deployment learning phase is conducted to construct signatures of system behavior by an application expert who is conversant with the workings of the system, such as a system administrator. First, signatures of correct system behavior are built based on *a priori* expert knowledge of the high-level phases of execution of the application. The phases in the execution life-cycle of the application are characterized by their workload characteristics. Then, a small number of runs of the target distributed system are executed, and black-box metrics are collected from these runsonly metrics observed in observably correct runs (i.e. no errors nor failure manifestations were observed) are considered. Next, the traces of these black-box metrics are annotated with the known phases in the execution life-cycle of the application, with periods of the time-series traces classified as belonging to one of the phases of execution of the application. The corresponding correlations of the black-box metrics then form signatures for each of the phases of execution of the application. Next, signatures of incorrect system behavior are built for known failure cases. For such known failure cases, we assume that realistic fault injection methods are available such that we are able to reproduce the exact manifestation of the failures in the distributed system (we demonstrate a few representative cases in Section **??**). Then, runs of the application with the faults injected are executed, and black-box metrics are collected from these runs. Given that the faults are known incidents, we will be able to ascribe to the correlations causal information. The representation of these is described in Section 2.3

2.3 From correlation to causality: Inference Graphs to represent *a priori* models

Signatures of correct and incorrect system behavior are represented using Inference Graphs, which represent the relationships between groups of metrics. Signatures of correct behavior do not contain any causal information,

and describe the correlations between metrics. Each metric or group of related metrics is represented by a vertex in the graph, and the edge weights of the graph represent the learned correlations between the metrics (or groups of metrics) in problem-free runs during the pre-deployment phase. The Inference Graphs of signatures of correct behavior are undirected graphs. Signatures of incorrect behavior encode causal information by specifying the direction of causation as edge directions in the Inference Graph. Based on his domain knowledge of the behavior of the system, and of the fault that was injected in the pre-deployment phase, the application expert can encode his knowledge about how the fault propagates in its manifestation across the various metrics. For instance, a deadlock could begin as a fall in CPU utilization, that causes a drop in disk activity in a write-intensive phase of execution, so that the direction of causation here is from CPU utilization to the volume of disk writes.

2.4 Online classification

The runtime phase of our algorithm involves continuously recording black-box metrics on each node, and building undirected Inference Graphs based on the correlations between observed metrics. This serves as a signature of the current state of the system. The current signature is compared against signatures of correct system behaviorif a match is achieved, then the system is determined to be behaving correctly and the diagnosis process stops. If the current signature fails to match any of the signatures of correct behavior, then the current signature is matched against the problem signaturesif a match is found, then the causal information encoded in the directed graph of the matched problem signature provides root-cause attribution to the offending subsystem of the system. If a match is not found, then the algorithm is only able to isolate the failure to the node, but is not able to perform root-cause analysis.

2.5 Augmenting signatures

Current signatures that match neither signatures of correct nor incorrect system behavior are then stored. The unmatched signatures can then be collected for off-site analysis by the application expert. This would involve first clustering unknown signatures, and computing the similarity of each cluster to existing signatures to provide candidate guesses to the application expert. Then, on determining that the given unknown signature is a problem signature, the application expert can proceed to add causality information to the signature. The signatures used for online diagnosis can thus be periodically updated and augmented by the application expert.

References

[1] P. Barham, A. Donnelly, R. Isaacs, R. Mortier. *Using Magpie for request extraction and workload modelling.* Proc. 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.

[2] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer. *Pinpoint: Problem Determination in Large, Dynamic Internet Services.* Proc. International Conference on Dependable Systems and Networks, Bethesda, MD, 2002.

[3] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox. *Capturing, Indexing, Clustering, and Retrieving System History.* Proc. 2003 Symposium on Operating Systems Principles, New York, NY.

[4] J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Proc. 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.

[5] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, A. Vahdat. *Pip: Detecting the Unexpected in Distributed Systems.* Proc. 3rd Symposium on Networked Systems Design & Implementation, San Jose, CA, 2006.

[6] J. Tucek, S. Lu, C. Huang, S. Xanthos, Y. Zhou. *Triage: diagnosing production run failures at the user's site.* Proc. 21st Symposium onOperating Systems Principles, Stevenson, WA, 2007.

[7] Hadoop. http://lucene.apache.org/hadoop, 2007.