CMU-PT/RNQ/0015/2009

Trustworthy and Resilient Operations in a Network Environment

Deliverable D3

First Specification of the Diagnosis Algorithm

Information and Communication Technologies Institute
Carnegie Mellon | PORTUGAL
AN INTERNATIONAL PARTNERSHIP

Executed by: **Electrical and Computer Engineering**

Direction/Department: **CMU**

Date: **02-11-2011**

Version Number	Owner	Change Control	Date
0.1	CMU	Document creation with the first draft specification of the diagnosis algorithm	27-10-2011
0.2	CMU	First review with content fixes and improvements	02-11-2011

Additional information:

Author/s:	...
Beneficiaries contributing on the deliverable:	CMU
WP contributing to the deliverable:	WPx
Estimation of pm spent on the deliverable:	X
Nature of the deliverable:	Report
Total number of pages:	23

— Document generated on 2 November 2011 at 10:04 —

Contents

List of Figures

Executive Summary

This project aims to develop a fault intrusion tolerant framework for monitoring cloud infrastructures in a trustworthy way. It targets to fullfil typical datacenter scenarios as those of Portugal Telecom. The framework will improve resiliency and guarantee the needed trustworthiness in a specific environment identified by PT.

We present the First Specification of the Failure Diagnosis algorithm henceforth referred to as *RAMS* for the deliverable D3 proposed in the TRONE project.

Detecting failures in large-scale distributed systems is challenging, as modern datacenters run a variety of applications and systems. Current techniques for detecting failures often require training to detect failures, have limited scalability, or are not intuitive to sysadmins. We present *RAMS*, a lightweight and scalable algorithm for distributed systems which detects failures using only correlations of operating system metrics collected transparently. The detection algorithm is based on our hypothesis of server application behavior, and hence does not require training, and can perform detection with complexity linear in the number of nodes, with results that are intuitively interpretable by sysadmins. Further, with some training, *RAMS* can identify the category of a problem that has previously been seen and determine the root cause of the problem. As part of our work for year 2 of the project, we plan to show that *RAMS* is versatile, and can diagnose several faults in i) multi-tier web request systems that often run in datacenters, ii) in Hadoop MapReduce systems that are often used to parallelize search, data mining and machine learning algorithms, and further show how *RAMS* is intuitive to sysadmins.

The rest of this deliverable is structured as follows. We briefly introduce the need for failure diagnosis techniques in Section 1. Then, we present a brief survey of related work on failure diagnosis techniques. The goals of our failure diagnosis algorithm are covered in Section 2 including the problems that we plan to target and also the assumptions that the diagnosis technique relies upon to work correctly. A detailed description of the diagnosis algorithm is presented in Section 4 along with hypothesis on which the failure diagnosis algorithm is based upon and some empirical evidence in support of our hypothesis. Finally, we discuss some of our future steps in Section 6.

1 Introduction

Businesses often rely on large-scale cloud computing systems to support Internet and telecommunication services such as e-Commerce, VoIP and business analytics. To satisfy the high availability requirements of these systems, there are real-time operations teams that diagnose problems by monitoring both low-level alarms derived from the equipment, *e.g*, server and network errors, as well as end-to-end indicators such as customer complaints. These operations teams work to ensure that major outages or blackouts are rare, and that they are resolved quickly when they occur. On the other hand, performance degradations or request failures affecting a subset of end users—occur more frequently and are much more elusive to diagnose. For example, Thereska et al [1] describe a TCP buffer overflow at a switch which resulted in performance degradation when striping data over more than 5 servers that was difficult to diagnose manually. Sambasivan et al [2] describe a bug in their distributed storage system in which the hash table responsible for storing mappings from filehandles to object names on disk stored mappings only in the lower bucket resulting in increased query times. Thus there is a significant need and interest in developing techniques that can detect failures and performance degradation problems rapidly.

Detecting failures and performance problems (hereby collectively referred to as "failures") in large scale distributed systems is highly challenging. As distributed systems grow, the data to be analyzed grows as well. In addition, interactions between components become more complex. Distributed systems have been growing to meet larger processing demands, e.g. due to high volumes of web requests in multi-tier Internet services, and large datasets in data-intensive systems e.g. MapReduce [3]. This growth has been facilitated by pay-as-you-use compute facilities e.g. Portugal Telecom's cloud computing service and Amazon's Elastic Compute Cloud (EC2). This growth has created new challenges in quickly detecting failures in large systems. Algorithms for failure detection need to efficiently handle larger and more complex datasets from larger systems, and from different types of systems.

Failure detection and diagnosis in distributed systems has been the subject of much recent research. Some techniques extract/infer application-specific information e.g. requests paths, to detect anomalies [4, 5, 6, 7, 8, 9], but this may be computationally expensive, and the instrumentation is often invasive and infeasible on production systems. Others infer failures using application-agnostic system metrics [10, 11, 12, 13], but they reason about metrics from all nodes in the system at once, which may not scale to large systems.

While many of the above-mentioned techniques may be effective at isolating failures, many are too expensive to be "always-on" due to invasive instrumentation and high overheads.

Instead, we target lightweight techniques to quickly detect if there is a failure in the system, before heavyweight techniques are used. We propose a new method, *RAMS*, for failure detection on distributed software systems, that is lightweight, scalable and versatile to be applicable to multiple types of systems. *RAMS* consists of two algorithms: (1) *RAMS-FD* is a lightweight failure detection algorithm which provides first-pass alarms to notify sysadmins of failures, allowing them to use more sophisticated techniques which are too expensive to be used in an "always-on" fashion. *RAMS-FD* is based on our *RAMS* **hypothesized model (§ 5.1) of normal, fault-free system behavior of server applications**. *RAMS-FD* is lightweight: it has low instrumentation overhead ($< 1\%$), using only few (≈ 10) widely-available OS performance counters. It is also scalable: (i) it uses simple computations which are linear in the size of the input data, and (ii) it can detect failures in a distributed system

We plan to evaluate the *RAMS* algorithms on two major classes of distributed server applications: (i) Apache/J2EE/MySQL-based RuBiS auction benchmark, which is a multi-tier web request processing application, and (ii) on the Hadoop [14] MapReduce [3] parallel distributed data-intensive processing system.

2 Related Work

Machine learning has been commonly used for failure diagnosis in distributed systems [4, 5, 7, 10, 11, 12, 13]. Given knowledge of failed requests (e.g. those with Service Level Objectives, or SLOs, violated), supervised learning techniques localize failures to their originating node or metric [4, 5, 10] based on models learned from training data. *RAMS-FD*'s detection does not require any training as it is based on the *RAMS* hypothesis, although the *RAMS-DT* diagnosis requires training. Another class of techniques extracts request paths in the system for diagnosis [6, 7, 8, 9]. *RAMS* uses OS-level metrics directly for diagnosis.

Another class of techniques solves the more fundamental problem of generating alarms to notify users of a failure, when knowledge of request failures is unavailable [11, 15, 16]. This is useful for systems with long running jobs or novel user-programmable workloads, e.g. MapReduce, or in the case of partial failures, when the problem is a nascent one which has not escalated into an SLO violation. *RAMS* falls in this class of techniques, and solves both instances of the problem–for both Hadoop and for multi-tier web request systems. Like *RAMS*, [11] uses OS-level metrics. They focus on macro datacenter state, whereas *RAMS* diagnoses problems on individual nodes. *RAMS-FD* is also more scalable than [11] as a failure detector as it requires only local state for each node.

Similarly, [15] uses metrics from all nodes in a system, while *RAMS-FD* uses only node-local metrics for diagnosis. Similarly to *RAMS*, [16] uses correlations, but they correlate behavior across nodes, incurring $O(n^2)$ complexity, while *RAMS* correlates behavior between metrics on the same node, which is more scalable. [17] focused on a method to cheaply collect system metrics to enable reconstruction of past incidents for investigation. Their correlation-like "rules" for inferring system problems, were not generalized, unlike the *RAMS* hypothesis.

Some diagnostic approaches have used regression to automatically discover correlations between metric pairs [18, 19]. However, these approaches do not scale well to large numbers of nodes and metrics as they search for metric correlations both locally, and remotely between nodes. *RAMS-FD* exploits semantic knowledge to analyze a small number of metrics, providing scalable diagnosis for large-scale systems. Cherkasova et al [20] and Stewart et al [21] exploited queuing theory and regression to model the relationship between resource-usage and transaction response times in Internet applications. This allowed them to distinguish between workload changes and anomalies for transaction types in their training sets. *RAMS-FD* relies solely on OS metrics to detect problems allowing it to be easily ported across systems. While [22], like *RAMS-DT* uses decision-trees to diagnose failures, *RAMS-DT* is a second-order algorithm applied to detection alarms, while [22] was applied directly to

application logs. *RAMS-DT* applies our earlier work, BliMeE, on using decision-trees on detection alarms [23] to *RAMS-FD* detection alarms. Some techniques have diagnosed failures on Hadoop using white-box information from Hadoop's natively generated logs [24, 25, 26], rather than the black-box OS-level metrics that *RAMS* uses. [12, 13] used OS metrics to diagnose failures in Hadoop, albeit using a peer-comparison approach with $O(n^2)$ complexity. *RAMS* is more scalable, and has wider applicability to multi-tier web request processing systems as well.

3 Problem Statement

3.1 Goals and Non-goals

The *RAMS* algorithms aim to perform failure diagnosis on server applications in distributed systems that is:

Lightweight, Transparent: We aim to use metrics that can be collected with minimum overhead and transparently, i.e. without modifying target applications. This would allow the metrics to be continuously collected without impacting system performance, and enable our algorithms to be used in production settings, where sysadmins are unlikely to allow invasive or high-overhead instrumentation.

Scalable: Our algorithms must have low computational complexity to enable them to scale up to diagnose large distributed systems. We aim for *RAMS-FD* to require only information from a single node to diagnose that node, so that diagnosis can be distributed across the system. We also aim for *RAMS-DT* to efficiently classify failures, with low computational complexity.

Versatile: We aim to detect failures in multiple types of systems. This would enable our algorithm to be used in large, complex modern datacenters running multiple types of applications.

Effective Diagnosis: Our algorithm aims to be effective in detecting failures in our target systems–we aim to minimize false-positives while detecting as many failures as possible. We measure the performance of *RAMS-FD* using the F_1 score [10], and we aim to maximize F_1 score. We also aim for *RAMS-DT* to accurately classify whether a node is free from fault, or if it is faulty, to classify the closest previously observed fault it is suffering from.

Non-goals: We do not present an online implementation of the *RAMS* algorithms. Instead, we focus on presenting and evaluating the diagnosis of our algorithms. As there is only one master node in a Hadoop cluster, we currently focus on failures on the potentially many of slave nodes, and defer master node failures to future work.

3.2 Fault Model

RAMS targets performance problems: faults that result in a slowdown, causing the processing of tasks in the system to take a longer time than without the fault, causing increased runtime and reduced system throughput. Performance problems can also be considered as *partial failures*, since they do not result in a system crash. We do not consider crashes as

these can be detected using simpler methods e.g. heartbeats. Nonetheless, we envision that *RAMS* would be able to detect crashes before the underlying fault has resulted in an outright crash. We do not target value faults from executions terminating with incorrect results.

While *RAMS* targets performance problems it can also be used to diagnose some of the security related problems as well. Several security problems, such as Denial of Service (DoS) attacks, can manifest as performance problems and can, therefore, be diagnosed using *RAMS*. One of the important issues that this project will try address in Years 2 and 3 will be the effectiveness of *RAMS* in correctly attributing the root cause of the performance degradation to a security attack and the kind (DoS etc.) of the security attack.

3.3 Assumptions

We assume that the target application under diagnosis is the dominant source of activity in the operating system on each node, as we use system-level OS metrics *(This assumption can be discharged by tracking per-process performance counters, but this incurs slightly higher overheads to collect)*. This is likely to be the case in virtualized environments where each VM hosts a single service.

4 Target Systems

Data-Intensive Processing MapReduce [3] is a framework that enables distributed, data-intensive, parallel applications by enabling a job described as a Map and a Reduce function to be decomposed into multiple copies of Map and Reduce tasks and a massive data-set into smaller partitions, so that each task processes a different partition in parallel. Hadoop has a master-slave architecture, with a single master and multiple slave hosts, as shown in Figure 1. Hadoop consists of an execution layer which executes Maps and Reduces, and the Hadoop Distributed Filesystem (HDFS), an implementation of the Google FileSystem [27]. The master host runs the JobTracker daemon, which schedules task execution on slaves and implements fault-tolerance using heartbeats sent to slaves, and the NameNode daemon, which provides the namespace for HDFS. Each slave host runs the TaskTracker daemon, which executes Maps and Reduces locally, and the DataNode daemon, which stores and serves data blocks for HDFS. Each Hadoop daemon is a Java process, and natively generates logs which records error messages, as do typical logs, as well as system execution events, such as the starts and ends of Maps and Reduces. We currently only target detection of faults on the slave nodes.

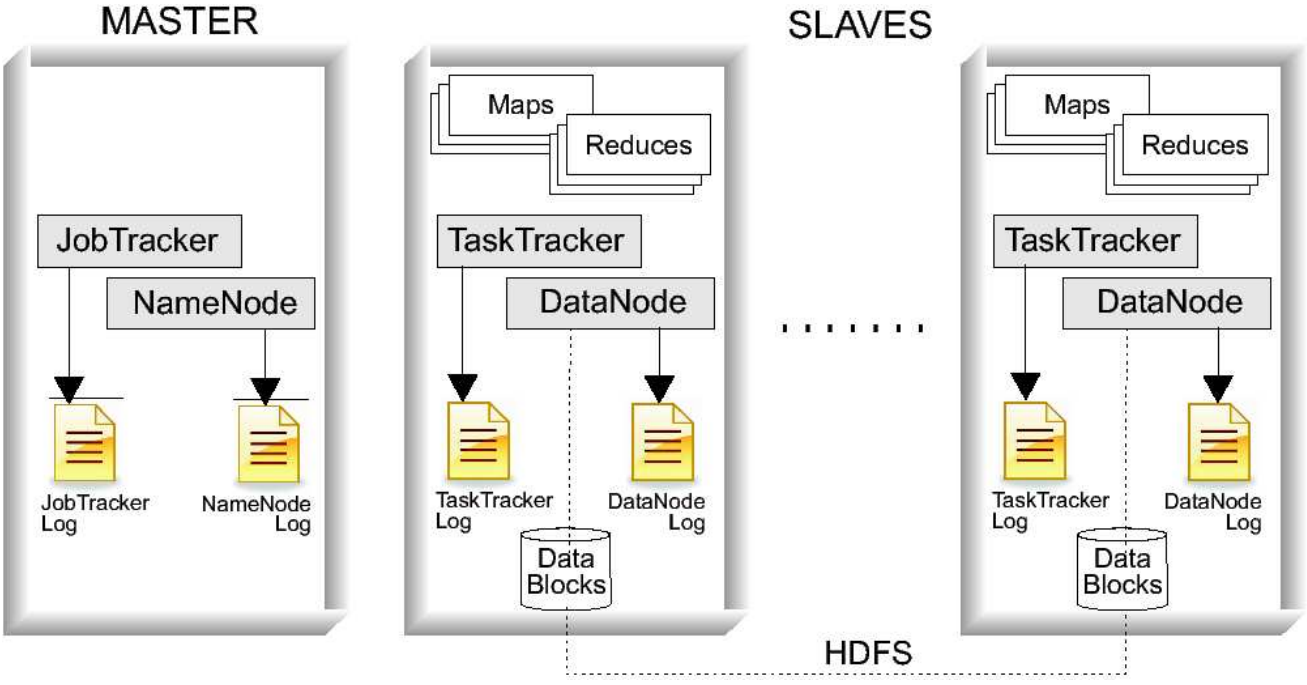


Figure 1: Architecture of Hadoop.

Multi-Tier Web Request Processing RuBiS [28] is an auction website benchmark modeled after the popular eBay auction website. RuBiS has a three-tier architecture, consisting of a web-server, an application server (using Java Servlets and Enterprise JavaBeans, or EJB) and a database server. In our setup, we used the Apache HTTPD web server, the JBoss J2EE (Java 2 Enterprise Edition) application server for handling the business logic of the application, and the MySQL relational database. Our setup consisted of 1 web server, 3 JBoss servers in round-robin load-balanced mode, and a single MySQL database server.

5 Approach and Implementation

5.1 The *RAMS* Hypothesis

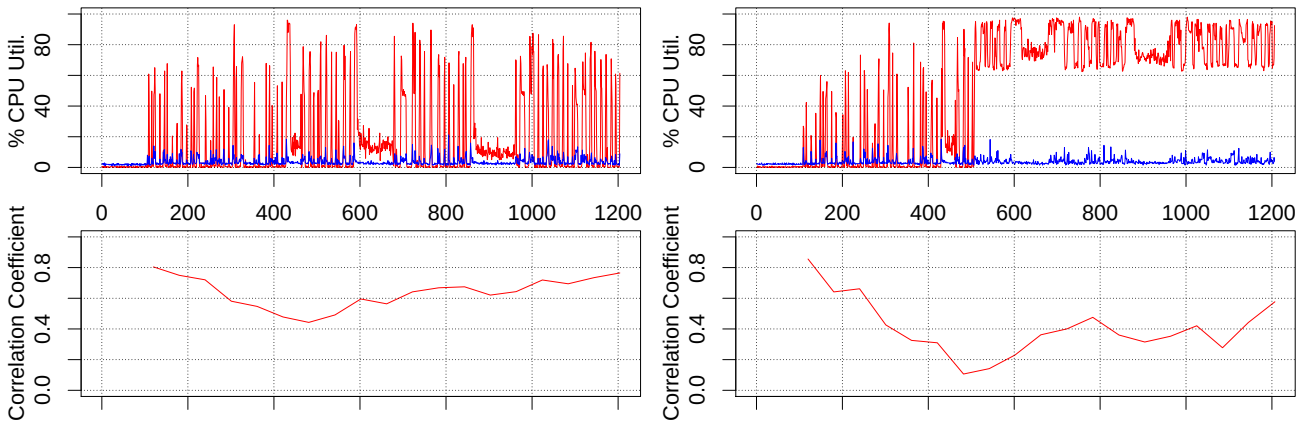
The *RAMS* hypothesis proposes that at a conceptual level, normal, fault-free processing to service a user request in a server application comprises, and alternates between, two phases: (i) a **communications** phase, when the application receives instructions from the user via the network (potentially indirectly, e.g. database receiving instructions from the web server in multi-tier web request processing, or slave nodes receiving instructions from a MapReduce master node), reads data from disk, or passes results back to the user via the network, or to disk, and (ii) a **compute** phase, when the application performs some operation based on the received inputs and instructions. At an operational level, the **compute** phase is marked by increased user-space CPU activity, while the **communications** phase is marked by increased activity in one or more of the system resources: disk, network, or kernel-space CPU activity to service the disk and network operations. Hence, application activity alternates between these two phases at a micro-level (e.g. in time-scales of micro- to milli-seconds, at the granularity of a single thread of execution). Then, we hypothesize that, taken at a macro-scale, this alternating activity of multiple interleaved threads induces correlated behavior at the macro-level between user-space CPU activity and system resource consumption.

Further, we hypothesize that in the presence of failures in the system, at the micro-level, user-space **compute** activity would show a marked deviation in its relationship with the indicators of the resources used during the **communications** phase (i.e. disk, network, kernel-space CPU activity). In the window of observation, either processing activity is impeded, and the **compute** phase dominates the **communications** phase, e.g. when there is a hang in the processing of the user job, or the **communications** phase dominates the **compute** phase, e.g. when there are problems with the disk, network, and other system resources, or when the **compute** task terminates prematurely, leading to a quick return to the **communications** phase. These micro-level disruptions then lead to macro-level disruptions in the correlated behavior between user-space CPU activity, and the system resources, i.e. disk, network, or kernel-space CPU activity.

5.2 Illustration

We illustrate the intuition behind the *RAMS* hypothesis of server application behavior with an example. Figure 2 shows a trace of the user-space CPU utilization, U, and the resource measure, the kernel-space CPU utilization, K, on two slave nodes in the same Hadoop

MapReduce cluster processing the same MapReduce job. We inject an external CPU load which consumes 70% of CPU utilization to simulate a fault on one node. Figure 2(b) shows a trace of the faulty node, where U increases significantly and remains high after the fault is injected 500 seconds into the experiment, while Figure 2(a) shows the trace of a fault-free node, where U and K exhibit similar behavior. Comparing the movement of the correlation coefficient $\rho_{U,K}$ for user- and kernel-space CPU utilization between the faulty and fault-free nodes, we can also see that $\rho_{U,K}$ remains significantly higher for the fault-free node as compared to the faulty node, and $\rho_{U,K}$ falls significantly after the fault is injected. This difference in the behavior of $\rho_{U,K}$ on faulty versus fault-free operation, provides support for the *RAMS* hypothesis.



(a) Normal node

(b) Faulty node

Figure 2: Time-series of User-space CPU utilization (red), and the resource metric chosen, Kernel-space CPU utilization (blue) (above), and the correlation between the two metrics (below), for the duration of one experiment. Horizontal axes show time elapsed in seconds. A CPU hog was injected on 1 node in a 5-node Hadoop cluster. Correlation falls when the fault is injected 500 seconds into the experiment.

5.3 *RAMS-FD*: Failure Detection Algorithm

Based on our hypothesis, the *RAMS-FD* algorithm computes the Pearson's correlation coefficient (ρ) between user-space CPU utilization and a resource metric for one of the resources (disk, network, kernel-space CPU utilization) used in the **communications** phase, and raises an alarm indicating a failure when the coefficient falls below a threshold, T. This computation is done independently for each of node in the system being diagnosed, for each time-series of sampled metrics collected on that node. $\rho \in [-1.0, 1.0]$ measures the degree of correlation, or linear dependence, between two variables. Its value is the ratio of the

covariance between two variables to the product of their standard deviations. For instance, the Pearson's correlation coefficient between user-space and kernel-space CPU utilization is as follows:

$$\begin{aligned} \rho_{U,K} &= \frac{covariance(U,K)}{\sigma_U \sigma_K} \\ \text{where } U &= \text{samples of user-space CPU utilization} = u_1, u_2, \ldots, u_n \\ K &= \text{samples of kernel-space CPU utilization} = k_1, k_2, \ldots, k_n \end{aligned}$$

In total, we compute **correlation coefficients** between user-space CPU utilization, U, and 7 resource-related metrics. Together with U, these raw metrics are sampled from `/proc` using the `sar` tool once per second:

1. $K = system\%$: Kernel-space CPU utilization
2. $DR = rtps$: Disk read transactions per second
3. $DW = wtps$: Disk write transactions per second
4. $NR = rxpck$: Network packets received
5. $NW = txpck$: Network packets transmitted
6. $MR = pgpgin$: Memory pages paged in
7. $MW = pgpgout$: Memory pages paged out

We refer to each of these 7 correlations as *RAMS-FD* detection metrics. In addition, we create compound measures of the conformance of the system's behavior to the *RAMS* hypothesis. This allows a combination of activity in multiple resource categories (disk, network, kernel-space activity) across time (e.g. user activity U may be strongly correlated with disk metrics at one point and network metrics at another), whereas a correlation between U and any of the above metrics measures only correlation with a single resource category. We create two compound metrics which take the maximum of the correlation between U and multiple metrics. They are:

$$\begin{aligned} \rho_{U,DiskNet} &= max \quad \{\rho_{U,DR}, \rho_{U,DW}, \rho_{U,NR}, \rho_{U,NW}\} \\ \rho_{U,DiskNetSys} &= max \quad \{\rho_{U,K}, \rho_{U,DR}, \rho_{U,DW}, \rho_{U,NR}, \rho_{U,NW}\} \end{aligned}$$

To raise alarms, we consider detection windows of D consecutive correlation coefficients, and raise an alarm if any of the D correlation coefficients falls below the threshold value T. The detection window also smooths the raw metrics to remove spurious correlations not indicative of true processing e.g. those caused by perfect sawtooth patterns during idle activity. To reduce the noise from the sampled parameters, we use a low-pass mean filter: we take the means of metric in small windows (in the order of less than 10s, discussed in §5.4) before computing the correlation coefficient.

Hence, for each node (host) in a system under detection by *RAMS-FD*, the algorithm takes as input the time-series's of user-space CPU utilization and of 1 of the 7 resource-related metrics (and the 2 additional compound metrics), and outputs a list of times when alarms are raised, when the correlation coefficients generated in the detection window D result in $\geq F$ threshold violations. For a distributed system with N nodes, N separate lists of violation times are generated. *RAMS-FD* does not need any training. It takes a user-specified threshold, T, and reports correlations which fall short of the threshold.

5.4 *RAMS-FD* Detection Algorithm Parameters

In total, there are 5 tunable parameters in the *RAMS-FD* algorithm. They are:

1. *LW, Low-pass Window Width*: Raw metric samples to take mean of
2. *LS, Low-pass Window Slide*: Raw metric samples to slide low-pass filter by
3. *W, Correlation Window Width*: Samples (after low-pass filter) to correlate
4. *S, Correlation Window Slide*: Samples to slide detection window by
5. *D, Diagnosis Window*: Correlation coefficients to consider

Then, the latency of the algorithm (i.e. minimum time between when alarms can be raised) in the steady state is the product $LS \times S$, i.e. the time interval between when *RAMS-FD* can raise each alarm is $LS \times S$ seconds, while the amount of information considered in each flag raised is $LW \times W$. Finally, we consider the threshold, T, to be a run-time user parameter that is user-tunable according to whether the user would prefer more alarms to be returned while suffering higher false-positives. For instance, in times when the sysadmin is inundated by other tasks, he can increase the threshold to reduce the amount of attention he needs to pay to the system, while he can reduce the threshold to investigate more alarms when he has the luxury of time to do so.

5.5 *RAMS-DT*: Problem Classification using Decision Trees

In the *RAMS-DT* algorithm, we learn decision trees from the detection alarms from *RAMS-FD* to classify the type of failure encountered according to past labeled failures. *RAMS-FD* returns 9 binary detection alarms for each node in a system under diagnosis, one for each category of system resources. These independent alarms can be reasoned about collectively to form a more complete diagnosis. We use decision trees to classify nodes by their collective behavior, as observed from the 9 *RAMS-FD* alarms, to determine if the node's behavior is similar to those in a previously diagnosed and labeled failure. One decision-tree is learned for each workload under diagnosis, as unlike *RAMS-FD* which is based on the general *RAMS* hypothesis, *RAMS-DT* performs best when decision-trees are tailored to the workload.

The *RAMS-DT* decision-tree is a binary-tree, where each interior node consists of one of the *RAMS-FD* detection metrics, and the left or right branch is taken depending on whether an alarm was raised for the metric. The decision-tree is learned from length-9 vectors of *RAMS-FD* alarms from a node in a distributed system, and a corresponding label of whether that node is fault-free, or the name of the fault it is suffering, for the duration of the *RAMS-FD* alarms. Then, the leaves of the decision-tree are used to classify subsequent nodes. The vector of detection alarms for the node is used to traverse the decision tree to a leaf node. Then, the *RAMS-DT* decision-tree returns the most frequently occurring training label at the leaf node in the tree. An approximate confidence in this diagnosis can also be obtained from the "purity" of the training nodes: the proportion of training nodes with the returned label. When this proportion is low, users can be prompted to carry out further manual investigation. We defer this aspect of *RAMS-DT* usage to future work.

In a deployment setting, we expect sysadmins would first respond to the basic *RAMS-FD* detection alarms, and perform in-depth investigation, after which they would close the case by ascribing a label to the episode. This label can then be associated with the *RAMS-FD* alarms collected from the episode. Over time, a decision-tree can be learned from this built-up collection of labeled *RAMS-FD* detection alarms, and used to classify future episodes.

5.6 Complexity and Scalability

As *RAMS-FD* uses only information from each node to compute alarms for that node, the detection can be carried out independently of all other nodes. The computation for detection on each node can be carried out on that node itself. This saves the bandwidth for transmitting metrics to a central location, and reduces the complexity of the global failure detection computation to a constant in the size of the distributed system.

In addition, *RAMS-FD* relies on computing the Pearson's correlation coefficient. The multiplications and divisions required are linear in the number of samples of the two variables involved. The number of samples involved in is determined by the correlation window size, W, which is in turn independent of the number of nodes in the system. Hence, the computation of each coefficient is constant in the number of nodes in the system. On the whole, diagnosing a distributed system with k hosts/nodes will involve computing k coefficients; hence, *RAMS-FD* scales linearly with the size of the system, and is highly scalable. In addition, *RAMS-FD* has no added training overhead.

After decision-trees have been learned for *RAMS-DT*, classification will involve only a traversal of the learned trees, which are at worst linear in the size of the training data, but in reality its size will be limited by the 9 *RAMS-FD* detection alarms used to split its nodes, resulting in a small tree for traversal. As the focus of *RAMS* is on the scalable *RAMS-FD* detection, the learning of *RAMS-DT* decision-trees is critical to the lightweight nature of *RAMS-FD*.

6 Summary and Future Work

We have presented *RAMS*, a hypothesis on system behavior under fault-free conditions, the resulting *RAMS-FD* algorithm for detecting failures in server applications, and the *RAMS-DT* algorithm for considering *RAMS-FD* alarms collectively using decision-trees to diagnose the root cause of a failure.

We plan to evaluate the efficacy of using *RAMS-FD* to detect failures in the Hadoop MapReduce data-intensive processing system, and in multi-tier web request processing systems, on the RuBiS online auction benchmark. We plan to investigate whether *RAMS-FD* is able to detect resource contention faults on Hadoop, and is able to detect application exceptions, server bugs, and hang faults on RuBiS. In addition, we plan to evaluate whether the results we achieve using correlation based metrics are intuitively interpretable to sysadmins, and can overcome stigmas about how automated diagnosis techniques are too complex to use. We also plan to demonstrate that the *RAMS-FD* is versatile by evaluating it across two systems. Finally, we plan to evaluate whether the *RAMS-DT* companion algorithm to *RAMS-FD* can take advantage of the multiple detection metrics to generate a root cause of a fault or a security attack that has been previously observed.

In future, we also plan to validate *RAMS* on more workloads and target systems, and to implement *RAMS* as a true black-box diagnosis tool which can simply analyze OS-level metrics to perform diagnosis transparently to the application. We also plan to explore augmenting the *RAMS* algorithm with multiple diagnosis windows and exponentially weighted samples to detect faults that may be less persistent, such as application bugs.

References

[1] E. Thereska, A. Ailamaki, G. Ganger, and D. Narayanan, "Observer: keeping system models from becoming obsolete," in *Second Workshop on HotTopics in Automatic Computing (HotAC II)*, Jacksonville, FL, June 2007.

[2] R. R. Sambasivan, A. X. Zheng, E. Thereska, and G. Ganger, "Categorizing and differencing system behaviours," in *Second Workshop on HotTopics in Automatic Computing (HotAC II)*, Jacksonville, FL, June 2007.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004, pp. 137–150.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *IEEE Conference on Dependable Systems and Networks*, Bethesda, MD, Jun 2002.

[5] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, vol. 16, no. 5, pp. 1027– 1041, Sep 2005.

[6] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed system of black boxes," in *ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003, pp. 74–89.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004.

[8] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger, "Stardust: tracking activity in a distributed storage system," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 3–14, 2006.

[9] G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi, "Distributed diagnosis of failures in a three tier e-commerce system," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Beijing, China, Oct 2007.

[10] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, Oct 2005, pp. 105–118.

[11] P. Bodik, M. Goldszmidt, A. Fox, D. Woodard, and H. Andersen, "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *EuroSys*, Paris, France, Apr 2010.

[12] J. Tan, X. Pan, S. Kavulya, E. Marinelli, R. Gandhi, and P. Narasimhan, "Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments," in *12th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr 2010.

[13] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, Seattle, WA, Jun 2009.

[14] Apache Software Foundation, "Hadoop," 2007, http://hadoop.apache.org/core.

[15] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr 2010.

[16] H. Kang, H. Chen, and G. Jiang, "PeerWatch: A fault detection and diagnosis tool for virtualized consolidated systems," in *International Conference on Autonomic Computing (ICAC)*, Washington D.C., USA, Jun 2010.

[17] S. Bhatia, A. Kumar, M. Fiuczynski, and L. Peterson, "Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems," in *Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec 2008.

[18] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena, "Ranking the importance of alerts for problem determination in large computer systems," in *International Conference on Autonomic Computing, ICAC*, Barcelona, Spain, June 2009, pp. 3–12.

[19] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "System monitoring with metric-correlation models: problems and solutions," in *International Conference on Autonomic Computing, ICAC*, Barcelona, Spain, June 2009, pp. 13–22.

[20] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *IEEE Conference on Dependable Systems and Networks*, Anchorage, Alaska, June 2008, pp. 452–461.

[21] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *EuroSys*, Lisbon, Portugal, Mar 2007.

[22] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing*, New York, NY, May 2004, pp. 36–43.

[23] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis," in *IEEE International Symposium on Software Reliability Engineering (ISSRE), Industrial Track*, Mysuru, India, Nov 2009.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs," in *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct 2009.

[25] J. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured log analysis," in *2nd USENIX Workshop on Analysis of System Logs (WASL)*, Big Sky, MT, Oct 2009.

[26] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems," in *International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, Jun 2010.

[27] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system." in *ACM Symposium on Operating Systems Principles*, Lake George, NY, Oct 2003, pp. 29 – 43.

[28] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, Jun 2003.