Short Paper: Troubleshooting Distributed Systems via Data Mining

David A. Cieslak, Douglas Thain, and Nitesh V. Chawla *
Department of Computer Science and Engineering,
University of Notre Dame, USA

Abstract

Through massive parallelism, distributed systems enable the multiplication of productivity. Unfortunately, increasing the scale of available machines to users will also multiply debugging when failure occurs. Data mining allows the extraction of patterns within large amounts of data and therefore forms the foundation for a useful method of debugging, particularly within such distributed systems. This paper outlines a successful application of data mining in troubleshooting distributed systems, proposes a framework for further study, and speculates on other future work.

1 Introduction

Distributed systems are multiplicative tools: they multiply the ability of a user to perform useful work, but they also multiply the ability to make a mess. Consider the following common scenario: a user submits one million jobs to a system consisting of thousands of CPUs. After working on the jobs for some time, the system reports completion. However, the user unhappily discovers that a large fraction of the jobs have failed! How would an end-user even begin to sort out this mess? This is not merely an academic exercise. Regular users of large distributed systems are accustomed to these occurrences, particularly in the development stages of a system. For example, Grid3 [7] has reported a thirty percent failure rate for some categories of jobs.

Why do unexplained failures of this type occur at large scale? There are many different kinds of failure that are difficult to disentangle. First, a given job may not necessarily be compatible with arbitrary machines. Obviously, most jobs require a particular CPU and operating system. More subtly, users may not be aware that a particular job requires certain tools or libraries to be installed on the operating system. Other failures may be due to administrative activities: the firewall rules on a cluster might be adjusted, or a user may be denied access to resources previously available. Finally, some failures are due to pure chance: a cosmic ray disrupts a stored value, and the program crashes.

We propose that data mining techniques can be applied to the problem of large scale troubleshooting. If both jobs and the resources that they consume are annotated with structured information relevant to success or failure, then classification algorithms can be used to find properties of each that correlate with success or failure. In the one-million jobs example above, an ideal troubleshooter would report to the user something like: *Your jobs always fail on Linux 2.8 machines, always fail on cluster X between midnight and 6 A.M, and fail with 50% probability on machines owned by user Y.* Further, these discoveries may be used to automatically avoid making bad placement decisions that waste time and resources.

We hasten to note that this form of data mining is not a panacea. It does not explain *why* failures happen, or make any attempt to diagnose problems in fine detail. It only proposes to the user properties correlated with success or failure. Other tools and techniques may be applied to extract causes. Rather, data mining allows the user of a large system to rapidly make generalizations to improve the throughput and reliability of a system without engaging in low level debugging. These generalizations may be used later at leisure to locate and repair problems. In addition, the problem of distributed debugging, with its unique idiosyncracies and dynamics, lends itself as a compelling application for data mining research. Standard off-the-shelf methodologies might not be directly applicable for large, dynamic, and evolving system. It is desired to implement techniques that are capable of incremental self-revision and adaptation. The goal of our paper is to serve as a proof-of-concept and identify venues for compelling future research.

2 Proof of Concept

We have created a proof of concept of this idea by analyzing jobs submitted to an active Condor [9] pool of 250 machines with a wide variety of properties and several active users. To apply data mining, we must use the information available from this pool to determine attributes. Experiments must then be performed to validate that data mining is useful in distributed systems troubleshooting.

[*] Authors' e-mail: {dcieslak,dthain,nchawla}@cse.nd.edu

1-4244-0307-3/06/$20.00 ©2006 IEEE.

Condor makes use of ClassAds [15] to describe both jobs and resources in a structured way, so we do not need to add any additional gathering instrumentation. For example, Figure 1 represents fragments of job and machine ClassAds. Of course, each job and machine has many more properties that are not always apparent or interesting to the user. In our configuration, both jobs and machines have over 70 properties, describing items ranging from the vital (command line arguments) to the mundane (time since last reboot). We have constructed a tool to capture the information provided on each machine by ClassAds, which then converts this dynamically into a feature space based by translating each field into a separate dimension of this space. This allows for many descriptive properties of the machines, such as CPU, size of memory, available diskspace, owner, and cluster. As we will see below, these non-obvious properties and their interactions play an interesting role in troubleshooting.

Job	Machine
Cmd = “simul.exe”	Name = “c01.nd.edu”
Owner = “dcieslak”	Arch = “INTEL”
ImageSize = 40569	OpSys = “Linux”

Figure 1. Job and Machine Fragments

With a process in-hand to generate the attribute spaces as needed, we then proceed to construct a set of controlled experiments in which failure and failure conditions are unambiguous. This enables, verification of data mining to capture the proper failure conditions. To test this concept in a controlled fashion, we construct batches of 1000 jobs that fail in known ways. To simplify, we define *failure* to mean *exited with status!=0*, although failures may be detected by many other means. As the jobs complete, we extract feature vectors that capture the pair of job and machine, along with an indication of success or failure. With each dataset, we then applied two data mining techniques. C4.5 [14] decision trees are among the most popular data mining techniques and employ an entropy-based divide-and-conquer induction algorithm for tree construction. While C4.5 is used in most supervised learning applications, they are prone to overfitting without properly tuned pruning. Still, C4.5 is capable of generating highly accurate models. RIPPER [4] is a rule based data mining method that is more expensive than C4.5, but is much more noise tolerant. This method uses incremental pruning to reduce error and adopts a separate-and-conquer approach, which removes training examples with established rules via recursion. Our experiments we show Ripper to be more amenable for distributed debugging. It produced more accurate and comprehensibile model.

To evaluate the functionality offered by data mining, we implemented the following four different (simulated) failure scenarios. (1) Fail on a given operating system. (2) Fail on a given OS with insufficient disk space. (3) Fail on a given OS and on others with a 66% probability. (3) Fail randomly with 50% probability.

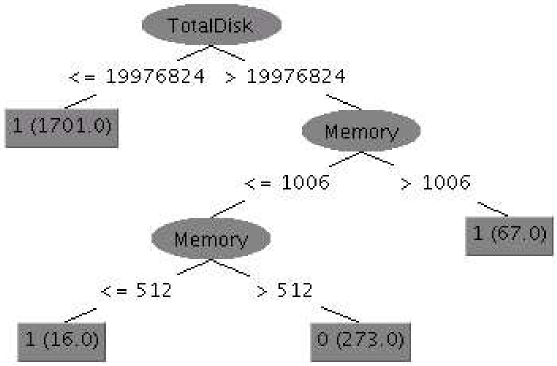

Figure 3. C4.5 Decision Tree

Figure 2 shows the response to these simulated cases. RIPPER produces compact rules, while C4.5 produces a classification tree, an example of which may be seen in Figure 3. In case (1), both classifiers predict that the job will fail if it runs on a SUN4U CPU, which is always associated with the SOLARIS OS in this pool. In case (2) both classifiers find that sufficient disk is needed to run the job, but capture the VirtualMemory property rather than the LINUX OS. In this particular pool, all LINUX machines match the VirtualMemory condition. In (3) RIPPER finds the condition most likely to lead to success (Arch==“INTEL”), even though that failure rate is still high. Finally, (4) demonstrates that random failures still result in some unexpected classifications by chance. While RIPPER was found to be the more robust method for rule learning, Figure 3 depicts a C4.5 decision tree learned from the third simulation and is useful for demonstrating the elegance of rules versus trees.

What is interesting about these results is that classification algorithms can produce results that are *correct observations* that can be *acted upon*, even though the results may be *unexpected*. Consider case (2): Even though the classifier identifies machines based on disk and memory properties, these properties may be used for future machine selection in order to avoid failures, because those properties are in fact correlated with the actual cause of the failure. A reasonable troubleshooting tool may observe properties and then modified the job requirements as a workload runs.

Of course, this assumes that the membership (and properties) of the pool do not change! Consider case (1): The troubleshooter proposes that will always fail on SUN4U, because the current pool only contains SUN4U-SOLARIS and INTEL-LINUX machines. If SUN4U-LINUX machines are added to the pool, they will (incorrectly) go unused. Or, if INTEL-SOLARIS machines are added, jobs will fail on them. Case (4) presents a danger. If jobs fail

	Induced Failure	RIPPER Output	C4.5 Output	Action Taken
1	**If** OS = Solaris **then** fail **else** succeed	(Arch = SUN4u) => ExitCode=1 (975.0/0.0) => ExitCode=0 (1996.0/0.0)	**nodes:** 3 **leaves:** 2	Assign all jobs to non-SUN4u machines
2	**If** OS = Linux **then** **If** Disk > 1e+12 **then** succeed **else** fail **else** fail	(TotalDisk ≥ 22955424) and (Disk ≤ 13989502) and (TotalVirtualMemory ≤ 2040244) => ExitCode=0 (273.0/0.0) => ExitCode=1 (1784.0/0.0)	**nodes:** 7 **leaves:** 4	Assign to machines with high disk space
3	**If** Arch = SUN4U **then** fail **else** fail 66% of the time	(Arch = INTEL) => ExitCode=0 (622.0/307.0) => ExitCode=1 (508.0/0.0)	**nodes:** 17 **leaves:** 9	Assign all jobs to Intel machines
4	**fail** 50% of the time	(KFlops ≥ 107527) => ExitCode=1 (244.0/111.0) (Disk ≤ 1557479) => ExitCode=1 (208.0/101.0) => ExitCode=0 (371.0/170.0)	**nodes:** 3 **leaves:** 2	Assign to any machines, but favor low KFlops and high Disk values

Figure 2. Diagnosis of Synthetic Failures

randomly, or due to some factor not captured by the instrumentation, a naive troubleshooter will unnecessarily restrict the set of machines to be used. If applied multiple times, the set will shrink until no machines are available.

Using this approach, some initial results on real jobs are promising. Evaluating several thousands jobs that previously ran on the pool, RIPPER identified an 80% failure rate on machines with less than 10 GB of disk and 80% success rate on machines with greater than 10 GB. With this information in hand, we explored individual failures on machines with less disk space, and discovered that Condor itself crashed in an unusual way when large output files consumed all space. By adjusting the job's requirements to avoid such machines, we may avoid such continuous crashes and make the resources available to other users. We intend to deploy this tool on more systems and more workloads to gain more experience.

3 Challenges and Opportunities

Based on these results, we believe that troubleshooting via data mining must be a *dynamic* and *continuous* activity in the spirit of autonomic computing. Such a troubleshooter cannot be run once and forgotten. It must continually monitor the productivity of a system and make adjustments at run-time. In order to deal with the challenges of a changing system mentioned above, decisions must be re-evaluated periodically and re-applied to the system. In this sense, troubleshooting can be posed as a control theory problem: an output signal must be kept within certain tolerances by applying controls that have delays and imprecise effects.

Thus far, we have employed *eager learning* methods, such as the discussed C4.5 and RIPPER classifiers, to construct a model using a set of training data in an off-line manner. Once the set of rules has been constructed, the original training dataset is no longer applied and each new record has no bearing in future decisions. However, the above description of troubleshooting clearly entails a *lazy learning* approach. Such methods postpone generalization until a query (in this case, when the user requests the grid to match a given program to a pool machine) and therefore have a larger set of information available to inform decisions. Examples of lazy learning include k-Nearest Neighbor and future studies of data mining assisted troubleshooting should consider this type of an approach. However, one disadvantage of lazy learning is that the training data must always remain available, which introduces separate concerns such as maintaining and storing a required information warehouse.

There are many more subtleties to be explored from a data mining perspective. We have only considered the possibility that machine properties are the cause of various failures. In a real system, it may be job properties: a faulty code might crash only on certain inputs. Or, it might be some combination of job and machine properties: varying job configurations may consume resources that cause a failure only on certain classes of machines. Or, there may be correlations between machines and external events: a cluster might be rebooted on a regular basis, causing jobs to failure; or, network firewall rules might inhibit certain applications on certain clusters. Each of these requires different classification techniques and introduces new sources of noise that must be accommodated. Thus, the desired characteristics of data mining techniques will be: incremental, prediction and learning under uncertainty, robust and efficient. Efficiency can be a characteristic as running a distributed debugger using data mining should not impose additional overhead on the system. Moreover, it should ideally be transparent to the end user. We believe the problem presents itself as an interesting hand-shake between two different streams of research reaping broad benefits. To understand and implement this better, we must instrument large scale systems under heavy load.

As the quality of data mining results is bounded by the ability of the constructed model to capture the nuance of information modeled, the described troubleshooting system performance is likely to be enhanced through more sophisticated features. Specifically, the information acquired for this survey was limited in *attriubutes* and *class*. In terms of class, this problem has been devolved into one of binary classes: identify successes and failures, where each is abstracted by Condor and defined by the user: Condor reports the final status of each submission and the user must interpret which of these implies success and failure. While quantifying separate success states is unlikely to improve results, increasing granularity in types of failure may yield information that is significantly more valuable to the user. This type of power allows the potential isolation of error source: faulty code, incompatible source libraries, bad input, system level issues, or randomized glitches. Additionally, the attributes surveyed impose further restrictions on the quality of predictions.

The ClassAd used to construct the attributes is user defined and maintained, and therefore there is no insurance to the accuracy of the information stored. Additionally, ClassAd tends to kept higher level information on each system (i.e. architecture, OS, memory), but significant clues to program failure may not be represented (i.e. the system's *perl* version). Therefore, classification is limited by the amount of detail provided by each computer's administrator. Even a full archive of such information may be insufficient for troubleshooting, but such a record would allow isolation of error at a system level. Further considerations for troubleshooting could be made to isolate the exact point of failure within a program on a given system and a recommended solution is reported to the user or used automatically.

4 Related Work

Debuggers such as Paradyn [12], PDB [11], and P2D2 [3] have extended the stop-modify-run mode of debugging to parallel and distributed systems. Although we have made use of the ClassAd mechanism in Condor, a variety of tools are available for logging and collecting data from distributed systems. Ganglia [10], NetLogger [8], MAGNET [6] and MonALISA [13] all allow for the construction of sophisticated monitoring networks and reduction services that allow for the aggregation of hundreds or thousands of performance measurements into a unified stream. A related method of debugging systems is to treat all components of a system as black boxes, using *structure* rather than *metadata* to debug problems. For example, one may examine the messages in a distributed system to infer causal relationships [16], to identify performance bottlenecks [1], or to suggest root-cause failures [2] without understanding the content of the messages themselves. A related technique is to search for deviant use patterns [5] in order to locate bugs.

5 Conclusion

To conclude, we believe that large scale distributed systems are sorely lacking in debugging and troubleshooting facilities. As others have noted [17], most users of the grid are limited by usability, not by performance. Troubleshooting via data mining represents one method that helps the user to draw general conclusions from large amounts of data. Many more tools and techniques are needed in order draw out specific reasons why individual components fail.

Acknowledgement. We thank Aaron Striegel for providing the computing resources used in this project.

References

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging of black-box distributed systems. In *ACM Symposium on Operating Systems Principles*, October 2003.

[2] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks*, 2002.

[3] D. Cheng and R. Hood. A portable debugger for parallel and distributed programs. In *Conference on Supercomputing*, 1994.

[4] W. Cohen. Fast effective rule induction. In *Proc. of Machine Learning: the 12th International Conference*, pages 115–123, July 1995.

[5] D. Engler, D. Chen, S. Hallem, A. Chaou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in system code. In *ACM Symposium on Operating Systems Principles*, October 2001.

[6] M. Gardner, W. chen Feng, M. Broxton, A. Engelhart, and G. Hurwitz. MAGNET: A tool for debugging, analyzing, and adapting computer systems. In *IEEE/ACM Symposium on Cluster Computing and the Grid*, May 2003.

[7] R. Gardner and et al. The Grid2003 production grid: Principles and practice. In *IEEE High Performance Distributed Computing*, 2004.

[8] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *IEEE High Performance Distributed Computing*, June 2002.

[9] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Eighth International Conference of Distributed Computing Systems*, June 1988.

[10] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30, July 2004.

[11] R. Mehmood, J. Crowcroft, S. Hand, and S. Smith. Grid-level computing needs pervasive debugging. In *IEEE/ACM International Workshop on Grid Computing*, Seattle, WA, November 2005.

[12] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[13] H. Newman, I. Legrand, P.Galvez, R. Voicu, and C. Cirstoiu. MonALISA: A distributed monitoring service architecture. In *Computing in High Energy Physics*, March 2003.

[14] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[15] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *IEEE Symposium on High Performance Distributed Computing*, July 1998.

[16] C. Schaubschlager, D. Kranzlmuller, and J. Volkert. Event-based program analysis with de-wiz. In *Workshop on Automated and Algorithmic Debugging*, pages 237–246, Septmber 2003.

[17] J. Schopf. State of grid users: 25 conversations with UK eScience groups. Argonne National Laboratory Tech Report ANL/MCS-TM/278, 2003.