Statistical Monitoring + Predictable Recovery = Self-*

Armando Fox and Emre Kıcıman, Stanford University
David Patterson, Randy Katz, Michael Jordan, Ion Stoica, Doug Tygar, University of California, Berkeley

March 9, 2004

Abstract

It is by now motherhood-and-apple-pie that complex distributed Internet services form the basis not only of e-commerce but increasingly of mission-critical network-based applications. What is new is that the workload and internal architecture of three-tier enterprise applications presents the opportunity for a new approach to keeping them running in the face of both "natural" failures and adversarial attacks. The core of the approach is anomaly detection and localization based on statistical machine learning techniques. Unlike previous approaches, we propose anomaly detection and pattern mining not only for operational statistics such as mean response time, but also for *structural* behaviors of the system—what parts of the system, in what combinations, are being exercised in response to different kinds of external stimuli. In addition, rather than building baseline models *a priori*, we extract them by observing the behavior of the system over a short period of time during normal operation. We explain the necessary underlying assumptions and why they can be realized by systems research, report on some early successes using the approach, describe benefits of the approach that make it competitive as a path toward self-managing systems, and outline some research challenges. Our hope is that this approach will enable "new science" in the design of self-managing systems by allowing the rapid and widespread application of statistical learning theory techniques (SLT) to problems of system dependability.

1 Recovery as Rapid Adaptation

A "five nines" availability service (99.999% uptime) can be down only five minutes a year. Putting a human in the critical path to recovery would expend that entire budget on a single incident, hence the increasing interest in self-managing or so-called "autonomic" systems. Although there is extensive literature on statistics-based change point detection [2], some kinds of partial failures, or "brown-outs" in which only part of a service malfunctions, cannot be easily detected by such techniques. For example, one of the authors experienced a bug such that after clicking to purchase a flight for April, a later visit to the "flight details" page showed the wrong flight date (in October) and no flight itinerary details at all. If the operational statistics such as response time for delivering this page are within normal thresholds, performance monitoring would not find this problem.

We believe a promising direction is to start thinking not in terms of normal operation vs. recovery, but in terms of constant and rapid adaptation to external conditions, including sudden workload changes, inevitable hardware and software failures, human operator errors, and in extreme cases, catastrophic failures or malicious attacks. In particular, we propose the broad application of techniques from statistical learning theory (SLT)—automatic classification, novelty/anomaly detection, data clustering, etc.—to observe and track *structural* behaviors of the system, and to detect potential problems such as the example above.

2 Approach and Assumptions

We assume typical request-reply based Internet services, with separate session state [11] used to synthesize more complex interactions from a sequence of otherwise stateless request-reply pairs. Past approaches to statistical monitoring of such services have primarily relied on *a priori* construction of a system model for fault detection and analysis; this construction is tedious and error-prone, and will likely remain so as our services continue to evolve in the direction of heterogeneous systems of black boxes, with subsystems such as Web servers, application logic servers, and databases being supplied by different vendors and evolving independently. We propose instead to build and periodically update the baseline model by observing the system's own "normal" behavior. The approach can be summarized as follows:

1. Ensure the system is in a state in which it is mostly doing the right thing most of the time, according to simple and well-understood external indicators.

2. Collect observations about the system's behavior during this time to build one or more baseline models of behavior. These models may capture either time-series behaviors of particular parameters or structural behaviors of the system.

3. If "anomalous" behaviors relative to any of these models are observed, automatically trigger simple corrective actions. If repeated simple corrective actions do not cause the anomaly to go away, notify a human. Since false positives are a fact of life with statistical approaches, we also need a strategy for quantifying and dealing with the cost of acting on false positives.

4. Periodically, go back to step 2, to update the model.

Each of steps 1–3 corresponds to an assumption, as follows.

A1. Large number of independent requests. If most users' interactions with the service are independent of each other (as they usually are for Internet services), and if we assume bugs are the exception rather than the norm, such a workload gives us the basis to make "law of large numbers" arguments supporting the use of statistical techniques to extract the model from the behavior of the system itself. Also, a large number of users per unit time means that large fractions of the service's functionality are exercised in a relatively short period of wall-clock time, providing hope that the model can be created and maintained online, while the system is running.

A2. Modular architecture for observation points. To use statistical or data-mining techniques, we need a representation of the data observations the model will operate on ("concepts" in the terminology of data mining) and a way to capture those observations. A modular service design, such as the componentized design induced by Java 2 Enterprise Edition (J2EE) or CORBA, allows us to crisply define a single user's time-bounded request-reply interaction with the service as a collection of discrete service elements or subsystems that participated in that interaction. For example, in J2EE, the unit of application modularity is the Enterprise Java Bean (EJB); a particular codepath through a J2EE application will "touch" some subset of EJB classes. Note that components themselves are opaque—we do not see intra-component method calls. This coarser grain maximizes the likelihood that the number of "legitimate" code paths through the system is much smaller than the number of permutations of components, making anomaly detection appealing. Note also that it is OK if the behaviors observed at different observation points are correlated with each other, or completely uncorrelated to any interesting failure: machine learning techniques called feature-selection algorithms can identify the subset of features most predictive of anomalies from a much larger collection of features. Lastly, collecting these observations must not materially interfere with service performance.

A3. Simple and predictable control points. If the model's predictions and analyses are to be used to effect service repair when an anomaly indicating a potential failure is detected, there must be a safe, predictable, and relatively non-disruptive way to do so. *Safe* means that correct application semantics are not jeopardized by actuating the control point. *Predictable* means that the cost of actuating the control point must be well known. *Non-disruptive* means that the result of activating a control point will be no worse than a minor and temporary effect on performance. These properties are particularly important when statistical techniques are used because those techniques will inevitably generate false positives. If we know that the only effect of acting on a false positive is a temporary and small decrease in performance, we can quantify the cost of "blindly" acting on false positives; this enhances the appeal of automated statistical techniques, since many techniques' sensitivity can be tuned to trade off false positive rates vs. false negative (miss) rates.

We now turn to how these assumptions might be satisfied in a real service. Note that A1 is trivially true for the services in question, whereas A2 and A3 lead to some interesting systems research.

3 Observation and Control Points

Since modifying every existing application to add observation and control points is cumbersome and unlikely, we limit our attention initially to *framework-intensive* applications[1]—those whose total code consists mostly of middleware (e.g. J2EE runtime services, libraries, etc.) with a smaller amount of application logic (though even simple applications typically contain 10K to 100K lines of such logic). By modifying the middleware, we can provide application-generic observation points without any extra work for application programmers. For example, we modified the source code of the JBoss open-source application server to collect and report code-path observations [4].

It is more difficult to add application-generic control points that are predictable, safe and non-disruptive. Crashing and rebooting a machine is certainly predictable, since crashing relies only on a simple external mechanism (the power switch), but it may be unsafe or disruptive or both, unless the application is known to be crash-only [3]. An alternative would be machine-level crashes in a system

[1] By framework we refer to a componentized middleware such as J2EE or CORBA.

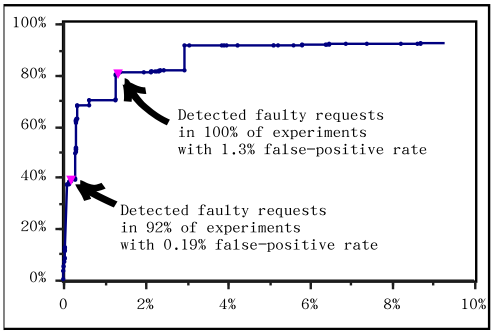
designed specifically so that a combination of overprovisioning and fast failover can mask the crash in the form of slight additional latency, though extra steps might have to be taken to ensure correctness (i.e. so that affected users see a performance blip instead of error messages).

4 SLT and Dependability

Having briefly addressed some important systems-building issues (to which we return shortly in the context of some concrete examples), we now discuss the core of our detection and diagnosis strategy. Statistical learning theory (SLT) provides a framework for the design of algorithms for classification, prediction, feature selection, clustering, sequential decision-making, novelty detection, trend analysis, and diagnosis. Its techniques are already being used in bioinformatics, information retrieval, spam filtering and intrusion detection. We propose a software architecture for integrating SLT pervasively into the computing infrastructure, as a tool for evaluating which SLT techniques are useful at detecting which kinds of problems. For concreteness, we describe two simple examples: one based on time-series models and another based on structural models.

4.1 Time Series Models

Time-series models capture patterns in a service's temporal behavior that cannot be easily characterized by a statistic or a small set of parameters. For example, the memory used by a server-like process typically grows until garbage collection occurs, then falls abruptly. We do not know the period of this pattern, or indeed whether it is periodic; but we would expect that multiple servers running the same logic under reasonable load balancing should behave about the same—the relative frequencies of garbage-collection events at various timescales should be comparable across all the replicas. We successfully used this method to detect anomalies in replicas of SSM, our session state management subsystem [11]. Each replica reports the values of several resource-usage and forward-progress metrics once per second, and these time series are fed to the Tarzan algorithm [9], which discretizes the samples to obtain binary strings and counts the relative frequencies of all substrings within these strings. Normally, these relative frequencies are about the same across all replicas, even if the garbage-collection cycles are out of phase or their periods vary[2]. If the relative frequencies of more than 2/3 of these metrics on some replica differ from those of the other replicas, that replica is immediately rebooted.



Detected faulty requests
in 100% of experiments
with 1.3% false-positive rate

Detected faulty requests
in 92% of experiments
with 0.19% false-positive rate

Figure 1: Detection rate vs. false positive rate for PCFG-based path-shape analysis of PetStore 1.3 running on our modified JBoss server. Relying on HTTP error logs would reduce the detection rate to about 78%. Compared to the uninstrumented application, our throughput is 17% less, request latency is about 40ms more, and analysis of several thousand paths takes a few seconds, suggesting that the approach is feasible as an online technique.

This works because SSM is deliberately optimized for fast reboot: it does not preserve replica state across reboots, and since some overprovisioning due to replication is inherent in its design, this control action is safe, predictable and non-disruptive. The net effect is that SSM as a system has no concept of "recovery" vs "normal" behavior; since periodic reboots are normal and incur little performance cost, the system is "always recovering" by adapting to changing external conditions through a simple composition of mechanisms.

4.2 Structural Models

Structural models capture control-flow behavior of an application, rather than temporal behavior. One example of a structural model is a *path*—the inter-component dynamic call tree resulting from a single request-reply interaction. We modified JBoss to dynamically collect such call trees for all incoming requests; these are then treated as parse trees generated by a probabilistic context-free grammar (PCFG) [12]. Later on, when a path is seen that corresponds to a low-probability parse tree, the corresponding user request is flagged as anomalous. In our initial testing, this approach detects over 90% of various injected faults with false positive rates around 3% (see figure 1). While our diagnosis results are not as good, with our decision trees identifying the correct cause of the anomaly only 50–60% of the time, it is striking that the technique performs as well as it does with no prior knowledge of the application's structure or semantics.

[2] Classical time-series methods are less effective when the signal period varies.

Since there is a nonzero false positive rate, we must make sure that any action we take is safe, predictable and non-disruptive. In this case, we respond by selectively "microrebooting" the suspected-faulty EJB's [4] without causing unavailability of the entire application. Although this work is still in progress, we have demonstrated that EJB microreboots are predictable and non-disruptive, and there is reason to believe they are safe because J2EE constrains application structure in a way that makes persistent state management explicit—most EJB's are stateless, and we are modifying JBoss to externalize the session state into SSM, which is itself optimized for safe and non-disruptive fast reboot.

5 Research Challenges

We have focused on recasting "recovery" as a kind of rapid adaptation, but a similar argument applies for other online operations such as resource management. For example, in a crash-only distributed hash table that we built [8], online repartitioning can be achieved by taking one replica offline (which looks like a failure and does not affect correctness), cloning it, and bringing both copies back online. Each will then have some stale data, but to the system, this looks the same as existing failure cases that are already handled by normal-case mechanisms. Hence no new machinery is required to implement growing, partitioning, or rebalancing as online operations analogous to "failure and recovery".

Most existing implementations of SLT algorithms are offline; our proposal may motivate SLT practitioners to focus on online and distributed algorithms. The above experiments show that even an unoptimized offline implementation of PCFG analysis can process thousands of paths in a few seconds. This in turn motivates the need for generic data collection and management architectures for statistically-monitored systems: even a simple (11K lines of code) application we instrumented produces up to 40 observations per user request, with 1,000 to 10,000 requests per second being representative of Internet services. Scalable abstractions for sliding data windows, sampling, fusion of results from different SLT models, etc. will have to be provided, as well as easy ways to create observation and control points without requiring intrusive modifications to every application.

Finally, although we have discussed applying SLT approaches primarily at the application level, we note that the needed infrastructure is largely in place for applying it at all levels of functionality all the way down to the hardware. A legacy of the Active Networking research agenda [14] is a new generation of user-programmable network devices for storage virtualization, server load balancing, and traffic management, which provide some of the observation and control points needed by our approach and allow us to make "law of large numbers" arguments required by assumption A1. Figure 2 shows a block-diagram architecture (parts of which we are already prototyping) for distributed network applications that exploit SLT-based monitoring at multiple levels.

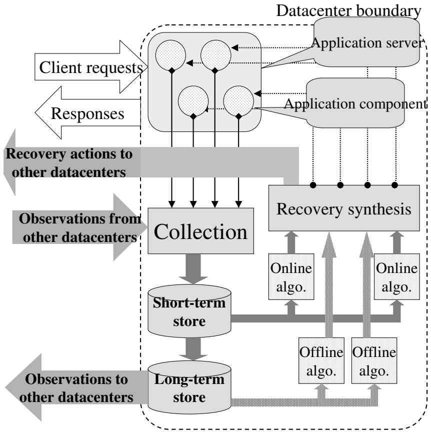


Figure 2: Internal high-level architecture of an intra-datacenter adaptive system. Application components (circles) export observation and control points to the SLT algorithms, with inter-datacenter exchange of observation and control points for geographically-distributed applications.

6 Related Work

Anomaly detection has been used to infer errors in systems code [5], debug Windows Registry problems [15], detect possible violation of runtime variable assignment invariants [7], and discover source code bugs by distributed assertion sampling [10]. The latter is particularly illustrative of SLT's ability to mine large quantities of observations for interesting patterns that can be directly related to dependability. System parameter tuning and automatic resource provisioning have also been tackled using PCFG-based approaches [1] and closed-loop control theory [13], although such approaches generally cannot detect functional or structural deviations in system behavior unless they manifest as performance anomalies.

The Recovery-Oriented Computing project [6] has argued that fast recovery is good for its own sake, but in the context of SLT, fast recovery is *essential* because it gives us an inexpensive way to deal with false positives. As such, ROC is a key enabler for this approach.

7 Conclusion

Our ability to design and deploy large complex systems has outpaced our ability to deterministically predict their behavior except at the coarsest grain. We believe statistical approaches, which can find patterns and detect deviations in data whose semantics are initially unknown, will be a powerful tool not only for monitoring and online adaptation of these systems but for helping us better understand their structure and behavior. We have argued that the structure of today's enterprise services supports the "many independent samples" assumption required for SLT to be effective, and that the challenge for systems research is to provide pervasive observation and control points that can serve as the "sensors and actuators" connected to the SLT algorithms.

As early steps in tackling that challenge, we described two applications of this technique: application-specific work on SSM, which has observation points built-in and allows whole-machine crashing as a control point, and application-generic work on the JBoss application server, with observations based on J2EE application paths and control points based on microreboots. Although our initial results are promising, in the long view these are just persuasive proofs-of-concept that invite much deeper exploration of the approach. A generic platform for pervasive integration of SLT methods, themselves the subject of broad and vigorous research, would hasten the adoption of SLT into dependable systems, which we believe would in turn provide a new scientific foundation for the construction of self-managing systems.

Acknowledgments

The ideas in this paper have benefited from advice and discussion with George Candea, Timothy Chou, Moises Goldszmidt, Joseph L. Hellerstein, Ben Ling, Matthew Merzbacher, and Chris Overton.

References

[1] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: real-time modelling and performance-aware systems. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[2] Michèle Basseville and Igor V. Nikiforov. *Detection of Abrupt Changes—Theory and Application*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1993.

[3] George Candea and Armando Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[4] George Candea, Pedram Keyani, Emre Kiciman, Steve Zhang, and Armando Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[5] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Lake Louise, Canada, Oct 2001.

[6] David A. Patterson et al. Recovery-Oriented Computing: motivation, definition, techniques, and case studies. Technical Report CSD-02-1175, University of California at Berkeley, 2002.

[7] Sudheendra Hangal and Monica Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[8] Andy C. Huang and Armando Fox. A persistent hash table with cheap recovery: A step towards self-managing state stores. In preparation.

[9] E. Keogh, S. Lonardi, and W Chiu. Finding surprising patterns in a time series database in linear time and space. In *In proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 550–556, Edmonton, Alberta, Canada, Jul 2002.

[10] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[11] Benjamin C. Ling, Emre Kıcıman, and Armando Fox. Session state: Beyond soft state. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 04)*, San Francisco, CA, March 2004.

[12] Christopher D. Manning and Hinrich Shutze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, 1999.

[13] S Parekh, N Gandhi, JL Hellerstein, D Tilbury, TS Jayram, and J Bigus. Using control theory to achieve service level objectives in performance management. *Real Time Systems Journal*, 23(1–2), 2002.

[14] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. In *ACM SIGCOMM '96 (Computer Communications Review)*. ACM, 1996.

[15] Yi-Min Wang, Chad Verbowski, and Daniel R. Simon. Persistent-state checkpoint comparison for troubleshooting configuration failures. In *Proc. International Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003.