Toward Self-Healing Multitier Services

Brian Cook
Duke University and IBM
bcook@cs.duke.edu

Shivnath Babu*
Duke University
shivnath@cs.duke.edu

George Candea
École Polytechnique Fédérale de Lausanne
George.Candea@epfl.ch

Songyun Duan
Duke University
syduan@cs.duke.edu

Abstract

Are self-healing database-centric multitier services utopia or just a hard puzzle? We argue for the latter and aim to identify the missing pieces of this puzzle. We advocate robust and scalable learning-based approaches to self-healing that we expect to work well for a large class of multitier services. We identify performance-availability problems (PAPs) as the most relevant target for self-healing, and argue that PAPs are best addressed macroscopically, outside the realm of individual tiers. Finally, we lay out a research agenda for learning-based approaches to self-healing, to enable wider deployment of self-healing multitier services.

1 Introduction

For a long time in the history of computing there were databases with custom-written clients. The 90s brought about the web and "thin clients," and databases started hiding behind web servers. As the scale of such services increased, stored procedures made their way out of databases into a new tier—the application server—running so-called "business logic." The recent emergence of compute utilities (e.g., Amazon Elastic Cloud, Sun Grid) are adding yet another dimension to the architecture of services we have come to expect at our fingertips. While more architectural churn can be expected in future, it has become clear that database-centric computing infrastructures will be multitiered, requiring what once was one database administrator (DBA) to become a growing organization of specialized professionals.

This multitier computing infrastructure consists of services our society can no longer do without (e.g., Amazon, eBay, Google). Users now expect the same ubiquity and reliability from these services as that offered by the phone system and electricity grid. These services are required to meet *service-level objectives*, or *SLOs*, that specify what an acceptable level of service is [16]. For example, an SLO for an online brokerage may stipulate that all transactions complete within 1 second, regardless of how much middleware, databases, or networks are involved. Dependable and predictable databases are not enough to meet all SLOs.

Today's services have difficulty meeting their SLOs. A recent study [21] found that 72% of the top-40 web sites suffer user-visible failures, such as items not being added to shopping carts or various error messages. Walmart.com experienced a 10-hour outage during the 2006 Thanksgiving traffic surge [27]. Such deviations from correct and desired behavior, or *failures*, can cause user dissatisfaction and substantial financial loss; for instance, a 22-hour outage at eBay in 1999 cost the company more than $3 Million in customer credits and $4 Billion in market capitalization [13].

Failures occur both due to system faults (such as process crashes) as well as performance bottlenecks (such as executing a suboptimal database query plan due to stale statistics), in all tiers of the service. Almost always, the root cause is the fallibility of humans, e.g., they introduce software bugs, misconfigure systems, fail to update statistics in a timely fashion, or replace the wrong hardware. It is therefore compelling to build systems that self-heal *across all tiers* and reduce the day-to-day involvement of humans in the operation of the service.

Furthermore, recovery from service failure ought to be quick when it occurs, because every minute counts (e.g., brokerages and banking firms can lose up to $75,000/minute of downtime [16]). While there are many *mechanisms* readily available for fast recovery (e.g., microrebooting misbehaving components [6], killing runaway queries), there is a dearth of suitable *policies* to invoke these mechanisms automatically, efficiently, and correctly on failure. Without policies and automated ways to derive policies, humans remain in the failure-recovery loop; limiting recovery to slower human timescales rather than machine timescales.

This paper describes a research agenda to identify robust policies that work in practice and that can be learned automatically. In support of this agenda, we develop a categorization of solutions for self-healing. While most previous work on self-healing focused on either recovery from

*The author was supported in part by a grant from IBM.

Failure	Candidate fix
Deadlocked threads	Microreboot EJB [6], kill hung query
Java exceptions not handled correctly	Microreboot EJB [6]
Aging [26]	Reboot at appropriate level to reclaim leaked resources [26]
Suboptimal query plan	Update statistics for tables in query [1], re-optimize physical design (e.g., [12])
Read/write contention on table block	Repartition table to balance accesses across partitions [12]
Buffer contention	Repartition memory across various buffers [24]
Bottlenecked tier	Provision more resources to tier [25]
Source code bug	Reboot tier/service, notify administrator

Table 1. Sample failures and fixes in a multitier J2EE service

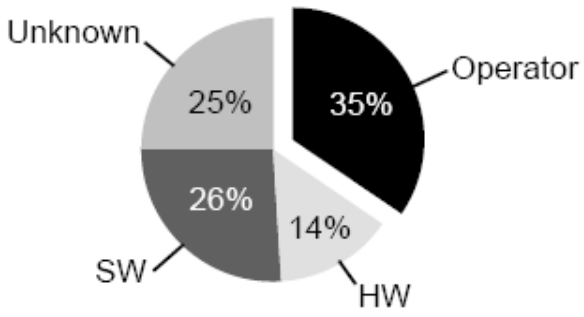

Figure 1. Causes of failures in three large multitier services (based on [18])

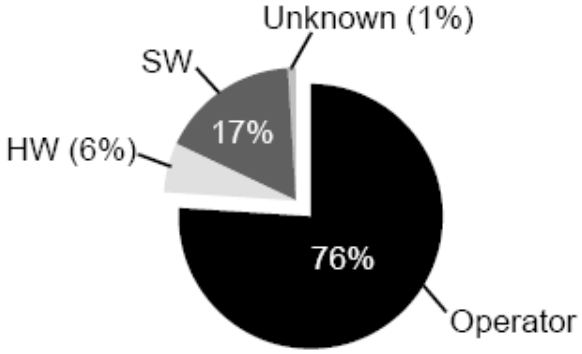

Figure 2. Time to recover from failures in three large multitier services (based on [18])

faults (e.g., [7]) or repair of dynamic performance bottlenecks (e.g., [25]), we treat performance-availability problems (PAPs) as a unit. Finally, we provide a roadmap of problems we believe need to be solved to complete the puzzle of self-healing database-centric multitier services.

Example 1 *Examples during presentation are drawn from a multitier web service named* RUBiS *[20]—an auction site written as a J2EE application [17] and modeled after eBay—running on the JBoss application server. JBoss includes an embedded web server. A MySQL server comprises the database tier. A J2EE application consists of reusable Java modules called Enterprise Java Beans (EJBs). Users interact with a J2EE application through servlets and Java Server Pages hosted on the web server, which invoke methods on the EJBs. In turn, these methods may call methods on other EJBs, submit queries or updates to the database tier, and so on.*

2 Failures in a Multitier Service

When a failure is detected in a multitier service, an effective *fix* needs to be identified and applied quickly. Table 1 lists sample failures related to hardware/software and the corresponding fixes for a multitier J2EE-based web service. *Microreboots* are fine-grained reboots of application components, usually done orders of magnitude faster than full service restarts; details are in [6]. Note from Table 1 that some failure types are specific to each tier, but others can arise in more than one tier. Furthermore, some failures (e.g., bottlenecks) can shift dynamically across tiers [25].

Hardware and software are not the only perpetrators of failures: the humans who configure, manage, and operate the service can make mistakes. [18] reports a study of service dependability, where they analyze error logs and failure-tracking databases from three large-scale multitier web services. The results are summarized in Figure 1: human operator error is clearly the most prominent source of failures.

Operator-induced failures tend to take longer to recover, as it is the human component of the system that needs to recover from the failure it has caused. Fortunately, humans can adapt and learn on their own. [18] reports how long it took to recover from the various categories of failures in the three services they studied, as shown in Figure 2.

3 Manual Vs. Automated Healing

A common approach used to identify fixes for failures today, which we term the *manual rule-based approach*, works

as follows. Domain experts create rules that map symptoms of different types of failure to specific fixes that should be applied when these symptoms are observed. Typical rules have an *if-then* format and involve thresholds, e.g., "if the miss rate in the database buffer-cache over the last 1 hour exceeds 35%, then increase the cache size." Typically, these rules are established prior to production and cannot be changed thereafter without human intervention.

Static predefined rules work well for simple services where all possible failures are known in advance or where a universal quick fix can solve most problems. However:

1. As services expand in features, size, and complexity, it becomes hard to foresee all possible failures. In such scenarios, the rules may be incomplete and could become incorrect over time since they do not evolve as the workloads or the underlying system configuration change.

2. Such rules are usually well understood on a per-tier basis only (e.g., [12]); these per-tier rules may have complex, unpredictable, and unwanted interactions in a dynamic multitier service.

3. To guarantee correct behavior in dynamic settings, rules are often made coarse-grained at the expense of quick recovery, e.g., "do a full database restart if any failure is observed."

The problems with a manual rule-based approach to fix identification motivate an *automated learning-based* approach that works as follows:

- Collect data about configuration, activity, and failures from preproduction and production runs of the service.

- Use the collected data to learn (i.e., generate or parameterize) *synopses* representing the service's behavior. Such synopses include statistical (e.g., Bayesian network, clustering) and performance (e.g., queuing network, failure-propagation paths [5]) models learned from data, as well as operators for data transformation (e.g., aggregation, feature selection) [28].

- Query the current synopses for the best fix F when a failure is observed, and apply F to the service. Check whether F recovers the service to a working state, and update the synopses with the newly-gathered data. If F fails to recover the service, then query the updated synopses for a new fix. Repeat this process until a correct fix is found or a threshold is reached when a general costly fix (e.g., full service restart or manual intervention) can be applied to recover the service.

The rest of this paper discusses different techniques to implement an automated learning-based approach.

4 Automated Identification of Fixes

4.1 Prerequisites and Caveat

Detecting failures: A self-healing service requires robust ways to detect failures as soon as they happen [4]. (TellMe Networks, a service operator, estimates that over 75% of the time they spend in recovering from an application-level failure is spent detecting the failure [7].) Some services have user-activity monitors and SLO-compliance monitors that detect potential failures by monitoring changes in service-level metrics, e.g., the number of searches done per minute. If such external metrics are not available readily, then metrics internal to the system can be monitored [7].

Universal set of fixes: One of the prerequisites for a self-healing service is a complete set of fixes for all possible failures. This requirement may seem unreasonable, but in the extreme case, a fix can be as general as alerting an administrator that manual intervention is needed, or performing a full service restart.

Detecting success/failure of fixes: After applying a fix, a self-healing system needs robust ways to determine whether the fix worked. Failure detection techniques can be used here, but care should be taken to let the service recover fully.

Automation irony: As described by psychologist J. Reason [19], human tasks are processed at one of three cognitive levels: skill-based (common repetitive tasks), rule-based (symptoms are pattern-matched to previous instances and the corresponding solution is invoked), and knowledge-based (reasoning from basic principles). Most administration and healing of systems occurs at the first two levels. Automating these tasks leaves human operators less prepared to handle tasks at the knowledge-based level, because they lose practice; as a result, the exceptional situations can be handled neither by human nor machine. The balance between intra-system visibility and automation needs to be considered carefully in a self-healing service.

4.2 Data Collection

A learning-based approach to fix identification benefits from the collection of different classes of data about service performance and failures:

- Multidimensional time-series data containing values of status variables, performance counters, and configuration parameters over time. Example attributes include CPU utilization, number of EJB calls, number of index accesses, and number of requests that violated SLOs.

- The path (control and data flow), resource utilization, and timing of requests through the multitier service.

- Data on the success and failure of attempted fixes.

Invasive Vs. noninvasive data collection: It is important to balance the benefits of collecting more data for analysis with the overhead of data collection. Furthermore, only "noninvasive" instrumentation data collected with common profiling tools—with no changes to application or system software—may be available from proprietary or legacy systems. It is typical that large multitier services contain software from many different vendors, e.g., an Apache web server, a BEA WebLogic application server, and an Oracle database server [10]. It is unlikely that such services will support a uniform invasive instrumentation framework that can collect, e.g., flow and timing data about requests flowing through all tiers of the service. As we will see in Section 4.3, techniques for fix identification differ in their data requirements.

Passive Vs. active data collection: To guarantee that the synopses learned are fairly representative of actual service behavior, it may be inadequate to rely solely on data collected through passive observations of the service in production use, e.g., by monitoring logs. Instead, the system may need to be *stimulated* actively for comprehensive data collection [22, 23]. For example, during preproduction (e.g., testing and deployment), the service can be subjected to different types and rates of workloads, and injected with various failures; while recording data about observed behavior. Active stimulation techniques have been developed to learn intercomponent dependencies and failure propagation paths (e.g., [5]). Workloads and failures for multitier services have been studied extensively—as evident from the number of surveys, benchmarks, and tools available (e.g., [2, 18, 20])—lowering the barrier to develop effective active stimulation techniques.

For clarity of presentation, we will assume that the data collected from the service is a multidimensional row-and-column time-series with schema $X_1, X_2, \ldots, X_n$. Attributes $X_1, \ldots, X_n$ are metrics of performance or failure, either measured directly from different tiers of the service or derived from measured metrics.

4.3 Building and Querying Synopses

Broadly, automated learning-based techniques for fix identification use one of two approaches:

- **Diagnosis-based** approaches first diagnose the cause of the failure, then suggest a fix based on the cause found. We discuss three diagnosis-based approaches based, respectively, on: (i) anomaly detection, (ii) correlation analysis, and (iii) performance-bottleneck analysis.
- **Signature-based** approaches do not attempt to diagnose the cause of each failure. Instead, a *classifier* is learned to associate *signatures* [11] representing unique symptoms of each failure with an effective fix for that failure. We discuss a technique that we are developing, called *FixSym*, that implements a signature-based approach.

4.3.1 Diagnosis via Anomaly Detection

Anomaly detection (e.g., [8]) seeks to identify irregularities in a service based on a characterization of its regular, or *baseline*, behavior. Three phases are involved:

- Collect data about the service.
- Establish the baseline behavior of the service and its components.
- Detect and classify anomalies, which are deviations of the current behavior from the baseline.

We give a simplified example, based on [8], to show how anomaly detection can be used for fix identification.

Example 2 *Suppose the data from the application-server tier contains attributes representing the number of times an EJB of one type calls an EJB of another type. Let N_b be a* baseline window size *and N_c be a* current window size, *with $N_c \ll N_b$. We can analyze data about EJB method invocations from the last N_b minutes to build a baseline that captures how calls from each EJB type are split across the other EJB types. Then, the EJB method invocations from the last N_c minutes can be monitored to determine when the behavior of one or more EJBs deviates significantly from the baseline behavior. (Deviation can be detected, e.g., using the χ^2 statistical test; see [4].) Such a deviation indicates potential EJB failure, so a likely fix is to microreboot the EJB [7].*

The biggest strength of anomaly detection is its ability to find fixes for new failures experienced by a service (i.e., failures encountered for the first time), as well as failures that occur rarely. Potential disadvantages include:

- Since monitoring data available from the service may be limited (see Section 4.2), anomalies can escape detection. In Example 2, invasive data collection at the level of EJB method invocations was required to detect and fix failures in the application-server tier.
- Capturing the baseline behavior of a complex multitier service is a nontrivial task. It is hard to determine a good value for the baseline window size N_b. To avoid contamination, the baseline behavior may need to be captured when the service is not experiencing significant failures. Furthermore, the baseline may need to be updated as workloads or the system configuration change.

- There is a delicate balancing act for the current window size N_c. Short N_c can lead to many false positives (spurious anomalies detected), while large N_c can lead to false negatives (undetected anomalies).

4.3.2 Diagnosis via Correlation Analysis

The correlation-based approach is similar to the anomaly-detection-based approach except in how the cause of a failure is diagnosed. Correlation analysis proceeds by identifying attributes in the data that are correlated strongly with (or predictive of) a failure-indicator attribute; illustrated next.

Example 3 *Suppose the data collected from the service contains attributes $X_1, \ldots, X_n$ representing performance metrics from various tiers, and a* failure-indicator attribute *Y (e.g., representing SLO violations). Correlation analysis—e.g., by building a Bayesian network as in [10] or by clustering the data as in [8]—recommends fixes by identifying X_i that are correlated strongly with Y. For example, if an attribute representing method invocations of an EJB is correlated with failure, then a likely fix is to microreboot the EJB. Similarly, if the number of accesses to an index is correlated with failure, then the index can be rebuilt.*

Advantages of correlation analysis include simplicity, ease of implementation, and efficiency. The biggest disadvantage is that correlation between two attributes X and Y can be inferred from data only if a reasonable number of training data records indicate this relationship. Therefore, correlation-analysis may fail to find fixes for failures not seen previously and for failures that occur rarely.

4.3.3 Diagnosis via Bottleneck Analysis

Bottleneck analysis (e.g., [12]) can diagnose failures caused by bottlenecked resources that arise frequently in multitier services [25]. Anomaly detection and correlation analysis may fail to pinpoint the root cause of such failures. However, bottleneck analysis can be done on multidimensional time-series data only if extra information is provided about the structure of the service as represented by the attributes, e.g., a relationship specifying that an attribute representing request response time is derived from other attributes representing the time requests occupy each resource.

Example 4 *The techniques proposed in [12] can determine bottlenecks in the database tier, e.g., read and write contention on a table block. A possible fix for such contention is to repartition the table and balance accesses across different partitions. The techniques in [1] can detect when transactions are bottlenecked by suboptimal query plans due to stale statistics, and recover by scheduling statistics updates.*

Procedure *FixSym*
Input: Set of candidate fixes $F = \langle F_1, \ldots, F_k \rangle$; $X_1, \ldots, X_n$ are attributes representing the performance and failure metrics collected from different tiers of the service;

```
1.  /* initialize the synopsis; domain knowledge may be used */
2.  init_synopsis(S);
3.  while (true)
4.     Wait for next failure data point f = ⟨X1=x1,..., Xk=xk⟩;
5.     fixed = false; count = 0;
6.     /* loop until a correct fix is found or threshold is reached */
7.     while (!fixed and count < THRESHOLD)
8.        /* use current synopsis to determine probable fix */
9.        probFix = suggest_fix(S, f, F);
10.       /* apply the chosen fix to the service */
11.       apply_fix(probFix);
12.       /* check if the applied fix fixed the failure */
13.       fixed = check_fix(probFix);
14.       /* update the synopsis with the new data point */
15.       update_synopsis(S, f, probFix, fixed);
16.       count = count + 1;
17.    end while
18.    if (!fixed) /* threshold exceeded, no fix found yet */
19.       Restart the service and notify the administrator;
20.       Update synopsis S with fix found by the administrator;
21.    end if
22. end while
```

Figure 3. FixSym (signature-based approach)

4.3.4 FixSym: A Signature-based Approach

FixSym is a new signature-based approach we are developing that makes two important changes to the diagnosis-based approach:

- FixSym uses all data used by the diagnosis-based approaches to learn synopses, and additionally incorporates data on the success and failure of attempted fixes.

- FixSym focuses on finding a correct and efficient fix for a failure based on information about fixes that worked previously and ones that did not work; without attempting to diagnose the root cause of the failure.

FixSym works with multidimensional time-series data that contains attributes $F_1, \ldots, F_k$ representing the result of attempted fixes, in addition to regular performance and failure metrics $X_1, \ldots, X_n$ from different tiers. FixSym uses this data to learn a synopsis S that best captures the relationship among $X_1, \ldots, X_n$ and $F_1, \ldots, F_k$ so that S can predict an effective fix given observed values of $X_1, \ldots, X_n$. Intuitively, S identifies a subset Ω of attributes in $X_1, \ldots, X_n$ that classify the symptoms of working and failed states of the service in the best manner. The values of attributes in Ω

Comparison metric	Manual	Diagnosis-based approach			Signature-based (FixSym)
		Anomaly-detect.	Correlation anal.	Bottleneck anal.	
Ability to find correct fixes	Depends on expert's knowledge	Depends on accuracy of diagnosis			Depends on historical evidence & synopsis
Run-time data requirements	Almost nil	Performance & failure metrics need coverage to pinpoint cause of failure		Fine-granularity data needed	Need symptom data, results of fix attempts
Time to find fix	Very fast	Fast	Accurate synopses can be slow [3]	Medium	Accurate synopses can be slow (Sec. 5.2)
Scalability	Poor	Easier to scale as systems expand in size, number of interconnections, and dependencies			
Adaptivity as system evolves	Hard to maintain	Online base-lining needed	Online synopsis-learning needed	Very adaptive (e.g., [25])	Online synopsis-learning needed
Ease-of-use	Poor for complex, dynamic services	Hard to tune i/p parameters	Very good	Needs domain-knowledge i/p's	Good
Handling new/ rare failures	Only if failures fore-seen & rules added	Good	Bad (needs enough representative samples)	Good	Bad (learns from attempted fixes only)

Table 2. Comparison of different approaches to automated fix identification

denote the signature of these states. S associates a successful fix with each failure signature (symptoms).

Figure 3 gives an illustration of FixSym. Each observed failure data point is input to the current synopsis to determine a fix. This fix is applied to the service and a check is made later to determine whether the fix worked or not. (These checks are discussed in Section 4.1.) The resulting data point—failure data point and result of attempted fix—is used to update the current synopsis. If the attempted fix failed, then the updated synopsis is used to determine a new fix, and FixSym proceeds as before. This process continues until a correct fix is found or a threshold is reached when a general and expensive fix (e.g., a full restart and/or notifying the administrator) is applied to get the service back to a working state.

Compared to diagnosis-based approaches, FixSym is less dependent on which performance and failure metrics are being collected. (Recall from Section 4.2 that there may be constraints on the data that can be collected from different tiers of the service.) Note that FixSym uses these metrics to represent symptom signatures. FixSym will work better with more representative data, but it can use whatever data is available. In contrast, diagnosis-based approaches need specific types of data to work. However, FixSym may fail to find a fix for a previously-unseen failure if the symptoms of this failure are very different from those seen so far.

Example 5 *Database servers maintain statistics about stored data in order to choose good execution plans for queries [1]. Unless these statistics are updated in a timely fashion, they can become out of date under heavy transactional workloads; causing failures due to suboptimal query plans. FixSym can enable self-healing under such scenarios, e.g., using a pattern of the form: "when the values of variables X_{est} and X_{act} representing the estimated and actual number of records, respectively, returned by a query Q differ significantly, update statistics on all tables accessed by Q."*

5 Research Agenda

5.1 Does One Size Fit All?

Table 2 summarizes the pros and cons of different approaches to automated fix identification that can be applied to a multitier service. It is clear that no single approach dominates all others under all scenarios. For example, the signature-based approach is good at dealing with scenarios where same workloads and failures tend to recur. However, this approach can be ineffective at finding fixes for previously-unseen or rarely-seen failures. This disadvantage could be overcome in the following ways:

- Combining the signature-based approach with one or more of the diagnosis-based approaches that find the cause of a new failure to recommend a fix.
- Enabling human operators to input their knowledge about symptoms of failures and fixes for these failures.
- Developing an active-learning approach that attempts fixes for failures in a feedback-driven loop based on previous attempts.

Note that incorporating the signature-based approach into a diagnosis-based approach can improve the overall efficiency of the latter by avoiding time-consuming diagnoses when previously-diagnosed failures occur. These observations give some interesting directions for future work:

- Perform an empirical study of the different approaches for automated fix identification, in order to generate

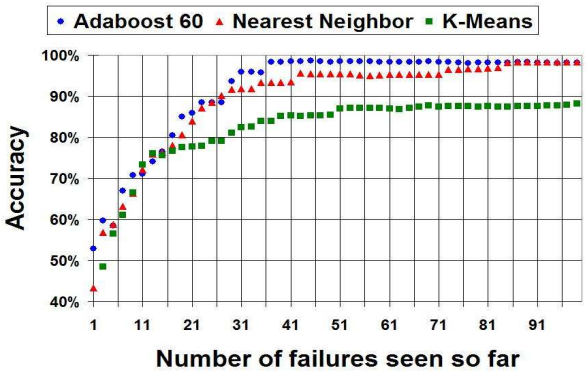

Figure 4. Synopsis comparison (accuracy)

a knowledge-base that a practitioner can use to pick the best approach based on the workload, environment, and requirements of her multitier service.

- Develop effective ways to combine the different approaches to leverage the strengths and mask the weaknesses of individual approaches.

- Develop an adaptive algorithm to pick the right combination of approaches to use automatically in any setting.

5.2 Picking the Right Synopsis

An important subproblem in automated fix identification is to pick a good synopsis from the large space of possible synopses from statistics, machine learning, and performance modeling. The main criteria for this choice is the need to balance an inherent accuracy Vs. running-time tradeoff in synopsis learning; Figure 4 and Table 3 report preliminary experimental evidence of this tradeoff.

Figure 4 represents results from an experiment where we evaluated the performance of FixSym when using three different synopsis techniques from machine-learning [28]:

1. *Nearest neighbor* is a simple machine-learning algorithm that maps a new failure data point f (recall Figure 3) to the data point f' that is closest to f among all failure data points observed so far. The fix recommended for f is the fix that worked for f'.

2. *K-means clustering* works by partitioning the failure data points collected so far into clusters based on the successful fix found for each point. A representative data point is computed for each cluster, e.g., the mean of all points in the cluster. Each new failure data point f is mapped to the cluster whose representative point is closest to f, and the corresponding fix is recommended for f. The clustering is redone after each failure is fixed successfully.

Synopsis	Time to generate 50 correct fixes	Accuracy at 50 correct fixes
AdaBoost 60	1740 seconds	98.5%
Nearest neighbor	90 seconds	95.5%
K-means	90 seconds	87%

Table 3. Synopsis comparison (running-time)

3. *Adaboost* is an *ensemble learning* technique that can produce accurate predictions by combining many simple and moderately inaccurate synopses (or *weak learners*). (See [14] for a detailed explanation of Adaboost.) The number 60 for Adaboost in Figure 4 and Table 3 is the optimal value in our setting for Adaboost's single configuration parameter, namely, the number of weak learners combined to generate the final synopsis. This number was found based on additional experiments not shown in this paper.

The experiments were conducted on a simulator for a multitier service that generates time-series data corresponding to different failed and working service states. On each failure, FixSym is invoked until a correct fix is found, as described in Section 4.3.4 and Figure 3.

The x-axis in Figure 4 shows the number of failures fixed successfully so far. Therefore, the x-axis corresponds to the number of training samples—each representing the symptoms of a failure and a successful fix—available for learning the synopsis. The y-axis shows the accuracy of the current synopsis computed on a fixed test set comprising 1000 failure states (symptoms) and correct fixes generated by the simulator. Notice that the ensemble synopsis—which is a state-of-the-art synopsis in machine learning—converges to good accuracy with much less training samples than the other synopses. Adaboost reaches 98% classification accuracy with 37 correct fixes. Nearest neighbor takes 85 correct fixes to reach 98% accuracy. K-means was inferior and only reached a final classification accuracy of about 87%.

However, Adaboost's superior accuracy comes at a significant cost in terms of running time, as illustrated in Table 3. Figure 4 and Table 3 illustrate an important challenge we must solve, namely, a self-healing service needs efficient synopsis-learning algorithms that balance the accuracy of recommended fixes with the time to generate these fixes. In [3] we report some promising work on this problem. Other synopsis-related challenges also arise in self-healing:

- **Online learning**: Unless the synopses are kept up to date efficiently as new data becomes available, accuracy can drop sharply in dynamic settings. While online learning of synopses is a hard problem, there have been some promising results recently (e.g., [29]).

- **Confidence estimates and ranking**: It becomes easy

to combine multiple approaches for fix identification, as suggested in Section 5.1, if each approach can give a confidence estimate for the fix it recommends for a specific failure; we can then rank the fixes and apply the most promising one. Synopses that give confidence estimates naturally with predicted values (e.g., Bayesian networks) are very useful in this setting.

- **Inaccurate, ambiguous, and negative data**: Recall that FixSym requires synopses to learn from unsuccessful fixes (negative training samples) in addition to successful fixes. In general, the self-healing domain poses some hard requirements on synopses, e.g., the ability to handle ambiguous and inaccurate data generated by unsuccessful fixes that were mistakenly classified as correct, and vice versa.

- **Inputing domain/prior knowledge**: Synopses will be more representative of service behavior if we can input domain knowledge during learning. Some synopses (e.g., Bayesian networks) provide easy (but narrow) ways to provide such inputs. Active stimulation during preproduction (see Section 4.2) provides many avenues to input domain knowledge. For example, a domain expert can guide which workloads to use, which types of failures to inject, and where to inject them; to generate data that can bootstrap synopsis learning.

5.3 Proactive Application of Fixes

We focused on a reactive approach to self-healing where fixes are selected and applied after failures strike. Some failures can force the service into a state where it is not possible to use or recover the service quickly [9]. In these settings, an approach where failures are predicted in advance and fixes applied proactively, can be more attractive. Such strategies need synopses that can forecast failures [3].

5.4 Control-theoretic Foundations

Since a self-healing service makes decisions based on data it observes about its own activity, the system design and implementation should consider control-theoretic issues like stability, steady-state error, settling times, and overshooting [15].

6 Conclusion

In this paper we motivated why failure recovery in database-centric multitier services is an acute problem and makes a compelling target for research on automated self-healing services. We presented a brief, but comprehensive, discussion of the important techniques—diagnosis-based and signature-based—and related issues involved in designing and implementing an automated self-healing multitier service. We showed how existing solutions fall short in several respects, e.g., being limited to a single tier, ability to handle specific types of failures only, requiring specific types of instrumentation data, and others. We presented a new signature-based technique, FixSym, that when combined with diagnosis-based techniques can outperform existing solutions for automated self-healing. Finally, we provided a roadmap of problems we believe need to be solved in this setting.

References

[1] A. Aboulnaga, P. J. Haas, M. Kandil, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated Statistics Collection in DB2 UDB. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, 2004.

[2] C. Amza, E. Cecchet, A. Chanda, S. Elnikety, A. Cox, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck Characterization of Dynamic Web Site Benchmarks, 2002. Technical Report TR02-388, Rice University.

[3] S. Babu and S. Duan. Automatic plan selection for forecasting queries, July 2006. (In submission).

[4] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, 2005.

[5] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications. In *Proc. of rd IEEE Workshop on Internet Applications*, June 2003.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proc. of the USENIX Symp. on Operating Systems Design and Implementation*, pages 31–44, 2004.

[7] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous Recovery in Componentized Internet Applications. *Cluster Computing Journal*, 9(1), Feb. 2006.

[8] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *Proc. of the 2004 Networked Systems Design and Implementation*, pages 309–322, 2004.

[9] Y. Coady et al. Falling Off the Cliff: When Systems Go Nonlinear. In *Proc. of IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, May 2005.

[10] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of the USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[11] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proc. of the ACM Symp. on Operating Systems Principles*, Oct. 2005.

[12] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in Oracle. In *CIDR*, 2005.

[13] *Yahoo! Cashes In On eBay's Outage*. http://www.ecommercetimes.com/story/545.html.

[14] Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *ICML*, Aug. 1996.

[15] J. L. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

[16] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Corp., 2001. http://www.research.ibm.com/autonomic.

[17] *Java EE at a Glance*. htpp://java.sun.com/javaee.

[18] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems*, pages 388–395, Mar. 2003.

[19] J. Reason. *Human Error*. Cambridge University Press, 1990.

[20] *Rice University Bidding System*. rubis.objectweb.org.

[21] *Business Internet Group*. The black Friday report on Web application integrity. San Francisco, CA, 2003.

[22] P. Shivam, S. Babu, and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *Proc. of the 2006 Intl. Conf. on Very Large Data Bases*, 2006.

[23] P. Shivam, S. Babu, and J. Chase. Active Sampling for Accelerated Learning of Performance Models. In *First Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, Jun 2006.

[24] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In *Proc. of the 2006 Intl. Conf. on Very Large Data Bases*, 2006.

[25] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning for Multi-tier Internet Applications. In *Proc. of 2nd IEEE International Conference on Autonomic Computing (ICAC)*, June 2005.

[26] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.

[27] *Christmas shopping crush stalls Walmart.com*. news.zdnet.co.uk/internet/0,1000000097,39284866,00.htm.

[28] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.

[29] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *DSN*, 2005.