Draco: Top-Down Statistical Diagnosis of Large-scale VoIP Networks

Soila P. Kavulya[†], Kaustubh Joshi[§], Matti Hiltunen[§], Scott Daniels[§],
Rajeev Gandhi[†],Priya Narasimhan[†]
Carnegie Mellon University[†], AT&T Labs - Research[§]

CMU-PDL-11-109
April 2011

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Large scale integrated services such as VoIP running over IP networks are the future of telecommunications. The high availability requirements of such services require scalable techniques for rapid diagnosis and localization of user-visible failures. However, state-of-the-art network event correlation techniques often produce alarms that cannot easily be correlated to customer visible impacts because they work in a "bottom-up" fashion starting from device-level events and working upwards. In this paper, we develop a contrasting "top-down" approach to problem diagnosis that starts from user visible defects such as call drops and works downwards by identifying the network level elements that are the most suggestive of the defects. Our prototype, called `Draco`*, uses statistical comparisons between good and bad system behavior to identify the underlying causes of problems without the need for any expert-provided rules or models, and without any prior training. This allows Draco to localize the causes of problems that have never been seen before. We have deployed Draco at scale for a portion of the VoIP operations of a major ISP. We demonstrate Draco's usefulness by provide examples of actual instances in which Draco helped operators diagnose service issues.*

Acknowledgements: We thank AT&T Labs for enabling this research and providing access to their `VoIP` system. We thank Mary Fernandez and Alek Remash from AT&T for their support and guidance. We also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, HP, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the NSF CAREER Award CCR-0238381 and grant CNS-0326453.

Keywords: diagnosis, distributed systems, scalable, VoIP networks

1 Introduction

In recent years, there has been a rapid convergence of previously distinct services such as data, telecom, and television onto a common platform driven by IP. Many ISPs offer a stable of "managed IP services" such as IPTV (e.g., [16]) and VoIP (e.g., [11, 17]) on top of their existing wireline IP networks, while standards such as LTE for cellular networks require the use of IP for all voice telephony (e.g., using the IP Multimedia System IMS [34]). Today, these services already serve tens of millions of users and are moving to hundreds of millions of users with the deployment of LTE. High availability and rapid troubleshooting is critical, and is a significant driver of user experience and business growth.

Managing a service platform providing such services provides numerous challenges due to the size and complexity of the system. The service platform often consists of hundreds of COTS elements of different types, developed by different manufacturers, and often managed by separate teams within the organization. A variety of different underlying problems may cause service requests (e.g., a VoIP phone call) to fail and the failures may be caused by the service elements, the underlying IP network, customer issues (e.g., misconfiguration or misuse), or combinations of the above. Failures are often intermittent (e.g., dependent on the system workload) making them harder to diagnose. Due to the size and the complexity of the system, there are often multiple independent problems occurring in the system at the same time. Typically the service operators have to rely on low-level alerts generated by the system elements (e.g., error logs, performance metric thresholds) or customer complaints. While the low-level alarms arrive close to real time and point to a specific elements, their volume may be overwhelming and they do not necessarily correspond with end-to-end service failures. The customer complaints often arrive later and do not point to a specific element or failure cause.

From a network operations standpoint, we contend that managed services provide new opportunities for network diagnosis and troubleshooting. Specifically, the traditional diagnosis based on low-level alarms lacks visibility to the system's end-to-end visibility. While network probes (e.g., [9, 25]) do provide some end-to-end visibility, they are highly sampled both in time and space. In contrast, managed IP services provide the network operator with visibility at scale across the entire stack - from user-visible application level events such as dropped calls that can be used to prioritize the most impactful network problems and minimize false positives, to low level network events such as server or route failures that can be correlated to the application level events to localize the root causes.

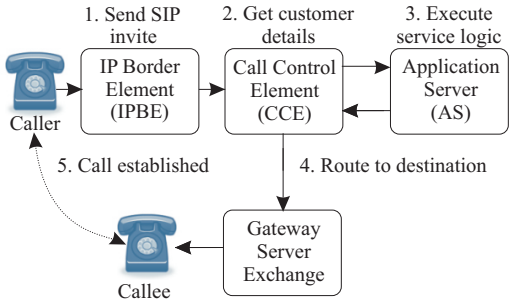
In this paper we present a statistical tool called `Draco`[1] that can perform "top-down" failure diagnosis of large managed services by starting with user-visible symptoms of a problem (e.g., dropped calls), and drilling down to identify groups of elements that are the most highly indicative of the symptoms. Draco addresses several crucial challenges. Unlike a lot of prior literature in fault localization (e.g., [4, 38]), Draco uses a domain-agnostic statistical approach that operates without the need for any user provided domain-knowledge or rules. Therefore, deployment of a new application is easy and requires little expertise. Second, Draco does not use historical norms or patterns to calibrate its internal models, and thus, can work on problems that have *never been seen before*. This capability makes the tool useful even for those systems that already deploy detection and correlation systems based on signatures, codebooks, or rules (e.g., [5, 10, 15, 38]). Third, Draco can identify failure modes that are activated under complex conditions involving multiple attributes (e.g., calls that pass through version 1 of server type A from vendor X and version 1.1 of server type B from vendor Y), making it useful for debugging interaction and protocol issues. Fourth, Draco can support large systems that have multiple independent and concurrent failures active at the same time by identifying them separately. Finally, the algorithms used by Draco are highly scalable - we have deployed Draco on a portion of network provider VoIP operations where it handles data incoming representing more than 20 million calls per day with room to grow.

[1] Draco is a genus of gliding lizards from Southeast Asia.

Table 1: A Generic Call Detail Record (CDR).

Attribute	Description
Timestamps	Call start and end times
Service	Type of service
Caller	Caller phone number and IP address
Callee	Callee phone number and IP address
Hostname	Network element name
Outcome	Successful, blocked or cut call
Defect code	Problem encountered, *e.g.*, timeout

Figure 1: A sample `VoIP` call flow for a call originating from a packet-switched network and destined to a circuit-switched network.



Draco works by extracting attributes associated with individual application-level operations (e.g., a single phone call) from application logs, and models each attribute as a binary predicate, e.g., request passed through network element X, or request originated from client IP address Y. Using an interative Bayesian algorithm, Draco builds distributions for each predicate's occurrence probability in both failed and successful operations *during the same time interval*, and identifies attributes that are the most indicative of failures as those whose failure distributions are most different from their success distribution. Subsequently, it searches for additional attributes that co-occur with the selected attributes and which further emphasize the differences between failures and successes thus allowing identification of problems due to complex conditions. When multiple problems exist in a system, Draco outputs a ranked list of indicted components and attributes.

We have deployed Draco on a portion of wireline VoIP services provided by a major ISP in order to help operators perform chronics analysis of dropped and blocked calls (defects) quickly and efficiently. In doing so, we have used call detail record (CDR) data that is already produced during the course of normal operation by the service's network elements. In this paper, we evaluate the quality of the diagnosis produced by the Draco algorithms in two ways: a) by performing fault injection experiments that use actual CDR success traces but which inject a variety of precisely controlled synthetic failure events so that ground truth is known, and b) by cataloguing actual incidents on the VoIP network that Draco was able to identify and which were subsequently confirmed by network operations personnel. Our results indicate that Draco is able to quickly identify the attributes which are indicative of failure with high levels of coverage (up to 99%) while maintaining a very low levels of false positives (up to 7%). Our experiments show that it outperforms state-of-the-art statistical techniques [22] by up to 45% when diagnosing complex problems involving multiple attributes.

Draco makes the following contributions over the existing literature in large-scale failure diagnosis:

- A model-free approach for diagnosing multiple independent, and composite problems affecting subsets of requests in a distributed system.
- A flexible automated tool that can be easily extended to incorporate additional sources of information, e.g., software versions, QOS data.
- A scalable implementation capable of handling millions of calls.
- A post-mortem analysis of failures that allows operators to discover hidden dependencies between components, and identify chronic problems by analyzing problems over multiple time windows.
- Validation on real-world `VoIP` CDR-based datasets.

The paper is organized as follows: Section 2 provides a brief background on VoIP networks and describes the VoIP dataset. Sections 3 and 4 discuss the design and implementation of our diagnostic tool. Section 5 presents the results of our fault simulation experiments, and highlights case studies in production use where Draco helped in identifying the root causes of chronic defects. Section 6 compares our results to related work. Section 7 concludes.

2 Background

Voice over Internet Protocol, *i.e.*, VoIP is an information service that delivers voice communications and enables voice convergence with other data applications and devices. The two popular protocols for VoIP are Session Initiation Protocol (SIP) and H.323. SIP is a signaling protocol whose syntax is similar to HTTP. The popularity of SIP is growing due to its ability to easily combine voice and Internet-based services. H.323 is a more mature signaling protocol typically deployed in the backbone of the VoIP network.

VoIP calls traverse different network elements based on the service type (*e.g*, 3-way calling, call forwarding, or call waiting) and whether the call is originating from or is destined to a packet-switched or circuit-switched network. Figure 1 illustrates the signaling path for a call originating from a packet-switched network and destined to a circuit switched network (*e.g*. PSTN). The caller initiates the call by sending an invite via the Internet Protocol Border Elements (IPBE) which supports SIP and H.323 signaling at the endpoints. The call is then routed to the Call Control Element (CCE) which retrieves the caller's routing and billing information and forwards the invite to the Application Server (AS). The Application Server executes the service logic, determines the routing number for the destination and routes the call back to the CCE. The CCE uses the routing number and identifies the egress gateway exchange server. The gateway exchange server routes the call from the packet-switched and to the circuit-switched network used by the callee.

Each network element (e.g., IPBE, CCE) in the VoIP system generates a call detail record (CDR) locally for each call that passes through them. There are two types of CDRs namely: a) *Success CDRs*, , and, b) *Defect CDRs* when a call fails during call setup (`blocked call`), or when a call fails during data transfer after the connection is established (`cutoff calls`).

2.1 Dataset Description

We obtained several months of VoIP call logs from a major ISP. The service provider offers a portfolio of VoIP services ranging from network hosting solutions to premises-based solutions where the customer owns and manages their Private Branch Exchange. The portion of the network provider's VoIP network that we analyzed handles millions of calls each day, and exploits redundancy to offer availability guarantees that are comparable to traditional voice networks. Service availability is measured by the number of failed calls per million attempted calls, *i.e.*, defects per million.

Each network element generates a call detail record for every call that passes through it. The call detail records consist of multiple call attributes, such as the hostname of the server, the type of VoIP service, and any error codes observed, as outlined in Table **??**. These raw CDRs from each network element are consolidated into end-to-end traces of the calls. The average size of the consolidated logs is 2.4GB/day, and each log contains between 1500-3000 unique call attributes pertinent to diagnosis. Calls follow a diurnal pattern with peak traffic between 12pm-10pm GMT which corresponds to office hours on the east coast of America as shown in Figure 2.

Operators are responsible for diagnosing failed calls due to platform problems that occur within the provider's network. Operators are also interested in problems that fall outside the scope of the provider's network, such as problems in the circuit-switched network, and issues with customer premises equipment, so that they can alert the relevant personel to fix the problem. Platform problems account for 4% of failed

Figure 2: The average number of calls per 5-minute interval follows a diurnal pattern. Call counts were obscured to preserve privacy.

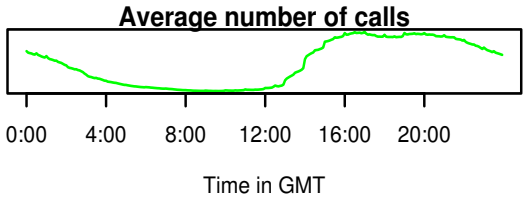


Table 2: Examples of platform problems in our logs.

	Examples of problems
1.	Configuration problem at server incorrectly routes calls from a certain make of phones.
2.	Bug in third-party software causes network element to reject calls.
3.	Application server overload leads to blocked calls.
4.	Race condition at server leads to blocked calls.
5.	Application server run out of memory.
6.	Customers use wrong codec to send faxes abroad.
7.	Debug tracing overloads servers during peak traffic.
8.	Blocked circuit identification codes on trunk group.
9.	Server crash causes brief outage.
10.	Intermittent problem at server due to software bug.

calls.

Table 2 lists examples of problems experienced. Problems observed in the data have the following characteristics: a) due to high-availability guarantees, problems are primarily "brown-outs" which affect a small subset of callers; b) problems are typically due to a combination of two or more call attributes, for example, server problems might only affect subscribers of a given service; c) due to the scale of the system, multiple independent problems may exist at any given time. We sought to develop a diagnostic tool that would effectively localize these problems, without the need for user-provided domain knowledge or tools.

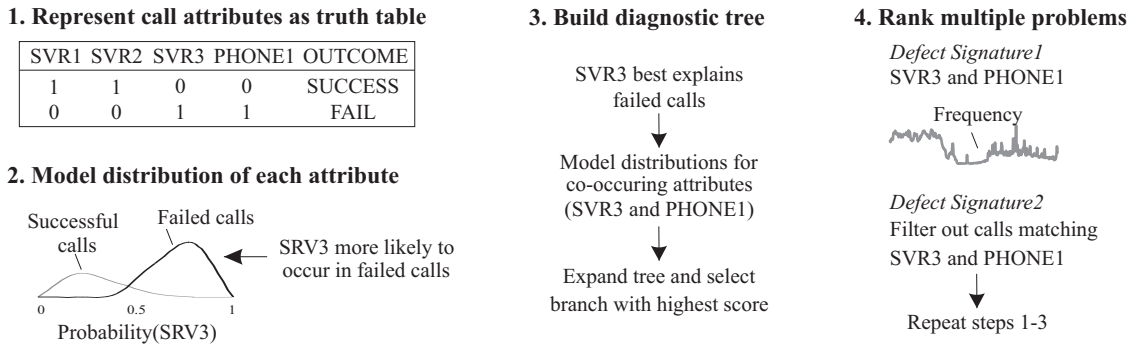
Motivating example Customers of a given service experienced difficulties making and receiving calls following a planned maintenance activity involving a configuration change. The issue prevented customers whose phones used IP addresses instead of fully qualified domain names from registering with the network. To restore service, the configuration change was backed out, and calls were redirected to the standby server.

To effectively debug this problem, operators needed to know the server, the types of customers, and the `VoIP` service affected. At present, operators primarily rely on myriads of alarms based on local events at servers, *e.g.*, CPU threshold exceeded. These alarms do not necessarily correspond to end-to-end failed calls, and can result in false alarms that mislead operators.

3 The Design of Draco

We developed `Draco` a tool that allows operators to localize problems based on real customer impact, *i.e.*, failed calls. `Draco` is capable of diagnosing unanticipated problems such as failed upgrades and system overload, and does not rely on extensive domain knowledge such as models of system behavior and pre-determined rules to localize problems. Instead, `Draco` analyzes the end-to-end flow of calls through the system, and infers the source of problems based on the success or failure of calls. `Draco` can also easily incorporate new data sources, such as software versions. We facilitate root-cause exploration via a dashboard

Figure 3: An overview of steps used by Draco's top-down, statistical diagnosis algorithm.



that offers operators different views of the diagnostic output. For example, operators can view only those problems affecting a particular customer or service.

Draco uses an iterative Bayesian diagnosis approach to identify the call attributes most correlated with failed calls. The diagnosis algorithm proceeds in four steps as illustrated in Figure 3. First, we extract call attributes from the call detail records and model them using a truth table. Second, we compute the distribution of each attribute in failed and successful calls. Third, we iteratively select attributes that best differentiate failed calls from successful calls. Lastly, we rank multiple problems that might be present in the system based on their severity.

3.1 Extract Call Attributes

Draco extracts call attributes from the consolidated end-to-end call logs and models them as a truth table as illustrated in Step 1 of Figure 3. The consolidated logs consist of stitched end-to-end call detail records listing attributes, such as hostnames, error codes, and VoIP services described in Table **??** and highlighted in the log snippet below.

```
#Snippet from logs
callno1|service3|ATTEMPT.SIP.2.3.47.487.1|
       svr1loc1|ingress.IP3-egress.svr4loc2
callno2|service6|Ans.STOP.SIP.3.0.0.0.0.term|
       svr2loc1|srv1loc5|phone1
```

We augment the attributes extracted from the consolidated logs with synthetic attributes that increase the scope of problems we can diagnose as described below:

1. *Creating wild-card attributes.* We use wild-cards to create synthetic attributes that can detect problems that affect all the servers in a high-availability cluster. For example, the synthetic attribute, $svr*loc1$, would represent the high-availability server at $location1$, consisting of the primary server, $svr1loc1$, and the standby server, $svr2loc1$.

2. *Splitting ingress and egress fields.* The call detail records contain ingress and egress fields that identify the preceeding, and the next network element in the call path. By extracting the hostnames from these fields, we can detect problems due to timeouts where the call detail record of the faulty network element is missing.

3. *Categorizing attributes.* We categorize attributes in the call detail records into defect codes, network elements, telephone numbers, and `VoIP` services. Operators can use these categories to filter diagnostic output, for example, operators might only be interested in diagnostic output that relates to network elements and telephone numbers.

Draco treats call attributes as a bag of features and is agnostic to the order in which calls traversed the network elements. To generate the truth table, we flag an attribute as `true` if it occurred in a call and `false` otherwise.

3.2 Model Attribute Distributions

We model each attribute in the truth table as a "coin toss", or binomial distribution, with a probability `p` that the attribute is `true` and `1-p` that the attribute is false. The probability `p` is computed by maintaining simple counts of how often the attribute was present in calls. We then model the conditional probability of each attribute considering only successful or failed calls using a Beta distribution. The conditional probability, listed in Eq. 1, is the probability that an event will occur given that one or more other events have occurred.

$$Pr(A/B) = \frac{Pr(A\,and\,B)}{Pr(B)} \tag{1}$$

Beta distributions are used extensively in Bayesian statistics and represent the posterior probability of a binomial distribution after observing $\alpha - 1$ successes (with probability p of success) and $\beta - 1$ failures (with probability $1 - p$ of failure). We opted for a Bayesian approach as it allows us to increase confidence in our belief about the probability distribution of failures and successes as we observe more calls. For example, we ascribe greater confidence to our belief after observing 50 failures out of 100 calls, compared to 1 failure out of 2 calls even though both scenarios have the same underlying probability, p, of 0.5.

For diagnosis, we identify attributes more likely to occur in failed calls than in successful calls by measuring the difference between the distribution of the attribute in failed and successful calls, as illustrated in Step 2 of Figure 3. We use a composite score based on the Kullback-Leibler (KL) divergence [23] to measure the difference between the two probability distributions. The KL-divergence between two probability distributions, X and Y, is the expected number of extra bits required to code samples from X using a code based on Y.

The first measure of the composite score is drawn from Liu et al [24] and measures the difference between the probability distribution of an attribute, θ, in failed calls, X_f, and the probability distribution of the attribute in successful calls, X_s, using the KL-divergence as shown Eq. 2. A large KL-divergence indicates that the attribute is more likely to occur in failed requests than in successful requests.

$$score1 = KL(p(Attribute/Failure)\,||\,p(Attribute/Success)) \tag{2}$$

The probability distributions used in the KL-divergence are Beta distributions whose parameters are computed by counting the occurrences of the attribute in failed and successful calls as described in Eq. 3.

$$\begin{aligned}
&p(Attribute/Success) = Beta(a,b)\\
&p(Attribute/Failure) = Beta(c,d)\\
&a = 1 + \sum successful\,calls\,with\,attr, \theta\\
&c = 1 + \sum failed\,calls\,with\,attr, \theta\\
&n = total\ calls,\ b = n - a + 2,\\
&d = n - c + 2
\end{aligned} \tag{3}$$

When ranking problems, Eq. 2 performed poorly with problems caused by the interaction of multiple attributes, particularly when one of those attributes occurred more frequently than the others. We augmented the initial score in Eq. 2, with the score in Eq. 4 which measures the difference between the probability distribution of failed calls given an attribute, θ, and the probability distribution of failed calls when we exclude the attribute.

$$score2 = KL(p(Failure/Attribute) \,||\, p(Failure/Not\,Attribute)) \tag{4}$$

The probability distributions used in the KL-divergence are Beta distributions whose parameters are computed by counting the occurrence and non-occurrence of attributes in failed calls as described in Eq. 5. This equation focuses solely on the failure distributions, while the initial score in Eq. 3 considers both the success and failure distributions. We added a constant, H, to Eq. 5 to prevent large biases in the score which occur when all calls with a given attribute fail. The composite score used to identify likely causes of problems is the geometric mean of Eq. 2 and Eq. 4.

$$\begin{aligned} p(Failure/Attribute) &= Beta(c, a+H) \\ p(Failure/Not\,Attribute) &= Beta(d, b+H) \end{aligned} \tag{5}$$

We illustrate the effect of the composite score using a simulated example. Suppose a configuration change in server A caused calls originating from customer C to fail. Assume that the number of failed calls that passed through server A is 3000, while the number of successful calls is 25000. The number of failed and successful calls that pass through A, and originate from customer C is 2000 and 2500 respectively. Due to other problems that might exist in the system, the total number of failed and successful calls is 5000 and 1000000 respectively.

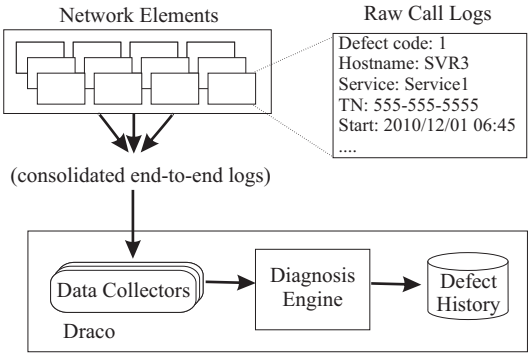
The probability of $p(Failure/A)$ is 0.11, while the probability $p(Failure/A\,and\,C)$ is 0.45. We expect that initial score, Eq. 2, would indict both A and C because $p(Failure/A\,and\,C)$ is significantly higher. However, the score for A using Eq. 2 is 789295, whereas the score for A and $C = 494491$ because A has a higher frequency than A and C. Our composite score, which is the geometric mean of Eq. 2 and Eq. 4, compensates for this by considering the $p(Failure/\ Attribute)$. The composite score yields the correct ranking with A and C ranked higher with a score of 522160, compared to A alone whose score is 281242. In this example, we used $H = 100 = 0.01\%$ of total calls.

For completeness, we list the formula for the KL-divergence between two Beta distributions in Appendix **??**. The KL-divergence can be easily computed using math libraries such as R [30].

3.3 Build Diagnosis Tree

The iterative Bayesian algorithm enables us to diagnose problems involving multiple attributes. We start by computing the composite KL-divergence score for each attribute, and select the attribute that best distinguishes failed calls from successful calls. We then filter calls that match that attribute and re-apply the composite KL-divergence score to determine whether a combination of attributes yields a higher score. For example in Figure 3, we would first select *svr*3, and then search for attributes that co-occur with *svr*3, such as *phone*1, which improve our confidence in the diagnosis. The result of this search is a diagnostic tree with depth, D, and node degree, r, which represent the maximum number of attributes that we wish associate with a given problem. We indict the branch in the tree with the highest KL-divergence score as the cause of the problem, which in our example is *svr*3 *and phone*1.

Figure 4: Architecture of diagnostic tool.



3.4 Rank Multiple Problems

We identify multiple problems that might exist in the system by excluding all calls that match the indicted branch, *e.g.*, *svr3 and phone1*, and repeating the algorithm for a fixed number of iterations to yield the top K branches that explain problems in our system. Operators can use the categories described in Section 3.1 to filter the diagnostic output, for example, operators can specify that they are interested in root causes that indict network elements or telephone numbers.

4 Draco Implementation

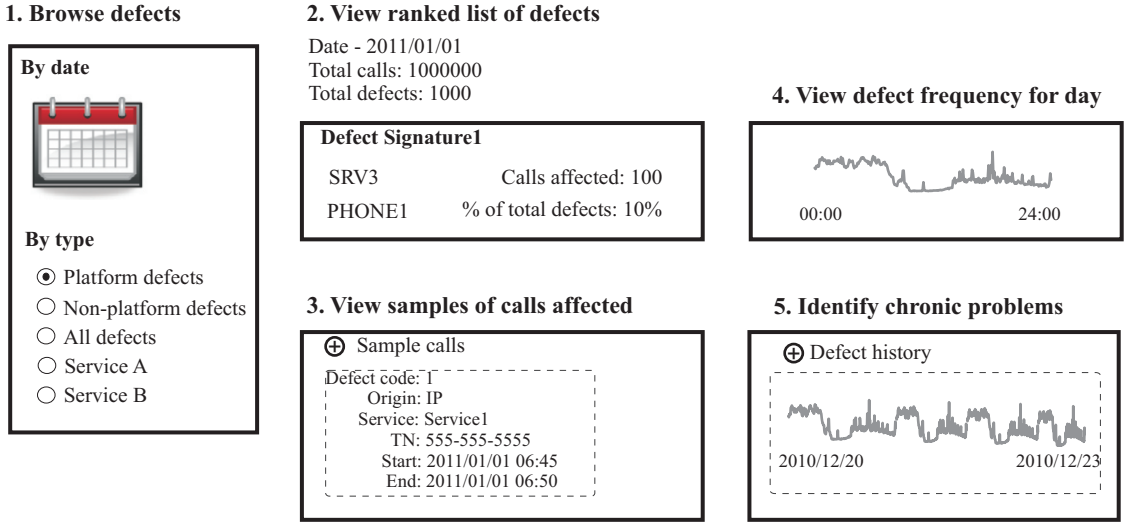
We implemented a prototype of `Draco`, written in *C*, which comprises of data collectors that accept the consolidated end-to-end call logs and extract call attributes (see Figure 4). The data collectors send data to the diagnosis engine which implements our Bayesian algorithm, and saves the diagnostic output to the defect history database. Each data collector supports one or more data formats specified using configuration files that increase the flexibility of our prototype. The diagnosis engine can receive data from concurrent input sources (e.g. multiple collectors) to reduce the amount of time needed to load data.

The diagnosis engine considers each end-to-end call record as an event. The engine collects and manages events based on a user controlled time window of length T seconds. Timestamp information in the event data is used to determine the bounds of the window; as new data is received the window progresses forward and old events are aged off. To allow more control over the aging process, particularly important for real-time use, the window is divided into n slices each of t seconds and events are aged out of the window using these smaller slices. For example, in a real-time environment a window of an hour might be divided into twelve, 300 second, intervals making the analysis on a "rolling hour," rather than a fixed hour, possible.

Performance was a primary concern while architecting the diagnosis engine as it is necessary to manage thousands of attributes from the `VoIP` system in real-time. Within the engine the data is organized hierarchically using a master data structure to manage "global" system information such as time window and slice information as well as references to attribute and event hash tables. Attributes and events are managed using independent data structures grouped as unsorted linked lists, and can also be referenced via a hash table using the attribute name or event key.

For each attribute, a series of success and failure counts are maintained based on the time slices. Managing the counts by time allows them to be adjusted as the time window rolls forward without the need to recount across all unexpired events. A reverse index which maps the attribute back to the events which list it is also maintained. The use of the reverse index allows for the quick computation of success and failure counts with regard to events that are common to a set of attributes. The reverse index is also organised using

Figure 5: Draco's dashboard allows operators to visualize signatures of problems, view samples of calls affected, and identify recurrent problems.



the time slices so that references are easily aged away as the window moves forward.

4.1 Visualization

We have deployed Draco on a portion of wireline `VoIP` services provided by a major ISP to help operators perform chronics analysis of dropped and blocked calls. Operators access the prototype via an interactive web-based user interface. Figure 5 illustrates how the web-interface facilitates the operator's workflow.

1 The operator selects the date and the types of problems they are interested in analyzing, *e.g.*, they can filter for platform problems that occurred within the provider's network.

2 Next, operators are directed to an interactive web-interface interface that ranks the top-20 problems diagnosed by `Draco` that match their filter. At the top of the page, `Draco` highlights the total number of calls on that date, as well as the number of failed calls. Below this is a ranked list of defect signatures that shows the attributes that best explain the problem. Operators can gauge the severity of the problem by checking the number of calls affected, or by checking the total percentage of defects explained by the signature.

3 Operators can gain more insight on the nature of the problem by viewing samples of calls affected via a drop-down option. The call samples display additional information from the call detail records, such as telephone numbers and call durations, that might not be captured by `Draco`'s defect signature.

4 A plot showing the frequency of the problem is displayed on the right, providing insight on the duration and severity of the problem.

5 Another drop-down option allows operators to identify chronic problems by visualizing their historical occurrence.

5 Evaluation

We validated our approach using data from fault simulation experiments, and the diverse set of real incidents from our dataset listed in Table 2. Fault simulation allowed us to investigate the effectiveness of `Draco` under a variety of precisely controlled synthetic faults. We also compared `Draco` against `Pinpoint` [22], which is a state-of-the-art diagnostic approach that relies on truth tables and decision trees to identify attributes that most indicative of failures.

We implemented `Pinpoint` using *Weka* [13], an open-source suite of machine learning algorithms written in Java. Due to the scale of data, we made the following changes to allow our unoptimized version of `Pinpoint` to run in a reasonable amount of time: first, we sampled 2% of successful calls but left failed calls intact, and second, we used the fast decision tree learner, *REPTree*, available in *Weka*. We then diagnosed problems by examining each branch in the decision tree whose leaf node classified failed calls, and ranked the branches based on number of failed calls.

We ran `Draco` on the full dataset and analyzed the top-10 problems diagnosed. We limited the width and depth of each tree generated to diagnose a single problem to 4. We specified that each problem diagnosed should contain a network element or a telephone number. We set the parameter, H, in our composite scoring function discussed in Section to 0.01% of total calls.

5.1 Fault Simulation Study

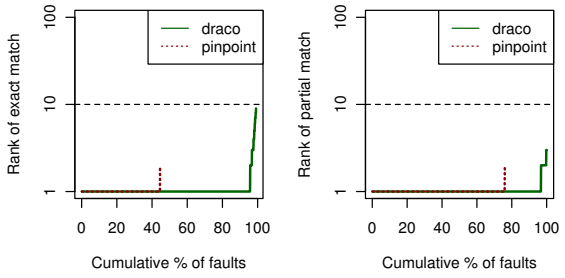
We simulated faults using one week's worth of actual CDRs of successful calls. We divided the CDRs into 1-hour intervals to yield 168 data samples. We simulated faults by changing the labels of successful calls, which contained attributes of interest, to failed calls. The simulated faults lasted for a duration of 20 minutes in each hourly sample. We sought to answer the following questions through fault simulation: a) how does varying the fault probability affect the effectiveness of diagnosis? b) how effectively can we diagnose complex failures involving multiple attributes? c) can we identify multiple concurrent faults? and d) how does noise affect the effectiveness of diagnosis?

To evaluate these scenarios, we simulated three types of faults namely: a) single faults associated one attribute, b) single faults associated with a combination of 2 to 4 attributes, and c) multiple faults, ranging from 2 to 4 concurrent faults per hour. For the single fault scenarios, we set the fault probability to 0.01, 0.15, or 0.75; whereas for multiple fault scenario, we associated each fault with a combination of 1 to 4 attributes and randomly set the fault probabilty to range from 0.01 to 1. The total number of experiments for the fault simulation study was 1176, *i.e.*, 7*168. We also investigated the effect of noise due to imperfect failure detection by failing 0.0001 of calls that matched only a subset of the relevant attributes, in addition to faults we had injected earlier. For example, if a fault involved 3 attributes, we added noise by failing 0.0001 of calls which matched only 2 or 1 of these attributes.

$$
\begin{aligned}
Avg.Precision &= \frac{\sum_{r=1}^{K}(P(r)*rel(r))}{Number\,of\,correct\,causes} \\
P(r) &= \frac{|Correct\,causes\,of\,rank\,r\,or\,less|}{r} \\
rel(r) &= 1, if\,diagnosed\,cause\,is\,relevant
\end{aligned} \tag{6}
$$

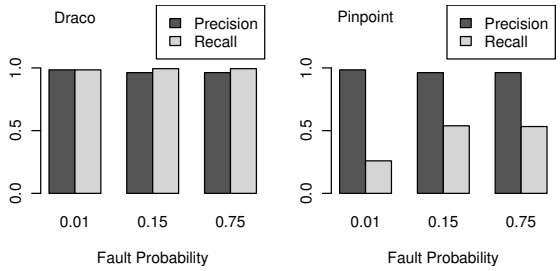
We evaluated the effectiveness of `Draco` and `Pinpoint` based on the rank of the correct cause of the fault in the diagnostic output, and computed *recall* and *mean average precision*. Recall is the fraction of injected faults that were correctly identified in the top-10 causes identified by Draco. Mean average precision is a measure of the false positive rate, which is typically used to analyze the quality of ranked search results. We computed the average precision for each fault injection experiment using Eq. 6. The mean average

Figure 6: Rank of correct root-cause for exact matches (a), and partial matches (b). We consider the top-10 culprits identified by `Draco` and `Pinpoint` when computing precision and recall.



(a) Exact matches (b) Partial matches

Figure 7: Effect of fault probability for exact matches. `Draco` performs better than `Pinpoint`, and is relatively unaffected by fault probability.



(a) Draco (b) Pinpoint

precision is the arithmetic mean of the average precision scores for the set of experiments. A high precision indicates low false positive rates.

5.1.1 Results of simulation experiments

The results of our fault simulation experiments are highlighted below.

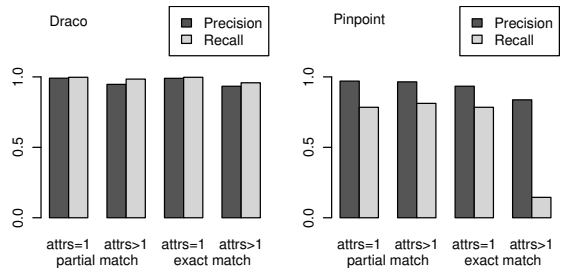
Draco correctly diagnosed 99% of single faults, with 96% precision `Draco` was effective at identifying the exact signature of the fault injected. Figure 7(a) shows the rank of the correct root-cause for single faults when we consider exact matches that contained all the affected attributes. `Draco` places the correct cause at the top of the list of likely causes 96% of the time. Some root-causes were not ranked at the top of the list because they contained one or more spurious attributes that were closely coupled with faulty attribute, for example, a server pair that was always invoked together. In contrast, the performance of `Pinpoint` was significantly lower as it correctly identified only 44% of injected faults with a precision of 96%.

We investigated the reason for `Pinpoint`'s poor recall by considering partial matches where at least one of the affected attributes is identified as shown in Figure 7(b). In this case, `Pinpoint` diagnosed 76% of injected faults with a precision of 93%. This implies that `Pinpoint`'s performance is degraded when failures involve multiple attributes because it identifies some, but not all relevant root-causes.

Draco is robust to variations in fault probability Figure 8(a) shows that `Draco` correctly identifies the root-cause of injected faults despite variations in the fault probability. `Draco`'s precision and recall remains relatively constant at 96% and 99% respectively. Figure 8(b) shows that `Pinpoint` is less effective at diagnosing problems that occur with a low probability.

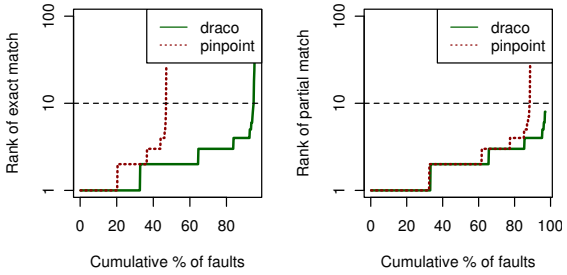
Draco correctly ranks complex failures due to component interactions Figure 9(a) shows that `Draco` is capable of diagnosing failures due to the interaction of two or more call attributes. `Draco`'s precision and recall are slightly degraded from 98% to 93%, and 99% to 96% respectively when considering exact matches for faults associated with multiple attributes. `Pinpoint` on the other hand, performs reasonably well with a precision of 96%, and a recall of 80% for faults associated with single attribute or partial matches. However, `Pinpoint`'s recall falls dramatically to 14% when the stricter criterion of exact matches is exerted for faults associated with multiple attributes as shown in Figure 9(b). We examined the decision trees generated by `Pinpoint` and we hypothesize that data pruning is eliminating relevant attributes from the tree. In addition,

Figure 8: Effect of number of attributes involved in fault. `Draco` ranks the correct cause well for both partial and exact matches.



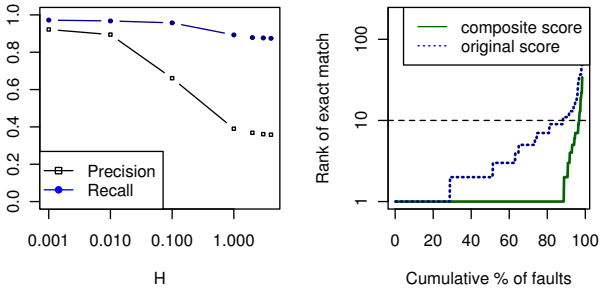
(a) (b)

Figure 9: Rank of correct root-cause for exact matches (a), and partial matches (b) when multiple faults are injected.



(a) (b)

Figure 10: Effect of noise on diagnosis. `Draco`'s precision and recall (a) are high when we set the constant H in Eq. 5 to 0.001 or 0.01. The ranking of the exact root-cause 11(b) shows that our composite score is more robust to noise than the original score proposed by [24].



(a) Draco (b) Pinpoint

the thousands of unique attributes present in our dataset, coupled with the data sampling that we performed to get pinpoint to run in a reasonable amount of time, might be causing the decision trees to suffer from the "curse of dimensionality".

Draco is effective at diagnosing multiple concurrent faults We evaluated the effectiveness of `Draco` and `Pinpoint` at diagnosing multiple concurrent faults. The number of faults ranged from 2 to 4 and the number of attributes affected varied from 1 to 4. Draco correctly identified 92% of injected faults within the top-4 likely causes as shown in Figure 10(a). `Pinpoint` correctly identified only 46% of injected faults within the top-4 likely causes when considering exact matches. For partial matches, `Pinpoint`'s performance improves to cover 85% of injected faults identified in the top-4 likely causes as shown in Figure 10(b).

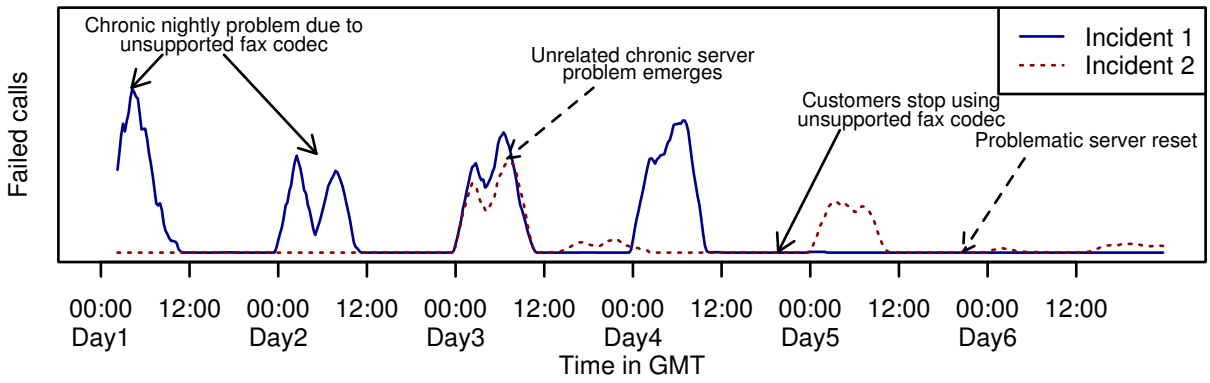
Draco is robust to noise We also investigated the effect of noise on the effectiveness of diagnosis. We had observed that the effectiveness of the scoring function proposed by Liu et al. [24] decreased for faults due to a combination of multiple attributes, particularly when noise causes one of those attributes to occur slightly more frequently than the others. We developed a composite score, described in Section 5, which compares the distribution of failures in calls where an attribute is present, against the distribution of failures in calls where the attribute is absent.

We added noise to our fault simulation dataset by failing 0.0001 of calls that matched only a subset of

Table 3: Average data load and diagnosis times.

	Log size	**Load time**	**Diagnosis time**
Pinpoint simulated-1hr	271±234MB	74±65s	16±34s
Draco simulated-1hr	271±234MB	8±7s	4±8s
Draco real-1day	2.4±1GB	7±1min	8±2min

Figure 11: Chronic defects in the production system diagnosed using `Draco`. Failure counts were obscured to preserve privacy.



the relevant attributes. We then analyzed the effect of varying the constant, H, in Eq. 5 that prevent large biases in the score when all calls with a given attribute failed. For example, if an attribute only occurred in a single call and that call failed, Eq. 5 would incorrectly assign a very high score if the bias H=0. Figure 11(a) shows the effect of varying H from 0.1% of total calls to 400% of total calls. We observed that H=0.001 and H=0.01 yielded high precision and recall for faults due to a combination of attributes. As we increase H beyond 0.01, the score in Eq. 5 reduces and the performance of the algorithm becomes similar to the original scoring function proposed by Liu et al. [24].

Figure 11(b) compares the performance of `Draco`'s composite scoring function against the original score proposed by [24] for noisy data. We used H=0.01 of total calls. `Draco` performs better at ranking the exact root-cause than the original scoring function. The scoring function identified partial matches where at least one of the affected attributes when the data was noisy.

Draco is scalable `Draco` takes on average approximately 15 minutes to load and analyze one day's worth of data as listed in Table 3. This demonstrates that `Draco` is capable of analyzing millions of attributes in real time. In addition, we observed that `Draco` analyzes the data from the fault simulation study about 200% faster than `Pinpoint` since we sampled 2% of sucessful calls when running `Pinpoint`, and `Draco` runs 4 times faster. However, the comparison is not entirely fair as we are using an unoptimized implementation of `Pinpoint` written in Java, compared to an optimized version of `Draco` written in C. The complexity of our algorithm is $KN*D^r$, where K is the number of iterations, N is the number of calls, D is the depth of the tree, and r is the degree of nodes in the tree. The magnitudes of D and r are determined by the maximum number of attributes associated with a failure in the system. Based on our experience, we expect the magnitude of D and r to be less than 5, thereby yielding reasonable performance as the system scales.

Table 4: Rank of correct root-cause based on offline verification of known problems .

	Examples of problems	**Rank**
1.	Configuration problem at server incorrectly routes calls from a certain make of phones.	3
2.	Bug in third-party software causes network element to reject calls.	1
3.	Application server overload leads to blocked calls.	1
4.	Race condition at server leads to blocked calls.	10
5.	Application server run out of memory.	1
6.	Customers use wrong codec to send faxes abroad.	1-4
7.	Debug tracing overloads servers during peak traffic.	15
8.	Blocked circuit identification codes on trunk group.	1-8
9.	Server crash causes brief outage.	-
10.	Intermittent problem at server due to software bug.	-

5.2 Case Studies

During the last six months, Draco has been used to assist in identifying the root causes of chronic defects, that is, low impact defects that persist over periods of days or weeks. The team addressing chronic defects tracks defects based on defect codes generated by the network elements. Some examples of the successful use of Draco include:

Incident 1 Repeating increase in the number of defects during night hours was observed associated with defect code $stop.ip-to-pstn$.102.0.504.102 illustrated in Figure 11. Draco identified two different (business) customers as being associated with the bulk of the defects. While these customers accounted for large share of total defects, the defect rate observed by the customers were a fraction of one percent. After further analysis, it was determined that these two customers were attempting to send faxes overseas using unsupported codecs during US night time. After the customers were notified the daily defect count associated with this defect code decreased by 50%.

Incident 2 Draco identified an independent problem with a specific network element that occurred concurrently with incident 1 (see Figure 11), and accounted for over 50% of the remaining defects when failures due to Incident 1 were excluded. Again, overall only a fraction of one percent of the calls passing through this element were failing making the problem harder to identify. After the element was reset, the total number of daily defects associated with this defect code was reduced by over 50% and this element was no longer implicated in Draco output.

Incident 3 An increase in failure rate during business hours was observed for defect code attempt.pstn-to-ip.41.0.-.-. Draco identified a trunk group as being associated with up to 80% of these defects. At peak, 2-3% of the calls passing this trunk group would fail. Further analysis revealed 2 blocked CICs (Circuit Identification Codes) on the trunk group and as a result the problem would only affect calls assigned to these blocked CICs (in a round robin manner). Unblocking these CICs eliminated the problem and reduced total defects associated with this code by over 50%.

Offline verification of known problems We verified the performance of `Draco` using logs from the known incidents listed in Table 4 showed that `Draco` correctly diagnosed 8 out of 10 incidents. We ranked 5 out of the 10 problems as the top problem with a rank of 1. The ranking of chronic problems such as No. 6 and 8 varied. The other faults that were not ranked at the top had a lower severity. The incidents that we did not detect were a brief outage caused by a server crash and an intermittent problem due to a server

bug. Each of these incidents lasted under one minute and their occurrence was dwarfed by other failures that happened during the day; unfortunately this is unavoidable when running Draco in an offline mode to analyse data over a large time window. We are implementing an online version of Draco that is capable of diagnosing problems that occur in very brief intervals of time.

5.3 Discussion

Draco facillitated the diagnosis of acute and chronic problem in the VoIP network. Unlike, the low-level alarms, *e.g.*, CPU threshold exceeded, that are generated by network device that have high volumes and high false positive rates, the problems identified by Draco correspond to failures affecting customers. Draco's dashboard allows operators to determine the severity of the problem and track the progression of the problem over time. This allows operators to better target their recovery efforts.

The domain-agnostic statistical approach employed by Draco allows us to diagnose problems that have not been seen before. Draco is also capable of diagnosing failures that occur due to complex conditions involving multiple attributes. Draco is tailored towards diagnosing "brown-outs" where a subset of the calls flowing through the system fail. However, Draco cannot localize problems if the system experiences a catastrophic event that causes the failure of the entire system.

6 Related Work

A significant body of research has addressed diagnosis in distributed systems since the 1960s, when Preparata et al. [29] proposed the PMC model to identify faulty components by collating results of diagnostic tests across a distributed system. This research can be broadly categorized as *statistical*, *model-based*, and *knowledge-based* diagnostic approaches.

Statistical These approaches summarize and interpret empirical data using techiques such as correlation, histogram comparison and probability theory. Statistical approaches, unlike model-based approaches, are not capable of predicting future behavior. Draco uses a statistical approach that draws upon probabilistic concepts such as conditional probability [33] and the Kullback-Leibler divergence [23]. Draco also borrows from Bayesian software debugging [24] which infers components that are more closely associated with failures. These approaches require little domain knowledge can easily incorporate new components if the level of monitoring increases. Draco improves the scoring function for diagnosing problems due to multiple interacting components, and scalably handles thousands of attributes through its greedy, iterative approach to attribute selection.

Netmedic [20] combines dependency analysis and correlations in state perturbations across processes to localize problems. They focus on process or node-level problems and do not address problems involving multiple interacting components. Giza [25] couples a hierarchical heavy hitter detection mechanism with correlation analysis to discover causal relationships between symptom and diagnostic events. Giza relies on alarms to be generated at each component of interest, whereas Draco uses end-to-end testing where the success or failure of a call serves as an alert.

Other statistical approaches analyzing the distribution of request times across networked components [1], detecting outliers using peer-comparison [26, 28, 36], and analyzing logs [37] to identifier outlier error messages.

Model-based These approaches define a mathematical representation of their system and then test their system to see if it conforms. Model-based approaches are predictive, to varying extents, as they can extrapolate behavior for previously unseen situations. Regression models [8, 35] learn the relationship between

resource consumption and application behavior, and can predict response times for variations of known transaction types. However, the models need to be re-trained to cope with new transaction types, and, unlike Draco, they do not address multiple independent failures.

Some approaches generate graphical models of how problems propagate through the system [2, 6, 19, 21], and exploit this knowledge to infer the source of the problem. The inference graphs generated may incorrectly indict widely shared components if the shared dependency is not captured by the model. Other graphical approaches [18, 32] model how successes (i.e., probes/monitor paths) propagates. In contrast, Draco compares the distribution of components in failed and successful calls, and filters out widely shared components that appear frequently in both successful and failed calls.

Machine-learning approaches learn models of normal and abnormal system behavior using large volumes of labeled training data. Some approaches learn signatures of recurring problems [5,10,12] by correlating performance metrics, such as CPU and memory usage, with failed requests. However, these approaches do not fare well at automatically localizing problems that have not previously been diagnosed. Other approaches collect end-to-end traces of client requests in a manner similar to Draco, and exploit clustering [3], or decision trees [7, 22] to identify resource-usage profiles or components that are highly correlated with failed requests. These approaches suffer from the "curse of dimensionality" in which extensive amounts of training data is required to cope with the thousands of attributes present in the VoIP system.

Knowledge-based These approaches rely on expert knowledge, typically expressed as a set of pre-programmed rules, to diagnose problems. Chopstix [4] and Hauswirth et al's "vertical profiling" [14] use a rule-based approach based on the correlation of performance counters on individual nodes to detect problems. Yemini et al. [38] uses codebooks to describe the relationship between problem symptoms and corresponding root-causes. Commercial tools, such as IBM Tivoli Enterprise Console [15] and HP Operations Manager [27], allow users to augment rules to their existing algorithms. Pip [31] allows programmers to embed expectations about the system behavior in the source code. Pip then detects problems by comparing actual behavior against expected behavior. Unlike Draco's inference-based approach, knowledge-based approaches are unable to tackle unseen problems. Conflicts between rules may be difficult to identify and result in unwanted side effects.

7 Conclusion

In this paper, we developed a methodology to perform network diagnosis of integrated services in a top-down manner starting with user-visible symptoms such as dropped calls, and working downwards to network level causes. We presented Draco, a statistical fault diagnosis tool that implements this approach. Draco is a scalable tool that can analyze datasets consisting of tens of millions of records. It computes distributional differences between attributes present in successful user interactions and attributes present in defective interactions to highlight sets of features that are the most indicative of defects. Due to its statistical nature, Draco does not require extensive domain expertise in the form of rules or models, thus making it easy to port to multiple applications. By comparing successes and failures over the same window of time, it avoids the need for separate learning passes, and can thus diagnose problems that have never been seen before. We have deployed Draco for performing chronics analysis on the VoIP portfolio of a major ISP. We show using both simulated and actual incidents on the network that Draco can produce useful diagnoses with a high precision and recall that significantly improves on the current state-of-the-art.

References

[1] 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*, pages 74–89, Bolton Landing, NY, Oct 2003.

[2] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 13–24, Kyoto, Japan, Aug. 2007.

[3] 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*, pages 259–272, San Francisco, CA, Dec 2004.

[4] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 103–116, San Diego, CA, December 2008.

[5] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *European conference on Computer systems (EuroSys)*, pages 111–124, Paris, France, April 2010.

[6] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *International Symposium on Integrated Network Management*, pages 377–390, Seattle, WA, May 2001.

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

[8] 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*, pages 452–461, Anchorage, Alaska, June 2008.

[9] L. Ciavattone, A. Morton, and G. Ramachandran. Standardized active measurements on a tier 1 ip backbone. *IEEE Communications Magazine*, pages 90–97, July 2003.

[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*, pages 105–118, Brighton, United Kingdom, Oct 2005.

[11] C. Corp. Xfinity digital voice. http://www.comcast.com/Corporate/Learn/DigitalVoice/digitalvoice.html.

[12] S. Duan and S. Babu. Guided problem diagnosis through active learning. In *IEEE International Conference on Automatic Computing*, pages 45–54, Chicago, IL, June 2008.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, July 2009.

[14] M. Hauswirth, A. Diwan, P. Sweeney, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 251 – 269, Vancouver, BC, Canada, Oct. 2004.

[15] IBM. Tivoli enterprise console, 2010. http://www.ibm.com/software/tivoli/products/enterprise-console.

[16] A. Inc. Uverse. http://www.att.com/u-verse/.

[17] Q. C. I. Inc. Qwest voip long distance and toll-free service. http://www.qwest.com/business/products/products-and-services/voip-adv-voice/voip-ip-ld-toll-free.html.

[18] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting. Automatic model-driven recovery in distributed systems. In *IEEE Symposium on Reliable Distributed Systems*, pages 25–38, Orlando, Florida, October 2005.

[19] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: a tool for failure diagnosis in ip networks. In *ACM Workshop on Mining Network Data (MineNet)*, pages 173–178, Philadelphia, PA, Aug. 2005.

[20] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 243–254, Barcelona, Spain, August 2009.

[21] 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*, pages 185–198, Beijing, China, October 2007.

[22] 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*, 16(5):1027– 1041, Sep 2005.

[23] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22:79–86, March 1951.

[24] C. Liu, Z. Lian, and J. Han. How bayesians debug. In *International Conference on Data Mining*, pages 382–393, Hong Kong, China, Dec. 2006.

[25] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large iptv network. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 231–242, Barcelona, Spain, August 2009.

[26] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, page 88, Tampa, FL, November 2006.

[27] H. Packard. HP operations manager, 2010. http://www.managementsoftware.hp.com.

[28] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Blind Men and the Elephant: Piecing together Hadoop for diagnosis. In *International Symposium on Software Reliability Engineering (ISSRE)*, Mysuru, India, Nov. 2009.

[29] F. P. Preparata, G. Metze, and R. T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computing*, EC-16(6):848–854, December 1967.

[30] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[31] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[32] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik. Real-time problem determination in distributed systems using active probing. In *IEEE/IFIP Network Operations and Management Symposium*, pages 133–146, Seoul, South Korea, April 2004.

[33] S. M. Ross. *Introduction to Probability Models*. Academic Press, 9th edition, December 2006.

[34] M. Sauter. *Beyond 3G - Bringing Networks, Terminals and the Web Together: LTE, WiMAX, IMS, 4G Devices and the Mobile Web 2.0*. Wiley Publishing, 2009.

[35] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 71–84, Boston, MA, May 2005.

[36] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing Logs as State Machines. In *USENIX Workshop on Analysis of System Logs*, San Diego, CA, Dec. 2008.

[37] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *ACM Symposium on Operating Systems Principles*, pages 117–132, Big Sky, MT, October 2009.

[38] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *Communications Magazine, IEEE*, 34(5):82–90, May 1996.