2011

Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems

Karthik Nagaraj
Purdue University, Knagara@cs.purdue.edu

Jennifer Neville
Purdue University, neville@cs.purdue.edu

Charles Killian
Purdue University, ckillian@cs.purdue.edu

Report Number:
11-020

Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems

Karthik Nagaraj, Charles Killian, Jennifer Neville
Purdue University
{*knagara, ckillian, neville*}*@cs.purdue.edu*

Abstract

Diagnosis and correction of performance issues in modern, large-scale distributed systems can be a daunting task, since a single developer is unlikely to be familiar with the entire system and it is hard to characterize the behavior of a software system without completely understanding its internal components. This paper describes DISTALYZER, an automated tool to support developer investigation of performance issues in distributed systems. We aim to leverage the vast log data available from large scale systems, while reducing the level of knowledge required for a developer to use our tool. Specifically, given two sets of logs, one with good and one with bad performance, DISTALYZER uses *machine learning* techniques to compare system behaviors extracted from the logs and automatically infer the strongest associations between system components and performance. The tool outputs a set of inter-related event occurrences and variable values that exhibit the largest divergence across the logs sets and most directly affect the overall performance of the system. These patterns are presented to the developer for inspection, to help them understand which system component(s) likely contain the *root cause* of the observed performance issue, thus alleviating the need for many human hours of manual inspection. We demonstrate the generality and effectiveness of DISTALYZER on three real distributed systems by showing how it discovers and highlights the root cause of six performance issues across the systems. DISTALYZER has broad applicability to other systems since it is dependent only on the logs for input, and not on the source code.

1 Introduction

Modern, large-scale distributed systems are extremely complex, not only because the software in each node in the distributed system is complex, but because interactions between nodes occur asynchronously, network message delays and orderings are unpredictable, and nodes are in heterogeneous environments, uncoupled from each other and generally unreliable. Compounding this is the fact that the software is typically developed by teams of programmers, often relying on external components and libraries developed independently, such that generally no one developer is fully aware of the complex interactions of the many components of the software.

Transmission [30] and HBase [18] exemplify the scale of this type of software development. Transmission is an open-source implementation of BitTorrent. In 2008, after three years of development, it became the default BitTorrent client for Ubuntu and Fedora, the two most popular Linux distributions. In the last two years alone, 15 developers committed changes to the codebase, not counting patches/bugs submitted by external developers. HBase is an open-source implementation of BigTable [4], depending on the Hadoop [17] implementation of the Google File System [15]. HBase has grown very popular and is in production use at Facebook, Yahoo!, StumbleUpon, and Twitter. HBase's subversion repository has over a million revisions, with 21 developers from multiple companies contributing over the last two years.

Given the activity in these projects, it not surprising that, in our experiments, we observed performance problems, despite their mature status. In systems with many independent developers, large user-bases with differing commercial interests, and a long history, diagnosis and correction of performance issues can be a daunting task—since no one developer is likely to be completely familiar with the entire system. In the absence of clear error conditions, manual inspection of undesirable behaviors remains a primary approach, but is limited by the experience of the tester—a developer is more likely to ignore occasional undesirable behavior if they do not have intimate knowledge of the responsible subsystems.

Recent research on distributed systems has produced several methods to aid debugging of these complex systems, such as execution tracing [11, 14, 26], replay debugging [13], model checking [20, 24], live property testing [21], and execution steering [33]. However, these methods either require either extensive manual effort, or are automated search techniques focused on discovering specific *error* conditions.

To address the challenge of debugging undesirable behaviors (i.e., *performance* issues), we focus on comparing a set of baseline logs with acceptable performance to another set with unacceptable behavior. This approach aims to leverage the vast log data available from complex, large scale systems, while reducing the level of knowledge required for a developer to use our tool. The state-of-the-art in debugging the performance of request flows [1, 3, 5, 28] also utilizes log data; however, in contrast with this previous work, we focus on analyzing a wider range of system behaviors extracted from logs. This has enabled us to develop an analysis tool applicable to more than simply request processing applications. Other work in identifying problems in distributed systems from logs [32] is restricted to identifying anomalous local problems, while we believe that poor performance commonly manifests from larger implementation issues.

We present DISTALYZER, a tool to analyze logs of distributed systems automatically through comparison and identify components causing degraded performance. More specifically, given two sets of logs with differing performance (that were expected to have equivalent performance), DISTALYZER outputs a summary of event occurrences and variable values that (i) most diverge across the sets of logs, and (ii) most affect *overall* system performance. DISTALYZER uses *machine learning* techniques to automatically infer the strongest associations between system components and performance. Contributions of this paper include:

- An assistive tool, DISTALYZER, for the developer to investigate performance variations in distributed systems, requiring minimal additional log statements and post processing.
- A novel algorithm for automatically analyzing system behavior, identifying statistical dependencies, and highlighting a set of interrelated components likely to explain poor performance. In addition to the highlighted results, DISTALYZER also provides interactive exploration of the extended analysis.
- A successful demonstration of the application of DISTALYZER to three popular, large scale distributed systems–TritonSort [25], HBase & Transmission–identifying the root causes of six performance problems. In TritonSort, we analyzed a recently identified performance variation—the TritonSort developers surmised DISTALYZER could have saved them 1.5 days of debugging time. In follow-up experiments on Transmission and HBase, once we fixed the identified problems, their performance was boosted by 45% and 25% respectively.

The rest of this paper is organized as follows: Sec. 2 describes our model of logs. We describe design and implementation in Sec. 3 and 4 respectively, case studies in Sec. 5, related work in Sec. 6, and conclude in Sec. 7.

2 Instrumentation

DISTALYZER derives its analysis based on the data extracted from logs of distributed systems executions. Hence, we describe the process of obtaining and preparing the logs for analysis, before the actual design in § 3. Applying our modeling to the logs of systems requires that some amount of its meaning is provided to DISTALYZER. Inherently, this is because we are not seeking to provide natural language processing, but instead to analyze the structure the logs represent. Xu *et al.* [32] have considered the automatic matching of log statements to source code, which requires tight coupling with programming languages to construct abstract syntax trees. In contrast, DISTALYZER aims to stay agnostic to the source code by abstracting the useful information in the logs. We describe this in more detail below.

The contributions of logging to debugging are so deeply ingrained that systems typically are not successful without a significant amount of effort expended in logging infrastructures. DISTALYZER assumes that the collection of logs has not affected the performance behaviors of interest in the system. This is a standard problem with logging, requiring developers to spend much effort toward efficient logging infrastructures. Logging infrastructures range from free text loggers like *log4j* [22], to fully structured and meaningful logs such as Pip [26] and XTrace [11]. Unfortunately, the common denominator across logging infrastructures is not a precise structure indicating the meaning of the logs.

Consider Pip [26], a logging infrastructure providing rich meaning. Pip provides log annotations which indicate the beginning and ending of a task, sending and receiving of messages, and a separate log just as an FYI (a kind of catch-all log). Every log also indicates a path identifier that the log belongs to. With this amount of instrumentation, it is possible to construct path trees, which show dependencies between tasks within paths. This kind of instrumentation has been leveraged by Sambasivan *et al.* [28] to be able to compare the path trees in systems logs. Unfortunately, this detail of logging is both not sufficient (it does not capture the instances of value logging, and does not adequately handle tasks which belong to multiple flows), and not widely available. A more commonly used logging infrastructure, *log4j*, provides a much more basic scheme - logs are associated with a "type," timestamp, priority, and free text string. It then remains to the developer to make sense of the logs, commonly using a brittle set of log-processing scripts.

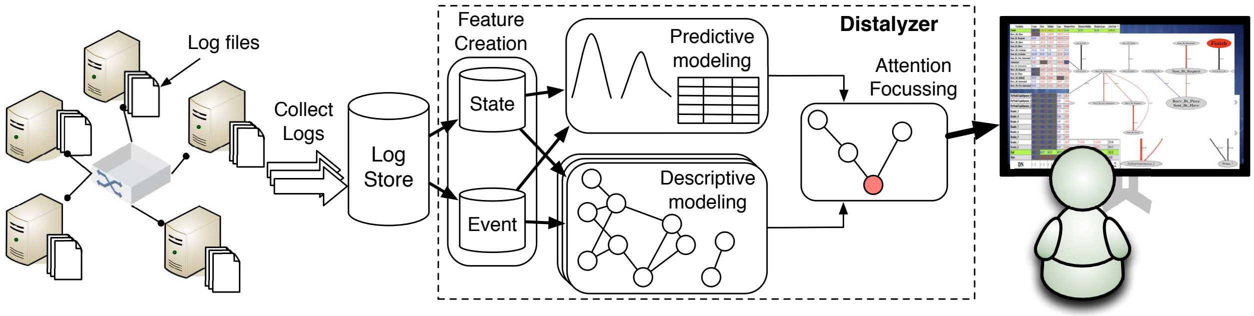


Figure 1: Four-step log comparison process in DISTALYZER leading up to a visual interface

As a compromise between fully meaningful logs and free-text logs, we work to find a middle-ground, which can be applied to existing logs without onerous modifications to the system being investigated. Our insight is that logs generally serve one of two purposes: *event log messages* and *state log messages*.

Event log message. An event log message indicates that some event happened at the *time* the log message was generated. Most of the logging for Pip falls into this category, in particular the start or end of tasks or messages. Other examples of such logs include logs that a particular method was called, branch of code taken, etc. These logs are often most helpful for tracing the flow of control with time between different components of the system.

State log message. A state log message indicates that at the time the log message was generated, the *value* of some system variable is as recorded. Typically, a state log message does not imply that the value just became the particular value (that would instead be an event log message), but merely that at the present time it holds the given value. State log messages are often printed out by periodically executed code, or as debugging output called from several places in the code. State log messages are often most helpful for capturing snapshots of system state to develop a picture of the evolution of a system.

Distinguishing state and event log messages is an important step, that allows us to tailor our modeling techniques to treat each in kind. We will commonly refer to these values as the *Event Variables* and the *State Variables*. A practical artifact of this approach is that we can simply use the system's existing infrastructure and logging code to generate logs, and then write a simple script to translate logs into state and event log messages in a post-processing step (§ 4.1). Adopting this approach makes it much easier to apply DISTALYZER to a wide range of existing systems, and avoids extra logging overheads at runtime. Additionally, it is possible to integrate our logging library into other logging infrastructures or code-generating toolkits that provide a distinction between state and event log messages, so that no post-processing phase would be required. Furthermore, this strategy allows the incorporation of external system activity monitoring logs for a richer analysis.

3 Design

This section presents the design of DISTALYZER, an operational tool capable of identifying salient differences between sets of logs, with the aim of focusing the attention of the developer on aspects of the system that affect overall performance and significantly contribute to the observed differences in behavior. DISTALYZER involves a multi-step analysis process as shown in Figure 1. The input to the workflow is a set of logs with tagged event and/or state log messages, separated by the developer into two classes C_0 and C_1 with different behavior on some performance metric P (*e.g.*, runtime). The choice of performance metric can be easily determined from Service Level Agreement (SLA) metrics. Some example choices for classes C_0 and C_1 are as follows:

- Different *versions* of the same system
- Different *requests* in the same system
- Different *implementations* of the same protocol
- Different *nodes* in the same run

Given the inputs, DISTALYZER use machine learning methods to automatically analyze the data and *learn* the salient differences between the two sets of logs, as well as the relationships among the system components. As a result of this analysis, DISTALYZER identifies and presents to the developer (for investigation) the most notable aspects of the system likely to contain the *root cause* of the observed performance difference. Specifically the system involves the following four components:

1. **Feature Creation**: A small set of event/state *features* are extracted from each log instance in both classes to make the data more amenable for automated analysis.
2. **Predictive Modeling**: The event/state variables are analyzed with statistical tests to identify which fea-

tures *distinguish* the two classes of logs. This step directs attention to the system components that are the most likely causes of performance difference.

3. **Descriptive Modeling**: Within a single set of logs (*e.g.*, C_0), the relationships among event/state variables are learned with dependency networks [19]. The learned models enhance the developer's understanding of how aspects of the system interact and helps to discard less relevant characteristics (*e.g.*, background operations, randomness).
4. **Attention Focusing**: The outputs of steps 2 and 3 are combined to automatically identify a set of interrelated variables that most diverge across the logs and most affect overall performance (*i.e.*, P). The results are graphically presented to the developer for investigation, not only indicating where the developer should look for performance bugs, but also insight into the system itself, obviating the developer the need to be an expert at all system interactions.

We describe each of these components in more detail below. We note that the user need not be aware of the internals of the statistical or machine learning techniques, and is given an understandable graphical representation of the variables likely to contain the root cause of performance differences. With a clear understanding of the root cause, the developer can spend more time on finding a good fix for the performance bug. In Section 5 we present results of using DISTALYZER to analyze TritonSort (different versions), BigTable (difference requests), and BitTorrent (different implementations).

3.1 Feature creation

The workflow starts with extracting a handful of feature summaries from the logs. The input is two sets of logs C_0 and C_1, classified by the developer according to a performance characteristic of interest P. For example, the developer may be interested in diagnosing the difference between slow (C_0) and fast (C_1) nodes based on total runtime P. DISTALYZER performs offline analysis on the logs, after they have been extracted from the system execution. We refer to each log instance as an *instance*. We assume that a similar test environment was maintained for both sets of logs, including workloads, physical node setup, etc. However, it is not necessary that both classes contain the same number of occurrences of a variable. Also, the two classes are not required to have disjoint (*i.e.*, non-overlapping) values for P.

DISTALYZER begins by calculating features from variables extracted from the log instances. Each instance i contains many event and state log messages, which first need to be summarized into a smaller set of summary statistics before analysis. The intuition behind summarizing is that this reduces the complexity of the system execution to a handful of *features* $\mathbf{X_i}$, that are much less prone to outliers, randomness and small localized discrepancies in log statements. Since DISTALYZER aims to find the source of *overall* performance problems (and not localized problems as in [32]), a coarse-grained set of features provides a better representative of each instance than every single value within that instance. A smaller set of features are less of a burden on the developer, and a richer set provides better coverage of different types of problems. DISTALYZER aims at striking the right balance between these objectives through our experiences and intuition debugging distributed systems. DISTALYZER constructs a set of summary statistics $\mathbf{X}$ from the timestamps of event log messages and the values of numeric variables for state log messages. Both are described below in more detail.

Event Features The timing of system events is often closely related to overall performance, as it can identify the progress of system components, the presence or absence of noteworthy events, or the occurrence of race conditions between components. We consider a set of event variables Y^e that are recorded in the log instances with timestamps. For example, an instance may refer to a node downloading a file in BitTorrent, where the event log may contain several recv_bt_piece events over time.

To summarize the timing information associated with a particular type of event Y^e in instance i, DISTALYZER constructs features that record the time associated with the *first*, *median* and *last* occurrence in Y_i^e. (All timestamps within a log instance i are normalized based on the start time of i.) Specifically, $X_{i.1}^e = min(\mathbf{Y_i^e}[\mathbf{t}])$, $X_{i.2}^e = median(\mathbf{Y_i^e}[\mathbf{t}])$, $X_{i.3}^e = max(\mathbf{Y_i^e}[\mathbf{t}])$. In addition, a fourth feature is constructed that counts the total number of occurrences of Y^e, $X_{i.4}^e = |\mathbf{Y_i^e}|$. Our experience debugging systems suggests that these occurrences capture some of the most useful, yet easily comprehensible, characteristics of system progress. They most commonly indicate issues including but not limited to startup delays, overall slowdown and straggling finishes.

In addition to the above features which consider the *absolute* timing in instances, we consider the same set of features for *relative* times. Since the instances from C_0 and C_1 may have different total times, normalizing the times within each instance to the range $[0, 1]$ before computing the features will yield a different perspective on event timings. For *e.g.*, in BitTorrent, it is useful to know that the last outgoing connection was made at 300sec, but for debugging it may be more important to know that it occurred at 99% of the runtime when comparing to another instance where the last connection was made at 305sec, but earlier at 70% of its total runtime. In this case, the divergence in the relative times of the events is more distinguishing. The top half of Table 1 outlines the set of event feature types considered by DISTALYZER.

State Features It is common that some of the system state variables may be directly or inversely proportional to the performance, and their divergence could either improve or substantiate the developer's notion gathered from events. We consider a set of state variables Y^s that maps to a list of values with their logged timestamps in an instance. For example, in BitTorrent, one of the state variables logged is the download speed of a node, which is inversely proportional to the total runtime performance. DISTALYZER does not attempt to understand the meaning behind the variables or their names, but systematically searches for patterns in the values.

To summarize the information about a particular state variable Y^s in log i, we construct features that record the *minimum*, *average* and *maximum* value in Y_i^s. Specifically, $X_{i.1}^s = min(\mathbf{Y_i^s})$, $X_{i.2}^s = mean(\mathbf{Y_i^s})$, $X_{i.3}^s = max(\mathbf{Y_i^e})$. In addition, to understand the variable values as the system progresses and also give the values context, DISTALYZER constructs features that record the variable values at one-fourth, half and three-fourth of the run. Similar to the events, the relative versions of these snapshots are also considered as feature types. The complete list of state feature types is listed in Table 1.

Event times
{First, Median, Last}
× {Absolute, Relative} occurrences
{Count}
State values
{Minimum, Mean, Maximum, Final}
{One-fourth, Half, Three-fourth}
× {Absolute, Relative} snapshots

Table 1: Feature types extracted from the system

Cost of Performance Differences Our analysis is focused on leveraging the characteristics of the average performance difference between the two classes, thus naïve use of the instances in statistical techniques will fail to find variables that distinguish performance in the *tails* of the distribution. For example, in a class of bad performance, there may be 2-3% of instances that suffer from significantly worse performance. Although these cases are relatively infrequent, the *high cost* of incurring such extreme bad performance makes analysis of these instances more important. DISTALYZER automatically detects a significant number of abnormally high/low values of the performance metric, and flags this to the developer for consideration before further analysis. Specifically, DISTALYZER identifies a "heavy" tail for P when the fraction of P_i outside $\bar{P} \pm 3\sigma_P$ is larger than 1.1% (*i.e.*, $4\times$ the expected fraction in a normal distribution). To more explicitly consider these instances in the modeling, we can re-*weight* the instances according to a *cost* function that reflects the increased importance of the instances with values in the tail, similar to other work on cost-sensitive learning [10]. This issue is discussed further in Section 5.2.1.

3.2 Predictive Modeling

In the next stage of the workflow, DISTALYZER uses statistical tests to identify the features that most distinguish the two sets of logs C_0 and C_1. Specifically, for each event and state feature X described above (*e.g.*, *first*(recv_bt_piece)), we consider the distribution of feature values for the instances in each class: X_{C_0} and X_{C_1}. DISTALYZER uses t-tests to compare the two distributions and determine whether the observed differences are *significantly* different than what would be expected if the random variables were drawn from the same underlying distribution (*i.e.*, the means of X_{C_0} and X_{C_1} are equal). If the t-test rejects the null hypothesis that the $\bar{X}_{C_0} = \bar{X}_{C_1}$, then we conclude that the variable X is *predictive*, *i.e.*, able to distinguish between the two classes of interest. Specifically, we use *Welch's t-test* [31], which is defined for comparison of unpaired distributions of unequal variances. We use a critical value of $p < 0.05$ to reject the null hypothesis and assess significance, adjusting for multiple comparisons with a Bonferroni correction based on the total number of features evaluated.

Our use of t-tests is motivated by the fact that we want to identify variables that distinguish the two classes on *average* across many instances from the system. Previous work [28] has used Kolmogorov-Smirnov (KS) tests to distinguish between two distributions of request flows. In that work, the bulk of the two distributions are the same and the KS test is used to determine whether there are *anomalous* values in one of the two distributions. In contrast, our work assumes that the log instances have been categorized into two distinct classes based on developer domain knowledge. Thus the overlap between distributions will be minimal if we can identify a variable that is related to performance degradation in one of the classes. In this circumstance, KS tests are too sensitive (*i.e.*, they will always reject the null hypothesis), and t-tests are more suitable form of statistical test.

Given the features that are determined to be *significant*, the magnitude of the t-statistic indicates the difference between the two distributions—a larger t-statistic can be due to a larger difference in the means and/or smaller variance in the two distributions (which implies greater separation between the two classes). The sign of the t-statistic indicates which distribution had a bigger mean–a positive value indicates that the first distribution had a larger mean. Among the significant t-tests, we return a list of significant variables ranked in descending order based on the absolute sum of t-statistic over all features. This facilitates prioritized exploration on the variables that best differentiate the two classes.

3.3 Descriptive Modeling

In the third component of the workflow, DISTALYZER learns the relationships among feature values for each class of logs separately. The goal of this component is to identify salient dependencies among the variables within a single class (*i.e.*, C_0)—to help the developer understand the relationships among aspects of the system for diagnosis and debugging, and to highlight the impact of divergent variables on overall performance P. Attempting to manually discover these relationships from the code is often difficult, because of the large size of these systems' code bases. It is also possible that observed variation across the classes for a feature is not necessarily related to performance. For example, a timer period may have changed between the classes without affecting the performance, and such a change can be quickly ignored if the dependencies are understood.

Since we are interested in the overall associations between the features in one class, we move beyond pairwise correlations and instead estimate the *joint distribution* among the set of features variables. Specifically, we use dependency networks (DNs) [19] to automatically learn the joint distribution among the summary statistics $\mathbf{X}$ and the performance variable P. This is useful to understand which sets of variables are inter-related based on the feature values. We construct DNs for the event and state features separately, and within each we construct two DNs for each feature type (*e.g.*, *First.Absolute*), one for instances of class C_0 and one for instances of C_1.

DNs [19] are a form of graphical model that represents a joint distribution over a set of variables. The primary distinction between Bayesian networks, Markov networks, and dependency networks is that dependency networks are an approximate representation of the joint distribution, which uses a set of conditional probability distributions (CPDs) that are learned independently. Consider the set of variables $\mathbf{X} = (X_1, ..., X_n)$ over which we would like to model the joint distribution $p(\mathbf{X}) = p(X_1, ..., X_n)$. Dependencies among variables are represented with a directed graph $G = (V, E)$, where conditional independence is interpreted using graph separation, as with Markov models. However, as with Bayesian networks, dependencies are quantified with a set of conditional probability distributions $\mathcal{P}$. Each node $v_i \in V$ corresponds to an $X_i \in \mathbf{X}$ and is associated with a probability distribution conditioned on the other variables, $p(x_i|\mathbf{x} - \{x_i\})$. The parents of node i are the set of variables that render X_i conditionally independent of the other variables ($p(x_i|pa_i) = p(x_i|\mathbf{x} - \{x_i\})$), and G contains a directed edge from each parent node v_j to each child node v_i ($(v_j, v_i) \in E$ iff $X_j \in pa_i$).

Both the structure and parameters of DNs are determined through learning the local CPDs. The DN learning algorithm learns a separate distribution for each variable X_i, conditioned on the other variables in the data (*i.e.*, $\mathbf{X} - \{X_i\}$). Any conditional learner can be used for this task (*e.g.*, logistic regression, decision trees). The CPD is included in the model as $\mathcal{P}(v_i)$ and the variables selected by the conditional learner form the parents of X_i (*e.g.*, if $p(x_i|\{\mathbf{x} - x_i\}) = \alpha x_j + \beta x_k$ then $PA_i = \{x_j, x_k\}$). The parents are then reflected in the edges of G appropriately. If the conditional learner is not selective (*i.e.*, the algorithm does not select a subset of the features), the DN will be fully connected. To build understandable DNs, it is desirable to use a selective learner. Since event and state features have continuous values, we use Regression Trees [9] as the conditional learner for the DNs. Regression trees are a type of decision tree that can handle continuous valued features. Their advantage over standard regression models is that they are selective models, so the features selected for inclusion in the tree will determine the structure of the DN.

Improvements The graphical visualization of the learned DN are enhanced to highlight to the developer (1) the divergence across classes (sizes of the nodes), (2) the strength of associations among features (thickness of edges), and (3) temporal dependencies among features (direction of edges). Specifically, each feature (node) in the DN is matched with its corresponding statistical t-test value. Since the t-statistics reflect amount of divergence in the feature, across the two classes of logs, they are used to size the nodes of the graph. Next, for the assessment of relationship strength, we use an input parameter m for the regression tree that controls the required (*i.e.*, minimum) number of training samples to split a leaf node in the tree and continue growing (*i.e.*, a large value of m leads to *shorter* trees because tree growth is stopped prematurely). The dependencies identified in a shorter tree are *stronger* because such variables are most correlated with the target variable and affect a larger number of instances. Thus, we weight each edge by the value of m for which the relationship is still included in the DN. Finally, we augment the DN graphical representation to include *happens-before* relationships among the features. If a feature value X_i occurs before feature value X_j in all log instances, DISTALYZER draws any link between X_i and X_j in the DN as a directed edge.

3.4 Attention Focusing

The final component of the workflow automatically identifies the most notable results of the predictive and descriptive modeling to present to the user. The goal of this component is to focus the developers attention on the most likely causes of the observed performance differences. The predictive modeling component identifies and presents a ranked list of features that show significant divergences between the two classes of logs. The divergence of a single feature is usually not enough to understand both the root cause of performance problems and their impact on performance—because performance problems often manifest as a causal chain, much like the domino effect. The root cause feature initiates the divergence and forces associated features (down the causal chain) to diverge as well, eventually leading to overall performance degradation.

We noticed that divergences tend to increase along a chain of interrelated features, thus the root cause may not have the largest divergence (*i.e.*, it may not appear at the top of the ranking). The descriptive modeling component, on the other hand, identifies the associations among features within a single class of logs. These dependencies can highlight the features that are associated with the performance measure P. To identify likely *causes* for the performance difference, DISTALYZER searches for a small set of features that are *both* highly divergent and have strong dependencies with P. The search procedure for finding the DN that highlights this set is detailed below.

The set of DNs vary across three dimensions: (1) event *vs.* state features, (2) feature type, *e.g.*, *First.Absolute*, and (3) the parameter value m_{min} used to learn the DN. In our experiments, we set m_{min} to one-third of the instances. The aim was to focus on the sufficiently strong relationships among features, and this choice of m_{min} consistently proved effective in all our case studies. However, m_{min} is included as a tunable parameter in the system for the developer to vary and observe the impact on the learned models. DISTALYZER identifies the most notable DN graph for the state and event features separately. Within a particular set, the attention-focusing algorithm automatically selects the feature type with the "best" DN subgraph. To score the DN graphs, they are first pruned for the smallest connected component containing the node P, and then the selected components are scored using Algorithm 1.

The intuition behind the score function is that it should increase proportionally with both the edge weights (strength of association) and the vertex weights (divergence across classes). The node and edge weights are normalized before computing this score. If the developer is interested in biasing the search toward features with larger divergences or toward stronger dependencies, a parameter α can be used to moderate their relative contributions in the score. The feature type with the highest scoring connected component is selected and returned to the developer for inspection.

Section 5 shows the outputs of DISTALYZER for real systems with observed performance problems, together with their interpretations. Apart from the final output of the attention focusing algorithm, the developer can also access a table of all the t-test values and dependency graphs for both the state and event logs. This is shown as the final stage in Fig. 1

Algorithm 1 Feature Scoring

Input: Log type: t (State / Event)
Input: Log class: c, Number of instances: N
Input: T-tests for all random variables in (t, c)
Input: DNs for all random variables in (t, c)
Input: Performance metric: P
  $feature\_graphs = \{\}$
  for Feature f: feature_types(t) **do**
    dn = $\mathsf{DN}_f(m_{min} = N/3)$
    cc = Connected-component in dn containing P
    $tree$ = maxSpanningTree(cc) rooted at P
    $score = 0$
    for Node n: $tree$ **do**
      $score$ += $\mathsf{T}_f(n) * dn$.weight(parentEdge(n))
    end for
    Append $(score, cc)$ to $feature\_graphs$
  end for
  return $feature\_graphs$ sorted by score

4 Implementation

We describe some implementation details for transforming text logs and developing DISTALYZER.

4.1 Processing Text Log messages

The BitTorrent implementations we considered were implemented in C (Transmission) and Java (Azureus), whereas HBase was implemented in Java. The Java implementations used Log4j as their logger. Transmission however used hand-coded log statements. HBase also used Log4j, but did not have any logs in the request path.

For each implementation, we tailored a simple Perl script to translate the text logs into a standard format that DISTALYZER accepts. We maintained a simple internal format for DISTALYZER. This format captures the timestamp, type of log, and the name of the log. For state logs, the format additionally includes the value of the log. We advocate adopting a similar procedure for analyzing any new system implementation. A developer with domain knowledge on the system should be able to write simple text parsers to translate the most important components of the log instances. To support the translation, we provide a simple library API for logging in a format accepted by DISTALYZER (shown in Fig. 2). At the beginning of each log instance, the translator calls setInstance, which indicates the instance id and class label for subsequent logs. It specifically requires marking logs as event or state logs at translation time by calling one of the two log methods.

```
setInstance(class, instance_id)
logStateValue(timestamp, name, value)
logEventTime(timestamp, name)
```

Figure 2: DISTALYZER log API

4.2 DISTALYZER

We implemented DISTALYZER in Python using the scientific computing libraries Numpy and Scipy. The two sets of logs are parsed to extract the event and state features for each log instance, and then the T-Tests and dependency networks are generated. The design allows adding or tweaking any of the event or state features if required by the developer. The Orange data mining library [9] provides regression tree construction, and we implemented dependency networks and connected-subgraph extraction over that functionality. The DOT language is used to represent the graphs, and Graphviz generates their visualizations. The implementation of DISTALYZER comprises of many embarrassingly parallel sub-tasks and can easily scale on multiple cores and machines enabling quick processing.

An interactive JavaScript based HTML interface is presented to the developer along with the final output. This immensely helps in trudging through the individual distributions of variables, and also to view the dependency graphs of all features. This has been useful in the post-root cause debugging process of finding a possible fix for the issue. To a good extent, this also helps in understanding some of the non-performance related behavioral differences between the logs. For example, in one case of comparing different implementations, we noticed that either system was preferring the use of different protocol messages to achieve similar goals.

5 Case Studies

Our goal in these case studies is to demonstrate that DISTALYZER can be applied simply and effectively to a broad range of existing systems, and that it simplifies the otherwise complex process of diagnosing the root cause of significant performance problems. We therefore applied DISTALYZER across three real, mature and popular distributed systems implementations. Table 2 captures the overview of the systems we considered. These systems represent different types of distributed system applications: distributed sorting, databases, and file transfers. We identified previously unknown performance problems with two of these systems, and worked with an external developer to evaluate usefulness of DISTALYZER in rediscovering a known performance bug with another. We describe the outputs of DISTALYZER and henceforth straightforward debugging process.

5.1 TritonSort

TritonSort is a large scale distributed sorting system [25] designed to sort upto 100TB of data, and holds four 2011 world records for 100TB sorting. We demonstrate the effectiveness of DISTALYZER by applying it over logs from a known bug. We obtained the logs of TritonSort from the authors, taken from a run that suddenly exhibited 74% slower performance on a day. After systematically and painstakingly exploring all stages of the sort pipeline and running micro-benchmarks to verify experimental scenarios, the authors finally fixed the problem. They said that it took “the better part of two days to diagnose”. The debugging process for the same bug took about 3-4hrs using DISTALYZER, which includes the implementation time of a log parser in 100 lines of Python code. A detailed analysis of the output of DISTALYZER and the debugging process on these logs follows.

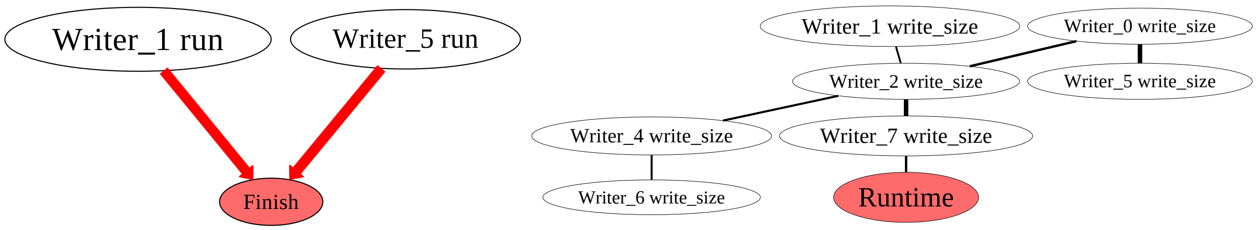
We had access to logs from a 34 node experiment from the slow run that took 383 sec, and also a separate run with the same workload that had a smaller runtime of 220 sec. These naturally fit into two classes of logs with one instance per node, which could be compared to identify the reason for the slowdown. These logs were collected as a part of normal daily testing, meaning no additional overhead for log collection. The logs contained both event and state log messages that represented 8 different stages of the system (Table 2). The performance metrics were identified as Finish and runtime for the event and state logs respectively, both indicating the time to completion. Fig. 3 shows the final dependency sub-graphs output by DISTALYZER for both event and state logs.

To briefly explain the visualization generated by DISTALYZER, nodes shown to be colored indicate the performance metric and the font size is proportional to the magnitude of the divergence. Edge thickness represents the strength of the dependencies between variables. Directed edges in event graphs indicate that a *happens-before* relationship was identified between the two bounding variables, as described in Section 3.4.

The best dependency graph for events was picked from the *Last* feature type, Fig. 3a. This demonstrates that variables Writer_1 run and Writer_5 run are both significant causes of the divergence of Finish. The final stage of TritonSort is the writer which basically handles writing the sorted data to the disk. Each stage in TritonSort is executed by multiple thread workers, denoted by the number in the variable. Therefore, this analysis attributes the root cause of slow runs to highly divergent last occurrences of the writer workers. A quick look at the distribution comparison of the two sets of logs in both the writers indicates that the slow run showed a difference of 90 sec. The performance metric and the writer run distributions showed an outlier with a larger time than the rest.

System Implementation	Types of Logs	Volume	Variables	Issues	Performance gain	New issues
TritonSort	State, Event	2.4 GB	227	1	n/a	×
HBase (BigTable)	Event	2.5 GB	10	3	22%	√
Transmission (BitTorrent)	State, Event	5.6 GB	40	2	45%	√

Table 2: Summary of performance issues identified by DISTALYZER



(a) Event

(b) State

Figure 3: TritonSort dependency graphs indicating the root cause of the slow runtime

Similarly, the dependency graph picked for the states is shown in Fig. 3b, where the performance metric Runtime is connected to the subgraph consisting of the write queue size of different writer workers. Although the figure was scaled down for space constraints, it is clear that the divergences of all the nodes are similarly high like total performance divergence. To understand the reason of this divergence, we looked at distributions for the Absolute Half (best feature) to learn that writers in the slow run were writing 83% more data. Thus, we concluded the slowness was caused by *slow writers*.

The actual bug had been narrowed down to the disk writing stage, which was indirectly slowing down earlier stages of the pipeline. It was further noticed that a single node was causing most of this delay, which finally led to the authors discovering that the cache battery on that node had disconnected. This resulted in the disks defaulting to write-through and hence the poor performance. Both the top ranked DNs output by DISTALYZER were useful in identifying the bug. We shared these dependency graph and interactive t-test tables with the author of the paper, who had manually debugged this problem. The output root cause was immediately clear to him, and he surmised “had we had this tool when we encountered this problem, it would have been a lot easier to isolate the difference between the bad run and a prior good one”.

5.2 HBase

BigTable [4] is a large-scale storage system developed by Google, holds structured data based on rows and columns, and can scale efficiently to a very large number of rows and column content. HBase [18] is an open source implementation of BigTable being developed by the Apache foundation. It runs on top of Hadoop Distributed Filesystem (HDFS), and has been tested on large scales and efficiency. HBase is implemented in Java and is tuned for high performance.

For our experiments, we used the Yahoo Cloud Storage Benchmark (YCSB) [8]. In our experiments, the read request latencies under “Workload D” had a notable heavy tail distribution. The minimum and median latencies are 0 and 2 msec respectively. However the mean latency is 5.25 msec and the highest latency is as high as 1 second, which is close to 3 orders of magnitude greater than the median. Moreover, more than 1000 requests have a latency greater than 100ms, which is not easily understood. We would like to be able to compare these slow requests to the huge bulk of fast ones, to identify these performance bottlenecks in HBase. This task is infeasible manually because these issues manifest only in large experiments (1 million requests), and a sufficiently large number of requests exhibit this behavior. After describing the experimental setup, we discuss the use of DISTALYZER in debugging three problems.

Experimental setup Our experimental testbed consisted of 10 machines with 2.33GHz Intel Xeon, 8GB RAM and 1Gbps Ethernet connections running Linux 2.6.35.11. Our HBase cluster consisted of a single master running on a dedicated machine, and 9 region servers (equivalent to BigTable tablet servers). The YCSB client was run on the same machine as the master (which was otherwise lightly loaded), with 10 threads issuing requests in parallel. Each request is either a read of all columns, or write all columns for a single row. 1 Million rows were pre-loaded into the table with each row’s size as 30kB. The workload consisted of 1 Million operations out of which 5% were writes.

The HBase implementation had no log statements in the request flow path, inspite of using the *log4j* logging library that supports log levels. Therefore, we manually added 10 event logs to the read request path, using the request row key as the identifier. The request logs from the different machines were gathered at the end of the run, to bucket log messages under different requests. The

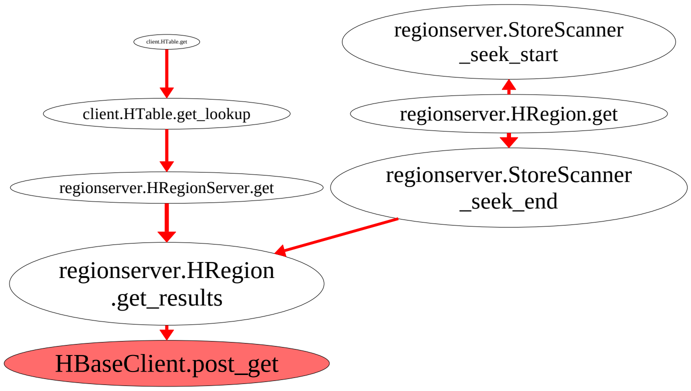


Figure 4: DN for unmodified HBase events

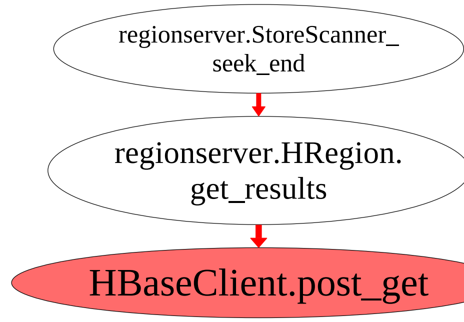


Figure 5: DN for HBase after fixing lookups

performance metric is the event that signifies the last step in request processing – HBaseClient.post_get.

5.2.1 Fixing the slowest outliers

On applying DISTALYZER to the logs, it detected the presence of a heavy tail in the performance metric (§ 3.1) and suggested re-weighting the instances. The weight function used to boost the instances with a large latency was $\lfloor w^{latency} \rfloor$. This is an exponential weight function and we chose a value of $w = 2^{(1/150)}$, with the idea that instances with $P < 150ms$ will have a weight of 1. Fig. 4 shows the dependency graph of the root cause divergence. All dependency edges turn out to be directed because all requests follow the same flow path through the system, and hence log the events in the same order. We identified two strong associations with large divergences leading upto the performance metric. Each of the chains is considered for root cause analysis, and we first chose to follow the path leading from client.HTable.get_lookup (the second chain is discussed in § 5.2.2). This chain starts at client.HTable.get which indicates that the HBase client library received the request from YCSB, followed by client.HTable.get_lookup representing the completion of lookup for the region server handling the given key.

This particular edge leads from a tiny variable to a variable with significant divergence, and domain knowledge indicates that no other event occur between them. The tiny variable is drawn as such because it does not differ considerably between the two classes of logs. As it is connected so strongly by a directed edge to the larger variable, this indicates the two systems' behavior is consistently *different* between the tiny event variable and the larger event variable. In this context, the edge represents the operation where the client needs to identify the particular region server that manages the row, and this is achieved by contacting the master who maintains the mapping. The distributions of this particular event in the t-test table shows that this event created gaps in the request flow of the order of 1000 ms.

When we looked at the logs of the regionserver at the same time these requests were being delayed, we noticed that the server was throwing a NotServingRegionException. This is given by the server when it does not serve a region that was specifically requested. This happens when a region was moved to another server during load balancing. The client possesses a stale cache entry for the region, and hence receives this exception. The client was catching this exception as an IOException, and treated it as a server failure. This triggers a backoff procedure that starts at 1 sec. According to the Bigtable description [4], the client immediately recognizes a stale cache and retries leading to an overhead of just 2RTTs. We came up with a fix for this issue, by adding code to treat exceptions with care and extracting the NotServingRegionException, and retrying immediately. This fixed the requests with latencies over 1 second.

5.2.2 Operating System effects

DISTALYZER was used again to analyze the new logs to find the cause of the other delays. Since the distribution skew was lesser than the threshold, the weighting function was not used anymore. The dependency graph is shown in Fig. 5, and closely resembles the right chain of Fig. 4. In fact, this root cause was also identified in the initial step as a second significant root cause, but was not chosen for inspection. This graph points to two variables chaining up as the root cause – regionserver.StoreScanner_seek_end and regionserver.HRegion.get_results.

The default Linux I/O scheduler since version 2.6.18 is Completely Fair Queuing (CFQ), that attempts to provide fairness between disk accesses from multiple processes. It also batches requests to the disk controller based on the priority, but it does not guarantee any completion times on disk requests. Since only the HBase process was accessing the disk on these machines, we believed that this scheduling policy was not well suited to random block reads requested by HBase. Another available I/O scheduler in Linux is the deadline scheduler,

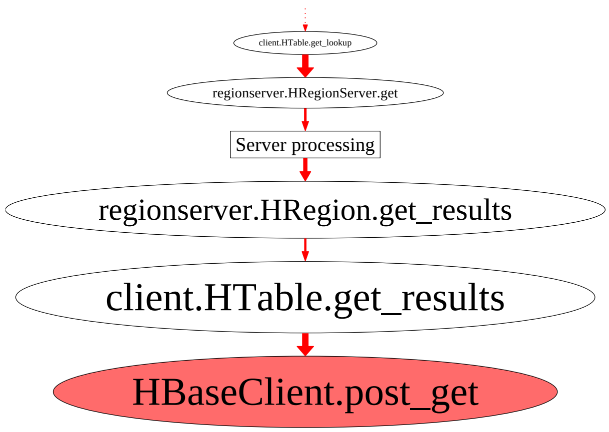


Figure 6: HBase DN showing large network divergence

which tries to guarantee a start service time for requests. Hence the deadline scheduler would be more suited toward latency sensitive operations.

After we applied the I/O scheduler change, we ran the same experiment again to understand if this caused changes in the latencies of the slow requests. The number of slow requests ($\geq$100ms) reduced from 1200 to just under 500, which is a 60% reduction. Also, the mean latency for the workload dropped from 5.3ms to 4ms, which is a 25% overall improvement in the read latency, confirming deadline is appropriate for these workloads.

5.2.3 Networking issues

After changing the disk scheduler, we again returned to investigate the remaining *slow* requests. We analyzed the new logs to obtain the DN in Fig. 6. The complete figure included some variables still related to disk latencies, which we ignore so as to find additional issues. For presentation, we replaced these disk-related variables with a "Server processing" box in the figure. The event variables of interest are regionserver.HRegionServer.get and client.HTable.get_results, in terms of the increase in divergence compared to their predecessors. We immediately realized that these events just follow a network operation, one for each hop of the RPC. We were using 1Gbps connections and neither the client or the servers were short of bandwidth at any time.

In the process of scanning through the networking code, it became clear that connections were persistent and only a single TCP connection was used per server. However, we found that the TCP NoDelay flag was set to false by default. HBase is most commonly used in high speed networks such as large server farms and data centers, which have very low latencies. Thus, we had expected HBase to disable Nagle's Algorithm by default, to spend some extra bandwidth at the cost of optimum network latencies. This change improved the latencies of all the slow network requests, and as a side-effect caused *all* of the request latencies to go down.

While DISTALYZER did not point us to complete solutions for the disk and network performance variances, it was extremely helpful in identifying the root causes so we could discover alternate configurations that improve the performance of these bottlenecks. All of the 3 reported root cause DNs were helpful in debugging HBase.

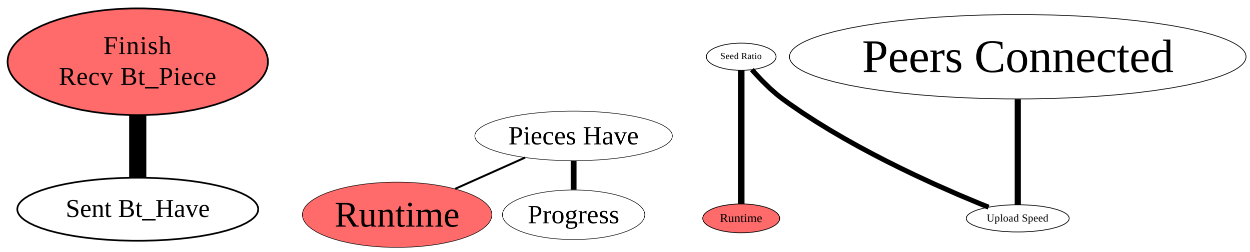
5.3 Transmission

Transmission implements the BitTorrent protocol, a distributed file sharing mechanism that downloads different pieces of a file from multiple peers. The protocol works by requesting a set of active peers for the file from a *tracker*, then individually initiates connections to them to download file pieces. By downloading from multiple peers simultaneously, clients can more easily download at large speeds limited only by its bandwidth. There are many popular implementations of this protocol, out of which we consider Azureus, another open source implementation for comparison. In some basic experiments, Transmission had a much worse download time compared to Azureus (552 sec vs. 288 sec).

Transmission [30] is a light-weight C implementation, and among all the free clients, it is known for its minimal resource footprint. Azureus [2] is one of the most popular free implementations of the protocol, developed in Java. It is an older and more mature implementation of the protocol and well known for its excellent performance. Unlike Transmission, it extends the basic BitTorrent messaging protocol for extra minor optimizations in communicating with supporting peers. Both are serious implementations of the protocol, and we expect a well tuned C implementation should perform no worse than a Java implementation. Using DISTALYZER, we were able to identify two performance bugs in Transmission that eliminated the download time difference completely.

Experimental setup Experiments consisted of 180 BitTorrent clients (30 clients per machine) attempting to download a 50MB file, providing ample interaction complexity in the global system. They used the same machines as described in Sec. 5.2. The swarm was bootstrapped with a single seeder, and each client was limited to an upload bandwidth of 250KB/s which is similar to common Internet bandwidths and makes ample room for running 30 clients on a single machine. Experiments were conducted with each implementation in isolation.

We built Azureus from its repository at rev. 25602 (v4504). Azureus had a detailed log of BitTorrent protocol messages during a download, and we added some state logs. The experiments used the HotSpot Server JVM build 1.6.0_20. We used version 2.03 of Transmission in our experiments, which contained debugging logs, and we simply activated the ones pertaining to the



(a) Event DN (b) State DN: 1st with score 0.89 (c) State DN: 2nd with score 0.84

Figure 7: Dependency graphs for unmodified Transmission

BitTorrent protocol. We identified the event and state performance metrics Finish and Runtime, respectively.

5.3.1 Faulty component affecting performance

In this particular case, the best features output by DISTALYZER in event and state dependency graphs Fig. 7a, 7b were dependencies between trivial divergences. These are in a sense false positives to the automatic root cause detection. More specifically, Fig. 7a was picked from the *Last* event-feature and shows the performance metric coalesced with the last piece receipt. The strong dependency to Sent_Bt_Have is justified by the fact that implementations send out piece advertisements to peers, as soon as they receive one more piece. Similarly, the state dependency graph in Fig. 7b shows strong dependencies between download completion time and the number of pieces download in half the run, and also the progress (which is in fact a factor of Pieces Have).

This led to considering the second ranked state graph in Fig. 7c, which in fact had a very close score to the highest rank. This DN was constructed from snapshots of the state variables at three-fourth of Transmission's download time. The interpretation is that Runtime is affected by the amount of data uploaded (seeded), which is essential in a symbiotic environment such as BitTorrent. This upload is trivially dependent on the upload speed, and finally associated with a very highly differing Peers Connected. This immediately takes the developer to the distributions of the number of peers, all nodes reporting 6 peers. Following this, the *Maximum* feature confirmed that while Transmission had only 6, Azureus had 50.

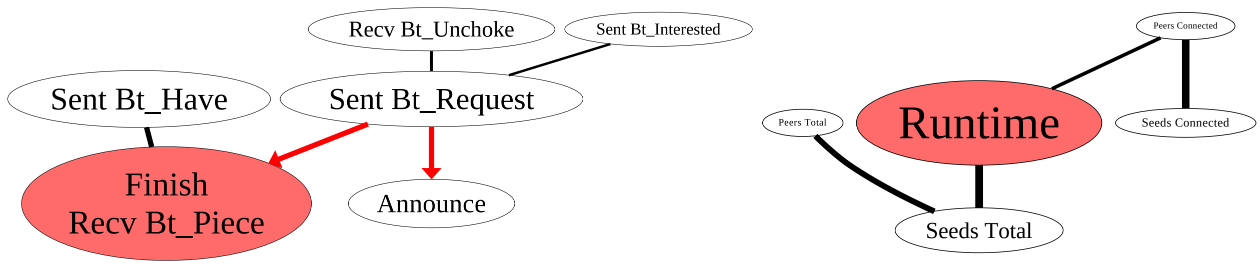
Fixing the bug To find the problem that limited Transmission's peer connectivity, we considered a single node's logs and fetched the set of unique IP:port pairs, and on looking at the values, we immediately realized that each peer had a different IP address. In our experimental setup with 6 physical machines, different nodes on the same physical machine were setup to listen on different ports and coexist peacefully. The bug was traced to the internal sorted set holding peers, whose comparison function completely ignored the port number of the peer. When a node hears about a new peer from the tracker, and it is already connected to a peer with the same IP address, the new peer is simply dropped.

On looking through forums and bug management softwares, we found that this inconsistency had actually been identified 13 months back, but the bug was incorrectly closed. We verified the authenticity of this bug and reopened it. The developers deemed this bug to be hard to fix, in terms of requiring changes to many modules. We argue that this is an important bug that limits Transmission from connecting to multiple peers behind a NAT box. In cases where multiple peers are situated behind a NAT box in an ISP, they would definitely want to download from each other and avoid the slow ISP link. This bug would prevent local connections, thus forcing them to connect to peers on the Internet.

5.3.2 Tuning the performance

Since the fix for the first bug was too tedious, we decided to circumvent the problem by assigning unique virtual IP addresses to each of the nodes. This did indeed solve the problem and made Transmission faster to an average download time of 342 sec, which was still much higher than 288 sec. DISTALYZER was used again with the new set of logs which produced the dependency graph output shown in Fig. 8. Considering the event DN in Fig. 8a, showing the highly divergent performance metric for the *Last* feature. Some of the features of this DN are similar to Fig. 7a that were discussed earlier.

The dependency between finishing and sending requests fits well with the protocol specifications, that a request for a piece must be sent in order to receive one. The Announce event happens *after* sending out requests, and hence de-values its possibility for root cause. The interested messages were a more probable cause of the differences (compared to un-choke) because one must first express interest in another peer after connection establishment. Only after this step does the remote peer unchoke it, thus opening up the connection to piece requests. This hypothesis was verified by viewing the distributions of Sent_Bt_Interested across all features. After knowing the root cause, the distribution for the offending variable



(a) Event (b) State

Figure 8: Dependency graphs for BitTorrent after fixing the NAT problem

in the *First* feature showed gaps of the order of 10 sec on Transmission, but was very small for Azureus.

We traced the code from the message generator to fix these large gaps, and found a timer (called rechokeTimer) that fired every 10 sec. For comparison, we found that Azureus had a similar timer set at 1 sec, thus giving it a quicker download start. The large divergence in sending interested messages could be fixed by shortening the timer value from 10sec to 1sec. Fig. 8b shows the state DN for the same logs for completeness, but it does not indicate a highly divergent root cause.

Performance gains We were able to apply a quick fix for this problem and the download times of Transmission were much better than earlier, dropping the mean completion time to 288 sec. The performance was upto 45% better than the first experiment. It should be noted that the more frequent timer did not affect the resource utilizations of Transmission, still using far fewer CPU cycles and memory than Azureus. Neither of these issues affected correctness, nor threw any sort of exceptions, and present themselves as subtle challenges to the developers. Overall, 5 DNs were reported for the two issues in Transmission, out of which 3 indicated trivial relationships between the components, but the other two were immensely helpful in understanding the root causes.

6 Related Work

Model checking aims to provide guarantees on program code against pre-specified properties. A number of techniques [16, 20, 23] have described different methods to assert program correctness. However, traditional model checking attempts to discover violations of clear failure conditions. There is also research in applying machine learning to logs of faulty executions, to categorize them [3,7] and also predict the root cause [5]. Conditions of performance degradation cannot be accurately modeled using these approaches, because it is rarely possible to specify performance as definite runtime predicates.

The formulation of debugging as an anomaly detection task has been applied in a variety of contexts. Magpie [3] and Pinpoint [5] model request paths in the system to cluster performance behaviors, and identify root causes of failures and anomalous performance. Fu *et al.* [12] propose the use of a Finite State Automaton to learn the structure of a normal execution, and use it to detect anomalies in performance of new input log files. Xu *et al.* [32] propose a mechanism to encode logs into state ratio vectors and message count vectors, and apply Principal Component Analysis to identify anomalous patterns within an execution. However, they completely ignore timestamps in logs and use the value logged, to identify localized problems within a single log file. On the other hand, DISTALYZER finds the root cause of the most significant performance problem that affects the *overall* performance. In contrast to all these systems, DISTALYZER aims to find the cause of performance problems in a major portion of the log instances, and hence uses t-tests to compare the average performance.

Request flows are a specific type of distributed processing, with a pre-defined set of execution path *events* in the system. Sambasivan *et al.* [28] aim to find structural and performance anomalies in request flows that are induced by code changes. Their approach of comparing different requests bears some similarity to our technique. However, as we illustrate through our case studies, DISTALYZER can be applied to request flow systems (HBase), as well as other types of distributed systems, by abstracting the logs into states and events. Although these specific applications of machine learning (including [1,3,5,6]) can leverage path structures, DISTALYZER can show the *most impacting* root cause among many performance problems.

There has been work in identifying the most important parts of systems logs through clustering of event frequencies [27]. Cohen *et al.* [6] use instrumentation data from servers to correlate bad performance and resource usage using tree-augmented Bayesian networks. Similarly, DISTALYZER can utilize system monitoring data as outlined in Section 2 to identify performance slowdowns due to resource contention using DNs.

Splunk [29] is an enterprise software for monitoring and analyzing system logs, with an impressive feature set. Although it provides a good visual interface for manually scanning through logs and finding patterns, it does not provide tools for rich statistical analysis on the data. Furthermore, there is no support for comparing two sets of logs automatically. We believe that Splunk is complementary to our work, and the concepts embodied in DISTALYZER could serve as a great addition to Splunk.

7 Conclusion

This paper proposes a technique for comparing distributed systems logs with the aim of diagnosing performance problems. By abstracting simple structure from the logs, our machine learning techniques can analyze the behavior of poorly performing logs as divergence from a given baseline. We design and implement DISTALYZER, which can consume log files from multiple nodes, implementations, runs and requests and visually output the most significant root cause of the performance variation. Our analysis of three mature and popular distributed systems demonstrates the generality, utility, and significance of the tool, and the reality that even our mature systems have undiagnosed performance issues that impact the overhead, cost, performance, or health of our systems. DISTALYZER allows us to find and solve these problems, where manual analysis has previously been unsuccessful.

Acknowledgments. We would like to thank Alex Rasmussen for assisting us with using DISTALYZER on TritonSort logs.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[2] Azureus bittorrent client. http://azureus.sourceforge.net/.

[3] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modelling. In *OSDI* (2004).

[4] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI* (2006).

[5] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-based faliure and evolution management. In *NSDI* (2004).

[6] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. S. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI* (2004), USENIX Association, pp. 16–16.

[7] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *SOSP* (2005), ACM, pp. 105–118.

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *SOCC* (2010).

[9] DEMŠAR, J., ZUPAN, B., LEBAN, G., AND CURK, T. Orange: From Experimental Machine Learning to Interactive Data Mining. In *PKDD*. 2004.

[10] ELKAN, C. The Foundations of Cost-Sensitive Learning. In *IJCAI* (2001), pp. 973–978.

[11] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[12] FU, Q., LOU, J.-G., WANG, Y., AND LI, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM* (2009), pp. 149 –158.

[13] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[14] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Usenix ATC* (2006).

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[16] GODEFROID, P. Model checking for programming languages using verisoft. In *POPL* (97).

[17] Apache hadoop project. http://hadoop.apache.org/.

[18] Hbase. http://hbase.apache.org/.

[19] HECKERMAN, D., CHICKERING, D. M., MEEK, C., ROUNTHWAITE, R., AND KADIE, C. Dependency Networks For Inference, Collaborative Filtering, and Data Visualization. *JMLR* (2001), 49–75.

[20] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI* (2007).

[21] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D^{3}S: Debugging Deployed Distributed Systems. In *NSDI* (2008).

[22] Apache log4j. http://logging.apache.org/log4j.

[23] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. In *SOSP* (2002), ACM.

[24] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* (2008).

[25] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI* (2011).

[26] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[27] SABATO, S., YOM-TOV, E., TSHERNIAK, A., AND ROSSET, S. Analyzing system logs: a new view of what's important. In *SYSML* (2007), USENIX Association, pp. 1–7.

[28] SAMBASIVAN, R. R., ZHENG, A. X., ROSA, M. D., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011), USENIX Association.

[29] Splunk. http://www.splunk.com/.

[30] Transmission bittorrent client. http://www.transmissionbt.com/.

[31] WELCH, B. L. The generalization of student's problem when several different population variances are involved. *Biometrika 34*, 1-2 (1947), 28–35.

[32] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009), ACM, pp. 117–132.

[33] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI* (2009).