2012 IEEE 18th International Conference on Parallel and Distributed Systems

Leveraging Cloud Infrastructure for Troubleshooting Edge Computing Systems

Michael Fagan
Department of CSE
University of Connecticut, USA
michael.fagan@uconn.edu

Mohammad Maifi Hasan Khan
Department of CSE
University of Connecticut, USA
maifi.khan@engr.uconn.edu

Bing Wang
Department of CSE
University of Connecticut, USA
bing@engr.uconn.edu

***Abstract*—Modern cloud-based applications (e.g., Facebook, Dropbox) serve a wide range of edge clients (e.g., laptops, smartphones). The clients' characteristics vary significantly in terms of hardware, operating systems, network connections, and software versions, just to name a few. Unfortunately, due to misconfiguration, outdated software, faulty hardware, or other reasons, many edge systems operate at suboptimal performance. Identifying poor performance and root causes is extremely challenging for the client of the cloud system. In this paper, we propose a novel troubleshooting service that leverages such heterogeneity to identify and debug performance problems on edge devices. First, by looking at many runs across many different clients, the service groups clients into different clusters based on performance. Next, the service enables logging on remote clients to collect run time traces, and subsequently identifies the root cause by analyzing logs automatically. We leverage high level features such as machine/OS type along with more low level kernel level statistics such as I/O rate and system calls. To demonstrate our system, we first introduce a configuration bug that was artificially injected in a recently built cluster by changing the TCP buffer size. Next, we present two real-life case studies, one related to I/O inefficiency on Android platform, and another misconfiguration bug in VirtualBox, that were identified using our tool.**

***Keywords*-automated debugging; system calls; middleware; cloud service; distributed computing;**

I. Introduction

Modern data centers serve millions of users each day. To access their services, edge clients use different computing platforms which vary significantly in terms of hardware (e.g., high end desktop vs. resource constrained smart phones), operating systems (e.g., Linux, Android, Mac OS, Windows), network characteristics (e.g., wireless vs. wired, 3G vs. 2G), and software versions (e.g., Firefox 12 vs. Firefox 13). For instance, some may use a laptop over a wired Ethernet connection to download a file whereas someone else may use a smart phone over a 2G wireless network to get the same file. Edge clients may experience suboptimal performance due to outdated software, misconfigurations, inferior quality hardware, poor quality network connections, among other reasons. Although some causes are beyond a user's control (e.g., poor quality Wi-Fi), many are fixable once identified (e.g., misconfiguration, outdated software). Since the client has no way of comparing his/her system's performance against others, it is almost impossible for the client to ascertain suboptimal performance to begin with. Even if they are dissatisfied with the quality of service, it is often beyond their capacity to identify the cause of "poor" performance. As a result, consumers often end up blaming the service providers for poor service, which is extremely damaging for the service providers reputation and business. Moreover, data centers end-up consuming more resources to serve requests due to edge clients' limitations.

In this paper, we identify several reasons that often lead to suboptimal performance on edge client devices. *First,* software developers often make certain assumptions regarding the target execution environment, which affects the design of the software. For example, the maximum expected network bandwidth may limit the size of the buffer the developer allocates at the network stack. However, once the target execution platform changes (2G is upgraded to 3G), the software may not be revisited to adapt to the changed condition, leading to suboptimal performance or faulty execution. Unfortunately, suboptimal performance caused by such outdated configurations is extremely hard to detect. *Second,* users often continue to use default configuration settings, which may not be the optimum setting for the user. *Finally,* users often use outdated software that may have inefficient implementations.

Fortunately, the cloud service provider is in a unique position to leverage a huge volume of diverse groups of clients and heterogeneity to identify and debug performance problems experienced on edge devices. In this paper, we present a novel troubleshooting framework that collects traces of software executions and system parameters from many different edge clients as needed, and attempt to answer questions like "Why does a download on client X takes longer than average?" or "Why is client Y's download operation getting terminated repeatedly?". To aid the edge clients, in this paper, we present the design and implementation of a middleware service that leverages kernel level traces of system calls to identify sub-optimal performance and perform root cause analysis.

The main idea behind our work is as follows. *First,* the anomaly detection engine is executed as a background service, which is responsible for monitoring high-level performance metrics such as download delay and throughput. Based on their performance, clients are grouped into different clusters where clients having similar performance are put in the same cluster. *Next,* once a cluster with suboptimal performance is

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.67

identified, the more heavyweight logging functionalities (e.g., *strace*) are enabled automatically on edge devices to collect system logs from clients that belong to different clusters. *Finally,* an automated troubleshooting module analyzes the collected logs from different performance clusters to identify the root cause (e.g., Why does a node in computing cluster X have high delay?). In this paper, the troubleshooting is offered as a proactive service. The presented middleware monitors software execution to identify potential anomalous conditions, and troubleshoot once such conditions are observed. Moreover, the same service can be used to identify faulty machines in data centers which may be responsible for the suboptimal performance on edge systems.

In this paper, we leverage kernel level tracing to provide debugging as a service for the following reasons. *First,* while different user applications perform different tasks (e.g., email client, web browser, parallel processing) and may share little in terms of design and implementation languages, they often run on the same (or similar) operating system, which often provides the necessary abstractions (e.g., system calls) for the application layers. Additionally, irrespective of the application, certain bugs often exhibit similar symptoms at higher levels. For example, a slow download may be caused by low bandwidth or inefficient I/O. However, without kernel level traces, it may become extremely hard to identify the real cause. Hence, characteristics of the system calls made by an application are used as a side channel to understand the behavior of the software. *Second,* as we leverage kernel level traces, using our tool does not require access to the application source code. *Finally,* it enables reuse of the service to troubleshoot different applications without implementing new tools for different applications.

To demonstrate the use of our tool, we first introduced a configuration bug in a recently built cluster by changing the TCP buffer space, causing one of the machines to take longer to download files. By collecting kernel level traces, we successfully identified the bug. Next, we troubleshoot a real life I/O inefficiency problem relating to network transfers on the Android platform. In short, file transfers were taking longer on Android phones compared to laptops when under similar network conditions. We identified the bottleneck to be the inefficient file system on Android devices. Finally, we present a misconfiguration bug in VirtualBox that repeatedly caused a download operation from the Google source code repository to be terminated after running for a while. The user spent numerous hours trying to diagnose the problem and wasted a few days on failed attempts. After the fact, we applied our tool and it successfully identified the problem to be a configuration issue in VirtualBox, which demonstrated the utility of our tool.

The rest of the paper is organized as follows. Section II describes related work. Section III describes the design and implementation of our system. Section IV presents the evaluation of our tool. Section V describes the limitations of our work and the future directions. Finally, Section VI concludes the paper.

II. Related Work

With ubiquitous adoption and scale, the economic and social losses incurred due to software bugs are on the rise. Not surprisingly, software debugging and troubleshooting have received significant attention recently. To summarize, prior work looked into source code verification [1], [2], [3], [4], statistical debugging [5], [6], [7], hardware failure analysis [8], resource bottleneck identification [9], run time analysis [10], [11], performance bug troubleshooting [12], and misconfiguration troubleshooting [13], [14], to name a few.

Recent work on cloud reliability and performance analysis includes efforts on benchmarking cloud services [15], hardware failure characteristics analysis [8], and fault tolerant protocol design [16]. Providing software testing as a service [17] has also been recently proposed, which aims to leverage computing infrastructure of the cloud to perform software testing in a more scalable fashion. In a not-so-recent prior work, to identify the components that are correlated to failure, Pinpoint [3] attempts tracing client requests across execution stages. Message level tracing is also being investigated [9] to identify high delay causal paths in multi-tier web applications, and to identify error propagation paths [7] at component levels. A tree augmented Bayesian network (TAN) is used to identify performance problems correlated to system states [5]. In another work [18], samples of healthy machines are used to identify the sick machines. Another group of recent work focuses on identifying interactive complexity across software components or multiple computing systems [6], [12]. However, none of these prior work attempts to identify suboptimal performance on edge clients by proactively leveraging service provider's infrastructure.

Numerous run-time monitoring techniques have been proposed to ensure quality-of-service at run-time [10], [11]. However, the correct/expected behavior must be supplied as input to use this approach. Such conditions may include pre-conditions or post-conditions of function calls, expected values of variables, temporal conditions or race conditions, which are often not known even to the developer of the system.

Source code verification via leveraging state exploration is another popular approach. Many automated debugging solutions use state exploration [1], [2], [3], [4], [19] as an underlying method to identify potential bugs. Delta debugging [4] is one of the most influential works on statistical source code debugging. Execution exploration works by simulating software executions to find error situations. Variable values are altered to narrow the aspects of the execution that leads to the error. This is done systematically to find as many bugs and their causes as possible. By this method, state exploration approaches are able to provide the user with bugs, the states in which they occur, and the values that lead to the issue. Another method to pinpoint a software flaw in the source code is via code annotation [20]. By adding annotation at the time of development, execution paths can be traced in the source code and thus the offending code locations can be narrowed. In the FATE and DESTINI [1]

approach, explicitly written expectations are used to find error conditions. Like annotations, the programmer creates extra code that explains how the code should behave in particular scenarios. They can note what variable values should be, how resources should be used, time of execution, as well as other aspects of code behavior. The software can then know when something is wrong because these assertions about behavior would be broken. However, proper documentation of behavior is needed for the method to be effective. There needs to be a balance between over-constraining and under-constraining the "expected" behavior so as not to have many false positives or bugs slip through the cracks. Moreover, with this approach, the tool needs to have access to the source code and requires significant manual effort. Use of such a tool is not suitable to troubleshoot poor performance caused by misconfiguration or use of outdated software, which is one of the targets in our work. Our method avoids this by simply not having behavior expectations. Instead, we look at the performance of a wide range of clients to determine the "best" possible performance for a group of users. We can then use this information to identify "troubled" edge clients.

In summary, although much prior work exist that focuses on troubleshooting performance problems on the service provider's side, to the best of our knowledge, none of the approaches attempt to leverage the service provider's infrastructure to troubleshoot edge systems proactively.

III. Overview

Conceptually, our tool is divided into three components, namely, a client-server communication component, a feature extraction component, and a troubleshooting component. The overall system architecture is illustrated in Figure 1. We elaborate on each of these components below.

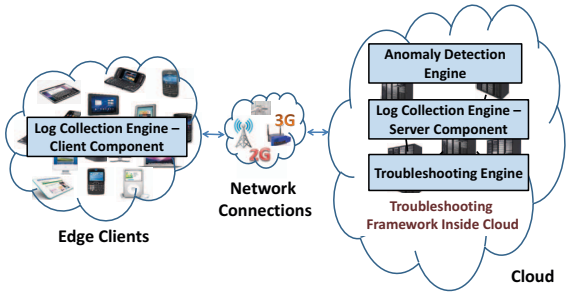

Fig. 1: System Architecture

A. Client-Server Communication Component

The network performance monitor begins by tracking transactions between the client and the server. These transactions are usually some sort of file download from the server. As such, the server can log some basic transaction information such as the client's operating system, the program used to download the data, and the total download time. This high level data is used by an anomaly detector to determine if the current client takes longer than other clients. Based on

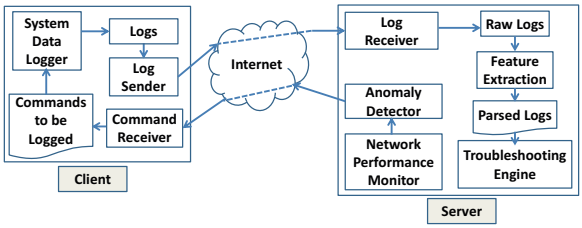

Fig. 2: Client-Server Communication Architecture

performance, the clustering engine groups different devices in different clusters dynamically. We cluster executions based on simple metrics such as delay. If an anomaly is detected, the remote log collection is initiated which works as follows.

Remote log collection requires the communication of logs between many edge clients and a central repository hosted on a service provider's servers. To facilitate this data transfer, we have developed a client/server communication software that monitors and logs executions once instructed, and sends the logs to the server for processing. This system is shown in Figure 2.

In short, the client is told which commands to log. One way to determine the program/command name on the client side is by using standard Linux commands such as *cat /proc/[pid]/cmdline* for all PIDs on the system. Since the client knows the server's URL or IP address, it can check the command line attributes for a reference to the server. If there is a match, the command name is marked for logging. There may be other ways (e.g., manually) to specify command or program name that needs to be logged. When the logger sees that a program marked for logging is running, it traces the program's execution using strace. This data is saved locally. Another monitor watches the directory in which these logs are stored. When the execution finishes, the client sends the log to the server over the Internet.

This client/server communication program efficiently gathers and transfers logs over the Internet to be analyzed by the server side classifier. All communications are done via TCP sockets to ensure a reliable transfer of logs. The client constantly listens for commands to come from the server that should be logged and the server constantly listens for logs coming from many different clients. Both the server and client applications are implemented in Java for easy portability to different machines and to take advantage of convient Java sockets.

B. Feature Extraction Component

After logs have been collected, they are fed to the feature extraction component, which prepares the information for log analysis. Once the server receives the log, the log parser extracts usable information such as time spent in each system call, the number of times each system call is invoked, the wait time between consecutive system calls of the same type (i.e. time between consecutive read() or write() calls), and the argument of some of these system calls. Once parsed, the extracted data is stored in a formatted file as a list of attributes.

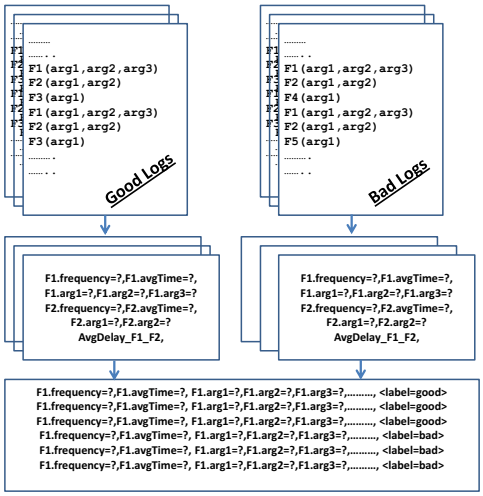

Fig. 3: Feature Extraction

Next, a subset of the stored attributes are selected and used to identify the root cause as follows.

Although various system call statistics (i.e., attributes) provide invaluable insight into an execution and are key to determining the cause of a problem, the parser generates hundreds of summary statistics. For effective troubleshooting, an efficient mechanism is needed to select a subset of attributes. However, selecting a subset of attributes from logs collected using the strace utility (or any other log collection tool) is quite challenging for the following reasons.

First, logs from different machines may have system calls with different names, but the same semantic meaning. This sometimes happens even in the same operating system. In Linux, both the poll() and select() system calls have the same semantic meaning for our purposes, but would appear differently in a log. To solve this problem, our tool lets a user specify a configuration file that maps such system calls. This mapping is reusable across multiple applications.

Second, some operating systems offer more functionality than others. In these cases, systems may have system calls that cannot be mapped to calls in another system because they do not exist. In these cases, we calculate an intersection of calls between all machines involved in an analysis. This intersection takes into account the mappings, if any, provided by the user. The algorithm is simple. One machine is chosen as the base set and only calls from that base set that also appear in all other considered machines' system call lists are included in the intersection list. Note, it is possible that this intersection method can miss system calls that are important to one machine's inefficiency compared to another. To help curb this problem, statistics for system calls that are not included in the intersection are summarized as well and reported to the user along with the rules generated by our classification algorithm. Therefore, a user will see if any of the system calls, which are not in the intersection, are significant to the overall executions and thus could be related to the problem. For each system call not included, we calculate the same statistics as for calls that are included.

Once the intersection is found, we calculate four attributes for each call. The number of times a particular system call is executed and the total amount of time spent executing that call are both saved. The number of consecutive calls to each system call (i.e. number of write() calls when a write() is called immediately after) and the average time between these consecutive calls are also added. Additionally, users may specify a predefined configuration file to describe a list of system call arguments (if any) that need to be included as attributes. For instance, in this paper, we consider the return values of read() and write() system calls. Finally, each log file is represented by a row of statistics in the database file. The overall process is illustrated in Figure 3. We start with a set of raw good and bad strace logs. This is represented at the top of Figure 3. Those files are then processed and statistics (i.e., frequency of each system call, time spent in each system call) are calculated for each file. Finally, statistics from different log files are saved in a formatted log file where each line represents the summary of a single log file. A good or bad label is added to identify whether the line is a summary of a good or bad log.

C. Troubleshooting Component

We explore simple decision tree algorithms to generate rules that may shed light on underlying anomalous behavior [21]. Once detailed logs have been gathered, parsed, and labeled as good or bad, our classification algorithm attempts to build a set of rules that may explain the root cause. These rules tell the user which (combination of) attributes are different between good and bad executions and thus can help a user identify the problem.

Algorithm 1 Classification Algorithm

Input: Log summaries labeled as good or bad
CommonAttributeSet
Output: Top 10 classification rules
$ClassAttributeSet \leftarrow \{Good, Bad\}$
for $i = 1 \rightarrow 10$ **do**
$DecisionTree \leftarrow generateTree(CommonAttributeSet, ClassAttributeSet)$
$RootAttribute \leftarrow DecisionTree.root()$
if $RootAttribute = NULL$ **then**
$break$
else
$printRule(DecisionTree)$
$CommonAttributeSet.remove(RootAttribute)$
end if
end for

The main idea behind the algorithm is as follows. Each row in the input dataset represents one execution log, and consists of a class attribute (i.e., representing either "good" or "bad" in our case) and a set of numerical and/or categorical attributes. Please note that the set of attributes that are fed to the

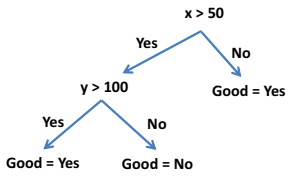

Fig. 4: Sample Decision Tree

algorithm are common across all the logs as explained earlier. The goal of the algorithm is to build a decision tree that can be used to predict the class attribute based on the other attributes. The decision tree implementation used in this paper, first, calculates the information gain of each attribute. Basically, the information gain of a given attribute 'X' is equal to the entropy, or randomness, of the raw training set minus the entropy of the training set given 'X' as a classification attribute. In other words, information gain is based on how well the attribute is at predicting the class. For details on information gain, interested readers are referred to prior work [21]. The algorithm choses the attribute with the highest information gain at each stage of the tree construction. Next, the tree builder splits the data into two sets using the chosen attribute. The new sets of data are then recursively split following the same process as above as long as the new split improves the prediction accuracy. Once a tree is created, each leaf node will represent instances of a particular class (e.g., good or bad). Each root-to-leaf path represents a rule that can be used to predict the class of the training instances that belongs to that particular leaf. Each node on a root-to-leaf path represents a condition that describes how it splits the data at that node. To get a rule, these node descriptions are combined with AND for a specific root-to-leaf path. Paths that end with leaf nodes labeled with the same class are then combined using OR. Figure 4 shows a sample decision tree. The rule generated for this tree would be, *if ($x < 50$) or (if $x > 50$ and $y > 100$) then the execution is good*. These rules are returned to the user so that they may identify the root cause behind the anomalous behavior.

Note, it is possible for the algorithm to pick an attribute in the first iteration that may not be the cause, but just happens to be the one with the highest information gain. Hence, once we build a tree, we then remove the root attribute, and attempt to rebuild the tree from scratch. However, after the removal of many attributes after multiple interations, possibility of generating junk rules may increase (e.g., due to over-fitting). Hence, we repeat the process a predefined number of times (e.g., 10 times in our case). In our current implementation, we report the top 10 rules to the user. In our experiments, we found this number to be large enough to gain enough information about differences between executions to guide us in the right direction to find the bugs. This value can be changed as needed for individual cases.

We implemented this system in Java as with all the other components of our tool. The decision tree construction is based on the jaDTi open souce decision tree implementation.

IV. Evaluation

To illustrate the wide applicability of our tool, we applied it in three different settings. In each case, the cause of the problem was different. In the first case, we introduced a configuration bug into a recently built cluster by changing the TCP buffer space, causing two of the six machines taking longer to download files. In the second case, the reported problem was a slow download speed on smart phones compared to laptops when they are under similar network conditions. In the third case, the problem is a configuration issue on VirtualBox. Our tool successfully identified the problem in all three cases. We elaborate each example below.

A. Case Study - I: TCP Memory Misconfiguration

Sometimes errors and misconfigurations during installation of software can cause performance issues down the line. These types of bugs, though very frustrating, are hard to locate. To test our tool, we created a scenario where a cluster of six server machines was set up, but two of the machines ended up with (for this experiment, intentionally) misconfigured TCP memory minimum, maximum, and threshold values, resulting in degraded network performance.

In our experiment, four of the machines were left with the default TCP memory values of 765792, 1021056, and 1531584. Two of the machines were set to have the values of 77, 102, and 153. We chose those incorrect values to be 10^{-4} of the appropriate values to simulate a programming error that used an incorrect multiplier in the setup of these machines.

We ran wget to fetch random 200MB sized files from a common server in the network and logged the statistics. Five different files of the same size were generated and each of those files was downloaded 25 times on each machine.

The logs were fed to our tool. Recall that the tool performs some pre-calculations based on the system call traces. For each trace, it calculates the number of times all the calls in the list are executed along with the sum of the time it took each call to execute across the entire trace. It then calculates the time between consecutive calls and averages these values for each combination of calls. For example, if a write() call immediately followed a write(), the difference in time from the start of the more recent write() and the end of the first write() was calculated and added to the running sum of total write() to write() times which was then used to calculate the average. For each system call, the result value was logged and averaged across an execution. This is important for some calls, like read()s, which return the amount of data read. For read() and write() calls, the tool also averages the value of one of the call's parameters, specifically the maximum amount of data that *could* be read or written by the call.

Figure 5 shows the download times for each experiment, color coded based on which machine the file was downloaded to. In this graph, a clear clustering can be seen between the two sets of machines.

Running the troubleshooting algorithm, including parsing and attribute selection, took approximately 130 sec. A Total of 958 attributes were fed to the classification algorithm. The

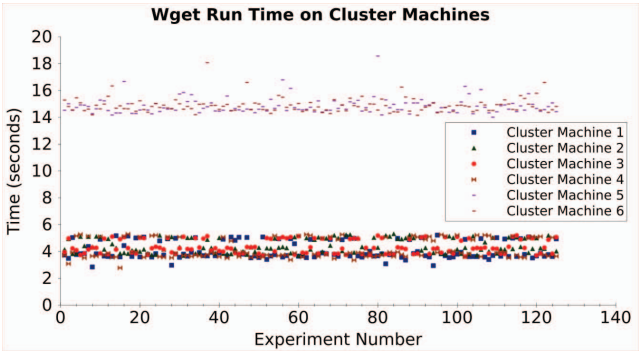

Fig. 5: Wget Run Times of Each Experiment on the Cluster Machines

Plot	Rule
6(a)	If numread $<$ 23631.5 then good = yes
6(b)	If numwrite $<$ 47116.5 then good = yes
6(c)	If timeselect $<$ 2.125 then good = yes
6(d)	If resultread $<$ 10823.0 then good = no
6(e)	If resultwrite $<$ 5367.0 then good = no
6(f)	If numselect $<$ 23624.5 then good = yes

TABLE I: Top 6 Rules for TCP Memory Bug

rules in Table I point out three system calls as key to the differences between the good executions and bad executions (the abbreviations used in the table are explained in Table II). We show six of the ten rules for the sake of space in Table I. Each of the rule in Table I has classification accuracy of 100% (i.e., each rule individually can classify all the training instances correctly). Four of the top rules all point to reading data from the socket and indicate that, in bad machines, a significant amount of extra time was spent on reading. The fourth rule in the table clearly shows that the average amount of data read into the buffer is always lower on the bad machines. Figure 6(d) highlights this difference. Figure 6 show the vast difference between these values in the execution logs. Since the issue was network related and our rules suggested that bad machines were able to read fewer amount of data each time and needed more read() calls to read the same amount of data as the good machines, it is not a jump to assume a system administrator would look into available memory to network applications. This would lead him to the tcp_mem file

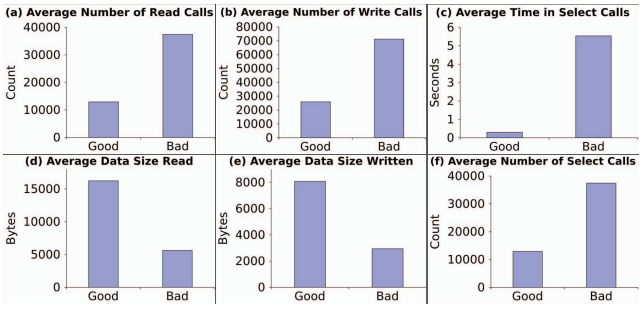

Fig. 6: Comparison of Key Rule Attribute Values in Good and Bad Machines

which sets the amount of memory available to TCP, which is where the bugs are.

Attribute	Interpretation
numread	Number of read calls
numwrite	Number of write calls
numselect	Number of select calls
timeread	Time in read calls
timewrite	Time in write calls
timeselect	Time in select calls
resultread	Data size read
resultwrite	Data size written
paramread	Max possible read size

TABLE II: Attribute to Interpretation Mappings for Case Studies I & II

B. Case Study - II: Android Network Delay

While studying the network performance of smart phones, a research group at the University of Connecticut (UConn) observed that smartphones had slower network speeds than other machine types (e.g., laptops, desktops) even when they are under similar network conditions (in the same wireless network, using the same network protocols, downloading data from the same server, and using the same TCP parameters). We were approached to further study the problem and attempt to identify a cause or causes for the delay. Interestingly, the situation is of importance to cloud service providers, where some clients (e.g., laptops) may have better performance than some other clients (e.g., smartphone users).

To reproduce the delay as identified by the prior research group, a simple set of tests were ran. The devices used for this case study include a Toshiba Portege R705-P35 Laptop with a dual core Intel i3 processor, 4GB of RAM, running CrunchBang Linux version 10, and a Samsung Nexus S smartphone with a single core Samsung Exynos 3110 processor, 4GB of RAM, running Android 4.0.4. Random files of sizes 1MB, 5MB, 10MB, 20MB, and 30MB were downloaded 10 times to the Android smartphone and Linux laptop using wget version 11. Figure 7 shows the average runtime of wget for each file over 10 trials on both a controlled and an uncontrolled network. In the controlled network, the server machine was connected directly to a wireless router and only one client machine was attached to the router during the experiment. In the uncontrolled experiment, the devices were using UConn's Wi-Fi network.

Plots	Rule
8(a)	If numread $<$ 5304.0 then good = no
8(b)	If resultread $<$ 2120.0 then good = yes
8(c)	If timewrite $<$ 0.366 then good = yes
8(d)	If paramread $<$ 14180.0 then good = no
8(e)	If numwrite $<$ 5328.0 then good = no
8(f)	If resultwrite $<$ 2116.0 then good = yes

TABLE III: Top 6 Rules for Android Bug

As can be seen in Figure 7, in both controlled and uncontrolled experiments, for the tested file sizes, the laptop outperformed the phone. The difference also increases as the file size increases. For troubleshooting, we downloaded a 10MB random

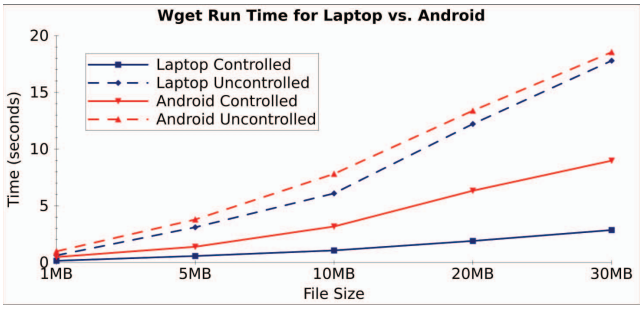

Fig. 7: Wget Run Times for Android and Linux Machines

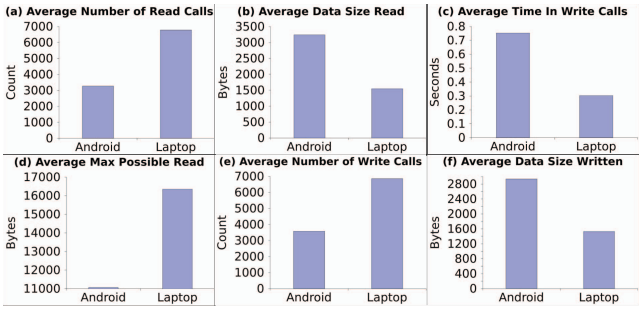

Fig. 8: Comparison of Key Attribute Values in Android and Laptop

file via wget 50 times on both the phone and laptop in the controlled setting as before and logged statistics on both devices. The logs were processed by our tool and the rules shown in Table III were generated (the abbreviations used in the table are explained in Table II). Each of the rule in Table III has classification accuracy of 100% (i.e., each rule individually can classify all the training instances correctly). Again, only six of ten are shown here for space concerns. Running the troubleshooting algorithm, including parsing and attribute selection, took approximately 3 sec. A total of 508 attributes were fed to the classification algorithm. In this case, a "good" execution is one that comes from a laptop, thus these rules differentiate between Android and laptop. All of the rules generated deal with read()s and write()s, except for two which identify open() and close().

Interestingly, as can be seen in Figure 8, the average number of write calls is significantly higher in laptop runs. However, write calls were much faster on the laptop. Hence, Android ends up wasting a significant amount of time in writing data. Over the course of a run, the I/O delay between consecutive write calls adds up for the Android and was the prime contributing factor of the delay. It is important to note that the laptop would read, and thus write less data per call, but do it all at a much faster rate than that on the phone. This led us to the conclusion that a file system issue was the primary cause of the slowdown, rather than networking issues which was the initial guess. This example highlights the importance of our tool, which can guide the developer in the right direction. Days of effort were wasted in tweaking network parameters to fix the problem, whereas the problem was something else. Currently, we are working to identify the root cause behind the "slow" I/O rate on Android. It is important to note that the cause of this "slow" I/O can be related to slow hardware and/or an inefficient file system software implmentation as well as other software issues.

C. Case Study - III: VirtualBox Misconfiguration Identification

Part of the process for the Android experiment required downloading the Android source code from the Google repositories. To do this, on machines running Windows, Ubuntu 11.10 was installed on top of VirtualBox 4.1.8 so that the convenient Linux binaries for *git* and *repo* could be used. When the download of the source was started (using the "repo sync" command) with all default parameters used in VirtualBox, it would run for a while, then fail, returning a fetch error. Numerous failed attempts were made.

Unlike the previous test cases, this case did not initially seem to be caused by some sort of low level misconfiguration or inefficiency. After all, the download was starting smoothly and was continuing for a while before being terminated. Initial guesses included the possibility of network limit on campus network, which resulted in few email exchanges. However, this proved not to be the case. Interestingly, downloads of the source were successfully completed on machines natively running Ubuntu relatives (CrunchBang). Further research showed that others were able to download the code in VirtualBox.

The guess was that the error was being caused by a problem in VirtualBox, maybe with some configuration that, combined with the University's network, caused the connection to die and thus resulted in the fetch errors. Eventually, the root cause turned out to be a simple one. When the NAT adapter type was used for the network connection, the download failed, but when the bridged connection was used, it succeeded. Fixing this misconfiguration on the machines that previously had fetch errors allowed the download to complete.

Looking back, we collected three VirtualBox configuration files, two where the download failed and one where it completed. Even with this small set, our tool was able to identify the network adapter as the first rule, and thus the best indicator as to whether a download will fail or not. The rule generated was *if adaptertype = bridged then good = yes*, directly identifying the configuration bug that has been extremely hard to detect. This example demonstrates that if users volunteer their configuration files whether they have a problem or not, these files can be used to help identify bugs related to misconfigurations using our tool easily.

V. Limitations and Future Work

Stracing can be a very intensive procedure since it requires constant monitoring of executions so as to catch the system calls and regular hard disk accesses to log the calls. Because of these invasive aspects of strace, we analyzed the execution overhead of stracing a program, in this case wget on both the laptop and Android phone.

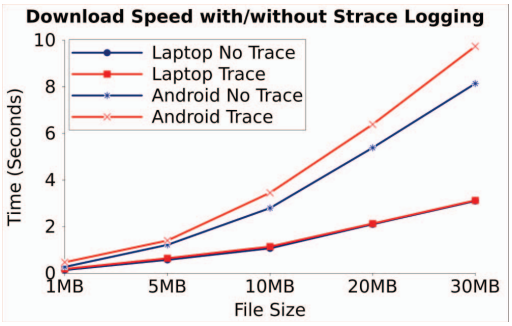

Fig. 9: Wget Execution Time Averages with and without Strace Logging

Figure 9 shows the average execution times of wget fetches for a 1MB, 5MB, 10MB, 20MB, and 30MB. For each data point, the file was fetched 25 times. As the graph shows, strace added very little overhead to the laptop executions. However, as can be seen see in Figure 9, the overhead of stracing on the Android platform was much more significant. For the 30MB files, 1.6 seconds on average were added. The likely reason for the higher overhead on the Android phone is related to the I/O bottleneck identified as the cause of the network slowdown. Fortunately, for a given edge client, stracing of a command will only have to be done once and so even a moderately large delay will be experienced only once by the user.

Another overhead issue is the communication of possibly very large strace files from clients to the central server. For example, some of the logs from sick machines in the cluster experiment reached over 20MB. The transmission of files of this size from many edge clients can become troublesome. To help curb this problem, we are exploring the possibility of parsing the logs on client machines. Doing so will increase system overhead for the client, but would vastly reduce the amount of data to be sent since the number of attributes that are generated is static. For instance, for the cluster experiment, a file containing data from a single parsed strace was only 2KB in size, a clear improvement over 20MB.

Finally, since the logs are not needed immediately, our client application could wait for an optimum time to transmit the logs. This could be a time when network usage is low, possibly late at night or early in the morning, thus limiting the effect on the client's user experience.

VI. Conclusion

In this paper, we present a tool that leverages service providers' infrastructure to troubleshoot edge computing systems. As we leverage kernel level utilities for troubleshooting, the tool does not require access to application source code. We evaluated our tool using one artificially injected bug and presented two real-life case studies. We also show that the overhead is tolerable and incurred only once for a particular case. To the best of our knowledge, this is the first paper to propose troubleshooting edge clients using heterogeneity to identify and debug performance issues on edge devices. We strongly believe that such a tool is mutually beneficial and of great importance for both the service provider and the clients.

Acknowledgment

This work is supported by the GAANN Fellowship under sponsor number P200A100141. The views and conclusions expressed in this paper are those of the authors and should not be interpreted as the opinion of the funding agency.

References

[1] "Fate and destini: A framework for cloud recovery testing," in *Proceedings of NSDI*, 2011.
[2] B. L. Trishul Chilimbi and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling?" in *Proceedings of ICSE*, 2009.
[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings of DSN*, 2002, pp. 595–604.
[4] A. Zeller, "Automated debugging: Are we close," *Computer*, vol. 34, no. 11, pp. 26–31, Nov. 2001.
[5] I. Cohen, M. Goldszmidt, T. Kelly, S. Julie, and J. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *Proceedings of OSDI*, December 2004.
[6] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: Troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of SenSys*, 2008, pp. 99–112, raleigh, NC, USA.
[7] G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi, "Distributed diagnosis of failures in a three tier e-commerce system," in *Proceedings of SRDS*, Beijing, CHINA, October 2007, pp. 185–198.
[8] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of SOCC*, New York, NY, USA, 2010, pp. 193–204.
[9] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of SOSP*, 2003, pp. 74–89.
[10] K. Havelund and G. Roşu, "An overview of the runtime verification tool java pathexplorer," *Form. Methods Syst. Des.*, vol. 24, no. 2, pp. 189–215, 2004.
[11] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Proceedings of ECRTS*, 1999.
[12] J. Heo and T. Abdelzaher, "Adaptguard: Guarding adaptive systems from instability," in *Proceedings of ICAC*, Barcelona, Spain, June 2009.
[13] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of OSDI*, Berkeley, CA, USA, 2004.
[14] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of OSDI*, ser. OSDI'10, Berkeley, CA, USA, 2010.
[15] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of SOCC*, New York, NY, USA, 2011, pp. 9:1–9:14.
[16] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A self-organized, fault-tolerant and scalable replication scheme for cloud storage," in *Proceedings of SOCC*, New York, NY, USA, 2010, pp. 205–216.
[17] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *Proceedings of SOCC*, New York, NY, USA, 2010, pp. 155–160.
[18] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," in *Proceedings of LISA*. Berkeley, CA, USA: USENIX Association, 2003, pp. 159–172.
[19] P. A. Nainar and B. Liblit, "Adaptive bug isolation," in *Proceedings of ICSE*, 2010.
[20] P. Reynolds, "Pip: Detecting the unexpected in distributed systems," in *Proceedings of NSDI*, 2006.
[21] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.