Taking the Blame Game out of Data Centers Operations with NetPoirot

Behnaz Arzani
University Of Pennsylvania
barzani@seas.upenn.edu

Selim Ciraci
Microsoft
seciraci@microsoft.com

Boon Thau Loo
University Of Pennsylvania
boonloo@cis.upenn.edu

Assaf Schuster
Technion
assaf@cs.technion.ac.il

Geoff Outhred
Microsoft
geoffo@microsoft.com

ABSTRACT

Today, root cause analysis of failures in data centers is mostly done through manual inspection. More often than not, customers blame the network as the culprit. However, other components of the system might have caused these failures. To troubleshoot, huge volumes of data are collected over the entire data center. Correlating such large volumes of diverse data collected from different vantage points is a daunting task even for the most skilled technicians.

In this paper, we revisit the question: how much can you infer about a failure in the data center using TCP statistics collected at one of the endpoints? Using an agent that captures TCP statistics we devised a classification algorithm that identifies the root cause of failure using this information at a single endpoint. Using insights derived from this classification algorithm we identify dominant TCP metrics that indicate where/why problems occur in the network. We validate and test these methods using data that we collect over a period of six months in the Azure production cloud.

CCS Concepts

•**Networks** → **Transport protocols; Network performance analysis; Network measurement;** *Network components; Network reliability; Network monitoring;*

Keywords

Data Centers; TCP; Network Diagnostics

1. INTRODUCTION

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil

DOI: http://dx.doi.org/10.1145/2934872.2934884

In today's data centers, a common issue faced by operators is troubleshooting faults in complex services. It is often unclear whether the cause of performance bottlenecks lies in the underlying network, client, or service-level application code/machine. Often times, the knee-jerk reaction is to first blame the network whenever performance issues surface.

The problem is exacerbated by two issues. First, the parties involved in diagnosing the errors (e.g. the service DevOps engineers, network operators) may operate as different entities either within or across companies and each party may lack easy access to the other's performance/debugging logs. This significantly increases debugging time and extends the mean time to recovery. Second, these failures are sometimes intermittent and non-deterministic, and hence difficult to reproduce without high fidelity, always on, monitoring probes in place throughout the entire infrastructure.

To highlight these challenges, we present a real-world example seen within our production cloud. Our public cloud offering is used by over 1 billion customers, whose applications reside within VMs in our data centers. In one particular scenario, the VM triggers an operation within the hypervisor, that requires the hypervisor to send a request to a remote service. Whenever the request/response latency increases (due to remote service failure, overload, or network slowdowns), an error occurs in the hypervisor which in turn causes the VM to panic and reboot, hence disrupting normal operations. We refer to this as "Event X or EX".

EX occurs intermittently and only on a subset of nodes at any given time. However, its occurrence is sufficiently frequent to degrade user experience. It is also unclear when a reboot happens, whether the error is caused by a remote service issue (overload, server failure), or a network problem (e.g. packet drops or router misconfigurations). In one example, when EX occurred the remote service DevOps team was first contacted. They suspected that EX may be occurring due to high request/response latencies, and blamed the network, and passed on the diagnostics to the network engineers. The network engineers observed normal RTT times in TCP traces at the time of failure and suspected that the problem is probably due to slow server responses, and handed

over the issue back to the server operations team. The iterations continued until the various teams involved pieced together the sequence of events that led to the service disruption. The entire process is time-consuming, expensive, and involves many engineers and developers spanning different organizations. To make matters worse, while the symptoms of EX may be the same, the actual cause may differ across occurrence, and fixing one occurrence of EX may not prevent others from happening in the future. Note that such lengthy debugging across multiple subsystems is an issue not unique to our production cloud, and has been reported elsewhere as well [26, 12].

While many network diagnosis tools have been developed in the past, all of them come short in some way when solving problems such as EX. Some require access to the entire system [28], others are too heavyweight and so cannot be used in an always on fashion [29], and finally some require information that the service/network are not willing to share [4, 28]. Many of the proposed concepts in these tools [4, 28, 31] have been designed, developed, and adopted by our organization. Their failure to diagnose EX is a motivation for a more effective tool.

In the ideal case, one would like to quickly pinpoint the *most likely* source of the failure within the infrastructure. The team (client, network, or remote service) that owns the failure can then provide a timely response, rather than have the error be passed around within organizations.

To achieve this goal, we propose NetPoirot[1] that aims to perform such failure attribution quickly and accurately. NetPoirot requires only a TCP measurement agent to be deployed at each client VM in the cloud. The key insight of NetPoirot is that different types of failures, albeit not network related, will cause TCP to react differently. For example, a slow reading remote service results in exhaustion of the TCP receive window on the sender VM, which itself triggers TCP zero window probing. Packet drops on a router result in an increase in the number of duplicate ACKs. High CPU load on the client result in fewer transmission opportunities for the client application and thus fewer data sent by TCP. These differences are not always easy to define, given the high correlation between the various TCP metrics. Additional information such as SNMP and network topology, if available, can be utilized to improve the accuracy and scope of the predictions made by NetPoirot. However, in practice, these information may be unavailable or expensive to collect in high fidelity. Hence, the goal of our work is to identify the extent to which failure diagnostics can be achieved simply by using data available through the clients.

NetPoirot make the following contributions:

Lightweight continuous monitoring. We develop a TCP monitoring agent that runs on all client machines within our data centers. These machines request services within and across data centers under our administration. The agent captures various TCP related metrics periodically. It is implemented in Windows, although a similar tool in Linux can be supported. Unlike more heavyweight approaches that require packet captures, NetPoirot's agent is lightweight and non-intrusive, requiring only aggregate TCP statistics to be periodically collected and measured. At runtime, the agent requires only 132 numeric values be examined (and then discarded) by the client machine every 30 seconds.

Machine-learning based classification. Using data collected from our agents, we study the extent in which TCP statistics can be used to distinguish various failures in data centers using decision-tree based supervised learning algorithms. We then identify the key parameters in TCP that are most representative of each type of failure in data centers.

Our approach is a significant improvement over techniques previously deployed at Microsoft such as SNAP [28]. SNAP manually reasons about each type of failure and devises a rule for each problem accordingly. Such manual inspections are limited in scope, cannot capture the dependency between the various metrics in the presence of failures, and are prone to human error. We found that many of the TCP metrics selected by NetPoirot in identifying failures are not even monitored by SNAP. For these reasons, in the design of NetPoirot we opted to use machine learning instead, which allows us to uncover more complex insights. Our use of decision trees allows us to identify the dominant set of TCP metrics that help to classify each failure. Furthermore, NetPoirot does not require any knowledge of the topology, or client to server mappings.

NetPoirot implementation. We have designed and developed a proof-of-concept implementation of NetPoirot that combines the above two ideas and is used for diagnosis at runtime. NetPoirot consists of a training phase, where a variety of faults are injected onto training VMs, in order to produce a diagnosis function as output. This function is then distributed to all machines in the data center to be used at runtime to identify the source of failures. NetPoirot can be used by both *customers* and *data center operators* as it does not require any information from the network or service.

NetPoirot evaluation. We perform an extensive evaluation of NetPoirot to validate its effectiveness. We evaluate the worst case performance of NetPoirot using data collected over a 6 month period in a production data center that hosts over 170K web servers and transfers 10Tbps of traffic. We have induced 12 different common types of failures of two different applications running on 30 different machines and observed the changes in TCP statistics collected from these machines. NetPoirot performs coarse-grained blame allocation on this data with high accuracy (96% for some failure types). NetPoirot's accuracy improves even further if its input information is augmented with additional information from the endpoint on which it resides. Our monitoring agent has currently been deployed on all the compute nodes within our data centers. We show that NetPoirot can accurately identify the entity responsible for a variety of failures.

The tradeoff that NetPoirot makes is that while it determines the entity (network, client, remote service) responsible for the failure, it does not pinpoint the exact physical device. We argue this is a worthwhile tradeoff as identi-

[1]Named after Agatha Christie's famous detective Hercule Poirot.

fying the entity is often times the operationally most time-consuming (hence expensive) part of failure detection, e.g. EX failures take from 1 hour (high severity) to days to diagnose. In fact, the development of NetPoirot was commissioned as a direct result of this problem in the case of EX.

In other words, NetPoirot *does not* eliminate the need for more heavyweight tools but instead helps pinpoint which tools to use and by whom. For example, to isolate sources of failure at the client, *the user*[2] may need to use gdb, top, etc. However, to pinpoint sources of failure in the network, network operators use TCPdump, EverFlow [31], etc. Furthermore, many failures are sporadic. Therefore, without always on monitoring, many of the available debugging tools will fail to diagnose the failure. In such situations, finger pointing is the most that one may hope for in blame attribution.

While we do not claim that we can diagnose *all* potential problems, our results indicate that it is possible to distinguish between a number of common network, server, and client failures with high accuracy. This is significant, as we do not need to use per-flow TCP metrics, nor do we need data from both endpoints. Since only aggregate metrics are collected, our methodology allows us to maintain some extent of customer privacy.

2. OVERVIEW

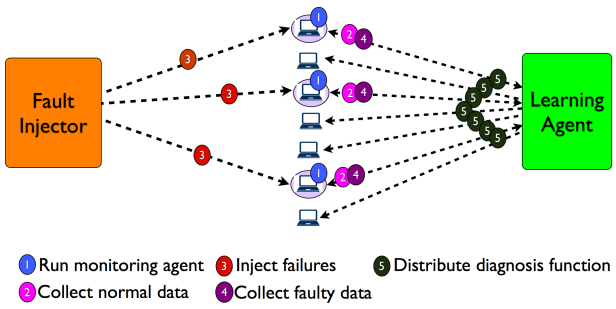

Figure 1: Overview of NetPoirot

We first begin with an overview of NetPoirot, which consists of two phases: *training* and *runtime*.

Training phase. Fig. 1 depicts the training phase. The training phase occurs in the production environment on a subset of machines, termed as the *training nodes*. During this phase, a TCP *monitoring agent* is installed on these nodes. Details on the monitoring agent are presented in Section 3. The agent can be either installed at the hypervisor or within individual VMs hosted by the training nodes. When there are no failures, agent statistics are periodically sent to the learning agent to capture the behavior of a non-fault scenario. From time to time, we use a *fault injector* to inject a variety of failures listed in Table 1. We define the "client" as the local machine that is communicating with a remote service, which we refer to as the "server". Failures range from increased CPU load, increased memory load, increased storage load (by generating excess I/Os on either the client or server using SQLIO), and also various forms of network-related problems such as throttling or packet drops.

[2]Here, by user we mean the owner of a VM in the data center or in the case of EX the operator maintaining the hypervisors. Note, that the operators maintaining the remote service will also utilize similar tools for diagnosis.

General label	Failure	How it is induced
Server	High CPU load on server	Application running kernel level CPU intensive operations
	Slow reading server	Modified the server application, added a random delay to its read operation
	High I/O on server	SQLIO [19]
	High memory load on server	Testlimit [20]
Client	High CPU load on client	Application running kernel level CPU intensive operations
	High I/O on client	SQLIO 1 [19]
	High memory load on client	Testlimit [20]
Network	Bandwidth throttling	Added rules in the A10's
	Sporadic packet drops	SoftPerfect [1]
	Packet reordering	SoftPerfect [1]
	Random connection drops	NEWT [21]
	High Latency	SoftPerfect [1]

Table 1: Table describing the types of faults we induced in our system for training.

For each injected failure, the monitoring agent's TCP statistics are collected and used by the learning agent to create a *diagnosis function*, trained using as the label, either the actual failure itself or its type (e.g. client, remote, network). The diagnosis function is the output of our supervised learning algorithm. The input to the function is the latest TCP statistics read from the end point, and the output is the entity most likely to be responsible for the failure. This function is broadcast to all VMs at the end of the training phase.

Runtime phase. At run time, all monitored nodes run the monitoring agent. Whenever failures are detected, each VM invokes the diagnosis function using digests collected locally and then generate as output the likely source of failure.

3. DESCRIPTION OF NetPoirot

In this section, we describe the various components of NetPoirot. NetPoirot is comprised of a monitoring agent that collects data that is then input to a learning agent. The learning agent uses this information to create a diagnosis function that is used at runtime. The following describes the details of each of these components in more detail.

3.1 Monitoring Agent

Our TCP monitoring agent is installed at each machine's hypervisor or within individual VMs. The installation is limited to only client machines communicating with various remote services within/across data centers. For example, if the remote service is storage, this precludes the need to run the agent on storage servers. The agent collects TCP statistics for all connections seen on its monitored node. Given that

metric	statistics calculated	abbreviation
Number of flows	$\mathcal{R}$	NumFlows
Maximum congestion window in δ	$\mathcal{S}$	MCWND
The change in congestion window in δ	$\mathcal{S}(*)$	DCWND
The last congestion window observed in δ	$\mathcal{S}(*)$	LCWND
The last advertised (remote)receive window observed in δ	$\mathcal{S}(*)$	LRWND
The change in (remote) receive window observed in δ	$\mathcal{S}$ (*)	DRWND
Maximum smooth RTT estimate observed in δ	$\mathcal{S}$ (*)	MRTT
Sum of the smooth RTT estimates observed in δ	$\mathcal{S}$ (*)	SumRTT
Number of smooth RTT estimates observed in δ	$\mathcal{S}$ (*)	NumRTT
Duration in which connection has been open	$\mathcal{S}$	Duration
Fraction of open connections	$\mathcal{R}$	FracOpen
Fraction of connection closed	$\mathcal{R}$	FracClosed
Fraction of connections newly opened	$\mathcal{R}$	FracNew
Number of duplicate ACKs	$\mathcal{S}$ (*)	DupAcks
Number of triple duplicate ACKs	$\mathcal{S}$ (*)	TDupAcks
Number of timeouts	$\mathcal{S}$ (*)	Timeouts
Number of resets	$\mathcal{S}$ (*)	RSTs
Time spent in zero window probing	$\mathcal{S}$	Probing
Error codes observed by the socket	$\mathcal{R}$	Error Code
Number of bytes posted by the application	$\mathcal{S}$	BPosted
Number of bytes sent by TCP	$\mathcal{S}$	BSent
Number of bytes received by TCP	$\mathcal{S}$ (*)	BReceived
Number of bytes delivered to the application	$\mathcal{S}$ (*)	BDelivered
Ratio of the number of bytes posted by the application to the number of bytes sent	$\mathcal{S}$	BPostedToBSent
Ratio of the number of bytes received by TCP to the number of bytes delivered	$\mathcal{S}$	BDToBRC

Table 2: Features captured by the monitoring agent during each epoch. We use $\mathcal{R}$ to show that the raw value of a feature is captured and $\mathcal{S}$ to show that we capture the statistics of that feature. (*) indicates normalized metrics.

our implementation is based on Windows, we will describe the agent based on Windows terminology. These statistics can also be collected in a Linux-based system.

The agent is implemented using Windows ETW events [18], a publish-subscribe messaging system in the Windows OS. A TCP ETW event is triggered every time a TCP related event, e.g. the arrival of a duplicate ACK occurs on any one of the connections currently active in the OS. The agent collects and aggregates events at the granularity of *epochs* so as to minimize bandwidth/storage overhead during training. Within every epoch, it receives ETW events, extracts relevant features and stores them in a hash table based on TCP's $5-$tuple. At the end of an epoch, the TCP metrics that depend on the transmission rate are normalized by the number of bytes posted by the application in that epoch. The normalized metrics are marked in Table 2. Each individual metric is then further aggregated by calculating its mean, standard deviation, min, max, 10^{th}, 50^{th}, and 95^{th} percentile across all TCP connections going to the same destination IP/Port. Given that NetPoirot only captures statistics from TCP connections using ETW events, it is potentially possible to instrument the hypervisor and the Windows OS (as part of Windows updates), so that the hypervisor captures these events and tags them with the process/customer ID and by doing so avoid having to install software on all customer VMs. These modifications are part of our future work.

We assume that identical failures happen within a single epoch, e.g. if a connection experiences failure A, then all other connections between the same endpoints in the same epoch either experience no failure, or also experience A. Thus, the epoch duration needs to be carefully tuned. Small epochs increase monitoring overhead as more data needs to be stored, but large epochs run the risk that sporadic failures of different types will occur within one epoch, affecting the accuracy of the learning algorithm. We currently use an epoch of 30s which we found to work well in practice. Fine tuning the epoch duration is part of our future work.

Table 2 shows the features maintained within an epoch by the monitoring agent. Our aggregation method reduces the amount of bandwidth required on the machines in the training stage[3] and has the added benefit of hiding the clients exact transmission patterns. Furthermore, when applications change their transmission pattern across connections in reaction to failures it allows for this change to be detected. In the other extreme, one may decide to use per connection statistics with more overhead but with the benefit of detecting why each individual connection has failed separately.

The agent imposes low runtime overheads. Based on our benchmarks, even in the absence of aggregation, when processing $500,000$ events per second, each agent uses 4% CPU load on an 8 core machine and less than 20 MB in memory.

3.2 Learning Agent

During the training phase, the learning agent takes as input TCP metrics gathered by monitoring agents on training nodes. At run time, it distributes the learned model to all clients to be used for diagnosis. The model has to quickly classify epochs with the appropriate labels to indicate whether it is a remote (Server), local (Client), or Network issue.

The learning agent uses decision trees as its classification model. In a decision tree, each internal node conducts a test on an attribute, each branch represents the outcome of the test, and the leaf nodes represent the *class labels*. The paths from root to leaf represents the classification rules.

In our setting, the internal nodes correspond to one of the aggregated TCP metrics being monitored. The learning phase determines the structure of the decision tree, in terms of the choice of attributes and the order in which they are used for testing along the path from the root to label (this ordering is determined by the information gain of features in

[3]Without aggregation, the client needs to transmit $31n$ features every epoch to the learning agent where n is the number of connections during that epoch. With aggregation, this number is reduced to 130.

the dataset). The specific nature of the test at each node, i.e. the inequality tests, is also determined in this phase.

As noted by prior work [8, 3, 9], the structure of decision trees allows for further understanding of the attributes that identify each failure. For this reason, we found decision trees more attractive to use than other machine learning approaches. We will elaborate further on this in Section 4.

Fig. 2 shows an example of a decision tree, that distinguishes packet reordering from normal data. Leaf colors in the figure represent the labels of the training data that ended up in those leaves. Most leaves are "pure", i.e. all the data in those nodes have the same label. Leaf 2 shows an "impure" leaf that has a mix of both labels. In such situations, the tree picks the majority label in the leaf as its diagnosis.

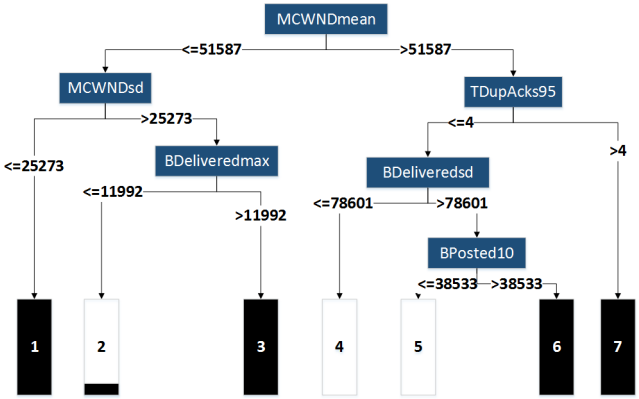

Figure 2: Example tree. The white/Black leave colors illustrate the labels of the training data that end up in that leaf.

Based on the concept of decision trees, our learning agent requires three enhancements for improved stability and accuracy:

Random forests. Our learning agent uses an enhanced type of decision tree, known as *random forests* [5][4]. In random forests, multiple decision trees are generated from different subsets of the data, and the classification is majority-based, where a majority is defined based on a cutoff fraction specified by the user. For example, a cut-off of $(0.2, 0.8)$ indicates that for class 1 to be chosen as the label, at least, 20% of the trees in the forest need to output 1 as the label as well. Random forests improve stability and accuracy in the face of differences in machine characteristics and outliers.

Multi-round classification. To improve accuracy, we do *rounds* of classifications. First, the forest is trained to classify Network failures only. The Server and Client failures in the training set are labeled as non-faulty (Normal) in this phase. Next, the Network failure data is removed from the training set, and a new forest is trained to find Server failures with Client failures labeled as Normal. Finally, the Server data is also removed and a forest is trained to identify client-side failures. At run time, data is first passed through the first forest, if classified as Network, the process terminates. If it is classified as Normal, it is passed through the second forest. Again, if it is classified as Server failure the process terminates. If not, the data is passed through the third forest and is assigned a label of Normal/Client. In machine learning, such multi-round classifications are referred to as *tournaments*. In traditional tournaments, different decision trees are used in pair-wise competitions. Our tournament strategy is a modification of standard tournaments, as they did not work well in our setting.

Per-application training. Applications react to failures differently. One application may choose to open more TCP connections when its attempts on existing connections fail while others may keep retrying on the ones currently open. Some form of normalization, such as that we use for the monitoring agent, helps avoid dependence on the transmission rate of the client itself. However, it does not help avoid this particular problem given that the effects of application behavior go beyond the transmission rate but also influence the number of connections, their duration, etc. Indeed, these behaviors themselves improve NetPoirot's accuracy as they provide more information about the failure. Hence, it is advised to train NetPoirot for each application separately. We argue that unless applications change drastically on a daily basis, there is sufficient time in between major application code releases and deploys for the model to be updated.

Two-phase tree construction with cross-validation. Each forest is constructed in two phases. First, given the training set, we determine basic parameters of the forest, e.g., its cut-off value and a minimum number of data points required in a leaf node. The latter is required to bound the tree sizes and to avoid overfitting. Once these parameters are determined, the training set is used to create the actual forest.

One of the pitfalls of any machine learning algorithm is the danger of *overfitting*, where the trained model is tailored to explicitly fit the subsequent testing set. This leads to poor future predictive performance. To avoid overfitting, we apply a standard machine learning technique, namely a modified variant of *cross validation* (CV). In a nutshell, the first phase is accomplished using *N-fold* [7] CV which estimates error using subsets of the training data, while the second phase builds the model using all the training data.

In the classic form of N-fold CV, the training data is randomly divided into N subsets (folds). Iterating over all folds i, in each iteration i is omitted from the training set and the model is trained over the remaining $(N-1)$ folds. The trained model is then tested using the i^{th} fold. The estimated errors in each iteration are averaged to provide an estimate of the model's accuracy. N-Fold cross-validation, however, if used on our data set runs the danger of overestimating accuracy as models will learn *specific* machine/network characteristics when data from one machine *leaks* between folds. Therefore, we divide data from each machine into its own unique fold. We define *CV error* as the average error of CV when each fold contains data from a single machine.

To show why this is important, we tested cross-validation on our data set using both methods[5]. Using vanilla cross-validation, we observe an error of 1.5%. However, when we

[4]Version 4.6-10 in R version 3.2.1.

[5]This was done without the use of tournaments.

partition the data based on machine label, we get 10.55% error. This further indicates that if data from the same machine is used for both training and at runtime one may get much higher predictive accuracy than those reported in this paper.

Normalization. We normalize TCP statistics that depend on the data being sent. Namely, features marked with $(*)$ in Table 2 were divided by the number of bytes posted by the application in that epoch in order to minimize dependency on the application's transmission pattern.

4. TCP BEHAVIOR UNDER FAULTS

The hypothesis behind NetPoirot is that the different types of failures, albeit not network related, cause TCP to react differently. To study this further, we examine the changes in various TCP parameters in the presence of different failures. Decision trees allow for not only classifying faults, but also illuminating features (in this case TCP metrics) affected by each failure. It is this basis that allows us to develop NetPoirot and why we use decision trees in this section.

The algorithm used in NetPoirot is agnostic to the choice of decision tree algorithm as we use random forests which rely on weak classifiers as their basis. With regards to the results in this section, we experimented with different types of decision trees, and will present our results based on a decision tree algorithm called *C5.0* [16]. We used the 0.1.0.24 version of C5.0 in the 3.2.1 version of R. C5.0 is based on *information gain* and aims to greedily reduce the amount of uncertainty with respect to the data's label. Nodes higher up the tree provide the most information gain (the most reduction in uncertainty) with respect to the output.

For each type of failure, we train a decision tree from data that includes that failure as well as Normal data. For each failure, we select the top three TCP metrics (features) in the tree. For each of these top features, we also measure the correlation between its value and the actual ground truth. This is done by computing the Pearson correlation (PC) between each feature value and the corresponding ground truth label (encoded as 0 for normal, and 1 for faulty) when that feature value was recorded. The PC value is then computed across all epochs for the duration in which the failure occurs.

PC is a measure of the *linear* correlation between the feature values and the labels (failure type/Normal) and provides further insight into the level of (linear) dependency between the features. PC's value ranges between $(-1, 1)$. The closer the absolute PC is to 1, the higher the linear correlation and therefore the easier it is to identify such correlations.

While PC is not required by NetPoirot for classification, we calculate the PC value so as to provide insight into how easy it would be for an operator to simply decipher the relationship between the failure and the metrics used by the decision tree. The human brain is relatively good at identifying linear relationships between variables. Our results show that often times, the relationship is non-linear, meaning that a manual classification approach is not likely to work.

4.1 Fault Injection and Workloads

Failure injection. Supervised learning requires labeled data for training. In order to train our model, we injected failures in the communication pattern of two different types of applications running on a subset (30) of the VMs in four of our production data centers located in West and Central USA, North and West Europe. Our network serves over 1 billion customers and handles petabytes of traffic per day. Given that networks in a data center environment are highly symmetric, training on a small subset of machines in a given cluster is sufficient for high accuracies at runtime. Some data centers already purposefully inject faults on a regular basis in their production environment in order to evaluate their degree of fault tolerance [25]. The traffic of the nodes in the path of these failures can be observed for training purposes.

Application workload. Applications are designed with a degree of fault tolerance, and, therefore, react to any failures that might occur. Given that NetPoirot uses per application training, such reactions result in additional changes in the TCP metrics that help NetPoirot isolate the type of failure. However, to isolate the impact of failures on TCP behavior (and not applications), we experiment using two applications that were designed to remain passive when failures occur. The first is a *duplex* application, where we recorded a hypervisor's communication to a remote service (the one that causes EX) for 6 hours. This trace is then replayed to and from the server. There is no fault tolerance logic in the application. Therefore, the duplex application mimics the behavior of a typical application in our production data center but without any additional logic that might affect the TCP connection. The second application is a *simplex* application that opens 128 connections[6] to a server and sends a constant number of bytes on each connection every second. We use three sending rates: 100, 500, 10000 Bps.

These applications are designed to capture the *worst case* performance of NetPoirot. For example, the simplex application involves communication in one direction. Therefore, some metrics that depend on communication from the server back to the client would not be captured. The duplex application, on the other hand, is an extreme application that does not react to failures. Overall, the simplex application results in decision trees with higher CV errors (Section 3.2) as a result of fewer metrics being influenced by the failure.

Our data is gathered over a period of 6 months (July-December 2015). All datasets are labeled with the corresponding machine ID where the data is collected, to be used for cross-validation (CV) described in Section 3.2.

4.2 Results

We inject failures listed in Table 1 to study their impact on TCP. Tables 3 and 4 summarize our main findings. We limit our investigation to the top three features (those which provide the most information gain) of a decision tree trained on the input dataset. These features are shown in each row in Tables 3 and 4. For each feature, we include in () the corresponding PC value as described earlier. We also measure the CV error, which represents the accuracy of the decision tree. For the purpose of this section, we do not apply the normalization described in Section 3.2 in order to gain more visibility into the direct impact of TCP on the raw metrics.

[6]This number is based on the number of connections opened by clients connecting to one of our services.

General label	Fault	Features Selected	CV Err
Server	High CPU load on server	Probingsd(PC=0.27) Timeouts95(PC=0.009) Duration50(PC=0.22)	6
	Slow reading server	LCWNDmin(PC=0.78) Duration50(PC=0.21) Durationmax(PC=0.15)	3
	High I/O on server	BDelivered95(PC=0.58) BReceivedmax(PC=−0.72)	0.2
	High memory load on server	BDToBRC(PC=0.01) BReceived(PC=−0.74)	0.1
Client	High CPU load on client	Duration95(PC=−0.33) Duration50(PC=−0.03) Durationmin(PC=0.4)	0.6
	High I/O on client	BReceivedmax(PC=−0.75) Probingsd(PC=−0.61) BDToBRC95 (PC=0.01)	1.7
	High memory load on client	BReceivedmax(PC=−0.82) LCWND10(PC=0.24) BDelivered95(PC=0.3)	15
Network	Bandwidth throttling	BReceivedmax(PC=−0.78) Durationsd(PC=−0.14) BDelivered95(PC=0.26)	0.05
	Sporadic packet drops	BReceivedmax(−0.73) MRTTmax(0.12) BDelivered10(0.34)	3
	Packet reordering	MCWNDmean(PC=−0.07) MCWNDsd(PC=0.01) TDupAcks95(PC=0.79)	0.1
	Latency	MRTTmean (PC=0.91)	0

Table 3: Important features in identifying failures for the **duplex** application. CV Err. are percentages.

We observed that each type of failure can be defined succinctly using a few features. To validate this, we used a standard machine learning technique, *Principal Component Analysis (PCA)* [24], where we identify the highest Eigenvalues of the dataset for each failure type. The sum of the Eigenvalues of a dataset equals its variance. Interestingly, for almost all failures, the sum of the 2 highest Eigenvalues captured more than 95% of the variance in the feature set. This is important as it shows that the space of each failure (as represented by TCP metrics) is compactly representable on two dimensions[7].

We next discuss interesting highlights of our analysis.

High CPU load on the server. The failure was induced by a multithreaded program, where each thread performed a CPU intensive system level operation. NetPoirot used the standard deviation of the time spent in zero window probing, which occurs when the remote side of the connection runs out of receive buffer, as its top feature for the duplex application. In the presence of high CPU load, the server would not read from the receive buffer as regularly as normal operations resulting in higher variance in the size of the receive buffer and by extension the time spent in zero window probing at the client. Transmissions will also be delayed due to receive buffer limitations. This explains the prominence of these features in the selection process.

Slow reading server Detecting a slow reading server should be simple in principle if it results in zero window probing. We induced a random delay of 0-100ms before the server reads from the TCP socket in order to test this theory. Surprisingly, the decision trees use measures of the congestion window and duration instead. It seems that the secondary effect of the delay was more pronounced on these metrics.

High I/O load on the client side. To induce high I/O load, we use the SQLIO tool [19] from Microsoft. High I/O load on the client has all the markings of an application limited connection. The decision trees both use the maximum of BReceived, as well as the 95^{th} percentile of the ratio of BDeliveredToBReceived.

High memory usage on the server. The selected values point directly towards a problem on the server, these values include the 95^{th} percentile of BDeliveredToBReceived and the maximum of BDelivered. This shows that the reduction in memory on the server side has caused an impact on the amount of data transmitted from the server. Note, that such data is not available on the simplex application. Here, instead the decision tree relies on the number of connections, as well as the congestion window related metrics in order to do classification.

General label	Fault	Features Selected	CV Err.
Server	High CPU load on server	MRTT10(PC=0.87), MRTTsd(PC=−0.12), Timeoutssd(PC=0.03)	1.2
	Slow reading server	NumberofFlows(PC=−0.22), DCWNDsd(PC=0.16), LCWND95(PC=0.37)	13
	High I/O on server	NumberofFlows(PC=−0.08), MCWNDmax(PC=0.11), LCWNDmax(PC=0.46)	14
	High memory load on server	NumberofFlows(PC=−0.04), DCWNDsd(PC=0.08), LCWNDmax(PC=0.49)	16
Client	High CPU load on client	MCWND10(PC=−0.07), BpostedtoSentmin(PC=−0.05)	0
	High I/O on client	NumberofFlows (PC=0.32), MCWNDmean(PC=−0.31)	15
	High memory load on client	NumberofFlows (PC=0.57), MRTTmax(PC=−0.47)	13
Network	Bandwidth throttling	NLossRecovery95(PC=0.66)	0.02
	Sporadic packet drops	MCWND95(PC=−0.47), BPostedmean(PC=0.24), Durationmin(PC=−0.07)	4
	Latency	MRTTmax(PC=0.94)	0.02

Table 4: The important features for identifying each type of failure in the **simplex** application. CV Err. are percentages.

High memory usage on client. As part of our analysis, for all the failures, we used fast correlation-based filters for feature selection and compared the results with the top 3 features of the decision tree. We found that feature selection for this particular failure returned an unexpected subset of the features. It returned the mean time spent in zero window probing and its standard deviation. On all machines, the client had zero Probingmean when the memory usage on the client was increased, whereas it was positive in the normal data. The client, having less memory, is pushing less data to the server allowing it to "keep up".

[7]These dimensions can be derived from the original features using PCA.

Packet drops. Using a commercial tool called SoftPerfect [1], we induce 5, 10, and 30 percent packet drop rates on all connections to the service. Metrics pointing to TCP throughput are those mostly used to identify the failure. While packet drops do result in a decrease in throughput it is surprising that DupAcks are not the most prominent metric. Indeed, the maximum number of BReceived has twice as high information gain than any of the DupAck statistics. We plotted the CDF's for both these metrics. Both showed a significant difference between failure/normal data. And therefore, we can only explain this choice by noting that the impact of BReceived had a more pronounced effect on information gain compared to the number of DupAcks.

Connection drops. This type of failure is one of the easiest for our classification tool to identify as tracking SYN, SYN/ACK ETW events suffice in understanding whether all or only a random subset of connections are being dropped. Thus, these failures are identified with 100% accuracy. With clear indicators, that identify why they occur.

Overall takeaways. TCP reacts differently to different failures. Therefore, the top three features selected by the decision tree also vary across failures. PC values are not a good predictor of the importance of TCP metrics. In fact, the top three features on occasion have low absolute PC values, suggesting that the relationship between faults and TCP metrics may be more complex than a straightforward linear one. This makes manual classification difficult and motivates the need for automated approaches.

5. EVALUATION

We have developed a prototype of NetPoirot, which we deployed in a production data center with failure injection and application workloads as described in Section 4.1. The focus of our evaluation is to measure the accuracy of NetPoirot in identifying each failure type. To this end, we compute the *confusion matrix* [6] of the test set on the trained model. A confusion matrix illustrates what each class of failure in the test set is classified as by NetPoirot.

Within each class, we further report the precision and recall, defined as follows:

Precision is defined as the ratio of true positives divided by the sum of true positives and false positives. It is a measure of reliability. For example, if the precision of the network failures are reported as 96%, it means that when NetPoirot allocates responsibility to the network for a failure it is the culprit with 96% likelihood.

Recall is the ratio of true positives to the actual number of instances in a class and is a measure of NetPoirot's ability to recognize that an entity is indeed responsible for a failure.

We will first describe the performance of NetPoirot when only TCP metrics from the client side are used. We then show that by augmenting network information with high-level counters, e.g. CPU load, on the *same client machine* one can achieve almost perfect classification.

We use the workload described in Section 4 and partition the measured data into two sets for training and testing. The partition is done by using *disjoint* sets of machines from different data centers for the training and test sets in order to avoid any bias in favor of NetPoirot. This also allows us to illustrate that high precision/recall can be achieved even without having data from the specific machines/data centers in the training set. The datasets used for training and testing are roughly the same size and contain aggregated information for over 37 million connections.

General label	**Fault**	P(**fault** | **error**$\cap$ **General label**)	P(**error** | **fault**)
Server	High CPU load on sever	66.36%	24%
	Slow reading server	2.81%	4.06%
	High I/O on server	6.36%	9.66%
	High memory load on server	24%	17.51%
Client	High CPU load on client	4.78%	9.86%
	High I/O on client	33%	17.61%
	High memory load on client	61.52%	11.2%
Network	Bandwidth throttling	96.06%	20.06%
	Sporadic packet drops	0.21%	0.13%
	Packet reordering	3.71%	1.52%
	Latency	0	0

Table 5: The classification errors of NetPoirot in each general label broken down in terms of faults.

5.1 Overall Accuracy

Duplex application. Fig. 3 shows the confusion matrix of NetPoirot when tested on the duplex application. We focus here on identifying the entity responsible for a failure. This could be the Client, Server, Network, or Normal (non-failure). Individual failure classification is deferred to Section 5.2. For better visualization, we show the confusion matrix as a bar graph in each class, where labels on the x-axis show the ground truth. The bars show what each failure was classified as by NetPoirot. The bar matching the ground truth label on the x-axis represents the value of recall. For example, Normal (green bar) has a high recall value close to 100% (height of the green bar). Precision values are reported on the x-axis next to the ground truth labels in parenthesis.

We make the following observations. First, network failures are the easiest to diagnose from TCP statistics and the most reliable among the three types of failure classes. NetPoirot has 99% precision for network failures, indicating that an output of "Network" from NetPoirot can be trusted. Server failures are the hardest to identify, given the lack of direct access to server side information. Recall was 82% for these failures, with the majority of errors going to Normal.

To understand the source of our errors, we looked into the specifics of the misclassified subclasses. Table 5 shows this information, limited to the misclassified data points from Fig. 3. $P(\text{fault} \mid \text{error} \cap \text{General label})$ describes the probability of a failure given that it was erroneously classified by NetPoirot and that it belonged to a particular class (Network/Server/Client). In other words, this column shows the

breakdown of the error in each class to show how much was attributed to each particular failure. The second column, $P(\text{error} \mid \text{fault})$, shows the probability of classification error given a failure type. This shows the likelihood that a failure of a particular type will be misclassified by NetPoirot.

As can be seen, the error within each class is usually less than 20%. However, *High CPU load on server* seems to be the most problematic failure type with 24% misclassification as normal data. Also, note that even though Network failures have a high recall of 90%(as shown in Fig. 3), almost all the error in the remaining 10% can be attributed to bandwidth throttling. We found that within this failure, throttling at 50 and 1 Mbps caused the most problems as they did not significantly disrupt application performance.

The above shows the performance of NetPoirot when it relies *only* on TCP metrics from the client. We can achieve near 100% precision/recall on Server and Client failures by augmenting network information with CPU/IO/Memory load from *only* the client machine (this is indicated by the high precision in the Network class). We can then locally check whether a client-side problem has occurred. If not, NetPoirot can check whether the failure is due to a Network problem. If both tests are negative, by elimination, the Server is the cause of failure. Note that simply relying on client information without TCP metrics (and NetPoirot) would not work – it would provide high precision/recall for client failures but fail (0% recall) for all other failures.

Finally, NetPoirot exhibits high precision in all classes of failures indicating that the entity output by NetPoirot can be trusted with high probability as the source of the failure.

Simplex application. The duplex application shows the worst case performance of NetPoirot on typical applications that have bidirectional communication. We also examine a more extreme situation where data is only transmitted from the client machine. Fig. 4 shows that while precision and recall remain high for network failures, it is more difficult to differentiate between Client/Normal and Server/Normal. We note, however, that the errors are not uniformly spread across all failures. In fact, NetPoirot achieved a recall of 99.9% in detecting high CPU load on the server. However, high I/O and high memory on the server side contribute to the majority of the misclassifications. This is because the simplex application lacks information that relates to data transmitted by the remote application, given that communication is only in one direction. To understand why this information is critical, we observe from Table 3 that in the case of the duplex application, the amount of data transmitted by the remote application (BReceived) plays a significant role in identifying the failure related to high memory on the server side.

We present the simplex application as an extreme scenario. Given that most applications have bidirectional communication similar to the duplex application, we focus on the duplex application for the rest of this section.

5.2 Individual Failure Classification

NetPoirot is primarily designed to identify the entity responsible for a failure. As a more ambitious goal, it is highly desirable to also identify the actual failure type itself. To test

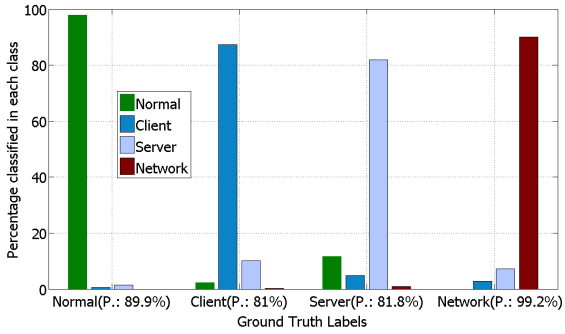

Figure 3: Confusion matrix on the **duplex** application's failure. Recall on each class is as follows: Normal: 97.94%,Client: 87.32%,Server: 81.89%, Network: 90%. Precision values are included in the x axis for each class.

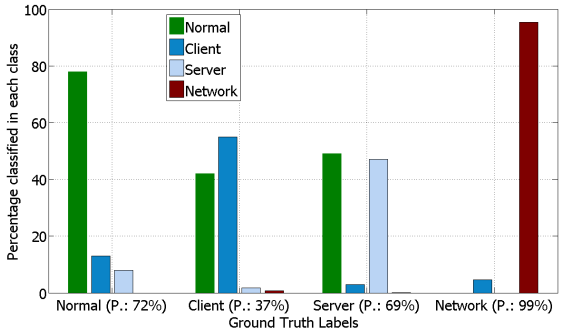

Figure 4: Confusion matrix on the **simplex** application's failure. Recall on each class is as follows: Normal: 78%,Client: 47%,Server: 55%, Network: 95.4%. Precision values are included in the x axis for each class.

the extent to which NetPoirot can accurately classify specific failures (as opposed to responsible entities), once the failure class is identified (Section 5.1), we use detailed failure labels to train an additional diagnosis function to identify the type of failure within the entity. The results are shown in Table 6. The third column only uses TCP metrics at the client, while the fourth column includes additional client-side information such as CPU, IO, and memory.

Table 6 shows that NetPoirot can classify the majority of failures, particularly on the network and server. In fact, NetPoirot is accurate enough in these classes so that we simply used a standard random forest trained for the failures without having to resort to running tournaments. For the client side failures, however, NetPoirot does not perform as well and we have to run tournaments. Distinguishing between high I/O and memory load on both the client and server proved extremely difficult without client-side information as TCP behavior remains largely the same in the presence of either failure. They are presented as a single class in Table 6. We note that, simply by augmenting the TCP statistics used by NetPoirot with client-side information, one can achieve high accuracy in identifying the individual failure type. While these accuracies are not high enough to provide definitive an-

swers as to what caused the failure, they can serve in guiding the operators/users towards a starting point. These results are encouraging as they show that individual failure types have distinguishing markers in the TCP metrics captured by NetPoirot. As part of our future work we plan to analyze these markers and investigate whether we can modify NetPoirot so as to allow for more detailed diagnosis.

General label	Fault	With only client network stats	With all client stats
Server side	High CPU load on server	Precision:99.69%, Recall:75.54%	Precision:99.78%, Recall: 76%
	Slow reading server	Precision:83.21%, Recall:95.66%	Precision:83.63%, Recall:96.44%
	High I/O or Memory on Server	Precision:65.78%, Recall:98.75%	Precision:76.97%, Recall:98.75%
Client side	High CPU load on client	Precision:75.64%, Recall:88.05%	Precision:100%, Recall:100%
	High I/O or Memory on Client	Precision:82.63%, Recall:98.48	Precision:100%, Recall:100%
Network	Bandwidth throttling	Precision:91.4%, Recall:79.94%	Precision:92.14%, Recall:85.53%
	Sporadic packet drops	Precision: 75.54%, Recall: 97.72%	Precision: 75.54%, Recall: 97.72%
	Packet reordering	Precision: 99.73%, Recall: 66.53%	Precision: 99.73%, Recall: 66.84%
	Latency	Precision:99.47%, Recall: 100%	Precision: 99.47%, Recall=100%

Table 6: Detailed fault classification with and without additional client side information.

5.3 Untrained Failures

NetPoirot's design is based on supervised learning and thus requires labeled data for training. This means that it should not be able to detect failures for which it was not specifically trained. Thus, it is important to understand NetPoirot's typical behavior in the presence of such failures. The situation can occur in one of the following two ways:

Dormant failures: A previously unknown type of failure is present during training and is labeled as Normal.

Unknown failures: A failure occurs for the first time during runtime.

While we cannot anticipate what the "unknown/dormant" failures would be, we attempt to illustrate this behavior by purposefully changing our original training data to reflect each of these behaviors. Ideally, we would like these failures to be either classified as Normal by NetPoirot, or as their ground truth class (actual entity).

To emulate dormant failures, we mislabeled each class in our training data as Normal before training NetPoirot. Similarly, to emulate unknown failures, we remove failed classes from the training data. We then investigate what entity will be output as being responsible for these unknown failures. Fig. 5 shows that dormant failures result in most of the dormant failure being classified as Normal (what we are hoping for). This shows that in the presence of dormant failures, an output of Normal from NetPoirot may require further investigation to uncover the source of the problem. Fig. 5 also shows that NetPoirot is resilient to unknown failures, as failures from each entity have similar characteristics[8]. In the few failure types where significant misclassifications occur, they usually result in classification to Normal.

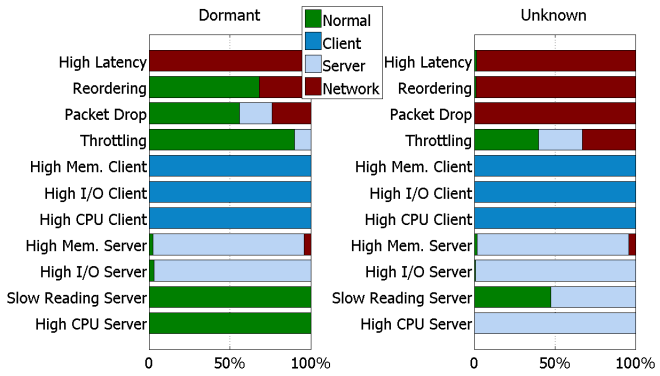

Figure 5: NetPoirot performance on dormant and unknown failures, when all client statistics are used.

5.4 Sensitivity Analysis

We do a series of sensitive analysis to explore NetPoirot's ability to detect failures across data centers.

So far we have shown the accuracy of NetPoirot when the data center locations of the machines used for testing only *partially* overlap with those used in training. Here we investigate whether the location of the machines used for training and testing influences the performance of NetPoirot.

Sensitivity to Cross Data-center Effects. In this experiment, we train NetPoirot for the duplex application using client machines in data centers on the west coast and one of the southern states. We then test the result on data from machines in the same southern state, and also on those in a midwest data center. Table 7 summarizes the precision and recall for test data collected for two machines in the southern data center, and two machines in the midwest. We observe that NetPoirot is mostly resilient to the location on which it has been trained. Interestingly, server-side failures are the most sensitive to location. We conjecture that this may be caused by cross-traffic effects as RTT variance increases. Further investigation is needed as part of our future work.

Sensitivity To Failure Duration. We next investigate NetPoirot's ability to identify short-lived failures. Table 8 summarizes our results for three failures (high IO at server, high IO at client, and high network latency). For training, we use failures that last for the application's lifetime. For each test case, we inject failures with durations ranging from less than 30 seconds, to 5 to 6 minutes. We allow at least a 15 minute gap between each failure. Our results indicate that NetPoirot is not overly sensitive to the duration of the failure. Note

[8]Fig. 5 shows the results for when both client and network statistics are used. Our results showed that NetPoirot performed almost equally as well when using only network information.

General label	Southern State	Midwest
Normal	Precision: 98.85% Recall: 98.2%	Precision: 83.34% Recall: 99.07%
Client	Precision: 100% Recall: 100%	Precision: 100% Recall: 100%
Server	Precision: 84.71% Recall: 94.67%	Precision: 97.75% Recall: 78.91%
Network	Precision: 98.85% Recall: 88.99%	Precision: 100% Recall: 98.82

Table 7: Performance breakdown by machine location. All client statistics are used for classification.

the relatively higher accuracy in identifying high I/O on the server side as compared to Table 5. This is because the VMs we use for testing and training in this scenario are all from the same data centers.

Duration	Client (High IO)	Server (High IO)	Network (High Latency)
< 30s	100%	100%	100%
[30s,1min]	100%	99.99	100%
$[1, 3]$min	100%	96.96%	97%
$[3, 5]$ min	100%	99%	100%
$[5, 6]$ min	100%	96.01%	100%

Table 8: Sensitivity to failure duration. Recall numbers are shown here.

Sensitivity to Per-connection Training. NetPoirot is trained on a per-application basis, where the decision-tree based forests are built using TCP metrics aggregated across all connections of an application to the same service. This aggregation method has three advantages: 1) It allows us to take advantage of the differences across connections to better detect failures, 2) It allows us to capture an application's reaction to failure, and 3) It reduces the runtime/training overhead of NetPoirot.

General label	Normal	Client	Server	Network
Precision	74.2%	100%	48.16%	89.53%
Recall	85.77%	100%	41.46%	77.82%

Table 9: Performance of NetPoirot when used for per connection classification.

We explore another method of training, where we build the forest on a per-connection basis. Table 9 shows the precision/recall achieved by NetPoirot, on failure detection using the duplex application. Here, NetPoirot is trained using per-connection metrics. We observe that training on a per-connection basis requires $100X$ increased learning time. Hence, we are restricted to only 40% of the earlier training set. We observe that the recall and precision numbers are lower. For example, server-side failures are the hardest to classify, with a recall of 41.46%. Overall, per-application aggregation is a more accurate and efficient approach, and the alternative approach should only be considered if the operator requires per-connection diagnostics information.

5.5 Real Application Analysis

Our test applications present challenging scenarios that help us explore the limits of NetPoirot's ability to detect failures. As noted in Section 4.1, these are extreme applications as they do not modify their communication patterns in response to failures. Here, we explore the performance of NetPoirot on two real-world applications, one based on video streaming, and another based on traces from our production data center containing EX failures.

YouTube video streaming. We tested NetPoirot on data collected while streaming YouTube videos in a browser. On 9 of our VM's located in three different data centers, we induced Client/Network failures[9] while streaming YouTube Videos. Table 10 shows the results for when only network statistics are used. Compared to our results in Section 5.1, we observe that NetPoirot does significantly better on the streaming application than on the duplex and simplex applications.

General label	Normal	Client	Network
Precision	97.78%	99.7%	100%
Recall	99.68%	98.25%	99.37%

Table 10: Performance of NetPoirot when used to identify failures when streaming YouTube videos.

Production applications experiencing EX. We run an experiment to validate NetPoirot's effectiveness in identifying causes of EX failures based on real production traces. We use Syslog entries from our production machines to identify when an EX event caused a VM reboot. As training and test sets, we extract information captured by our monitoring agent,which is deployed on all compute nodes in our data centers, during the same time period . The monitoring agent used to capture this data is an older version of NetPoirot and does not report metrics such as time spent in zero window probing which we added in subsequent releases. We additionally use resolved tickets to identify the cause of failures.

We use a large subset of machines (162/175) for training as EX occurrences on any one machine tend to be low (typically at most 1), though in aggregate, they occur frequently. The remaining machines are used for testing. We lack direct access to the ground truth labels and have to use a combination of log analysis and resolved tickets to identify the start-time/type of the failure that lead to EX. Note that in a real-world deployment, we would have injected faults as described in Section 2 in the training phase to induce and learn about all types of EX failures, leading to a more accurate failure labeling process.

In the training set, we mark the duration of each failure based on a simple heuristic. We know when the failure ended as the VM experiencing EX reboots (detectable from Syslog), we set the failure start time to 3 minutes prior to this reboot. We do not use data after the reboot. 3 minutes is a conservative estimate that we apply. However, this is at best an estimate. After talking to our engineers, they confirmed

[9]Server failures are excluded as we do not control the servers.

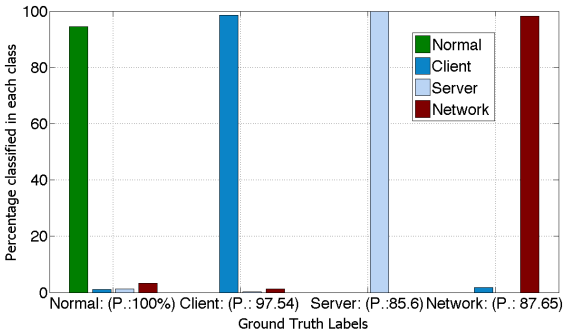

Figure 6: Confusion matrix on EX. Recall on each class is as follows: Normal: 94.53%,Client: 98.54%, Server: 100%,Network: 98.2%.

that the duration of failures that cause EX are highly variable and can last from a couple of seconds to minutes.

To improve label accuracy, we first pass each class of failure in the test set through a classifier that identifies *that* failure (e.g. Client) from Normal data. We then label the data based on the output as failure/Normal making sure that each class occurs with contiguous time stamps. Similarly for the data that we label as normal, based on the 3 minutes timeframe, we first pass it through each of these classifiers and label it as failure if its timestamp is a continuous extension of the failure on the same machine. This allows us to extend the failure labeling past the 3 minute conservative estimate.

Fig. 6 shows the confusion matrix similar to the format described in Section 5.1. Since client information was not collected, the results are based on NetPoirot when using network statistics alone. We make the following observations. First, despite lacking the actual ground truth on failures and the time interval in which they occur, NetPoirot never misclassifies failures as other types of failures but instead returns a label of Normal. This is important as an entity is not blamed when it was not at fault.. Second, given course granularity labels of failures vs normal, NetPoirot is able to achieve very high recall values, consistently above 94%, and in fact, achieves 100% recall for server failures. This suggests that NetPoirot is a promising approach at pinpointing the entity causing EX.

6. DISCUSSION

Our results in Section 5 demonstrates the effectiveness of NetPoirot in pinpointing the source of failures in a data center. We perform a range of sensitivity analysis, on different applications to fully explore the efficacy of NetPoirot. In doing so, we have identified possible extensions of NetPoirot, as well as gained a better understanding of the scenarios in which it is most useful. We briefly discuss these possibilities here, as well as point out directions for future extensions.

NetPoirot uses lightweight endpoint monitoring, is non-intrusive, and incurs minimal overhead to machines in a data center. Despite being lightweight, NetPoirot is able to perform accurate failure detection of the entity responsible for the failure, and even the type of failure itself. This precludes identification of the actual device, e.g. the physical router that causes a network problem. Another limitation of NetPoirot is its reliance on TCP metrics, which precludes failures observed by UDP. Given that most traffic within data center uses TCP, a wide range of applications (e.g. web apps, database applications) can still benefit from NetPoirot.

NetPoirot is a lightweight failure identification tool for data center applications. We have identified several avenues to further improve our results via more sophisticated learning techniques. These include:

Cross-application learning. NetPoirot requires per-application training, which we argue is feasible, given that learning can be done offline for each new application being deployed in a data center. One interesting idea we are exploring is the use of a concept in machine learning, known as *transfer learning* [23], in which the feature space of one application can be modified so that it can be used in NetPoirot for identifying the cause of failures in another application.

Non-production training. NetPoirot exploits the fact that data centers tend to have homogeneous setups, where failures can be induced on a subset for machines for supervised learning. This has been explored by others [25]. As an enhancement, we plan to explore training in a staging environment. Our sensitivity analysis in Section 5.4 suggests that NetPoirot is resilient to learning across data centers, suggesting it is possible to limit the training to a single cluster within a data center, and still apply the results to another cluster with similar configuration.

Improving accuracy. In Section 3.2 we reported that traditional CV on our data set yields an error of 1.5%. Cross-validation error where each fold comprises of data from only a single machine is much higher (11%). This suggests that *if* samples of faulty/normal data from a machine exist in the training data, the classification error on that machine dramatically decreases. Thus, continuously changing the machines used in training should improve the accuracy of NetPoirot even further over time.

7. RELATED WORK

Anomaly detection in distributed systems [10, 13, 11, 15] detect when a failure has occurred, while the goal of NetPoirot is to find the entity responsible for the failure.

Inference and Trace-Based Algorithms [4, 2, 31, 12] either require (a) data not locally available to the client at runtime, (b) knowledge/inference of the probability distribution of failure on each device in the system, (c) high resource consumption at runtime, or (d) knowledge/inference of application dependence on the different network/service devices. Each of these requirements raises the barrier of adoption, as compared to NetPoirot.

Fault Localization by Consensus [22] violates the above data locality requirement, but does not require any information from the network or service. The work assumes that a failure on a node common to the path used by a subset of clients will result in failures on all or a significant number of those clients. Therefore, if many clients in different locations report a failure, it is most likely caused by the service, whereas if only a single client fails the problem is likely lo-

cal to that client. Fig.7 illustrates why this approach fails in the face of problems such as EX. Here, we use the Network Emulator Tool (NEWT) [21] to induce a 5% packet drop rate on all TCP connections to the remote service on a single machine in our stage cluster. Even though the 5% drop rate was present over the entire 6 hour period, only 3 EXs occurred. These events happened when data transmissions increased. This shows that even though a failure may be present in the network, not all clients will observe it at the same time or in the same way. NetProfiler [22] would erroneously, classify such a problem as a client side problem. These approaches require further information in order to provide reliable fault localization. We do, however, envision a hybrid of this approach with NetPoirot to improve diagnosis accuracy.

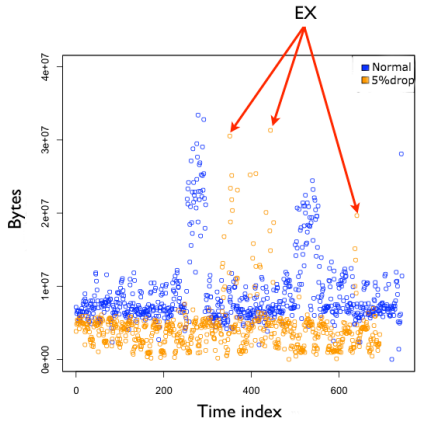

Figure 7: Each dot shows BPostedmax in time and is representative of a 30s epoch.

Fault Localization using TCP statistics [17, 27, 29, 28] targets using TCP metrics for diagnosis. [17] requires heavyweight active probing. [27] uses learning techniques (SVM), however, it relies on packet captures from both end points and is limited in the scope of failures it detects (network and application problems only). T-Rat [29] infers TCP information from packet captures. It is used to understand why a TCP connection is rate limited. Our diagnosis goals are broader. T-Rat is too heavyweight to be used as always on. SNAP [28] requires sensitive information, such as topology, path information, and switch counters to identify applications with frequent problems. NetPoirot eliminates the need to share such information with clients. SNAP manually reasons about the relationship between each failure and the TCP metrics it collects and devises detection rules accordingly. Such reasoning is limited in its scope. NetPoirot instead automates the creation of these rules through decision tree based algorithms.

Tomography Based Approaches [30, 14] correlates active measurements across multiple paths to determine the location of failures, e.g. [30]. These techniques usually involve active probing, which increases overhead in a data center.

Learning Based Approaches [8, 9, 3] do failure detection. NetPoirot also uses machine learning techniques, but the application domain is different (home networks [3] and mobile video delivery [9]). [8] uses decision trees in order to locate the device responsible for a failure by observing the path traversed on the tree. It requires request to server mappings. NetPoirot does not require this additional information.

8. CONCLUSION

In this paper we present NetPoirot, a diagnostic tool that allows for identifying the cause of performance problems in data centers. Our approach uses lightweight non-intrusive continuous monitoring of TCP metrics only on client machines, and machine-learning based classification. Based on extensive evaluation over a 6-month period in a production environment across multiple data centers, we show that NetPoirot can accurately identify entities responsible for a wide range of failures, achieving in some cases, above 96% accuracy for some failure types.

9. ACKNOWLEDGEMENTS

This work was done while Behnaz Arzani was an intern at Microsoft and was supported in part by grants NSF CNS-1513679, NSF CNS-1218066, NSF CNS-1117052, DARPA/I2O HR0011-15-C-0098 and CNS-0845552. We would like to thank our shephard Ramesh Govindan, our anonymous reviewers, as well as Sanchit Aggarwal, Omid Alipourfard, Ang Chen, Monia Ghobadi, Ran Gilad-Bachrach, Andreas Haeberlen, Nick Iodice, Dave Maltz, Jitu Padhye, Jennifer Rexford, Lynne Salameh, Narasimhan Venkataramaiah, and Nikos Vasilakis for their useful comments/feedback on our work.

References

[1] Soft Perfect Connection Emulator. https://www.softperfect.com/, 2012. [Online].

[2] ADAIR, K. L., LEVIS, A. P., AND HRUSKA, S. I. Expert network development environment for automating machine fault diagnosis. In *Aerospace/Defense Sensing and Controls* (1996), International Society for Optics and Photonics, pp. 506–515.

[3] AGARWAL, B., BHAGWAN, R., DAS, T., ESWARAN, S., PADMANABHAN, V. N., AND VOELKER, G. M. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI* (2009), vol. 9, pp. 349–364.

[4] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review* (2007), vol. 37, ACM, pp. 13–24.

[5] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[6] BURGESS, M. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming 60*, 1 (2006), 1–26.

[7] BURMAN, P. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika 76*, 3 (1989), 503–514.

[8] CHEN, M., ZHENG, A. X., LLOYD, J., JORDAN, M., BREWER, E., ET AL. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on* (2004), IEEE, pp. 36–43.

[9] DIMOPOULOS, G., LEONTIADIS, I., BARLET-ROS, P., PAPAGIANNAKI, K., AND STEENKISTE, P. Identifying the root cause of video streaming issues on mobile devices.

[10] FU, Q., LOU, J.-G., WANG, Y., AND LI, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on* (2009), IEEE, pp. 149–158.

[11] GABEL, M., SATO, K., KEREN, D., MATSUOKA, S., AND SCHUSTER, A. Latent fault detection with unbalanced workloads. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[12] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 139–152.

[13] HUANG, L., NGUYEN, X., GAROFALAKIS, M., JORDAN, M. I., JOSEPH, A., AND TAFT, N. In-network pca and anomaly detection. In *Advances in Neural Information Processing Systems* (2006), pp. 617–624.

[14] HUANG, Y., FEAMSTER, N., AND TEIXEIRA, R. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review 38*, 5 (2008), 53–58.

[15] IBIDUNMOYE, O., HERNÁNDEZ-RODRIGUEZ, F., AND ELMROTH, E. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR) 48*, 1 (2015), 4.

[16] KUHN, M., AND JOHNSON, K. *Applied predictive modeling*. Springer, 2013.

[17] MATHIS, M., HEFFNER, J., O?NEIL, P., AND SIEMSEN, P. Pathdiag: automated tcp diagnosis. In *Passive and Active Network Measurement*. Springer, 2008, pp. 152–161.

[18] MICROSOFT. Windows ETW. https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx, 2000. [Online].

[19] MICROSOFT. SQLIO. http://www.microsoft.com/en-us/download/details.aspx?id=20163, 2012. [Online].

[20] MICROSOFT. TestLimit. http://blogs.msdn.com/b/vijaysk/archive/2012/10/27/tools-to-simulate-cpu-memory-disk-load.aspx, 2012. [Online].

[21] MICROSOFT. BWorld Robot Control Software. https://chocolatey.org/packages/newt, 2013. [Online].

[22] PADMANABHAN, V. N., RAMABHADRAN, S., AND PADHYE, J. Netprofiler: Profiling wide-area networks using peer cooperation. In *Peer-to-Peer Systems IV*. Springer, 2005, pp. 80–92.

[23] PAN, S. J., AND YANG, Q. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on 22*, 10 (2010), 1345–1359.

[24] PEARSON, K. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11 (1901), 559–572.

[25] TSEITLIN, A. The antifragile organization. *Communications of the ACM 56*, 8 (2013), 40–44.

[26] WELLS, P. M., CHAKRABORTY, K., AND SOHI, G. S. Adapting to intermittent faults in multicore systems. *ACM SIGOPS Operating Systems Review 42*, 2 (2008), 255–264.

[27] WIDANAPATHIRANA, C., LI, J., SEKERCIOGLU, Y. A., IVANOVICH, M., AND FITZPATRICK, P. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on* (2011), IEEE, pp. 261–266.

[28] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *NSDI* (2011).

[29] ZHANG, Y., BRESLAU, L., PAXSON, V., AND SHENKER, S. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM Computer Communication Review* (2002), vol. 32, ACM, pp. 309–322.

[30] ZHAO, Y., CHEN, Y., AND BINDEL, D. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM Computer Communication Review* (2006), vol. 36, ACM, pp. 219–230.

[31] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., ET AL. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 479–491.