NetAlytics: Cloud-Scale Application Performance Monitoring with SDN and NFV

Guyue Liu, Michael Trotter, Yuxin Ren, Timothy Wood
The George Washington University
District of Columbia, USA
guyue, trotsky, ryx, timwood@gwu.edu

ABSTRACT

Application performance monitoring in large data centers relies on either deploying expensive and specialized hardware at fixed locations or heavily customizing applications and collecting logs spread across thousands of servers. Such an endeavor makes performance diagnosis a time-consuming task for cloud providers and a problem beyond the control of cloud customers. We address this problem using emerging software defined paradigms such as Software Defined Networking and Network Function Virtualization as well as big data technologies. In this paper, we propose NetAlytics: a non-intrusive distributed performance monitoring system for cloud data centers. NetAlytics deploys customized monitors in the middle of the network which are transparent to end host applications, and leverages a real-time big data framework to analyze application behavior in a timely manner. NetAlytics can scale to packet rates of 40Gbps using only four monitoring cores and fifteen processing cores. Its placement algorithm can be tuned to minimize network bandwidth cost or server resources, and can reduce monitoring traffic overheads by a factor of 4.5. We present experiments that demonstrates how NetAlytics can be used to troubleshoot performance problems in load balancers, present comprehensive performance analysis, and provide metrics that drive automation tools, all while providing both low overhead monitors and scalable analytics.

CCS Concepts

•**Networks** → **Middleboxes / Monitoring / Performance;**

Keywords

Network Function Virtualization, Software Defined Network

Middleware'16, December 12-16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-4300-8/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988344

1. INTRODUCTION

The growing popularity of cloud computing has led to large-scale, distributed applications, complicating both performance and correctness monitoring. Modern web applications have evolved from single server applications to relatively simple multi-tier applications to larger scale distributed systems. Now, a growing trend is to deploy such systems as “microservices," wherein a large application is broken into many smaller components each deployed in a virtual machine or container. While such an approach may simplify development by encouraging modularity, it poses significant challenges to system administrators who must deploy, configure and analyze the performance of these systems.

Unfortunately, understanding the performance and ensuring the correctness of distributed applications is a growing challenge for system administrators. Debuggers are typically capable of analyzing a single process, while manually examining logs for dozens or even hundreds of components in a large distributed system quickly becomes overwhelming. Similarly, analyzing performance bottlenecks grows more arduous as a system grows, especially as web services expand from simple multi-tier architectures to distributed configurations. While cloud platforms have greatly simplified application deployment, they have led to more complex management concerns.

At the same time, there is a growing desire for the cloud infrastructure to provide transparent services to users, allowing them to easily take advantage of features such as load balancing or fault tolerance. Ideally, the monitoring and management of distributed systems would be performed by the platform provider in as application-agnostic a way as possible. As a result of the separation between developers and IT administrators, system administrators troubleshooting an application may not understand the finer details about how different components interact or which log files to examine on each server. Thus, there is a need for monitoring systems that can be deployed in an ad-hoc manner without requiring modification to end systems or tight integration with applications.

Recently, software-based networking technologies are changing the way network data can be routed and inspected. Software Defined Networking (SDN) allows a centralized controller to dictate how packets are routed on a per flow basis, simplifying redirecting packets when they need to be handled in a special way. When combined with new Network Function Virtualization (NFV) techniques, packets can be routed to network functions: software elements that can analyze and even modify packets with efficiency comparable to hardware solutions. Consequently, a new opportunity opens for flexible monitoring of specific network flows.

With NetAlytics, we explore how SDN, NFV, and big data streaming analytic tools can combine to build a powerful monitoring and debugging platform for distributed systems. NetAlytics allows a system administrator to specify a simple query defining types of traffic to monitor and data to gather, as well as how that data should be analyzed. The query is transformed into a set of SDN rules that direct the desired traffic to dynamically instantiated NFV monitors that efficiently extract the target data. This data is then aggregated and sent through a highly scalable streaming analytics engine, allowing system administrators to quickly get back meaningful insights about their networks and the applications running within them.

One of the biggest challenges to developing a practical solution to this problem is the large amount of data. For example, the traffic of a large data center can easily go beyond 100s of Gbps when considering all links in the topology. Given this scale, there is no existing tool that can store all of the data and analyze them in real time, not to mention how much congestion the extra monitoring traffic may cause if it all must be routed to storage servers. Therefore, the goal of NetAlytics is to unobtrusively identify and extract necessary data and analyze application performance at real time.

NetAlytics makes the following contributions:

- NFV-based network monitors that are precisely deployed to efficiently monitor packet flows.
- A control framework that uses SDNs to split traffic and send targeted flows to the monitoring system.
- A query language that automatically produces a set of SDN rules and deploys NFV monitors.
- A processing engine for real-time analytics on the aggregated data, providing insights into application performance and network behavior.

We have implemented NetAlytics utilizing DPDK [3] to build efficient software-based monitors capable of processing traffic rates of 10Gbps or more. The monitored data passes through an Apache Storm [1] deployment to provide scalable, real-time analytics. We demonstrate that our platform prototype incurs minimal performance overhead, use large scale simulation to test our placement algorithm, and describe several scenarios where NetAlytics simplifies detecting configuration errors, analyzing performance, and automating resource management based on live network data.

2. BACKGROUND

2.1 Software-Based Networks

Software Defined Networking allows a logically centralized software controller to manage a set of distributed data plane elements. In the past, the data plane primarily consisted of routers and switches but has been evolving to include both hardware and software middleboxes such as intrusion detection systems (IDS), WAN optimizers, deep packet inspection (DPI) engines, etc. The SDN controller provides a set of rules to the data plane indicating how to forward through the network the different flows which are typically matched by a set of IP header fields. These rules can either be proactively installed, or the controller can be invoked when a data plane element receives a packet that it does not already have a rule for. As a result, the SDN controller can customize how to route individual flows using its own application logic. Thus, the SDN controller creates a much more flexible network than previous routing protocols that typically determined routes at much coarser grain based on criteria such as weighted shortest path [6, 10, 13, 25].

Network Function Virtualization is a complementary technology that allows data plane elements to be written as software components running in virtual machines. While virtualization often incurs high overhead, especially for I/O activities, recent research [17, 18, 33] has demonstrated several techniques for bypassing these overheads. Combined with recent high performance network interface cards (NICs) and multi-core CPUs, one can now run a software data plane that has performance comparable to hardware ASICs while providing much greater flexibility and easier deployment [7, 15, 19, 24, 27].

With NetAlytics, we take advantage of both of these technologies to provide a flexible software infrastructure within the network that can redirect important flows to efficient monitors.

2.2 Scalable Real-time Analytics

Given the high volume of data originating from these monitors, an application running on a single machine would not do. Nor could we wait to accumulate the data into large batches and launch long-running MapReduce [11] jobs against them for analysis. Instead, we opt for a scalable real-time framework that is capable of processing the data efficiently and reliably. There are several options and we choose Apache Storm [1] because it is publicly available, easy to use and gave reasonable performance.

Initially developed to analyze trending topics, news feeds and other real time events at Twitter and later open sourced in 2011, Apache Storm is capable of processing millions of messages per second [2]. Storm conceptualizes its workflow as a directed acyclic graph (DAG) wherein one processor emits data to other processors in the graph. To use Storm terminology, a graph is a “topology” whose root nodes, or “spouts”, feed other nodes, or “bolts”, in the topology. A Storm cluster runs many of these topologies. It can also dynamically scale the topology as nodes are started and stopped and gracefully handle node failures.

3. NETALYTICS DESIGN

NetAlytics is designed to be:

- Transparent: Require no per-host changes, while still

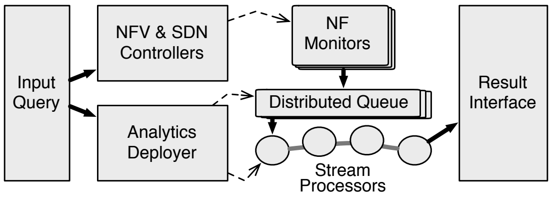

Figure 1: NetAlytics pipeline

providing application-level performance monitoring.

- Real-time: Responsively collect and analyze data on demand for interactive analysis.
- Scalable: Monitoring, data aggregation, and stream analysis layers all must scale independently based on resource needs.
- Efficient: Monitors and analysis engines should be run only when and where they are needed .

The overall flow of our system is shown in Figure 1. System administrators have an interface wherein they input a query with syntax similar to SQL that defines the data they are interested in and how it should be processed. This query is transformed into a set of SDN rules installed by the SDN controller and a group of NFV-based monitors that must be installed by an NFV Orchestrator. The query also specifies composable processing elements that the analytics engine deploys. The results of the processing are then returned to the administrator, providing real-time insights into their network and its applications.

3.1 Efficient NFV Monitoring & Parsing

NetAlytics takes advantage of high performance software I/O libraries to efficiently gather data from inside packets on commodity servers. Each monitor runs flexible parsers that extract a desired region of a packet and can be customized for specific protocols at different layers, e.g., TCP/IP, HTTP, or Memcached. These software-based monitors can be deployed as virtual network functions by an NFV orchestrator, allowing them to be started exactly when and where they are needed.

When a monitor is instantiated, it is instructed to run one or more *parsers*, capable of extracting information related to a given protocol or application. NetAlytics provides a set of common parsers as shown in Table 1. In addition, system administrators can develop their own parsers with a simple interface: they define a packet handler function called when each packet arrives and make use of the monitoring library's output functions to emit the desired information extracted from a packet or group of packets.

NetAlytics parsers are lightweight processes since they must be able to keep up with network line rates, e.g. over 14 million packets per second on a 10 Gbps link with 64 byte packets. As a result, our parsers simply extract a small amount of data from each packet or produce aggregate statistics about flows. More complex data processing is deferred to the streaming analytics system discussed below. For example, we provide a `http_get` parser that can extract the URL of an HTTP GET request. The parser periodically releases a list of URLs to be analyzed in more depth, e.g. combining the list with TCP connection timing information gathered by another parser, and ultimately ranking them to find the fastest and slowest pages.

The parsers emit data tuples that become the input to the analytics system. Typically, a parser emits either a tuple that is miniscule compared to the full packet it was derived from or a single tuple per group of packets. Thus, the amount of data extracted from packets and sent to the analytics engine is significantly smaller than the size of the raw packets. As a result, NetAlytics is more efficient than existing network analytic systems that often mirror entire packets or packet headers. NetAlytics further reduces the overhead of transmitting data tuples by aggregating tuples produced by all parsers and having the monitor send them in batches (potentially after being compressed).

The first element in each tuple is an ID field, usually calculated as a hash of the packet's n-tuple, although parsers may use a different ID when emitting data aggregated across multiple flows. The ID field enables information from multiple parsers to be aggregated by the processing entities, e.g. they might use the flow ID to combine TCP connection time information gathered by one parser with HTTP GET request information from a second.

Parsers	**Layer**	**Description**
tcp_flow_key	Net	extract src_ip, dst_ip, src_port, dst_port
tcp_conn_time	Net	detect SYN/FIN/RST flags
tcp_pkt_size	Net	calculate tcp packet size
memcached_get	App	parse memcached get request
http_get	App	parse http get request and response
mysql_query	App	parse mysql query and response

Table 1: Common NetAlytics parsers

3.2 Data Aggregation and Analysis

Data processing in NetAlytics occurs in two layers: a distributed queuing service that aggregates data and the real-time analytic engines that perform actual processing. We build our system on top of Kafka [21] and Storm [1] which respectively provide these services. Separating these into two layers provides important scalability benefits, allowing each layer of the system to be independently replicated based on available capacity and the performance requirements of the analytics engine itself.

An aggregation layer is necessary because typically the analysis that must be performed on the tuples released from the monitors cannot be executed at the same rate that data is produced. Parsers, potentially distributed across multiple monitoring hosts, send their data to one of the Kafka servers. Kafka is a distributed queuing service wherein data tuples can be buffered by topic. Using Kafka, we can fuse together data streams from parsers replicated at different points in the network. Alternatively, if several different unique parsers are running, each will receive its own data buffer since the parser type is used to select a buffer. The aggregation level is particularly useful for NetAlytics queries that only need to run for a short period but may gather a substantial amount of data. In this case the Kafka servers can act as a large memory buffer while the analytic engine slowly processes the data.

The Storm analytics engine is deployed as a topology consisting of a data "spout", that requests data from Kafka, a series of processing nodes, and a data sink that produces the final result. The topology dictates how simple analytic building blocks such as counters, rankers, histogram generators, etc. combine to transform the input data into the desired result. NetAlytics provides topologies for several common processing tasks, and we name the topology by connecting a set of blocks' name listed in Table 2, e.g. diff_group takes two streams (e.g., the start and end times of a TCP flow) and calculates their difference value, and then groups the results by some attribute (e.g., the destination IP). System administrators can easily create more by combining the building blocks within these topologies in new ways.

Blocks	Description
top-k	get k largest value of the stream
max/min	get the smallest/largest value of the stream
sum	get the total sum value of the stream
avg	get the average value of the stream
diff	get the difference value of two streams
group	group the results by one or more attributes

Table 2: Common NetAlytics topology building blocks

3.3 Specifying Queries

NetAlytics provides a query language for system administrators to easily describe what data must be gathered and how it should be processed. The query syntax is designed to be easily translated into a set of OpenFlow rules that will direct the appropriate flows to the monitors. Table 3 shows the query language syntax.

query ::= *parser-clause addr-clause attr-clause process-clause*
parser-clause ::= **PARSE** *parser-list*
parser-list ::= parser_name | *parser-list*, parser_name
addr-clause ::= **FROM** *address-list* **TO** *address-list*
address-list ::= *address* | *address-list, address*
address::= ip:port | subnet:port | hostname:port | *
attr-clause ::= **LIMIT** *limit-rate* **SAMPLE** *sample-rate*
limit-rate ::= amount_of_time | number_of_packets
sample-rate ::= interval | auto | *
process-clause ::= **PROCESS** *processor-list*
processor-list ::= *processor* | *processor-list, processor*
processor ::= (processor_name: *argument-list*)
argument-list ::= *argument* | *argument-list, argument*
argument ::= argument_name=value

Table 3: Query language syntax

The primary keywords are `PARSE, FROM/TO, LIMIT` and `PROCESS`. `PARSE` specifies a list of one or more parsers that are deployed on monitors. The `TO` and `FROM` statements identify the traffic flows of interest; * means all hosts or all ports within the specified host. LIMIT dictates the number of packets to monitor or how long the monitors and the processors should run. In addition, a sampling rate to apply at the monitor can be specified, which is enforced by hashing each packet's n-tuple to do sampling by flow, not packet. The sampling rate can be specified by `SAMPLE`. "auto" means the sample rate is determined by the feedback-driven sampling mechanism, or if its value is *, sampling is disabled. Finally, the `PROCESS` clause specifies the Storm topology file to deploy. Parameters for the Storm topology can also be provided in the query. Consider these example queries:

```
PARSE tcp_conn_time, http_get
FROM 10.0.2.8:5555 TO 10.0.2.9:80
LIMIT 90s SAMPLE auto
PROCESS (top-k:  k=10, w=10s)
```

Here two independent parsers, `tcp_conn_time` and `http_get`, will be installed in monitors that are routed traffic between 10.0.2.8:5555 and 10.0.2.9:80. They will run 90s with an automatically adjusted sample rate. Their output will be processed by the `top-k` topology.

```
PARSE http_get FROM * TO h1:80, h2:3306
LIMIT 5000p SAMPLE 0.1
PROCESS (diff-group:  group=get)
```

In this example all connections to host h1 or h2 are parsed by `http_get` with 10% of flows being sampled to gather 5000 packets and processed by the diff-group topology.

3.4 Query Instantiation

When a user submits a query, it is interpreted to determine which parsing and processing engines need to be instantiated and how flows need to be mirrored and rerouted.

The `FROM/TO` clauses identify the types of monitored flows, typically by specifying flows destined for or originating from a particular IP and port. The values from these clauses in the query are translated into the *match* portion of an OpenFlow rule. Currently, we assume that all queries contain a `FROM` and/or `TO` clause, since we use them for monitor placement. Expanding NetAlytics to support more generic network-wide monitoring is possible but requires manual monitor placement.

The `PARSE` portion of the query dictates which parsing modules need to be deployed. These parsers are deployed onto monitoring hosts as described in the next section. The monitoring hosts selected to run the required parsers define the *action* portion of the OpenFlow rule. In that rule, we create an action list with both the standard output port leading to the destination and a secondary output leading to the monitor. The query interpreter combines the match and action criteria to build a rule transmitted to the SDN controller via its Northbound interface which in turn is either pulled on demand by switches when they see new packets or proactively pushed to the switches by the SDN controller. The new rules mirror a copy of each matched packet to NetAlytics's monitoring infrastructure, permitting it to process packets without adding processing latency to the critical path.

The Storm topology indicated by the `PROCESS` clause determines what analytic components need to be initialized and connected together. Depending on the available resources, Storm components start on available servers in close proximity to the aggregators and monitors as described in the

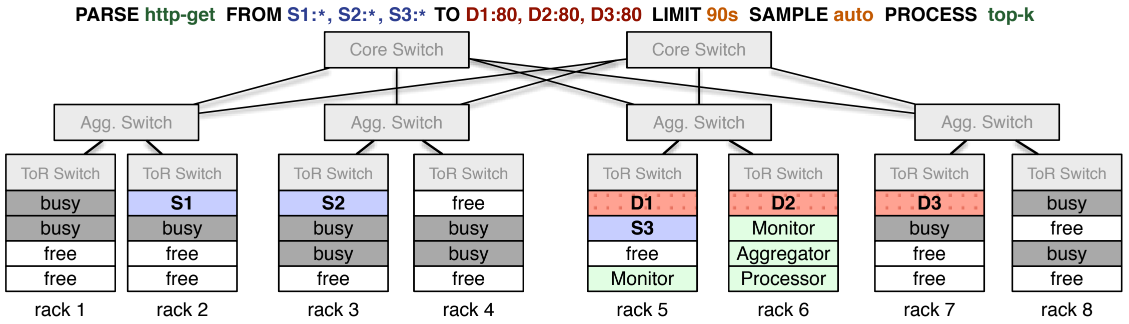

Figure 2: Based on a query, NetAlytics automatically places monitors, aggregators and processors to minimize network bandwidth and balance load.

next section. Though it is possible add operations such as join in the query language, we leave this as future work.

4. MONITORING OVERHEADS

NetAlytics's goal is to unobtrusively monitor applications, but the additional network traffic it creates by sending data for analysis can cause overheads for other applications. We provide two approaches for reducing this cost: careful monitor placement and feedback-driven sampling.

4.1 Monitor & Analytics Engine Placement

NetAlytics places monitors, aggregators, and processors in order to minimize the network bandwidth consumed by monitoring traffic. We assume the data center is structured in a tree structure, such as a Fat Tree [22, 5], and that NetAlytics has access to the data center network and server topology, as well as an IP-to-host mapping table for any IPs that a system administrator may want to monitor. Our placement algorithm currently assumes that NFV monitors are deployed on hosts at the leaf nodes of the tree, i.e. mixed with other standard servers. In the future, deploying NFV services on mixed hardware/software switches higher up in the network hierarchy may become feasible. We also try to minimize the resource consumption on those hosts caused by NFV monitors, thus we design different placement strategies for different trade-offs between network bandwidth and host resource consumption. Since our focus is on quickly deploying monitors for short term queries, we rely on fast heuristic algorithms rather than optimization formulations.

Monitors: Given a query, NetAlytics extracts all flows to be monitored according to the FROM and TO statement in the query. For each flow, NetAlytics first determines its source and destination host. For example, in Figure 2, there are three flows, (S1:*->D1:80), (S2:*->D2:80) and (S3:*->D3:80). NetAlytics then finds all Top of Rack (ToR) switches that contain the source and destination hosts. We say such switches `cover` that flow.

We have two observations about monitor placement. First, a flow f can only be monitored by a monitor under a ToR switch which covers f. In Figure 2, the flow (S1:*->D1:80) can be monitored by a monitor in either rack 2 or rack 5. Second, one monitor under a ToR switch sw is able to monitor all flows covered by sw. For instance, the monitor in rack 5 can track both (S1:*->D1:80) and (S3:*->D3:80). Based on these two observations, we implement two monitor placement strategies, random and greedy. Algorithm 1 details our two monitor placement algorithms.

Algorithm 1 Monitor placement

```
Input: $F$: flow set; $strategy$: monitor placement strategy
while $F$ is not empty do
    if $strategy = random$ then
        $sw \leftarrow$ random pick a ToR switch covering
                  at least one flow
    else if $strategy = greedy$ then
        $sw \leftarrow$ choose a ToR switch covering most flows
    $h \leftarrow$ host on switch $sw$ with minimal load
    $m \leftarrow$ place a monitor in $h$
    for each $f$ in flows covered by $sw$ do
        if m does not have enough capacity then
            $break$
        $m.monitor(f)$
        $F.remove(f)$
```

The random strategy randomly selects a ToR switch and places a monitor in the host with minimal load on that switch. Then it uses that monitor to track as many flows as the monitor can. This process repeats until all flows are monitored. Instead of a random choice, the greedy strategy always places a monitor on a ToR switch which covers the most flows. For example, in Figure 2, the greedy strategy places two monitors, in rack 5 and rack 6. The aim of greedy strategy is to reduce the number of monitors to be placed.

Analytics Engines: Unlike monitors, analytics engine placement does not have a position constraint. Data extracted from any monitor can be sent to any analytics engine, and bandwidth is less of a concern since we expect monitors to only send a small amount of data from each flow to the aggregation layer. We consider three strategies to place analytics engines. Since the placement strategy for aggregators and processors are very similar, we only discuss aggregators in the following scenario.

Initially we tried a purely random algorithm–for each monitor, randomly choose a host to place an aggregator. But this random approach performs too poorly in terms of both network and resource cost. Instead, we create a variant we call local-random. For a monitor, before creating a new aggregator, we check if there are some existing aggregators connected to the same aggregate switch with the monitor. If so, we choose that aggregator to process the monitor; otherwise a random host is picked for the new aggregator.

Next we consider a first fit strategy. This method will not place a new aggregator until all existing aggregators are saturated. Specifically, we keep using one aggregator to process monitors until the aggregator has no capacity left. Then we randomly choose a host to place a new aggregator. This minimizes resource cost (i.e., the number of servers used for aggregation and processing), but potentially incurs high network cost.

To reduce network cost we consider a greedy strategy that shares the same idea with the monitor placement. It first greedily finds an aggregate switch which connects to the most monitors and chooses a host nearby the monitor under that aggregate switch to place an aggregator. Again, the host is selected according to its free capacity. If there is no available host, it falls back to choose one from the whole hosts set. The detailed implementation of this greedy approach can be found in Algorithm 2.

Algorithm 2 Aggregator placement

```
Input: $M$: monitor set
while $M$ is not empty do
    find a switch $sw^*$ among aggregate switches which
    connects to the largest number of monitors in $M$
    $M^* \leftarrow$ set of monitors under $sw^*$
    $h \leftarrow$ a host under $sw^*$ with enough capacity
    if $h$ is $NULL$ then
        $h \leftarrow$ select one from all hosts with enough capacity
    $a \leftarrow$ place an aggregator in $h$
    for each $m$ in $M^*$ do
        if a does not have enough capacity then
            $break$
        $a.process(m)$
        $M.remove(m)$
```

Our goal with this greedy method is to avoid sending traffic up to the core switch level, since this wastes a more limited network resource. One possible placement of aggregators and processors is shown in Figure 2. This approach may result in a larger number of aggregators and processors being deployed than strictly necessary, but each will require fewer resources since the processing load is more spread out. If the number of used hosts is a concern, the first fit approach is preferable.

Our focus with these algorithms is on determining placements with different network bandwidth and host resource trade-offs. Once a placement is determined, we assume the position of aggregators and processors are fixed for that query. Dynamic management of the the analysis layer during a query is beyond the scope of this work, but techniques such as [28] could be used.

4.2 Feedback-Driven Sampling

By default, monitors send all packets through the parsers they are running, which in some cases may produce a higher tuple output rate than the analytics engine or even the aggregation layer can handle. To prevent overloading the other parts of the system and to minimize wasted network bandwidth, we propose an adaptive sampling scheme based on feedback from the aggregation layer.

Each aggregation server can observe the rates that new tuples are arriving from monitors and being sent to the processing engines. In Storm, the processors pull data from the aggregation layer. If the pull rate is less than the arrival rate of new data, then the aggregation buffers overflow, causing data to be dropped. This data loss wastes network bandwidth between the monitors and aggregators.

To prevent this situation, we use a back pressure mechanism wherein the aggregation layer observes its input and output rates to see if the system is overloaded. If the input rate is too high and memory is low, then the aggregator sends a status message back to the monitor indicating it has low buffer space. The monitor consequently adjusts its sampling rate, reducing the number of packets sent to the aggregator. Expanding upon this idea, the monitor could inform the SDN controller of the overload. The controller could even adjust its sampling rate so that fewer flows are sent to the monitor, reducing the bandwidth overhead both between the target server and the monitor, and between the monitor and the aggregation layer.

5. NETALYTICS IMPLEMENTATION

NetAlytics's monitoring system uses Intel Data Plane Development Kit (DPDK) [3], a library that provides high performance access to packets by bypassing the OS with a user space poll-mode driver. While our current prototype runs as a multi-process application directly on each host, encapsulating these processes within virtual machines using an NFV platform is a future endeavor in order to further simplify deployment [27, 24, 33, 18]. We use Kafka and Storm for the aggregation layer and stream processing engines and wrap them in a control framework that automatically starts the different components depending on the desired query.

5.1 Monitor Design

While NetAlytics monitors receive a mirrored packet stream and thus do not directly impose any latency on running applications, they must be able to maintain a high throughput and scale up efficiently as network speeds increase. In fulfillment of this goal, our implementation employs these key concepts:

Zero-copy, Lockless: DPDK allows multiple processes to access a packet with no copying and provides lock free data structures that prevent synchronization overheads.

Multi-level queuing: We architect our monitoring system as a hierarchy of queues, which improves scalability by dedicating cores and queues for each worker thread.

Batching: We use batching both within the monitor's pro-

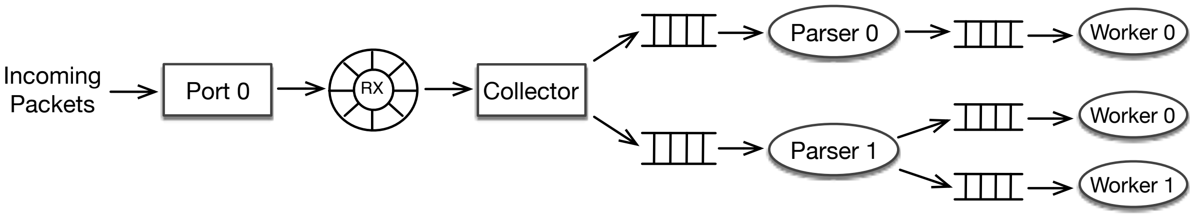

Figure 3: NetAlytics monitor architecture includes the collector, parsers, and an output interface

cessor pipeline to lower the overhead of queue manipulation operations and when preparing data tuples to be sent to the aggregators.

Sampling: We use adaptive sampling that drops packets early, reducing both CPU load and network bandwidth if the downstream analytics system cannot keep up.

5.2 Monitor Architecture

NetAlytics monitor framework includes the collector, parsers, and an output interface. It is built on DPDK 2.1.0 and takes advantage of its high performance packet processing mechanisms.

Collector: receives packets from the NIC and puts them into parser queues based on the sampling rate. The collector puts a pointer to each packet into the queues, i.e. it does not copy the packets themselves. The pointer eliminates copying overheads but allows every packet to be processed in parallel by all the different parsers running on the monitor. While we have found a single Collector thread sufficient for handling a 10Gbps input rate, one or more cores could be dedicated to each network port using Receive Side Scaling to load balance across the collector threads on higher rate networks.

Parser: contains the protocol logic to extract important packet data. Each parser runs as a separate process that uses shared memory to access the packet queues filled by the Collector. One parser process may run multiple worker threads; this provides scalability for computationally intensive parsing functions.

ProtocolLib: implements common functions to work with Ethernet, IP, TCP and UDP headers, in addition to payload data. As a result, new parsers can be written with minimal code. For example, our HTTP GET parser requires only 12 lines of application-specific code.

Output Interface: reorganizes processed data in a specific format such as JSON and outputs the message via a TCP socket or Kafka producer.

Figure 3 illustrates how packets flow through the monitor. The Collector polls the NIC for new packets which are DMA copied into shared memory. Then the Collector puts a copy of the packet descriptor (i.e., a small data structure with a pointer to the packet content) into the queues for all running parsers, so each parser can analyze the packet in parallel. In the simplest case, the parser may only have a single worker thread that examines the new packets and possibly emits a data tuple to the output interface. The more complex parser in the lower portion of the figure makes use of a two-level queuing system and multiple worker threads so that inspection of different packets can be scaled across multiple cores. The parser's dispatcher distributes packets among the set of workers using round-robin order or based on the packet flow ID to ensure consistent processing of flows, and we have a reference count on each packet so we know when all collectors and parsers have finished with it and it can be deleted.

5.3 Storm Topology

NetAlytics can use common Apache Storm patterns to analyze the data gathered by monitors. Here, we describe how our `top-k` topology is implemented since it is the most complicated processor used in our experiments.

The Storm topology performs top-k analysis on monitored data as shown in Figure 4. This flow expands upon the Storm-Starter project's Rolling Top Words topology, which is similar to the approach Twitter uses to produce trending topics and images. Once raw packets have gone through the monitors, a list of keys (e.g., HTTP GET URLs) are emitted and sent to the Kafka layer in batches.

Storm then uses multiple Kafka "Spouts" (i.e. data sources linked to the Kafka servers) to poll for new messages. The retrieved data is then emitted to the Parsing Bolts. The Parsing Bolts hash the raw string obtained from Kafka to get a signature. Each of these bolts emit the signatures with a respective count of one to a Counting Bolt selected based on the signatures. The hashing ensures that even if there are multiple Parsing or Counting bolts, counts for the same URLs will always be directed to the same instance. As a result, the Counting Bolts are capable of producing rolling counts of the signatures. Since we are interested in the global top-k instead of local signature frequencies, we combine the frequencies using the Ranking Bolts. The Ranking Bolts use a parallel reduction to construct rolling local top-k's and then combine them into the rolling global top-k. The final output list of the top-k signatures can then be stored in a database, used to update the query output display, or even matched back to URLs.

6. NetAlytics EVALUATION

In this section we evaluate the scalability of the NetAlytics architecture and the efficiency of our placement algorithms; use cases for the system are discussed in Section 7. Experiments are performed on a cluster of HP servers with dual Xeon 5600 2.67 GHZ CPUs (6 cores per socket), a Intel 82599EB 10G Dual Port NIC, and 32GB memory. All servers are connected by a 10 Gigabit Ethernet switch.

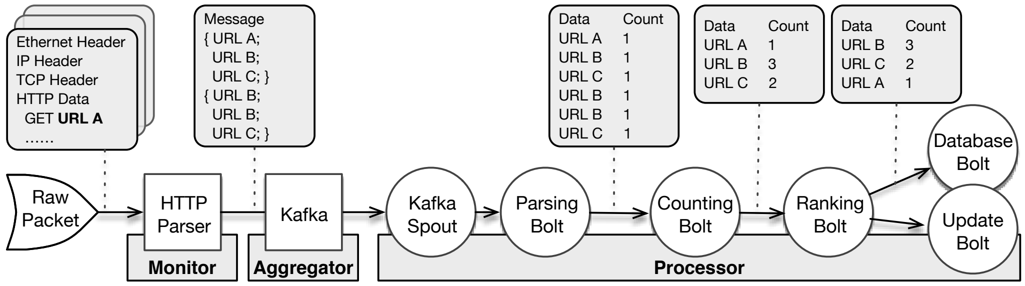

Figure 4: The flow of packets from the monitors through Storm.

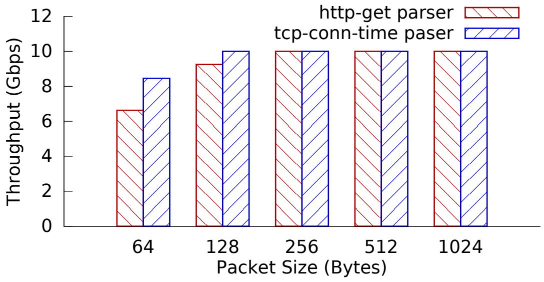

Figure 5: Monitor Throughput

6.1 System Scalability

The goal of NetAlytics is to scale to large data center networks where gigabits of traffic needs to be monitored and then processed with low latency.

Monitors: We first evaluate the performance of the monitor to demonstrate its ability to provide high speed packet parsing. We use one server to run the monitor and the other for traffic generation using PktGen-DPDK from WindRiver [30].

Figure 5 shows the achieved throughput when running two different parsers: `tcp_conn_time` which performs minimal processing and simply emits a data tuple when a SYN or FIN flag is seen and `http_get` which does a deeper packet analysis including copying out the URL requested in an HTTP stream. In both cases we only use one thread (i.e, one core) for the parser being tested. From the results, we can see the simple TCP parser can achieve a higher throughput, meeting the 10Gbps line-rate for packet sizes of 128 bytes. The more complex HTTP parser incurs greater overhead because of its string processing requirements but still is able to meet the line rate for packets sized 256 bytes or greater, illustrating the efficiency of NetAlytics's monitoring platform. Our prior work has shown that adding additional threads and NICs can scale the underlying NFV platform's throughput up to 68 Gbps when processing real traffic flows [44].

Processors and Aggregators: The analytics system based on Kafka and Storm also must scale to handle high traffic rates. We deployed four Kafka nodes, four Storm nodes and three Zookeeper nodes. Zookeeper can provide service discovery and coordination for both Kafka and Storm, and we use three Zookeeper nodes to ensure it won't be the performance bottleneck. We run our experiments using one monitor and keep the ratio of Kafka brokers and Storm workers as 1:2 so that storm spout and bolt don't need to share the same worker process. Figure 6 shows the maximum input rate can be handled by NetAlytics as we adjust the number of monitors, Kafka brokers and Storm workers. With 16 processes for the analytics subsystem, NetAlytics can process over 4 Gbps of incoming data. This modest CPU cost would be sufficient to analyze the 40 Gbps links common at the core of data center networks, assuming a 10:1 data reduction factor between the monitor and the aggregator.

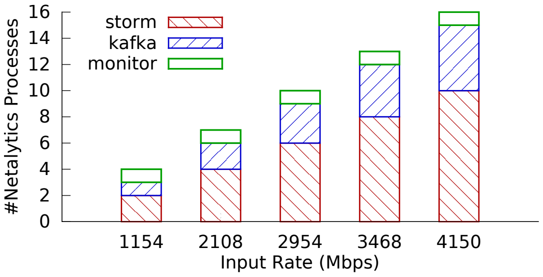

Figure 6: NetAlytics scales when increasing the number of processes, the minimum NetAlytics setup includes four processes (monitor, kafka, storm spout and bolt).

To reach this rate we optimize the configuration of Kafka and Storm to focus on throughput rather than reliability. Typically, Kafka provides reliable message delivery by persisting copies of all messages to disk, limiting throughput to the disk write rate (70 MB/s). Instead, we use a RAM disk to store Kafka's log and reduce the data retention window, which improves throughput by more than an order of magnitude. Since NetAlytics queries already involve sampling the data stream, the potential for message loss is not significant for our use case.

6.2 Placement Simulation

We have implemented all the proposed monitor and analytics engines placement algorithms in a simulator and use a common data center topology to evaluate our design.

Topology: Our simulations use a three-level fat tree topology with k=16, which contains 1024 hosts, 128 edge switches, 128 aggregate switches and 64 core switches described as in [5]. We consider memory and CPU resource constraints at the host when placing a NetAlytics node. The memory capacity of each host is a random number between 32 to 128 GB and the CPU capacity is a random number between 12 to 24. The utilization of both resources is between 40% to 80%.

Workload Generation: To simulate a data center workload, we use a staggered traffic distribution: 50% within the ToR

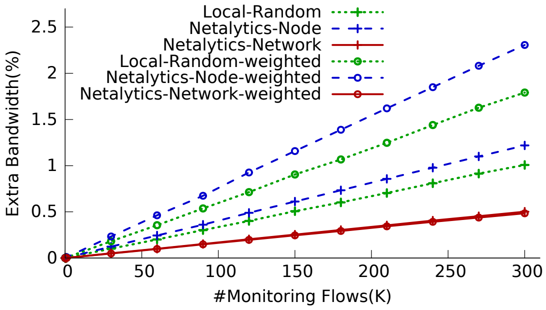

Figure 7: Network cost of placement algorithms

switch, 30% within the same aggregate switch, and 20% across a core switch (ToRP=0.5, PodP=0.3, CoreP=0.2) as in prior work [26]. Flow size distribution is generated by referring to [9]. The parameters of NetAlytics nodes are based on our system evaluation results: each monitor process can handle 10 Gbps traffic, one aggregator and two analyzer processes can handle 1 Gbps traffic. We assume each host has a 10 Gbps Ethernet NIC and has 2K to 20K flows. The total number of flows is around 1000K and traffic size is around 1.2 Tbps. In each experiment, we set the number of flows that need to be monitored and then randomly choose these flows from the total workload. At the monitors, only 10% data will be extracted and sent to the aggregators, and the aggregators will send all data to the processors. We run each experiment at least 10 times with random seed to get a stable average cost.

Network and Resource Cost: We use network cost and resource cost to evaluate different placement algorithms. The metric of network cost is the ratio of extra bandwidth consumed by NetAlytics to the original workload traffic. We use two ways to calculate this metric, one is Bandwidth Cost defined as the total bandwidth that all flows consume times the number of hops the flows need to go through from the monitors to the aggregators. The other one is Weighted-Bandwidth Cost which takes the network topology into consideration, and gives each hop a different weight based on which link it needs to use. We use a weight of 1 from host to ToR switch, weight 2 to the aggregate switch, and weight 4 for core links. This metric is important since not all links are equal in the data center, and cross-rack traffic is more expensive and should be discouraged. The resource cost is measured by the total number of NetAlytics processes which includes all running monitor, aggregator and processor processes.

Placement Algorithms: We compare three placement algorithms Local-Random, Netalytics-Node and Netalytics-Network. In the Local-Random algorithm, the optimized random strategy is used for placing both monitors and analytics engines. The goal of Netalytics-Node algorithm is to minimize the number of nodes used for monitoring and processing, and it uses the random strategy to place monitors and first fit strategy for analytics engines. Netalytics-Network tries to minimize traffic overhead given the resource constraints, and it uses the greedy strategy for all placements. The simulation results demonstrate the trade-off between traffic and resource consumption.

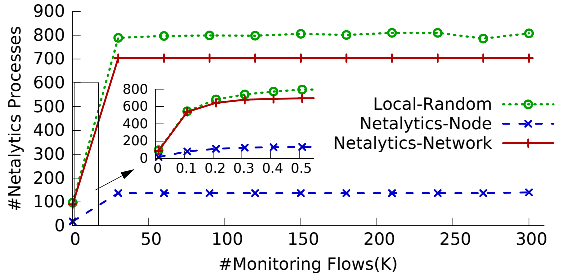

Figure 8: Resource cost of placement algorithms

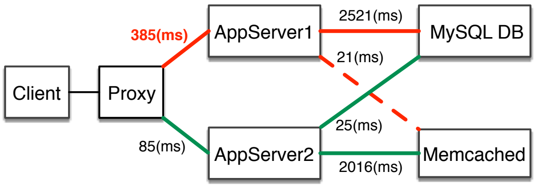

Figure 9: Architecture and response times for each tier

Netalytics-Network Placement incurs the least amount of extra traffic. Figure 7 shows the network cost of three placement algorithms. The extra bandwidth increases linearly with the number of flows, but Netalytics-Network algorithm consumes the least network bandwidth. Most importantly, the two lines of Netalytics-Network almost overlap which means it successfully limits most NetAlytics traffic within the rack. This is the result of its greedy coverage strategy that tries to minimize the distance between monitors and analytics engines. Netalytics-Node performs the worst because the first fit placement requires monitored data to be redirected across the whole topology. Local-Random improves the analytics engines' locality, so its network cost is lower than Netalytics-Node.

Netalytics-Node Placement consumes the least amount of resource. Figure 8 shows the resource cost of the three placement algorithms. As we expect, the resource consumption of Netalytics-Node is the lowest, as it always tries to fill current nodes before creating a new node. Though it's possible Local-random puts analytics engines near monitors, its randomness still causes higher recourse cost than Netalytics-Network. All algorithms level off with large number of flows. The reason is one monitor can handle more than 100K flows with average 10KB flow size. And due to data reduction, we only need a small number of analytics engines which are also insensitive to the number of flows.

7. NETALYTICS USE CASES

In this section we describe sample use cases where NetAlytics can help diagnose performance issues, analyze application performance at fine granularity, and automate resource management.

7.1 Multi-Tier Performance Debugging

Our first use case shows how NetAlytics can detect bottlenecks and debug network performance issues by monitoring response time and throughput statistics. We setup a two-

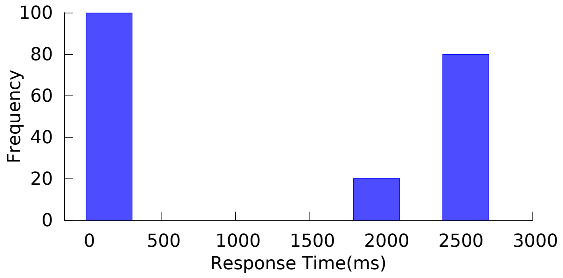

Figure 10: Bimodal response times at the client side

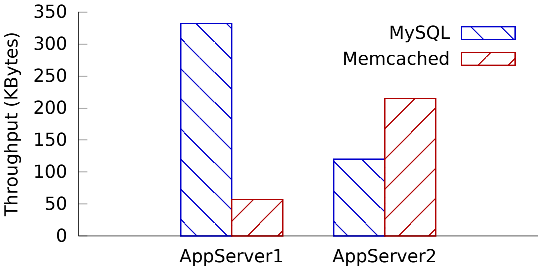

Figure 11: Throughput for mysql and memcached

tier web application illustrated in Figure 9; the application has a web front end proxy that load balances requests to a replicated application tier. The application tiers retrieve data either from a MySQL database or Memcached.

In the Figure 10, we can see the response time observed at the client side is anomalous, with a much higher percentage of requests having a longer response time than expected. To diagnose this problem, a common method is to monitor the CPU utilization on all servers. However, the CPU load on both App servers is around 12% since each is simply receiving an input request and then making a network call to either the MySQL database or the Memcached server. Checking the CPU usage on the database and cache only reveals that both servers have moderate load.

With NetAlytics, we first issue a query to start a `tcp-conn-time` parser and a `diff-group-avg` processor to determine the response time breakdown per-tier. The parser reports the start and end time of each TCP connection. The processor is a generic topology that can be configured for various purposes. In this case, it takes the difference of the start and end times emitted by the parser, groups them by source and destination IP, and calculates the average for each group. As shown in Figure 9, we find the response time between Proxy and App Server 1 is four times larger than the connections with App Server 2, although the times for reaching the database and cache are similar from both. This simple query has quickly helped the administrator narrow down the problem, although it is not yet clear why App Server 1 is incurring a higher response time.

We next start a `tcp-pkt-size` parser and a `group-sum` processor to check the throughput for each connection, and as shown in Figure 11, we find three times higher network usage between App Server 1 and MySQL, and four times smaller load to Memcached. From this, we can infer something is wrong with App Server 1's configuration, and it turns out it has been misconfigured and most of its requests are served by the MySQL database instead of the much faster cache, leading to the substantially higher overall response times observed from the proxy.

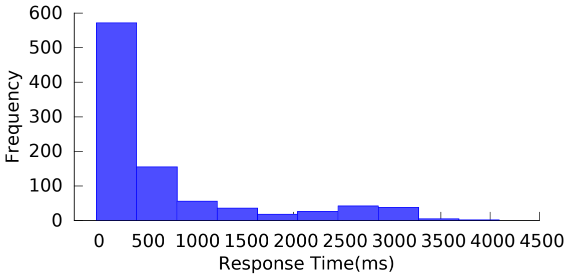

Figure 12: Client response times

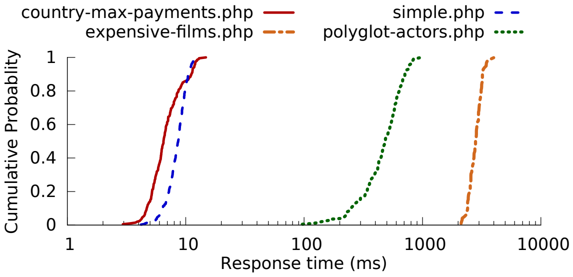

Figure 13: CDF of several web urls

This scenario illustrates how NetAlytics can help an administrator quickly measure per-tier performance at fine granularity. By allowing this kind of interactive debugging, it becomes far easier to determine the root cause of performance anomalies in a distributed system, all without requiring any application-level logging or server modifications.

7.2 Coordinated Performance Analysis

A second use case for NetAlytics is as an in-depth performance analysis tool that gives application developers insight into the the behavior of specific web pages and database queries. Here we analyze a PHP web application that executes queries against the sample MySQL Sakila DVD rental database. To understand its performance requires that NetAlytics dispatch queries that combine several different monitors spanning different network layers.

We first run the simple `PARSE tcp_conn_time FROM * TO h1:80,h2:3306 LIMIT 500s SAMPLE * PROCESS (diff-group: group=destIP)` query to gather response time data for all TCP connections to either the web or database host. Since our web application performs minimal processing on its own, the response times are dominated by the MySQL lookups, so the response times returned for each host are nearly identical. We plot the histogram of web response times in Figure 12.

Delving deeper, an administrator might issue a second query that makes use of two parsers:
`PARSE (tcp_conn_time, http_get) FROM * TO h1:80 LIMIT 500s SAMPLE * PROCESS (diff-group: group=get)`. NetAlytics can start both parsers which independently send the requested URL and the connection time to the processors, which will group the results based on the page requested, combining both network and application-level data. Consequently, developers or administrators can quickly see how the response times vary for

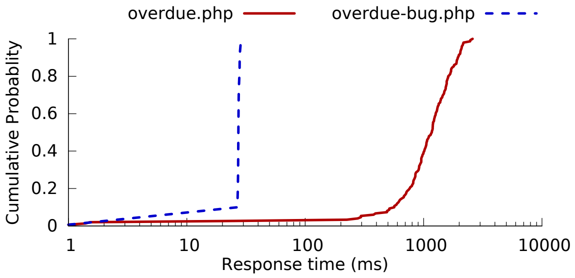

Figure 14: Response time cdf of overdue.php and overdue-bug.php

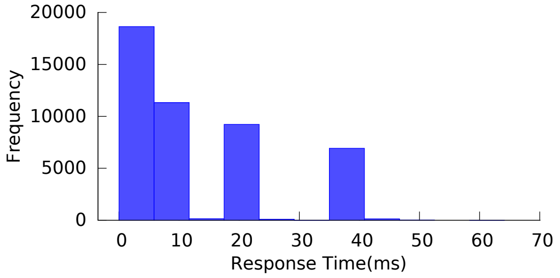

Figure 15: Response time histogram of MySQL query

different pages, as shown in Figure 13.

When running this test, we identified a bug in our own PHP code: an incorrect variable name in the script was causing the page to not issue the appropriate database queries. As a result, a page that should have had a very large response time was completing with minimal latency, as shown in Figure 14. While this is only anecdotal evidence, it illustrates how NetAlytics could be useful as part of a regression testing suite that analyzes real-time performance after an application is updated.

Finally, we examine the performance of the MySQL tier more closely by measuring the response times for individual SQL queries. Since MySQL permits several queries to be sent over a single TCP connection, measuring the full connection time hides the individual query times. We have implemented a `mysql` parser which observes a TCP stream to detect individual query/response pairs. This parser emits timing information on a per-query basis, as well as the query statement itself. Figure 15 shows the response time breakdown for individual SQL queries.

While it is possible to obtain most of this information by processing application logs, NetAlytics provides a non-intrusive and efficient solution. As a comparison point, we measure MySQL's throughput with and without the general query log enabled. The query log records response times for all queries, but we find that it lowers the the throughput for a simple statement from 40.8K to 33K queries per second, a 20% drop. In contrast, NetAlytics incurs no overhead on the actual application, and allows performance statistics to be gathered only when needed.

7.3 Real-Time Popularity Monitoring

Next we consider a richer setting where we show how NetAlytics can be used to continuously automate resource management based on live network data.

To illustrate the potential for this situation, we first analyze a real network trace released by Zink et al. [48] which contains information about client requests for YouTube video clips recorded at a network gateway. We use our `top-k` processor to analyze the T6 trace and find the Top-10 popular videos within each timing interval. Figure 16 shows the popularity of the 2nd and 3rd most popular videos over time, illustrating how even this top content sees fluctuations over time. These measurements suggest that real time network information could be useful for optimizing resource allocations and data replication management.

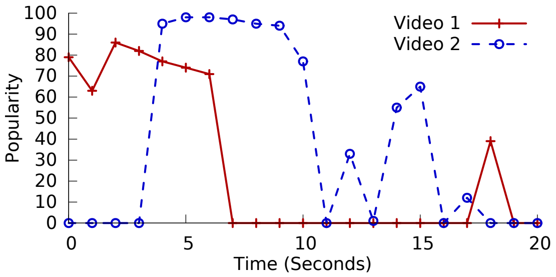

Figure 16: NetAlytics uses Storm to measure video content popularity over time (100 is most popular video).

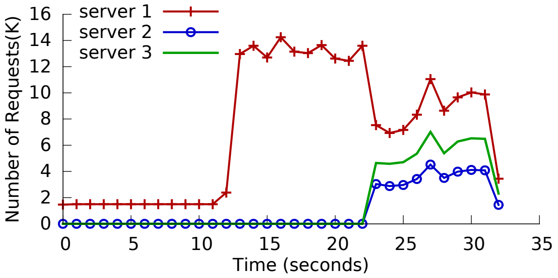

Figure 17: NetAlytics monitoring data can be used to dynamically adjust the number of replicas for popular content

To emulate such a scenario, we use a setup with a load balancer, three web servers, and two clients. Initially, we have the first client make HTTP GET requests for one of 1000 urls at a moderate rate, and at a certain point, we make the second client perform HTTP GET requests for 10 urls at a fast rate to simulate the top-10 hot content.

To gather and process this data, we issue a query to start a `http_get` parser and a `top_k` processor in NetAlytics. The parser examines the body of client HTTP GET request packets to determine the content being requested and the accessed video server. These URLs are then passed to the top-k analysis storm topology described previously in figure 4.

Given the sheer volume of data produced by the monitor, we cannot effectively analyze the results in real-time on a single machine. Instead, NetAlytics instantiates multiple Kafka and Storm nodes and routes the data gathered by the parser to the aggregators.

Rather than simply display the list of top URLs to a user, we tie NetAlytics in this experiment to an automated resource management system that automatically adjusts the number of web servers being used and dynamically replicates content across them. To do this, we utilize the top-k processor's ability to store the URLs of the most popular content into a Redis in-memory data store. We also use an Updater Bolt within the topology that checks if the frequency of a URL is above a configurable upper threshold. If so, it will add a server to the web server pool and replicate the popular content to it. Likewise, the Update Bolt will remove a server when the top-k frequency is below a configurable lower bound. In order to prevent rapidly increasing and lowering the number servers based on the rolling top-k's, we force the Update Bolt to back off for a predetermined amount of time before resuming examining the top-k for further adjustments.

In order to load balance incoming requests across the server pool, we use an NGINX proxy that obtains its configuration from the Redis database filled by the top-k Database Bolt. Thus we are able to implement a dynamic proxy which can quickly adjust to web traffic characterized by the NetAlytics platform.

The results are shown in Figure 17. During the first ten seconds, the workload is low and evenly distributed, but after 10 seconds the load on web server 1 rises when the second client starts sending requests for a set of 10 urls. NetAlytics automatically detects the surging popularity of this content and initializes two more servers to handle the load. As expected, after 23 seconds, part of the load on server 1 is redistributed to server 2 and 3.

Currently, this kind of content popularity information is usually gathered by using daily or weekly log data analysis tools such as Hadoop. As a result, resource allocations based on application-level data are done manually at coarse time intervals. With NetAlytics, we can provide these types of insights at much finer grained time intervals, with no instrumentation of the server applications themselves.

8. RELATED WORK

Distributed System Performance Diagnosis: Previous research projects have presented a variety of diagnosis tools and frameworks to monitor and debug distributed systems [39]. Common approaches includes monitoring performance metrics [20, 38, 45], instrumenting applications [8, 14, 23, 42] and annotating logs [31, 41]. For example, X-Trace [14] presents a framework to trace multiple applications at different network layers, but it needs to instrument all of the elements and reconstruct casual relations offline to do debugging. Pivot Tracing [23] introduces a novel happened-before join operator that combines dynamic instrumentation and causal tracing which means it can diagnose real-world issues more effectively, yet it is Java-specific and requires instrumentation of each application. Thus, many prior approaches are application specific and may rely on experts with deep knowledge and experience of the system, which makes them less practical in large deployments.

Several approaches for transparently determining application performance have been proposed [4, 32, 34, 47]. Projects like Aguilera et al. [4] treat each component of a distributed system as a blackbox and perform performance debugging by observing message traces. NetCheck [47] performs diagnosis by using a blackbox tracing mechanism to collect a set of system call invocation traces from the relevant end-hosts. NetAlytics has the same end goals as these works, but in this paper, we focus on how to efficiently deploy network monitors in software-based networks and use a scalable processing engine to analyze network data in real time.

Networking Monitoring and Debugging: Networking monitoring and control are always hot research topics in the networking communities. However, many operators are still using traditional tools like tcpdump [37], traceroute and NetFlow [12] to collect information from the end hosts or switches, pinpoint the network problems and manually modify the configuration.

The emergence of SDN provides the possibility to simplify these time-consuming tasks. Compared to traditional network, SDN provides network-wide visibility and allows for constantly monitors network conditions and reacts rapidly to problems [29]. Projects such as Hedera, MicroTE, OpenSketch, OpenSample and CherryPick [6, 10, 35, 36, 43] presented several approaches to measure and control the network such as detecting congestion and rerouting flows using OpenFlow [25] interface. However, these projects only focus on analyzing flow-level data or aggregated information, which makes them unapproachable to packet granularity problems such as faulty interfaces [46].

Recent projects include EverFlow, Planck, NetSight and OFRewind [16, 29, 40, 46] presented techniques to access the packet level information and achieved different levels of network visibility. For example, NetSight [16] records the full packet histories and provides an API for network analysis programs to diagnose problems. EverFlow [46] leverages commodity switch's "match and mirror" capability to trace specific packets and help network administrators troubleshoot DCN faults. NetAlytics takes inspiration from these works and focuses on how to gather and process packet-level data in a more flexible and efficient way. Our implementation leverages a set of emerging techniques and tools such as DPDK, Storm and SDN controllers. Compared to other approaches that may use one of them independently, we focus on how to automatically glue these layers together to provide analysis results in real time while incurring minimal performance overhead.

9. CONCLUSIONS

The number and diversity of distributed applications are growing in both public clouds and private data centers. Understanding the performance of these applications is a major challenge, particularly when it is not feasible or desirable to directly instrument the applications or the virtual machines they run in. We have described NetAlytics, a platform for large-scale performance analysis by processing network data. NetAlytics uses Network Function Virtualization to deploy software-based packet monitors into the network, and uses Software Defined Networking to steer packet flows to the monitors. The captured data is then aggregated and sent to a processing engine based on the Apache Storm real-time data analytics engine. Our NetAlytics prototype illustrates how transparent network monitors can help diagnose performance issues, analyze application performance at fine granularity, and automate resource management.

Acknowledgements: We would like to thank Niky Riga, Sarah Edwards and Vicraj Thomas for their discussions in the early stages of this project and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-1253575 and CNS-1422362.

10. REFERENCES

[1] Apache storm http://storm.apache.org/.

[2] Apache storm scalable http://storm.apache.org/about/scalable.html.

[3] Data plane development kit. http://www.dpdk.org/.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74. ACM.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19. USENIX Association.

[7] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling end-host network functions. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 493–507. ACM.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18. USENIX Association.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12. ACM.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10. USENIX Association.

[12] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 245–256. ACM.

[13] N. Feamster, J. Rexford, and E. Zegura. The road to SDN. *Queue*, 11(12):20:20–20:40.

[14] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation*, NSDI'07, pages 20–20. USENIX Association.

[15] X. Ge, Y. Liu, D. H. C. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. OpenNFV: Accelerating network function virtualization with a consolidated framework in OpenStack. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14. ACM.

[16] N. Handigol, B. Heller, V. Jeyakumar, D. MaziÃÍres, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85. USENIX Association.

[17] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mSwitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 1:1–1:13. ACM.

[18] J. Hwang, K. Ramakrishnan, and T. Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *Symposium on Networked System Design and Implementation*, NSDI 14.

[19] E. T. S. Institute. Network functions virtualization (NFV): An introduction, benefits, enablers, challenges & call for action. *White Paper*.

[20] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381. IEEE.

[21] J. Kreps, N. Narkhede, J. Rao, and others. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*.

[22] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901.

[23] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM.

[24] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473. USENIX Association.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

[26] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 157–170, Berkeley, CA, USA, 2013. USENIX Association.

[27] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136. ACM.

[28] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. Middleware15.

[29] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. pages 407–418. ACM Press.

[30] W. R. T. Report. Wind river application acceleration engine.

[31] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 9–9. USENIX Association.

[32] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 347–356.

[33] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112. USENIX.

[34] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia. Precise, scalable, and online request tracing for multitier services of black boxes. *IEEE Trans. Parallel Distrib. Syst.*, 23(6):1159–1167.

[35] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter. OpenSample: A low-latency, sampling-based measurement platform for commodity SDN. pages 228–237. IEEE.

[36] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 23:1–23:7. ACM.

[37] tcpdump. Tcpdump/libpcap http://www.tcpdump.org/.

[38] VMware. Resource management with VMware DRS. *Technical Resource Center*.

[39] C. Wang, S. P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi. Performance troubleshooting in data centers: An annotated bibliography. *SIGOPS Oper. Syst. Rev.*, 47(3):50–62.

[40] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 29–29. USENIX Association.

[41] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132. ACM.

[42] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 57–70. USENIX Association.

[43] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42. USENIX.

[44] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 2016.

[45] X. Zhang, Y. Zhang, X. Zhao, G. Huang, and Q. Lin. SmartRelationship: a VM relationship detection framework for cloud management. pages 72–75. ACM Press.

[46] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491. ACM.

[47] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. NetCheck: Network diagnoses from blackbox traces. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 115–128. USENIX Association.

[48] M. Zink, K. Suh, Y. Gu, and J. Kurose. Characteristics of YouTube network traffic at a campus network âĂŞ measurements, models, and implications. *Computer Networks*, 53(4):501 – 514. Content Distribution Infrastructures for Community Networks.