2018 IEEE 11th International Conference on Cloud Computing

A Toolset for Detecting Containerized Application's Dependencies in CaaS Clouds

Pinchao Liu, Liting Hu, Hailu Xu, Zhiyuan Shi, Jason Liu, Qingyang Wang[†], Jai Dayal[‡], Yuzhe Tang[§]
Florida International University, [†]Louisiana State University, [‡]Intel Corporation, [§]Syracuse University
Email:{pliu002, lhu, hxu017, zshi005, liux}@cs.fiu.edu, qywang@csc.lsu.edu, jai.dayal@intel.com, ytang100@syr.edu

***Abstract*—There has been a dramatic increase in the popularity of Container as a Service (CaaS) clouds. The CaaS multi-tier applications could be optimized by using network topology, link or server load knowledge to choose the best endpoints to run in CaaS cloud. However, it is difficult to apply those optimizations to the public datacenter shared by multi-tenants. This is because of the opacity between the tenants and the datacenter providers: Providers have no insight into tenants container workloads and dependencies, while tenants have no clue about the underlying network topology, link, and load. As a result, containers might be booted at wrong physical nodes that lead to performance degradation due to bi-section bandwidth bottleneck or co-located container interference.**

We propose 'DocMan', a toolset that adopts a black-box approach to discover container ensembles and collect information about intra-ensemble container interactions. It uses a combination of techniques such as distance identification and hierarchical clustering. The experimental results demonstrate that DocMan enables optimized containers placement to reduce the stress on bi-section bandwidth of the datacenter's network. The method can detect container ensembles at low cost and with 92% accuracy and significantly improve performance for multi-tier applications under the best of circumstances.

Keywords*—*Virtualization, Clustering, Dependency Analysis.

I. Introduction

Recently, there has been a dramatic increase in the popularity of Container as a Service (CaaS) clouds, including Microsoft Azure's Container Service [11], Amazon's EC2 Container Service [1] and Lambda offerings [2], and Google's Container Engine service [4]. CaaS is a form of container-based virtualization in which container engines, orchestration and the underlying compute resources are delivered to users as a service from a cloud provider. With CaaS, users can boot, organize, run, scale, manage and stop containers using a provider's API calls or web portal interface. Users purchase the containers from the cloud providers and run applications inside containers.

Many applications, such as RUBiS [16], Hadoop MapReduce [10], Storm [18] are multi-tier applications in which two or more components cooperate and communicate to jointly provide a certain service or accomplish a job. A typical multi-tier application, for instance, contains a presentation tier for basic user interface and application access services, an application processing tier for processing the core business or application logic, and a data access tier for accessing data, and finally a data tier for holding and managing data.

These CaaS multi-tier applications could be optimized by using network topology, link or server load knowledge to choose the best endpoints to be placed [23] [24] [25] [30] [34]. For example, when container sizes and application workloads are known, the interference between competitive multi-tier applications can be mitigated by mapping their containers to remote physical servers or racks. Power savings can be achieved by squeezing lightweight containers that use complementary resources into a small subset of physical servers and turning off the redundant servers. However, these optimizations can only apply to the private datacenter where the datacenter provider owns both applications and the underlying infrastructure, and cannot apply to the public datacenter shared by multi-tenants. This is because of the ***opacity*** between the tenants and the datacenter providers: tenants have no insight into the underlying datacenter network topology, link and load. Providers have no insight into the tenant's container workloads. As a result, naively placing the heavily communicating containers across the racks which have the bottlenecked bi-section bandwidth [20] [27] [33] can also lead to unacceptable latencies. Naively co-locating the containers that run interactive queries, and the containers that run batch jobs while eagerly consuming any idling resources on the same servers can also lead to unacceptable latencies.

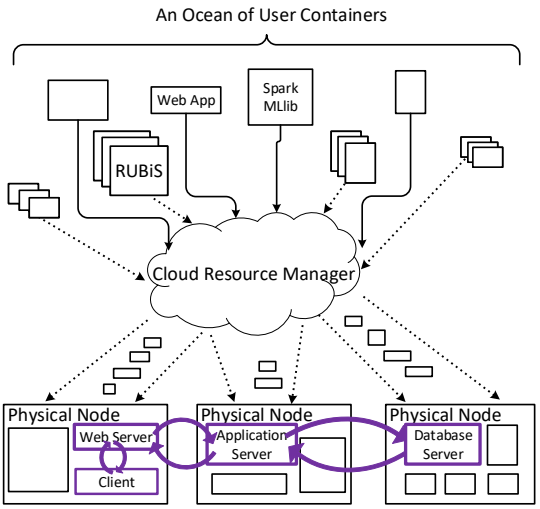


Fig. 1: A containers cluster use case.

Figure 1 illustrates a use case scenario. The cloud provider uses the container orchestration tools such as Kubernetes [8] for automating deployment, scaling, and management of containerized applications. It randomly places containers using a bin-packing algorithm [37] as long as the physical server can satisfy their resource requirements, without considering the dependencies between containers. In this case, client requests arrive at the container running the web server front end and are then forwarded to one of the containers running application servers, which in turn may request data from a backend container hosting a database.

As illustrated in Figure 2, an ensemble of frequently communicating, the 'chatty' containers are placed across

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00032

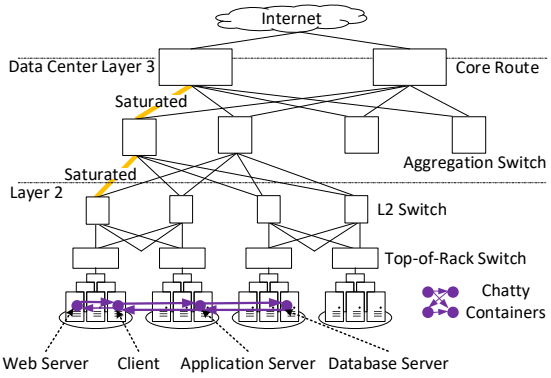


Fig. 2: Example of shared up-links from Top-of-Rack crash, causing performance degradation for many containers.

multiple racks. Such a placement can negatively affect the services provided by the ensemble. Since these are typically 5:1 to 20:1 oversubscribed, this can result in a worst-case available bi-section bandwidth as low as 125Mbps [28]. Furthermore, the higher level switches in the network topology cost much more, due to the amount of network bandwidth and numbers of ports they have to support. Recent studies [19] [27] [33] have shown that the servers in different racks have to share the up-links from top of rack switches (ToRs). First, as the shared up-links from ToRs become saturated, intra-ensemble communications may be delayed. Such delays can be further exacerbated by message re-transmissions due to time-outs. Second, the use of scarce, shared bandwidth can affect other services and ensembles, as evidences in applications like Hadoop MapReduce that experience slowdown due to file system-level data reorganization. Finally, a link failure can cause severe imbalances across paths and may require relocation of some of the containers in order to reduce over-subscription.

In this paper, we propose DocMan, a novel container API that (1) identifies potential container ensembles in cloud via resource utilization logs, (2) assesses the degree of "chattiness" among the containers in these potential ensembles, thereby (3) enabling optimized container placement to reduce the stress on bi-section bandwidth of the data center's network. DocMan has the following unique properties:

- ***Transparency and privacy-preserving*** — as a non-intrusive tool, DocMan does not require any code modification to the containers or applications.
- ***Lightweight*** — unlike the network sniffer tools that bring heavy burdens to the management software, DocMan has negligible CPU and memory overheads.
- ***Actionable*** — insights derived from running DocMan can help management software better co-locate container ensembles on underlying hosts.

Discovering container ensembles and their inter-container dependencies are quite challenging. A naive method that continuously gathers statistics about all communicating containers is prohibitively expensive. First, it would require introspection of all packets sent and received by the containers; this would induce notable CPU overheads and additional per packet latency of the tens of microseconds. Second, additional memory resources would be required to maintain statistics for every pair of IP addresses.

DocMan uses a three-step approach. Firstly, it acquires container-level statistics commonly available in container management systems (e.g., Kubernetes), such as the total numbers of packet in/out over time. Secondly, it computes the correlation coefficients among these statistics. Finally, it divides the corresponding containers into subsets (also called ensembles) using correlation values and a hierarchical clustering algorithm [36].

We have designed and implemented DocMan and evaluated its effectiveness on a testbed that consists of 20 physical servers with 114 Docker containers. These containers run a diverse mix of business, web, Internet services, and batch workloads. Experimental results show that DocMan can identify container ensembles with 92.0% accuracy, and improve the application performance in terms of throughput and latency by making sophisticated placement decisions. In particular, we observe an up to 625% improvement in the application throughput for a RUBiS instance, a 33.4% improvement in application throughput for Hadoop MapReduce instances, and 7.9% improvement for Spark Streaming instances under the best circumstances.

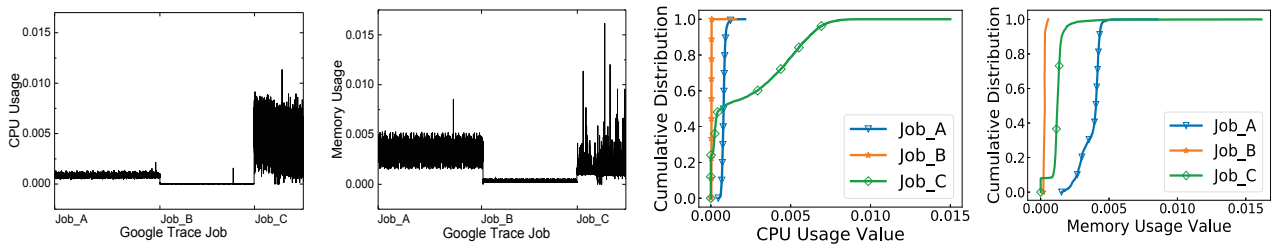
The remainder of this paper is organized as follows. Section II discusses background and motivation. Section III describes the DocMan design and implementation. Section IV presents the experimental setup and performance evaluation. Section V discusses the related work. We conclude with some directions for future work in Section VI.

II. MOTIVATION

The novelty of our work lies in that, instead of using an intrusive approach to detect containerized application's dependencies (e.g., via intercepting packet [21] [31] [32] or injecting code [22] [26] [35]), DocMan uses a ***non-intrusive*** approach via capturing the CPU, memory and I/O logs for detecting dependencies. The rationale behind our approach is that we observed that real-world containerized multi-tier applications exhibited strong correlation among their resource usage statistics.

We analyzed Google traces [5]. The log represents 29 day's worth of cell information on May 2011, on a cluster of 12.5k machines. The cluster uses Linux containers [9] for resource isolation and usage accounting. A job is comprised of one or more tasks, each of which is accompanied by a set of resource requirements used for scheduling (packing) the tasks onto machines. Each task runs within its own container. Tasks are scheduled onto machines according to the given lifecycle. We randomly selected 3 jobs from Google traces in the same period, each job containing 1005, 999, and 1997 tasks separately.

Fig. 3a and Fig. 3b show that the different jobs have different resource usage patterns, indicating that there exists the hidden dependencies within the tasks that belong to the same job. For example, the "CPU spikes" and "CPU valleys", or the "memory spikes" and "memory valleys" of the tasks that belong to the same job (i.e., container ensemble) consistently occur together. To summarize resource consumption range of tasks belonging to the different jobs, Fig. 3c and Fig. 3d show the Cumulative Distribution



(a) CPU rate. (b) Memory usage. (c) CDF of CPU rate. (d) CDF of Memory usage.

Fig. 3: Resources usage statistic for 3 jobs in Google trace log during a same period. (Job_A is Job4665892165; Job_B is Job6261947650; and Job_C is Job6486323509.)

Function (CDF) results. The results demonstrate that the tasks of the same job are highly likely to fall into the same range of CPU or memory utilization, which can be used to infer the intra-ensemble correlations discussed below.

III. DocMan Design

This section illustrates DocMan's design, which includes (1) monitoring of basic resource usage statistics of containers; (2) computing the correlation coefficients among these statistics to identify their distances; and (3) using hierarchical clustering algorithms to detect potential container ensembles.

A. Data Collection

The first step is to capture the following universally available system-level metrics about each container:

CPU: CPU usage per second in percentage terms (%).

Memory: Memory usage per second (MB).

I/O: Packets transmitted per second (KByte/sec).

These periodic measurements result in three time series signals per container. Before explaining the actual analysis being applied, we illustrate the utility of taking these measurements with a simple example.

Multi-tier applications (e.g., RUBiS [16]) typically use a request-response architecture, in which a client container sends a request to the front end (e.g., Apache), which assigns the work to an appropriate server (e.g., Tomcat) running the application logic. The application logic services the request by querying the backend (e.g., a database server like MySQL) to produce the necessary output, and sending the response back to the client. Therefore, given the nature of multi-tier applications, we can expect correlations between the *CPU*, *Memory* and I/O statistics for interacting containers. For example, an instant rise of CPU usage or packet flow rate in one container directly or indirectly triggers activities in other containers, thereby creating the correlations among these statistics.

To more clearly show the correlations among these statistics, we did an experiment on two RUBiS instances with 2000 clients and 500 clients workloads, respectively. Each instance has three containers, i.e., client, Tomcat server and MySql database. The container-level metrics of CPU, memory and I/O for these containers are continuously collected and compared in Fig. 4, where the system load is the normalized value of CPU, memory plus I/O with the same weight. The results show that the containers that collaborate together to accomplish the same task within the same instance tend to have similar trends of system load, while the containers that are responsible for different tasks do not exhibit correlations. The same logic can be applied to other applications as well, such as big data analytics application like Hadoop MapReduce applications [10], Storm applications [18], Spark Streaming applications [17], which are popular in CaaS clouds. For example, MapReduce adopts a Partition/Aggregate pattern which scale out by partitioning tasks into many sub-tasks and assigning them to worker containers (possibly at multiple layers). These worker containers are expected to work together to accomplish the task.

There are several design tradeoffs about data collection:

- *Metrics selection.* Although "I/O" (PacketIn and PacketOut) is a more intuitive metric to do correlation analysis among communicating containers, we also include CPU and memory as complementary metrics because they will provide extra information in certain scenarios. For example, if a bunch of containers do many-to-many but infrequent data exchanges, it is difficult to detect them using only I/O metric, especially when the interval between data exchanges is larger than the sampling window. In such cases, other metrics can capture the missing signals.
- *Sampling window.* Larger sampling window does not always means more accurate result. For example, if the large window accidentally covers the idle period when the container ensemble has no inter-communications. The final result may not be as accurate as smaller windows. On the other hand, if the sampling window is too small, it may fail to catch up the useful information timely. Therefore, we need to find the appropriate sampling window for each representative application.

B. Distance Identification

Each containers log can be organized as a vector of an array (α, β, γ). α is the *CPU* usage record; β is the *Memory* usage record; and γ is the I/O usage record, respectively. Then, the all containers generate a vector matrix for the cloud service provider to analyze. We choose the Pearson product-moment correlation coefficient (PMCC) [13] to measure the degree of correlation, giving a value between -1 and +1 inclusive.

The mathematical logic is as below: there are two vectors X and Y to demonstrate any two vectors which are

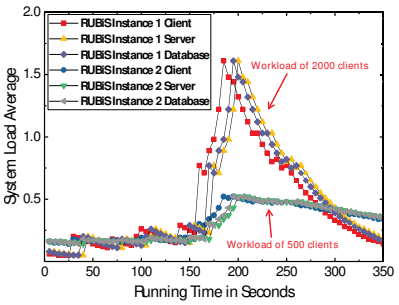


Fig. 4: Example of a multi-tier RUBiS application showing correlation on server usage. The different sets of RUBiS applications have their own resources usage trend.

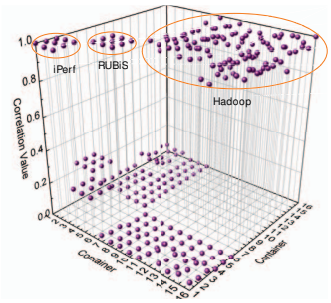


Fig. 5: 3D plot of correlation matrix between containers including iPerf containers, RUBiS containers, and Hadoop containers.

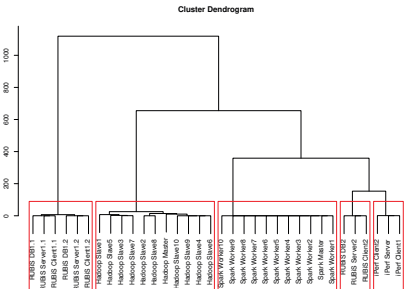


Fig. 6: Hierarchical tree generated by clustering algorithm.

generated by the containers. Each element is an array of resources usage record in the vector. $X = X_1, X_2, \ldots, X_n$, $Y = Y_1, Y_2, \ldots, Y_n$, then the correlation between X and Y is:

$$corr = \frac{\sum_{i=1}^{n} \left(\begin{pmatrix} X_{\alpha_i} \\ X_{\beta_i} \\ X_{\gamma_i} \end{pmatrix} - \begin{pmatrix} X'_{\alpha_i} \\ X'_{\beta_i} \\ X'_{\gamma_i} \end{pmatrix} \right) \left(\begin{pmatrix} Y_{\alpha_i} \\ Y_{\beta_i} \\ Y_{\gamma_i} \end{pmatrix} - \begin{pmatrix} Y'_{\alpha_i} \\ Y'_{\beta_i} \\ Y'_{\gamma_i} \end{pmatrix} \right)}{\sqrt{\sum_{i=1}^{n} \left(\begin{pmatrix} X_{\alpha_i} \\ X_{\beta_i} \\ X_{\gamma_i} \end{pmatrix} - \begin{pmatrix} X'_{\alpha_i} \\ X'_{\beta_i} \\ X'_{\gamma_i} \end{pmatrix} \right)^2} \sqrt{\sum_{i=1}^{n} \left(\begin{pmatrix} Y_{\alpha_i} \\ Y_{\beta_i} \\ Y_{\gamma_i} \end{pmatrix} - \begin{pmatrix} Y'_{\alpha_i} \\ Y'_{\beta_i} \\ Y'_{\gamma_i} \end{pmatrix} \right)^2}} \tag{1}$$

The calculation result *corr* is between -1 and 1. If it is 0, it means they have no relationship. Near -1 means that they are negatively related with each other and near 1 means that they are positively related with each other. Since the negative relationship is meaningless in practice, we update all the negative numbers to 0 in the output matrix.

Fig. 5 shows the result that is performed by Python package numpy[12] calculation. It illustrates that the containers are performing same tasks, which are having a higher value near to 1. Such as the value between container 0, 1, and 2, the respective values are 1.00, 0.99, and 0.99. In the 3D chart, the values are at the top part. However, comparing to other containers, they have a lower value which is near to zero, that means they do not have a direct relationship. Such as 1 and 15, the value is only 0.03. For 2 and 14, the value is only 0.04. These values are at bottom part in the chart.

We further define the concept of distance to describe the strength of dependencies between two containers:

$$Distance(X_i, Y_i) = \begin{cases} \frac{1}{corr_i}, & corr_i > 0 \\ \infty, & corr_i = 0 \end{cases} \tag{2}$$

The distance value matrix will be used as input information for the next step hierarchical clustering.

C. Hierarchical Clustering

Clustering[3] is the process of making a group of abstract objects into classes of similar objects. A cluster of data objects can be treated as one group. Two commonly used clustering algorithms are hierarchical clustering and *k*-means clustering. DocMan uses hierarchical clustering for the following reasons:

- Hierarchical clustering does not require the number of clusters in advance.
- It works well with both globular and non-globular clusters, while *k*-means fails to handle non-globular data.
- *k*-means clustering is sensitive to initial centroids. If the user does not have adequate knowledge about the data set, this may lead to the erroneous results.

The process of hierarchical clustering is as follows:

- Step 1: initially assign each container to a cluster, so that there are N initial clusters for N containers.
- Step 2: find the closest (most similar) pair of clusters and merge them into a single cluster.
- Step 3: compute distances (similarities) between the new cluster and each of the old clusters.
- Step 4: repeat Step 2 and Step 3 until all items are clustered into a single cluster of size N.

Concerning Step 4, of course, there is no point in having all N items grouped into a single cluster, but doing so results in the construction of the complete hierarchical tree, which can be used to obtain k clusters by just cutting its $k-1$ longest links. K can be based on the number of racks in the datacenter, or it can be chosen to make the inter-cluster distance less than a certain threshold.

IV. Experimental Evaluation

Our experimental evaluations answer the following questions:

- What is the accuracy rate of DocMan's toolset for identifying the dependencies among the containers?
- What are the potential performance gains of the containerized applications by applying the DocMan toolset for new placement?
- What are the runtime overheads of the DocMan toolset?

Our key evaluation results are as follows.

- The accuracy rate of the DocMan toolset averages 92%. Classification the area under the ROC (Receiver operating characteristic) curve is 0.93. ROC is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.
- By applying the DocMan toolset, for RUBiS instance, the latency is reduced by an average of 6 times and the throughput is increased by 72 times under the best circumstance. For Hadoop and Spark MapReduce tasks

`Word Count` and `Secondary Sort`, the latency is reduced by 21%.
- The DocMan toolset's CPU overhead is 55% lower than Wireshark and 14% lower than tcpdump. Its memory overhead is 97% lower than Wireshark and 30% lower than tcpdump.

A. Testbed

The test is performed on 20 dual-core dual-socket servers. Each of them have two Intel processors, 4GB of memory and 20GB hard drives. The 20 servers are equally distributed across 4 edge switches. All switches and NIC ports run at 1Gbps. The switches are connected with each other, with an oversubscription ratio of at most 4:1. All servers run on Ubuntu 16.04 with Docker version 17.06.

Experiments employ 3 instances of Hadoop MapReduce (30 containers), 3 instances of iPerf (9 containers), 3 instances of RUBiS (9 containers), 3 instances of Sparks Streaming (30 containers), 3 instances of Storm (27 containers), and 3 instances of Redis (9 containers), resulting in a total of 114 containers running a mix of business, internet services, and batch workloads.

B. Workload and Metrics

Apache Hadoop Mapduce(v2.7.1) [6] is an open-source software framework used for distributed storage and processing dataset of big data using the MapReduce programming model. We use three instances of 10 Hadoop MapReduce containers, one of them is master, the rest are slaves. The namenode process and the YARN cluster manager are launched on the master node, and each slave node is responsible for launching its own datanode process.

Apache Spark Streaming(v1.4) [17] is a fast and general engine for large-scale data processing. We use three instances of 10 Spark Streaming containers, one of them is master, the rest are slaves. The master process and the built-in standalone cluster are started on the master node, and each worker is responsible for launching the executor process.

iPerf3 [7] is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers, and protocols. We use three instances of iPerf containers. Within each instance, there are one server and two clients.

RUBiS(v1.4.2) [16] is an eBay-like benchmark. We use a PHP-based configuration of RUBiS, with a web server front end (Apache) and an application server (Tomcat) connected to a backend database (MySQL). The workload is generated by a web simulation client.

Apache Storm(v1.1.0) [18] is a distributed real-time computation system. It makes easy to process unbounded and real-time streams of data. We set three instances of 9 Storm containers, one of the containers is Nimbus, and three of them are ZooKeepers and the rest of them are Supervisors. Topologies of WordCount, Reach and RollingTopWords are running on them separately.

Redis(v3.2) [15] is an in-memory data structure store, used as a database, cache and message broker. It implements a large hash table on top of system calls. We use three instances of 3 Redis containers. In each instance, one of them is the server, the rest are clients.

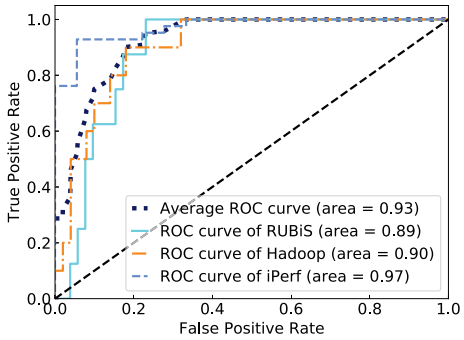


Fig. 7: ROC curve shows the accuracy of DocMan black box method. The overall accuracy area is 0.93, which is considered as excellent level.

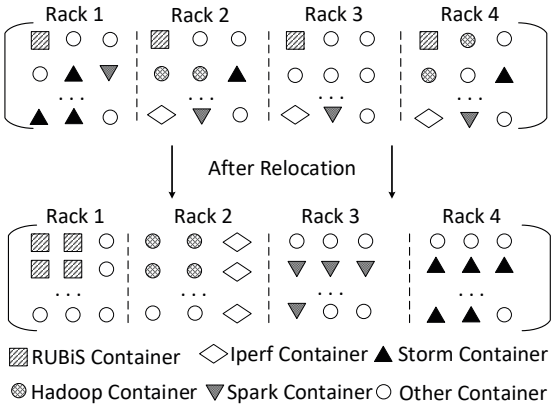


Fig. 8: Before and after containers mapping.

C. DocMan's Functionality Evaluation

We first evaluate the accuracy rate of DocMan in finding the right set of ensembles. Then, we evaluate the benefits of these findings for improving container's placement on hosts through the comparison of the applications throughput and response time between initial containers placement and final containers placement. Finally, we evaluate the runtime overhead, if any, induced by the DocMan toolset.

1) DocMan's Accuracy Rate

Fig. 6 shows the hierarchical trees constructed using the decreasing level of the dependencies strength. We firstly determine a $N*N$ correlation strength matrix, and then run the hierarchical clustering algorithm over the distance matrix. The hierarchical trees are generated by R [14], which shows the calculated dependencies between the containers being observed. It demonstrates that DocMan can effectively expose most of the underlying dependencies between the containers within our testbed. To fulfill the requirement of dependencies detection, DocMan black-box process does not require any code modification to the containers or the applications.

Fig. 7 shows the ROC curve of classifiers. The ROC curve is used to describe the accuracy rate of container classification results. It is created by plotting the true positive rate against the false positive rate. In our cases, it is more

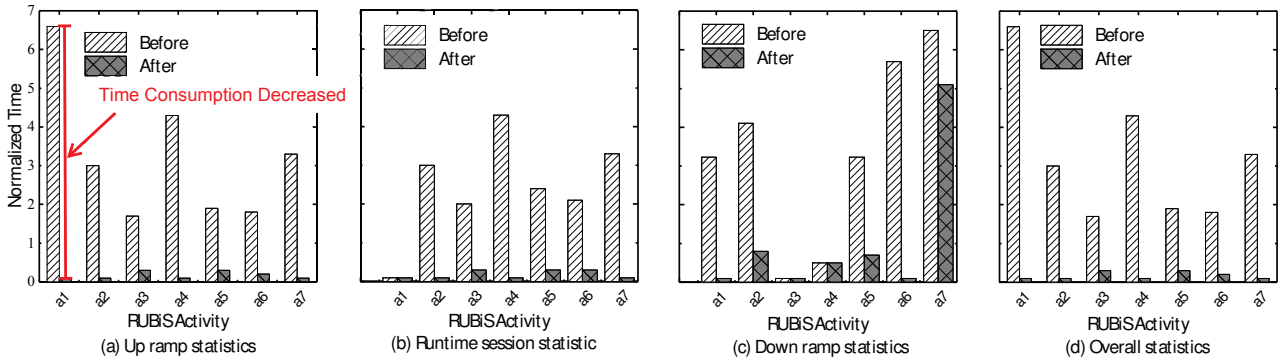


Fig. 9: Performance comparison for RUBiS on different running stages and overall statistics (RUBiS Activity: a1:Home a2:Browse a3:Browse Categories a4:Search Items In Category a5:Browse Regions a6: Browse Categories In Region a7: Search Items In Region).

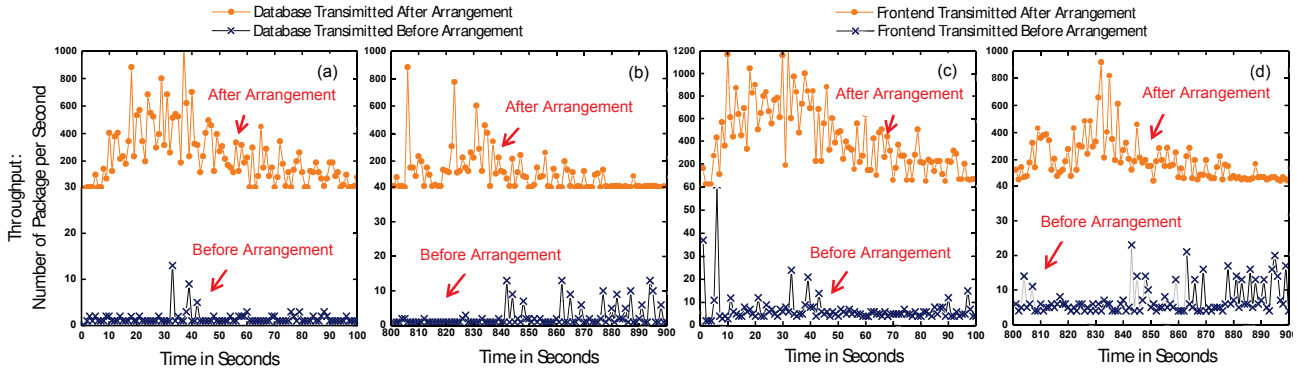


Fig. 10: Throughput comparison for RUBiS before arrangement and after arrangement. It shows after arrangement the throughput increased averagely 1.93 times than before arrangement. (a) (b) are the database transmitted packages throughput during up ramp period and the runtime period. (c) (d) are the front end transmitted packages throughput during up ramp period and the runtime period.

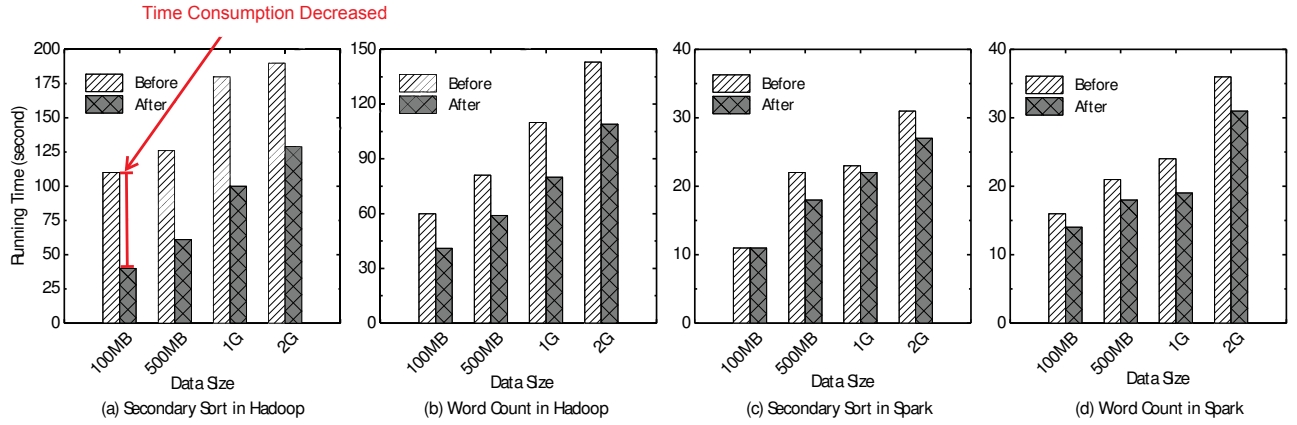


Fig. 11: Latency comparison of Hadoop MapReduce application and Spark Streaming application before arrangement and after arrangement.

complicated than the binary classifier since more than two applications can co-exist on the same rack. We depict three classifiers accuracy results of RUBiS, Hadoop MapReduce and iPerf, and the overall average accuracy line, which is calculated based on the separate accuracy results. The average accuracy is 92% and the overall underline average area of ROC is 0.93. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test. The result shown in the Fig. 7 is considered as an excellent one.

2) DocMan's Potential Benefits

After determining the container ensembles and the hierarchical clustering of container instances, we next evaluate the benefits of using this information as inputs for improving container placement on the hosts.

Fig. 8 shows the applications mappings to hosts and racks before and after arrangement driven by DocMan's insights. We set the bandwidth between different racks as 2Mbps limited bi-section to simulate the scenario that upper-level network traffics are saturated. According to the intra-ensemble dependencies, we put the same instance applications to one rack to avoid the saturated network traffics, then prove that the new placement can lead to better application performance. Here, we assume that CPU and memory resources are not bottlenecks.

TABLE I: Compare With Other Monitoring Tools

Monitor Tool	Code Base	Memory Cost	CPU Peak	CPU Average
DocMan	**630 Lines**	**0.35GB**	**7.8%**	**6.7%**
Wireshark	30MB	1.20GB	68.0%	14.3%
tcpdump	931KB	0.05GB	10.0%	7.5%

Fig. 9 and Fig. 10 present the comparison results for RUBiS instance before and after arrangement, in terms of the response time and throughput separately. There are three different RUBiS running periods statistic data from 2000 rounds basic website activities. The periods are up ramp, runtime session, down ramp, and wrap them up as the final overall statistic. Fig. 9 shows that the response time is reduced after arrangement. For example, in the overall statistic, the activity of "view item" is faster 7.25 times. Meanwhile, Fig. 10 demonstrates the throughput increased 1.93 times averagely after arrangement since the new placement effectively avoids the saturated network.

Fig. 11 presents the enhanced performance of Hadoop MapReduce and Spark instances. It shows that the performance for Hadoop MapReduce is averagely increased by 33.4%, and performance for Spark is averagely increased by 7.9%. We deploy `Word Count` and `Secondary Sort` into Hadoop MapReduce and Spark Streaming instances. `Word Count` data source is generated by a program which randomly picks words from a dictionary file which includes 5000 English words, and then counts the number of occurrences of each word in a given input set. `Secondary Sort` problem relates to sorting values associated with a key in the reduce phase.

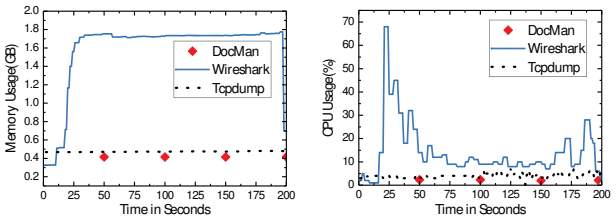
Experimental results clearly indicate the benefits of dependency-aware containers placement, thereby demonstrating that information obtained by using DocMan toolset can help improve placement and migration actions while reducing utilization of network resources in container as a service clouds.

3) DocMan's Overhead Analysis

Table I shows the comparison of DocMan and other two popular network monitoring tools, Wireshark and tcpdump. It demonstrates the attributes of the DocMan toolset's lightweight feature and low overhead. To analyze the overhead of DocMan toolset, the first phase of correlation detection has no extra overhead because the basic container level statistics are already available in container management systems (e.g.,Kubernetes). The second phase of distance identification has a complexity of $O(N^2)$ where N equals the number of containers, costing several seconds. The third phase of clustering has a complexity of $O(N^2)$ costing several seconds.

Figure 12 shows the overhead of DocMan toolset comparing with Wireshark and tcpdump. Wireshark is the worlds foremost and widely-used network package and protocol analyzer. Tcpdump is a common packet analyzer that runs under the command line. Figure 12a demonstrates that Wireshark costs 5 times more memory than DocMan or tcpdump. For DocMan clustering step, it runs regularly for every 50 seconds. The memory additional cost is stable and as low as 20MB.

Figure 12b shows the CPU usage statistics. To launch Wireshark, it incurs a CPU burst, which potentially impacts



(a) Memory usage. (b) CPU rate.

Fig. 12: Overhead comparation between DocMan, Wireshark and tcpdump.

other running container applications since it costs high CPU computation within a short period. Even it goes to stable status, it still consumes much more CPU resource than DocMan. CPU usage for tcpdump is similar with DocMan. However, users usually use tcpdump combine with Wireshark if they require a visualized result.

In summary, the overhead of the DocMan toolset is much lower than other popular network tools.

V. Related Work

We classify related literature into three different categories: Injecting code to obtain runtime information, based on communication pattern between cluster service provider and tenants to detect the dependencies, and optimizing cluster applications based on network packets inference.

Injecting code to obtain runtime information. Code injection technologies inject some extra code into the source code of target programs to capture runtime information. For example, Fay [26] uses runtime instrumentation and distributed aggregation to get software execution traces. vPath [35] provides path discovery by monitoring thread and network activities and reasoning about their causality. It is implemented in a virtual machine monitor, making it agnostic of the overlying middleware or application but it requires changes to the VMM code and the guest OS.

Based on communication pattern between cluster service provider and tenants to detect dependencies. Cloudtalk [19] lets users describe their tasks to the cloud and help them make appropriate choices for task placement. BtrPlace [29] is used as a planning tool to limit the number of applications or to predict the need to acquire new servers, meanwhile, provides a high-level scripting language, allowing service users to describe requirements. However, they did not consider the natural dependencies between tasks, and require users to have domain knowledge to provide accurate resources usage description.

Optimizing cluster applications based on network packets inference. Meng et al. [32] propose to initialize the applications in data center based on their network traffic information. They design a two-tier approximate algorithm to find the traffic patterns and adjust the network architectures to improve the cluster performance. Magpie [21] captures events from OS kernel, middleware, and application components and calculates the time correlation of these events. DocMan not only focuses on network metric but also focuses on other metrics, which enable more considerable factors for the provider to optimize cluster performance.

VI. Conclusion

In this paper, we study the dependency detection problem in a public CaaS environment. First, we identify that there exists hidden dependencies between containers that belong to the same application by monitoring their resource usage statistics at runtime. Second, we design a black-box toolset called DocMan to detect these dependencies with negligible overhead. Third, we evaluate the accuracy of DocMan with real-world containerized applications.

DocMan's methods are fully implemented, but additional work is required for using it to continuously detect and manage containers at cloud-scale. For example, we need to filter out background traffic noises (i.e., heartbeat packets), since such traffic might otherwise be interpreted as intra-ensemble communications. Further, it would be interesting to integrate DocMan into management solutions like Kubernetes and Docker Swarm. We also plan to leverage DocMan's insights to guide the container placement, thus improving containerized application's performance and amortizing the expenses related to their debugging and maintenance.

VII. Acknowledgment

The authors would like to thank the anonymous reviewers for their feedback that improved the paper significantly. We'd thank Florida International University Graduate & Professional Student Committee and School of Computing and Information Sciences for the travel award to present this work.

References

[1] Amazon ec2 container service. https://aws.amazon.com/ecs/.

[2] Aws lambda - serverless compute. https://aws.amazon.com/lambda.

[3] Cluster. https://en.wikipedia.org/wiki/Cluster_analysis.

[4] Google cloud platform container. https://cloud.google.com/container-engine.

[5] Google trace log. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[6] Hadoop. http://hadoop.apache.org.

[7] Iperf. https://iperf.fr.

[8] Kubernetes. https://kubernetes.io.

[9] Linuxcontainers.org. https://linuxcontainers.org.

[10] Mapreduce. https://en.wikipedia.org/wiki/MapReduce.

[11] Microsoft azure container service. https://azure.microsoft.com/en-us/services/container-service/.

[12] Numpy. http://www.numpy.org.

[13] Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[14] R project. https://www.r-project.org/about.html.

[15] Redis. https://redis.io.

[16] Rubis. http://rubis.ow2.org.

[17] Spark. https://spark.apache.org.

[18] Storm. http://storm.apache.org.

[19] Alexandru Agache, Mihai Ionescu, and Costin Raiciu. Cloudtalk: Enabling distributed application optimisations in public clouds. EuroSys '17, pages 605–619, New York, NY, USA, 2017. ACM.

[20] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin 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, Berkeley, CA, USA, 2010. USENIX Association.

[21] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[22] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. *SIGCOMM Comput. Commun. Rev.*, 43(4):231–242, August 2013.

[24] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[25] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[26] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. SOSP '11, pages 311–326, New York, NY, USA, 2011. ACM.

[27] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. SIGCOMM '10, pages 339–350, New York, NY, USA, 2010. ACM.

[28] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[29] Fabien Hermenier, Julia Lawall, and Gilles Muller. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Trans. Dependable Secur. Comput.*, 10(5):273–286, September 2013.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[31] Liting Hu, Karsten Schwan, Ajay Gulati, Junjie Zhang, and Chengwei Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 3–12, New York, NY, USA, 2012. ACM.

[32] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. INFOCOM'10, pages 1154–1162, Piscataway, NJ, USA, 2010. IEEE Press.

[33] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. NSDI'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[34] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM.

[35] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. USENIX'09, pages 19–19, Berkeley, CA, USA, 2009. USENIX Association.

[36] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[37] Minyi Yue. A simple proof of the inequality ffd (l) 11/9 opt (l) + 1, l for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7(4):321–331, Oct 1991.