Cross-Layer Topology Discovery Tool

COMPUTATIONAL AND DATA SCIENCE DEPARTMENT
INDIAN INSTITUTE OF SCIENCE, BANGALORE

Dhanya R Mathews

supervised by
J. Lakshmi, Mudit Verma, & Pooja Aggarwal

Abstract

As microservice architecture becomes prominent, existing fault management techniques to deal with service disruption become limiting mainly due to the amount of data needed to be analyzed. This report emphasizes the need to consider the cross-layer topology of the cloud service, including components involving different layers of the cloud stack to intelligently identify and correlate the observability data and assist in implementing efficient and more accurate fault management techniques that can provide better explainability. Towards this goal, the report presents a tool that derives cross-layer topology discovery for a cloud microservice application and discusses the benefits of using cross-layer service topology to implement effective fault management.

Keywords

Cloud Application Services, Cross-Layer Topology Discovery, Monitoring, Fault Management

CONTENTS

I. Introduction

In the recent past, cloud application services have seen an architecture shift from tightly coupled monoliths to loosely coupled microservices. With microservice architectures gaining prominence, the number of service components that constitute an application service has increased. A disruption in the service components or its dependent provisioning components across the different cloud service layers [1] can affect the application service quality. To effectively deal with service disruptions for cloud application services and to guard against SLA violations, such scenarios demand smarter autonomous methods to identify, localize, and mitigate faults or predict failures [2].

In order to deal with service disruptions autonomously, there needs to be an understanding of components involved in the service delivery across the different service architecture layers. As a cloud service disruption could be due to a failure or error in any of the components involved in its delivery, an awareness of those components can improve the ability of the service to deal with unexpected events that could potentially challenge its quality, thus improving its resilience. Further, end-to-end topology information can facilitate an understanding of how faults, errors, and failures propagate across the components in and across the different cloud service architecture layers [3].

Although individual components involved in the service delivery are well understood, the structure of the end-to-end dependency graph defines how individual components affect the overall performance [4]. As a fault in any component involved in the delivery of a service can challenge the service quality, the intelligence to deal with service disruptions need to be derived based on an understanding of the components involved in the delivery of the service and their state [3]. Topology discovery identifies the components and interconnections vital for service delivery and dynamically updates it to facilitate efficient monitoring. Efficient dynamic topology discovery is an antecedent for effective monitoring, and an effective monitoring framework is essential to facilitate autonomous SLA management. The service quality monitoring framework can use the information about the current topology to monitor the components and trigger autonomous failure management techniques. For example, in the case of an application service following a microservice architecture, the topology discoverer should identify the cross-layer service components and the dependent provisioning components dynamically. The monitoring framework can use this topology information to collect the observability data based on the SLO of interest for the identified components, monitor their state, and derive correlations among the observability data based on the interconnection. Further, such intelligent monitoring can better explain the impact of a fault or error, or failure and facilitate efficient proactive or reactive strategies to deal with service disruptions.

This report makes the following contributions:

1. Highlights the need for cross-layer topology discovery for cloud application services.
2. Present a tool that dynamically discovers the cross-layer topology of a microservice application.
3. Explains how a cross-layer topology discovery tool can facilitate efficient and explainable fault management.

II. Related Work

Monitoring and Topology discovery is an intriguing area of research in cloud computing. This section briefly discusses the state-of-the-art to understand the various approaches for mining the service components' topology in and across the service architecture layers. Most approaches that discover the service topology work at a given cloud service architecture layer. [5] deploys the Kubernetes manifest files for the microservice and mines the architecture of the microservice application. To identify the service interactions, this work deploys a monitoring container in all pods to sniff network packets sent to/from the container in the pod. Even though this tool was tried during the initial phase of this work to discover the service interactions, it was later replaced by Istio Kiali due to the time taken to discover the interactions. The authors of [6] present a hypervisor-based network monitoring system to track the connections between different VMs in a Data Center. Our work, instead of monitoring the network flows at the hypervisor, derives the VM interconnections based on the interactions among the services hosted on the pods inside the VMs.

[7] explores topology discovery with a different objective, i.e., to visualize and verify the deployment in a multi-cloud infrastructure. A VM dependency discovery system has been proposed in [8] to estimate the response time based on component interactions. IaasMon [9] presents an auto-discovery process for virtual and physical infrastructure by attaching a module to the communication middleware of OpenStack and then using the topology information to correlate monitored metrics.

In addition to the above research contributions, few commercial tools are available for discovering the service topology. Smartscape by Dynatrace [10] deploys an agent called OneAgent to discover the components involved in the service delivery in and across the cloud service architecture layers. Network Topology from Google [11] collects real-time telemetry and configuration data from Google's infrastructure to visualize your resources. Instana [12] uses end-to-end tracing based on Google dapper to model the dependencies between physical and logical components [13]. However, it isn't easy to use these tools for this work due to the unavailability of source code and APIs to integrate.

In the listed tools above, each of the referred tools was built for a specific use, and none of them have a comprehensive capability that provides API-based usage for integration into any monitoring or fault diagnosis frameworks. Further, the survey identified no open-source cross-layer topology discovery tool that works across the cloud stack and dynamically updates the topology information. An awareness of end-to-end service topology is essential for a monitoring framework to make informed decisions to ensure service resilience. The following sections briefly discuss a tool developed to dynamically discover the end-to-end topology for a cloud service and its benefits.

III. Topology Discovery Tool

This section presents the Topology Discovery tool, developed to discover the cross-layer topology of cloud application services dynamically. The motivation was to use some of the identified generic methods to develop a topology discovery tool that could generate necessary information about service components at runtime, including the effects of scaling or failure events. This topology information then becomes the input for any monitoring frameworks that can build the desired cross-layer insights. The tool also needs to find metadata information of interest to the monitoring module about the components identified. Though the current version of the tool has been implemented for a microservice application deployed on a Kubernetes cluster, the tool is planned to be extended to be used by any distributed systems and applications.

Fig. 1 demonstrates the different components and interconnections of interest for the topology discovery tool in and across the different cloud service architecture layers. As shown in the figure, at the service instances layer, the components of interest are the microservices, and the interconnection is the service interactions. At the virtual resources layer, the components of interest are the container on which the service is deployed, the Kubernetes pods and the virtual machines, and the metadata regarding the virtual CPUs, memory, and disk assignments. At the physical resources layer, the components of interest are the host and the physical resources. The current version of the tool works for virtual machines deployed on KVM. Also, the tool is developed using Python 3.8.

The topology discovery tool uses different command-line tools to derive the end-to-end topology for a cloud application service. The tool needs to identify the service interactions at the service instances layer and the interconnections across the different cloud service architecture layers. In order to find the service interactions, the tool allows the user to either provide a JSON file with the service interactions similar to a graph generated by Istio Kiali or has the capability to trigger the Locust test to generate traffic and use Istio Kiali to learn about the interactions, given the file locations are given as input to the tool. However, if the Locust test file is not provided by the developer or is not available and the user cannot derive a semantic meaning of the application, this can be a challenge. For example, for the Sock-Shop microservice application, the developer has provided the load test [14], and this can be used by the tool to derive service interactions. In other cases, the user either has to draft the service interactions file to be given as an input to the tool or has to depend on other open-source load tests, if available. The interconnections of interest among the different layers are : (i) the mapping of the service instance to the container, pod, and virtual node, and

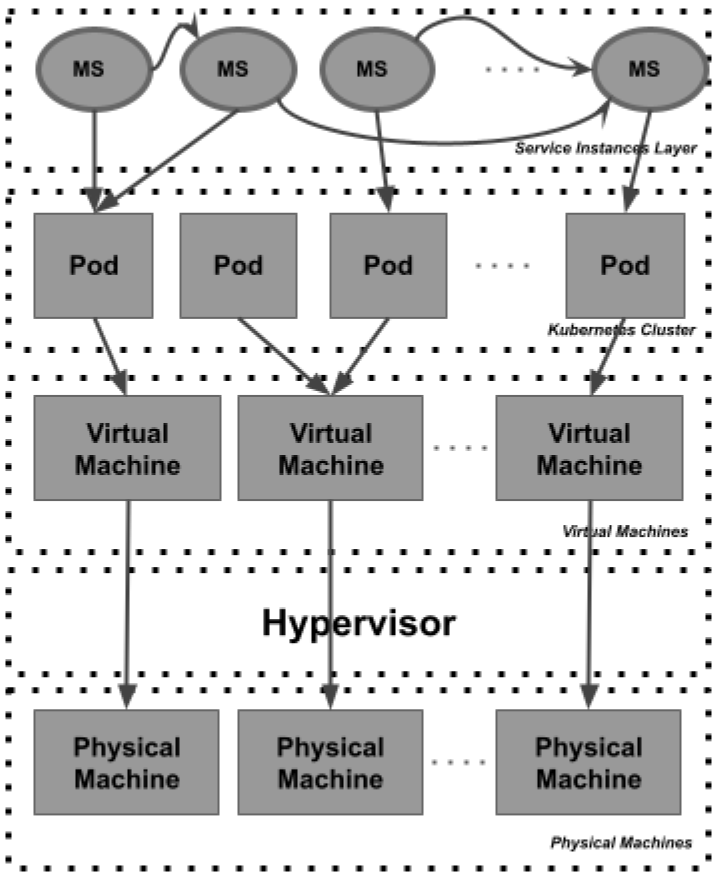


Fig. 1. Components of interest across cloud service architecture layers

(ii) the mapping of the virtual node to the physical node and physical resources. The tool uses the service name in order to identify the pod on which it is provisioned. To find the mapping between the virtual node and the physical node, the tool uses the name of the KVM VM. Once this information on the components and their interactions/interconnections are derived, the tool generates the initial topology information. After discovering the initial topology, the tool will then have to monitor the state of these components to identify events of interest that could potentially change the topology information. For this, the tool will deploy agents to track the Kubernetes events and libvirt events to discover the topology changes dynamically.

The topology information is generated by the tool in two different formats: JSON format and .gexf format. In the JSON file format, each object corresponds to a service, and the value identifies the services with which it interacts and the deployment components. The .gexf graph format aids visualization. The following sections will describe the architecture and working of the topology discovery tool in detail.

A. Topology Discovery Tool Architecture

This section describes the different components of the topology discovery tool and their working. To discover the end-to-end topology of a cloud application service, the topology discovery tool uses the following four components:

1. Topology Discovery Master
2. Topology Discovery Agent
3. Pod Events Tracker
4. VM Events Tracker

1) Topology Discovery Master: Topology Discovery Master is the controller or master process that aggregates the information from various sources, generates the initial topology information, and updates the topology information concerning the various events in the cluster. The master process takes the following as input:

1. Kubernetes configuration file of the remote cluster
2. Host informaiton file
3. Service interaction file or the Locust parameters
4. Kubernetes namespace of interest

The Kubernetes configuration file is used to connect to a remote cluster and get the details of pods, containers, and Virtual Machines. The Host information file contains the IP addresses of the physical machines in the cluster. This information is required in order to get the mapping between virtual machines and physical machines. The service interactions file or Locust parameters aid in understanding the interconnections at the Service Instances layer. Once this information is retrieved, the master process will aggregate these and generate the topology information in a JSON format and .gexf format. Following is the command to trigger the master process:

```
python3 td_master.py -i <host_information_file> -n <namespace> [-t <
    service_interaction_file> | -le <locust_endpoint> -lf <
    locust_file>] -c <kubernetes_configuration_file>
```

2) Topology Discovery Agent: The Topology Discovery Agent runs on physical machines in the cluster and communicates the information about the virtual machines and the virtual resources to the master process when queried. The agent process uses the virsh commands to generate the required information. The Topology Discovery Agent is triggered using the following command:

```
python3 td_client.py
```

3) Pod Events Tracker: Pod Events Tracker keeps track of the pods in the given namespace to identify events that result in state changes that could affect the topology information. The master process triggers this process after generating the initial topology information. The Pod Event tracker uses the Kubernetes python library to identify the events in the namespace. Once the tracker sees that the event is regarding a pod creation, modification or deletion, it marks those events. This process uses a push notification mechanism to inform the master process about the updates. It can be configured to send updates per event or aggregate the event for a given time period.

4) VM Events Tracker: The master process deploys the VM Events Tracker on the hosts in the host input file. It identifies the Virtual Machines' lifecycle events of interest using the libvirt APIs and pushes these updates to the master process.

Fig. 2 describes the interactions among the four components described above. The figure shows that the master process uses the information generated from different sources to generate and dynamically update the end-to-end cloud service topology.

B. Validation Tests

The topology discovery tool has been validated for three microservice applications with a diverse number of microservices: (i) Sock Shop [15], (ii) Train Ticket [16], and (iii) Hotel Reservation from the DeathStar [4] benchmark suite. The other microservice application from the DeathStar benchmark suit were not used because they lacked Kubernetes manifest files. The capability of the tool to identify dynamic topology changes has been validated by configuring Kubernetes pod autoscalers and simulating host lifecycle events. This section briefly describes the output that the tool generates.

Fig. 3 shows a sample execution of the Topology Discovery Master process. The figure shows that the tool generates output in both JSON format and a .gexf form. The figure also shows that the master process deploys the pod events tracker and VM events tracker and waits for update notifications from these processes.

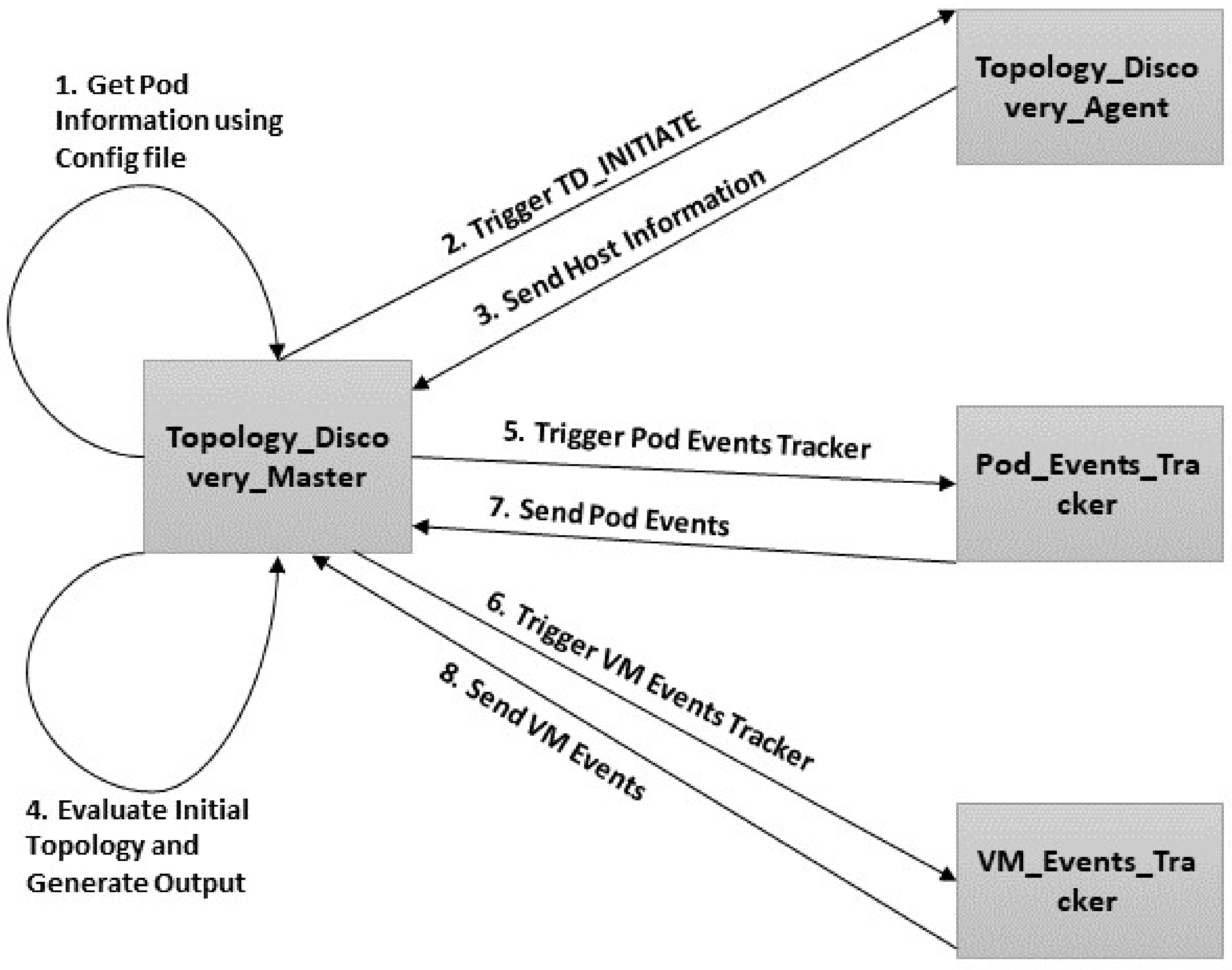
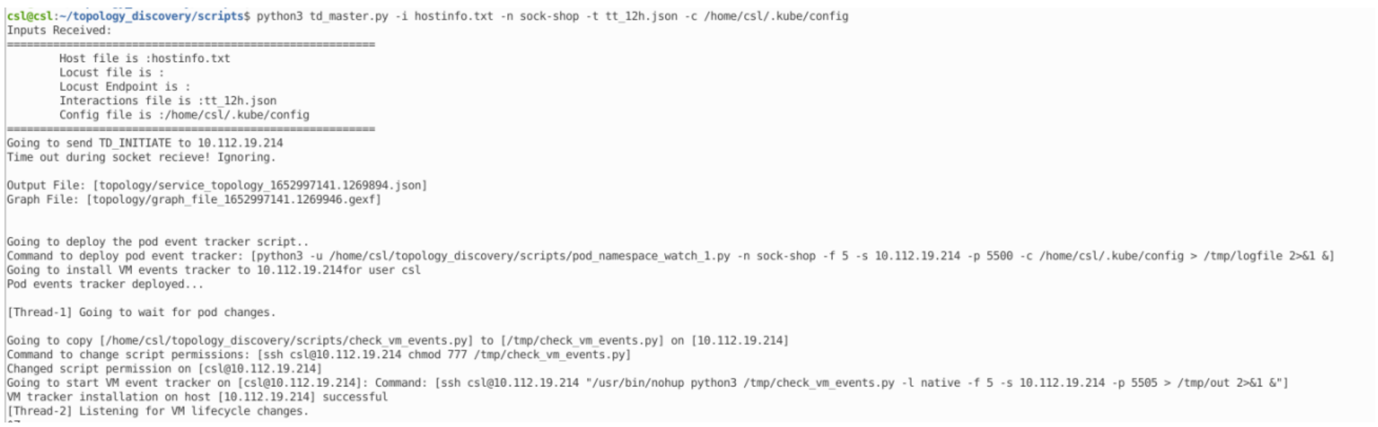
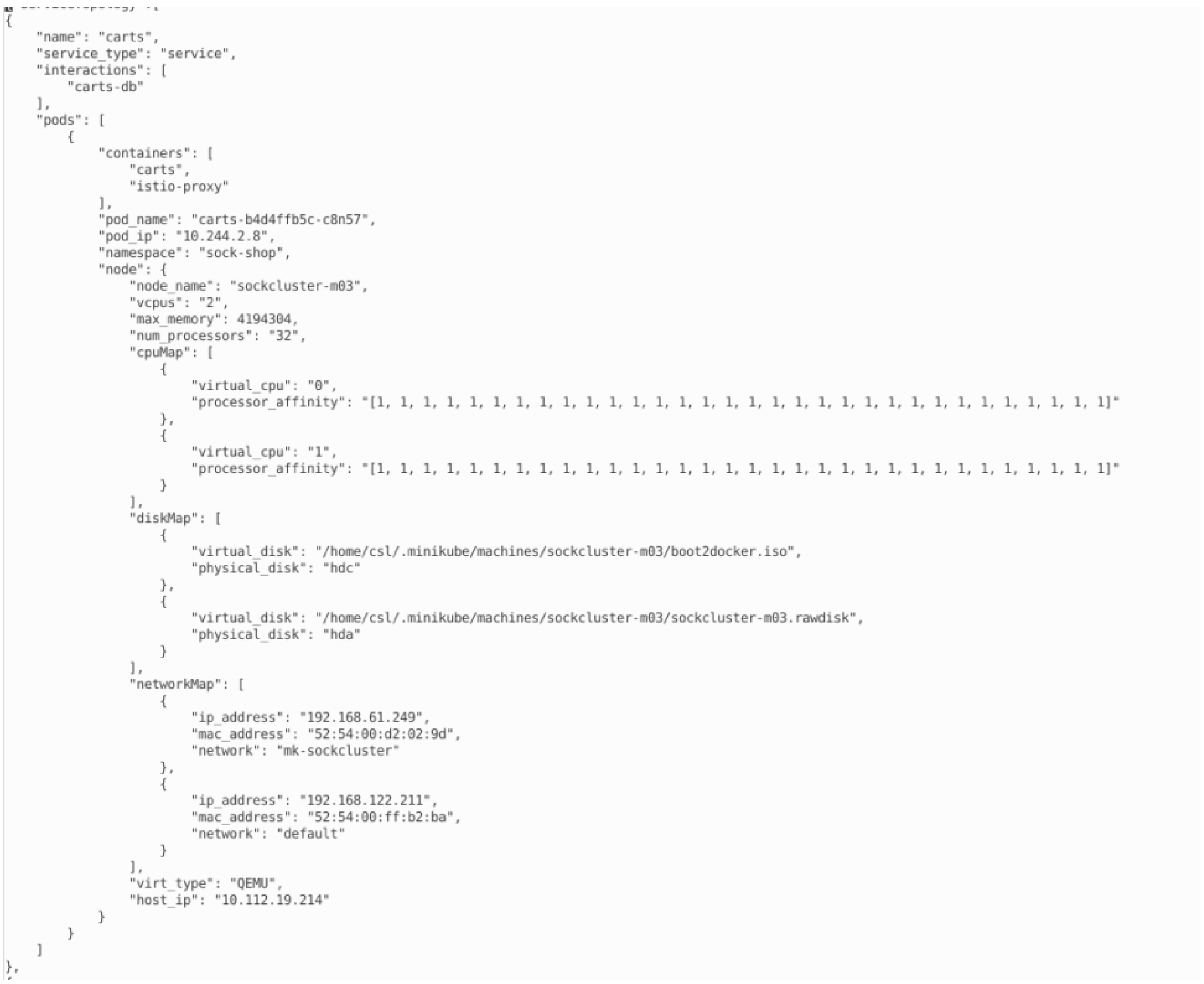

1. Get Pod Information using Config file
2. Trigger TD_INITIATE
3. Send Host Information
Topology_Discovery_Agent
Topology_Discovery_Master
5. Trigger Pod Events Tracker
7. Send Pod Events
Pod_Events_Tracker
6. Trigger VM Events Tracker
8. Send VM Events
VM_Events_Tracker
4. Evaluate Initial Topology and Generate Output

Fig. 2. Topology Discovery Tool Working



```
csl@csl:~/topology_discovery/scripts$ python3 td_master.py -i hostinfo.txt -n sock-shop -t tt_12h.json -c /home/csl/.kube/config
Inputs Received:
==========================================================
        Host file is :hostinfo.txt
        Locust file is :
        Locust Endpoint is :
        Interactions file is :tt_12h.json
        Config file is :/home/csl/.kube/config
==========================================================
Going to send TD_INITIATE to 10.112.19.214
Time out during socket recieve! Ignoring.

Output File: [topology/service_topology_1652997141.1269894.json]
Graph File: [topology/graph_file_1652997141.1269946.gexf]


Going to deploy the pod event tracker script..
Command to deploy pod event tracker: [python3 -u /home/csl/topology_discovery/scripts/pod_namespace_watch_1.py -n sock-shop -f 5 -s 10.112.19.214 -p 5500 -c /home/csl/.kube/config > /tmp/logfile 2>&1 &]
Going to install VM events tracker to 10.112.19.214for user csl
Pod events tracker deployed...

[Thread-1] Going to wait for pod changes.

Going to copy [/home/csl/topology_discovery/scripts/check_vm_events.py] to [/tmp/check_vm_events.py] on [10.112.19.214]
Command to change script permissions: [ssh csl@10.112.19.214 chmod 777 /tmp/check_vm_events.py]
Changed script permission on [csl@10.112.19.214]
Going to start VM event tracker on [csl@10.112.19.214]: Command: [ssh csl@10.112.19.214 "/usr/bin/nohup python3 /tmp/check_vm_events.py -l native -f 5 -s 10.112.19.214 -p 5505 > /tmp/out 2>&1 &"]
VM tracker installation on host [10.112.19.214] successful
[Thread-2] Listening for VM lifecycle changes.
```

Fig. 3. Topology Discovery Tool Sample Output



```
{
    "name": "carts",
    "service_type": "service",
    "interactions": [
        "carts-db"
    ],
    "pods": [
        {
            "containers": [
                "carts",
                "istio-proxy"
            ],
            "pod_name": "carts-b4d4ffb5c-c8n57",
            "pod_ip": "10.244.2.8",
            "namespace": "sock-shop",
            "node": {
                "node_name": "sockcluster-m03",
                "vcpus": "2",
                "max_memory": 4194304,
                "num_processors": "32",
                "cpuMap": [
                    {
                        "virtual_cpu": "0",
                        "processor_affinity": "[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]"
                    },
                    {
                        "virtual_cpu": "1",
                        "processor_affinity": "[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]"
                    }
                ],
                "diskMap": [
                    {
                        "virtual_disk": "/home/csl/.minikube/machines/sockcluster-m03/boot2docker.iso",
                        "physical_disk": "hdc"
                    },
                    {
                        "virtual_disk": "/home/csl/.minikube/machines/sockcluster-m03/sockcluster-m03.rawdisk",
                        "physical_disk": "hda"
                    }
                ],
                "networkMap": [
                    {
                        "ip_address": "192.168.61.249",
                        "mac_address": "52:54:00:d2:02:9d",
                        "network": "mk-sockcluster"
                    },
                    {
                        "ip_address": "192.168.122.211",
                        "mac_address": "52:54:00:ff:b2:ba",
                        "network": "default"
                    }
                ],
                "virt_type": "QEMU",
                "host_ip": "10.112.19.214"
            }
        }
    ]
},
```

Fig. 4. Sample object in the JSON output file

As described earlier, the JSON file output by the tool has services, their dependencies, and a few other metadata of interest. Fig. 4 shows the metadata associated with a service in the JSON file. The figure shows that the tool outputs the container, pod, host, and resource information for the c̈artss̈ervice in the SockShop microservice application. In addition to this information, the tool outputs the details of virtual compute, disk and network, and the associated physical resources. These details can facilitate the monitoring module to enhance the resilience of the cloud application service.

Fig. 5 shows the Gephi visualization of the .gexf file output by the tool for the Sock Shop application deployed on a 3-node Kubernetes cluster on virtual machines for experimentation. The different layers are denoted using a dashed rectangle in the figure. The nodes in the figure represent the components involved in the service delivery. The node labels indicate the name of the service components from the real deployment, and the solid lines recognize the interconnections (corresponds to bidirectional communications). The figure shows that the tool captures both the horizontal and vertical topology for a given application service. Due to space limitations, the results of capturing the dynamism in the setup and outcome for other microservices described above are not included.

Inorder to validate whether the tool captures the dynamism in the cluster, pod addition or deletion, pod autoscaling and VM lifecycle events were triggered. It was noted that the tool was able to capture the

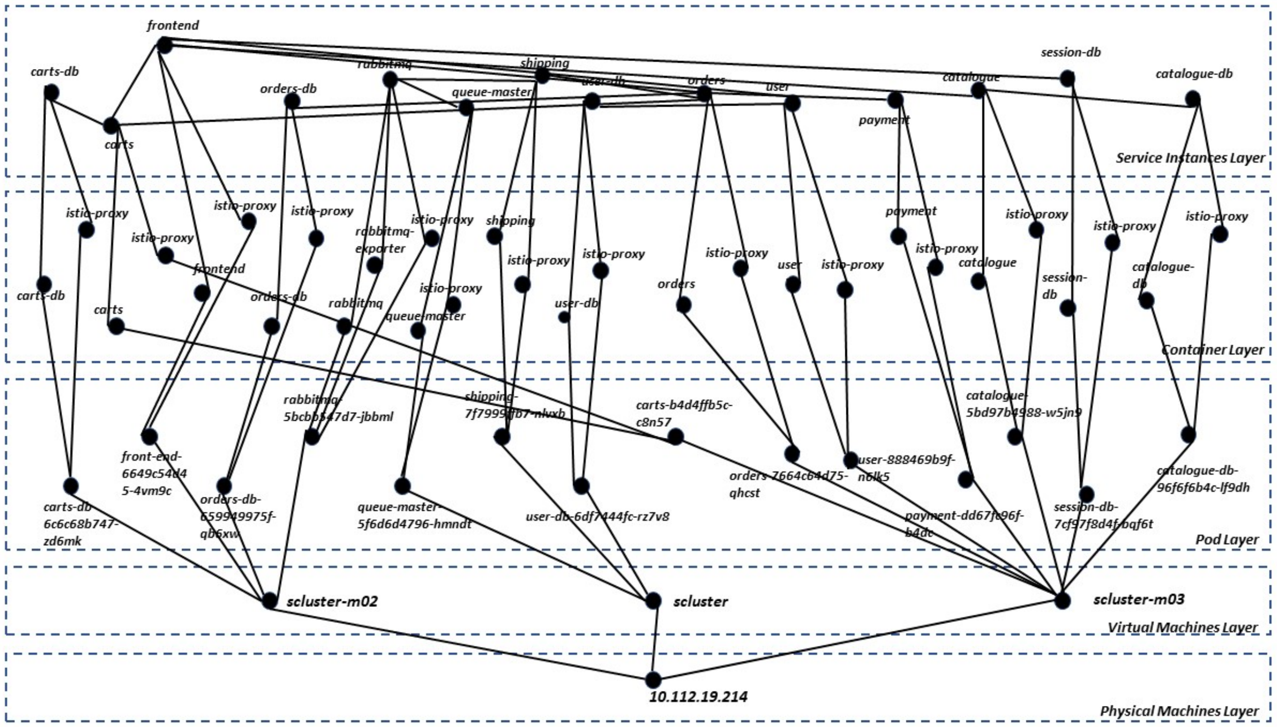


Fig. 5. Gephi visualization for topology discovery tool output for Sock Shop microservice application, deployed on 3-node Kubernetes cluster

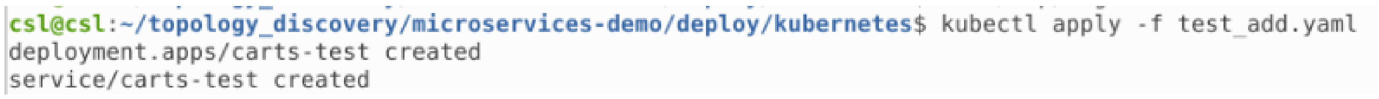
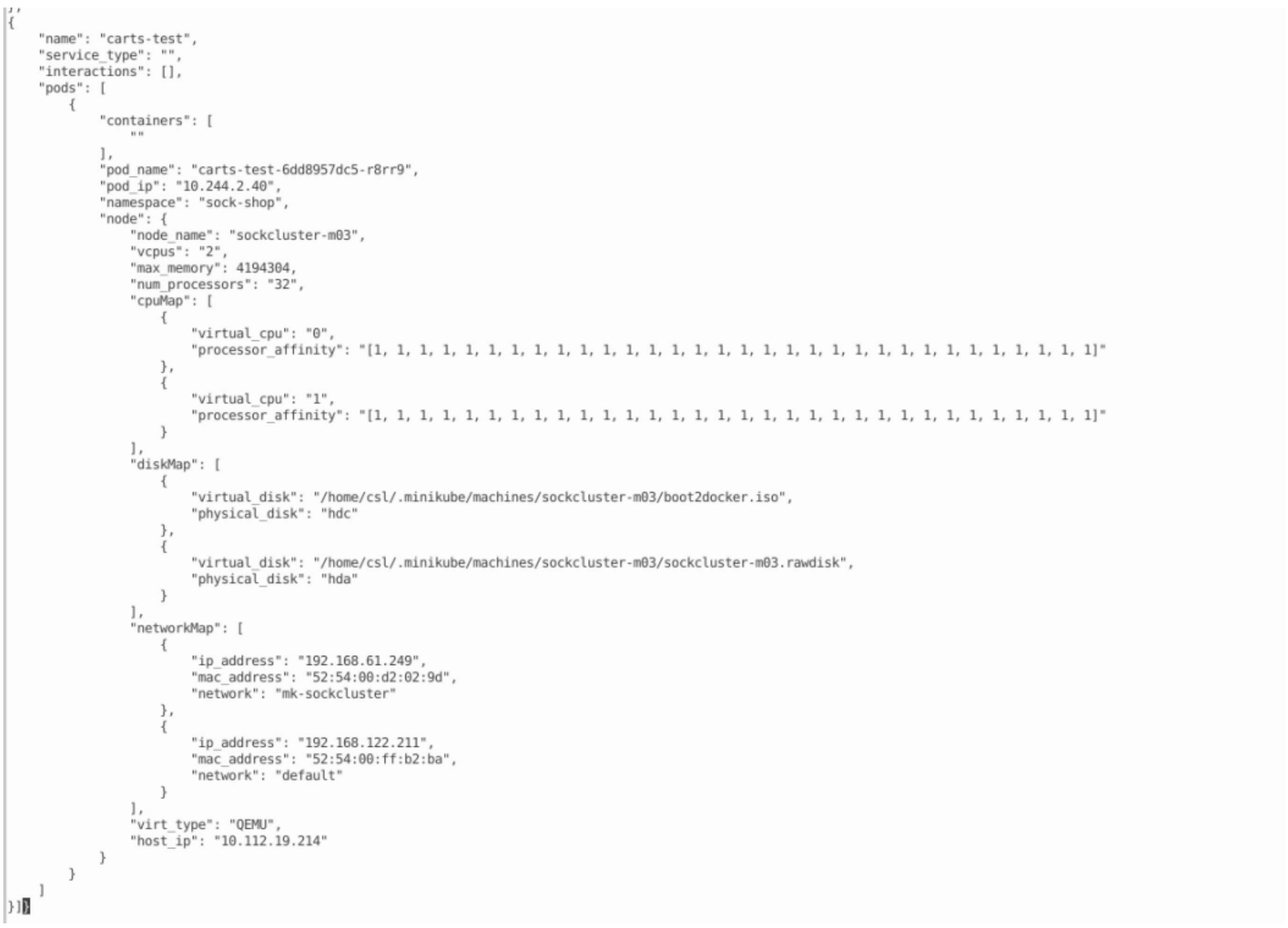

```
csl@csl:~/topology_discovery/microservices-demo/deploy/kubernetes$ kubectl apply -f test_add.yaml
deployment.apps/carts-test created
service/carts-test created
```

Fig. 6. Pod addition to validate the tool

changes. As an example, Fig. 6 shows how a new service pod was added. Once the pod was added, the Pod Events tracker, identified the event and communicated it with Topology Discovery Master. Fig. 7 shows the message on the master regarding the pod addition. Further, it was indeed verified from both the output files that this information is available.

IV. Effective Fault Management Using Cross-Layer Service Topology

A cross-layer service topology discoverer can help to make an end-to-end monitoring framework intelligent. Identifying service components across the stack can aid the monitoring module in selecting the observability data for the components which best describe an SLO of interest instead of looking at a whole spectrum of data across the components. As the tool also captures the dynamic changes in cross-layer topology, it allows the monitoring framework to work with the current state of the service deployment rather than stale data. The interconnections across the components in and across the different cloud service layers available in the cross-layer service topology information can facilitate determining the potential effect of a failure. For example, from Fig. 5, if a virtual machine fails, the topology awareness will facilitate identifying the components that will get affected and help decide on the remediation efforts before it leads to a user-perceived service disruption. As faults, errors or failures can propagate across the service components as given in [3], the causality for identifying the dependencies among the various observable data can be trusted only if they are justified through the derived cross-layer topology, thus providing more explainability. Further, the monitoring framework can use the interconnections to understand the correlation of failure across the components and analyze the scope for cascade failures or identify the blast radius. In the case



```
},
{
    "name": "carts-test",
    "service_type": "",
    "interactions": [],
    "pods": [
        {
            "containers": [
                ""
            ],
            "pod_name": "carts-test-6dd8957dc5-r8rr9",
            "pod_ip": "10.244.2.40",
            "namespace": "sock-shop",
            "node": {
                "node_name": "sockcluster-m03",
                "vcpus": "2",
                "max_memory": 4194304,
                "num_processors": "32",
                "cpuMap": [
                    {
                        "virtual_cpu": "0",
                        "processor_affinity": "[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]"
                    },
                    {
                        "virtual_cpu": "1",
                        "processor_affinity": "[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]"
                    }
                ],
                "diskMap": [
                    {
                        "virtual_disk": "/home/csl/.minikube/machines/sockcluster-m03/boot2docker.iso",
                        "physical_disk": "hdc"
                    },
                    {
                        "virtual_disk": "/home/csl/.minikube/machines/sockcluster-m03/sockcluster-m03.rawdisk",
                        "physical_disk": "hda"
                    }
                ],
                "networkMap": [
                    {
                        "ip_address": "192.168.61.249",
                        "mac_address": "52:54:00:d2:02:9d",
                        "network": "mk-sockcluster"
                    },
                    {
                        "ip_address": "192.168.122.211",
                        "mac_address": "52:54:00:ff:b2:ba",
                        "network": "default"
                    }
                ],
                "virt_type": "QEMU",
                "host_ip": "10.112.19.214"
            }
        }
    ]
}]}
```

Fig. 7. Event update reflected in the topology discovery JSON output

of fault localization, the connectedness through topology helps reduce search space further to identify the effect of the fault, derive better remediation methods, and reestablish disrupted services with a potentially limiting effect on SLO. Also, the topology information helps in choosing remediation methods that are better explainable and specific to a particular fault, rather than choosing a generic attitude for recovery from a failure. In case of recovery actions, the cross-layer topology awareness will help limit the downtime for the connected components. It will also provide accurate information on which components need to be re-initiated or which ones need to be notified. As the cross-layer topology information makes the monitoring module smarter, it helps mitigate fault by effectively taking proactive and reactive remediation methods. We believe that this will facilitate autonomously dealing with service disruptions more efficiently.

V. Conclusion and Future Work

This report identified the need to discover the cross-layer cloud service topology to deal with service disruptions effectively. The report also presented a tool to discover the cross-layer service topology. The tool is developed to generate the topology based on the initial deployment configuration and subsequently update it dynamically. The tool has been used to discover a few well-known microservice-based applications for validation. This tool is realized to enable the identification of service components across cloud service layers to develop an end-to-end monitoring framework. Future work will study the correlations between observability data generated by different service components across the cloud service layers and use those to infer the cause-and-effect relationships among the failures for developing effective models for end-to-end fault management.

References

[1] D. R. Mathews, M. Verma, P. Aggarwal, and J. Lakshmi, "Insights into multi-layered fault propagation and analysis in a cloud stack," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 714–716.

[2] D. R. Mathews and J. Lakshmi, "Service resilience framework for enhanced end-to-end service quality," in *Proceedings of the 18th Workshop on Adaptive and Reflexive Middleware*, ser. ARM '19. Association for Computing Machinery, 2019, p. 7–12.

[3] D. R. Mathews, M. Verma, P. Aggarwal, and J. Lakshmi, "Towards failure correlation for improved cloud application service resilience," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21. Association for Computing Machinery, 2021.

[4] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. Association for Computing Machinery, 2019, p. 3–18.

[5] G. Muntoni, J. Soldani, and A. Brogi, "Mining the architecture of microservice-based applications from their kubernetes deployment," in *Advances in Service-Oriented and Cloud Computing*, C. Zirpins, I. Paraskakis, V. Andrikopoulos, N. Kratzke, C. Pahl, N. El Ioini, A. S. Andreou, G. Feuerlicht, W. Lamersdorf, G. Ortiz, W.-J. Van den Heuvel, J. Soldani, M. Villari, G. Casale, and P. Plebani, Eds. Springer International Publishing, 2021, pp. 103–115.

[6] J. Hwang, G. Liu, S. Zeng, F. Y. Wu, and T. Wood, "Topology discovery and service classification for distributed-aware clouds," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 385–390.

[7] H. Wei, J. S. Rodriguez, and O. N.-T. Garcia, "Deployment management and topology discovery of microservice applications in the multicloud environment," in *Journal of Grid Computing*, 2021.

[8] A. Sangpetch and H. S. Kim, "Vdep: Vm dependency discovery in multi-tier cloud applications," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 694–701.

[9] J. Gutierrez-Aguado, J. M. Alcaraz Calero, and W. Diaz Villanueva, "Iaasmon: Monitoring architecture for public cloud computing data centers," in *Journal of Grid Computing*, 2016.

[10] "Smartscape," https://www.dynatrace.com/support/help/how-to-use-dynatrace/smartscape, (Accessed on 31/03/2022).

[11] "Network topology," https://cloud.google.com/network-intelligence-center/docs/network-topology/concepts/overview, (Accessed on 31/03/2022).

[12] "Instana," https://www.instana.com/, (Accessed on 31/03/2022).

[13] M. Novakovic, "Introducing the logical view," https://www.instana.com/blog/introducing-logical-view/, (Accessed on 31/03/2022).

[14] "Sock shop load test," https://github.com/microservices-demo/load-test, (Accessed on 31/03/2022).

[15] "Sock shop microservice," https://github.com/microservices-demo/microservices-demo, (Accessed on 31/03/2022).

[16] "Train ticket microservice," https://github.com/FudanSELab/train-ticket, (Accessed on 31/03/2022).