Communication-Affinity Aware Colocation and Merging of Containers

Nishadi N. Wickramanayaka, Chamath I. Keppitiyagama, Kenneth Thilakarathna

Abstract— **Microservice architecture relies on message passing between services. Inter-service communication introduces an overhead to the applications' overall performance. This overhead depends on the runtime placement of the services bundled in containers and it can be reduced by intelligently deploying the containers by considering the communication affinities between services. Researchers have attempted to colocate microservices and merge containers based on affinities. However, container merging has not been considered up to now. This study shows that the problem of service placement in a microservice application considering communication affinities, constrained by computational resources, can be mapped to an instance of the Binary Knapsack Problem. We propose a container colocation and merging mechanism based on a heuristic solution to the Binary Knapsack Problem. The proposed approach reduced the communication overhead of a benchmark application by 58.5% and as a result, execution time was reduced by approximately 13.4% as well.**

Keywords— **Affinity, Binary Knapsack, Colocation, Container Networks, Communication Overhead, Docker Containers, Microservices, Microservice Architecture, Inter-service Communication**

I. Introduction

Microservice architecture is used to develop software applications as suites of independently deployable small services that interact with each other [1]. It is often used to decompose an existing system rather than to compose a system anew using services offered by different enterprises. Microservices are typically deployed in containers with each service contained in a dedicated container [2] which are then hosted in multiple hosts. Hence microservices of an application exchange a significant amount of data, creating communication affinities. Affinity is defined as a relation between two microservices [6] which in this study given by the total amount of data exchanged between those two services over time. To place services in different containers, function calls in the monolithic application should be converted to network calls between the containers in the microservice architecture. Those network calls add an extra layer of networking with expensive operations such as packet encapsulation, decapsulation, address translations [3], which ultimately increase the services' request/response time [2], [4].

Correspondence: Nishadi N. Wickramanayaka [#1] (E-mail: *nishadinuwa1995@gmail.com*) Received: 24-12-2021 Revised:26-08-2022 Accepted: 30-08-2022

Nishadi N. Wickramanayaka, Chamath I. Keppitiyagama and Kenneth Thilakarathna are from University of Colombo School of Computing, Sri Lanka. (e-mail: *nishadinuwa1995@gmail.com, chamath@ucsc.cmb.ac.lk, kmt@ucsc.cmb.ac.lk*).

DOI: http://doi.org/10.4038/icter.v15i3.7251

Therefore, the resulting communication overhead adversely impacts the overall performance of the microservice application (μApp) despite the benefits of the architecture.

Hence what this paper focuses as the main research problem is that the degraded performance of μApps due to the communications between the services of the application. Container networks used to connect containers with each other play a major role in communication complexities of microservice architecture. Overlay networks are used for host-to-host communication when containers/services are deployed in different hosts whereas bridge networks are used when containers/services are deployed within the same host. Processes inside the same container communicate over the loopback interface. Overhead imposed by an overlay network is higher than the overhead imposed by a bridge network [3]. Loopback interface imposes the least overhead since it eliminates the intervention of bridge network as well. Therefore, placement of two services with high communication affinities in different physical nodes makes this situation worse [6]. Thus, it is evident that the container placement decisions of the μApps need to be taken carefully in the deployment time.

As long as microservice architecture is used to design an application it is impossible to completely eliminate the communication overhead incurred at runtime. Because even if all the services are located inside a single machine there will still be communication across address spaces. A possible way of addressing the problem is by reducing the communication overhead to a certain extent by making the deployment decisions carefully. Further, if mapping of the exact same design decisions to runtime is not strictly necessary, then alterations may be applicable to further reduce the communication overhead. The motivation behind this study is to increase the performance of a μApp, by reducing the overhead of inter-service communication by carefully analysing the runtime behaviour of a μApp and containers. Therefore, our main objectives were to explore the impact of container networks on μApps, to discover the possibilities of reducing the communication overhead of a μApp without changing the design of the application and finally to measure up to what extent the performance can be increased from the proposed solution.

In order to achieve these objectives, we present a novel mechanism of container colocation and container merging. Container colocation is defined as moving the services with high communication affinities into a single host. Colocation is constrained by the resources available on the hosts. We could map this problem to an instance of the Binary Knapsack Problem (BKP). Container merging is the process of executing services that are already colocated, in a single container. Through the colocation process the overlay network is reduced to a bridge network. Merging process reduces the bridge network to communications over the

This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits nrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited

container's loopback interface. As a result, communication overhead is reduced, and the performance of the application is improved. Also, this might reduce the number of hosts needed to execute the µApp.

The rest of this paper is organized as follows. Section II presents a review of the background related to the study. Section III presents the design and implementation of the solution. Results and the evaluation of the proposed approach are summarized in Section IV. Finally, Section V concludes and outlines some future directions.

II. BACKGROUND AND RELATED WORK

A. *Performance Degradation of µApps and Container Networks*

With the inclination of the industry towards cloud-based infrastructure, microservice architecture has received massive attention from the academic community. Hence ample amount of studies which compare the µApps with monolithic applications show the performance penalty resulted when using µApps [2], [4], [8]. The performance of microservices in container-based and virtual machine (VM) based environments has also been studied by Salah et al. [9]. Amaral et al. [10] evaluated the performance impact of two models of implementing microservices in a container environment: master-slave and nested-containers. They mention that nested-container model is hardly adopted in real-world applications since there are some trade-offs in terms of network performance.

Suo et al. [3] have done a thorough investigation on latency, throughput, scalability, and startup cost of various container networks on a single host and on multiple hosts in a virtualized environment. Out of four networking modes on a single host (none mode, bridge mode, container mode, host mode), bridge mode network incurred 18% and 30% throughput loss in upload and download respectively in facilitating each container to own an isolated network namespace, resulting all inter-container communications to go through the docker0 bridge. Out of four networking modes available on multiple hosts environments (host mode, NAT, Overlay network, Routing), both NAT and Routing incurred considerable performance degradation due to the overhead of address translation and packet routing. However, the overlay network caused a high-performance loss of 82.8% throughput drop and a 55% latency increase compared to the host mode. They explain the reason for the performance degradation of µApps in terms of bridge network and overlay network which are used to connect the containers. This study has done a comparison between several container networks in a single host and multiple hosts separately. They have not compared the single host container networks with multi-host container networks. Further, they have conducted the experiment in VMs where an additional network overhead is introduced to containers through the virtualization.

Yang et al. [11] have attempted to bridge the above gap by deploying the containers on both VM and bare-metal environments. The results confirm the overhead of VM environments with a throughput loss compared to bare-metal deployment. In all tests, the multi-hosts control group showed a significant throughput loss compared to the single host control group. Further, Kratzke [12] has analysed the performance impact of the overlay network in terms of encryption to HTTP-based and REST-like services. Even though these analyses show the impact of container networks on imposing communication overhead, they have only considered inter-container communications and none of these studies have considered intra-container communications. Hence, it is identified that intra-container communications should also take into consideration in order to further explore the ways of reducing the communication overhead.

B. *Container Placement Problem*

Based on the aforementioned studies, placement of the containers has been identified as one of the major reasons in creating these communication affinities. Hence, it is pertinent to explore the state of the art of container placement process in practice. However, identifying the best placement of containers is not an easy task. Existing container management tools implement several common placement strategies. Kubernetes [13] places a minimum number of microservices per host in the cluster [6]. This is called the Spread strategy. However, it can add latency to communication and lower the µApp's performance. Also, this does not take resource optimization into consideration during the deployment. Some management tools use the Bin-pack strategy: deploying a µApp in a minimum number of hosts so that it avoids the cluster resource wastage. Besides these commonly used two strategies, the Random strategy is also used where the management tool selects a host to deploy a microservice randomly. All these three strategies are available in Docker Swarm [14]. Irrespective of the strategy, management tools only consider the instantaneous resource usage of the service when they place them on hosts and rarely try to find an optimal setting. However, they do not consider the communication affinities between services resulting in placing microservices with high communication affinities in different hosts. Eventually, the large amount of network traffic that takes place between two services over the network can hinder the overall performance of the application.

Sampaio et al. [6] propose REMaP, a runtime microservices placement mechanism. They consider microservices' resource usage and as well as their affinities when placing the microservices in hosts. This problem is modelled as an instance of the multi-dimensional bin-packing problem. The objective of REMaP is to maximize the affinity score while deploying the microservices in a minimum number of hosts during the runtime. In solving the problem, they have used the First Fit as a heuristic in their approach since a runtime placement needs quick solutions. REMaP instruments the microservices to gather information required to take colocation decisions during the runtime. Though, we noticed the heavy cost of this instrumentation on the microservices. Hence, the benefits derived through colocation are negatively affected due to the instrumentation cost. Further, REMaP cannot handle data synchronization across different hosts after migrating a stateful microservice. Hence, the migration of stateful microservices may lead the µApp in an inconsistent state. In addition, runtime migration cost may not be justifiable compared to the benefits derived due to colocation. REMaP does not use the hints available in the configuration files about the resource usage or the relationships between microservices indicated in them. We further noticed that REMaP does not consider container merging at all.

Han et al. [15] propose a refinement framework for profiling-based microservices placement to identify and respond to workload characteristics. The resource requirements obtained through profiling has been fed into a greedy-based heuristic algorithm to make microservices placement. However, the main focus of this work is not a placement algorithm but a profiling-based framework for microservices deployment. Hence any placement algorithm can be adapted to their framework. Both aforementioned solutions depend on the data collected at run time. However, we have identified that some essential parameters that are required to take the placement decision are already available in configuration files even before the runtime of the µApp.

C. *VM Placement Problem*

Once the containers are mapped to VMs and communications between containers to the communications between VMs, virtual machine placement in physical machines (PMs) can be considered as the closest research area to the container placement problem. Tziritas et al. [7] propose a communication-aware graph-coloring algorithm, placing the VMs in the underlying system in an energy-efficient manner while optimizing the network overhead due to the VM communication inter-dependencies. However, that VM selection process cannot be directly mapped into the container selection process as their study pre-defines the number of servers to place VMs and container placement may not necessarily give the number of host machines to place the containers as the problem is to optimize the placement of services, thus the algorithm itself should be able to identify the minimum number of hosts to locate the services. However, it is possible to map their VM communication graph to a container/service communication graph.

Chen et al. [5] propose a different approach for the VM placement problem which is an affinity-aware grouping method for allocation of VMs into PMs based on a heuristic bin-packing algorithm. It groups the VMs based on the affinities among them and then allocates those identified VM groups into a minimum number of PMs using the bin packing strategy. One major limitation of this research is the generation of one large VM affinity group with total resource requests overstepping the PM resource limit. Sonnek et al. [16] present a decentralized affinity-aware migration technique that incorporates heterogeneity and VM communication cost to allocate VMs on the available physical resources. Their technique monitors network affinity between pairs of VMs periodically and triggers the migration if inter-server traffic exceeds intra-server traffic and uses a distributed bartering algorithm, to dynamically adjust VM placement such that communication overhead is minimized. Since the migration also has a cost, they refrain from migrating VMs if it results in only minor benefits.

III. METHODOLOGY

The proposed solution comprises of two phases:

- Container colocation: moving containers that are initially deployed in different hosts into a single host.
- Container merging: placing the services that are initially deployed in different containers inside a single container.

Containers deployed in multiple hosts are connected through overlay networks and containers in the same host communicate through a bridge network. As mentioned before, overlay networks impose a higher overhead than bridge networks [3], [11], [12]. Hence during the colocation phase services with high communication affinities are identified to deploy on a single host. This is not a trivial task since the colocation is constrained by the processing resources available on the host. In this study, we propose a novel approach to solve the colocation problem by mapping it to an instance of the BKP.

Current approaches to reduce the communication cost use only the container colocation process [6] which replaces the overlay network with bridge network. It is evident that the elimination of this bridge network should further reduce the overhead in inter-service communication. This study introduces a novel concept of merging the colocated containers to further reduce the communication overhead by eliminating the bridge network as well. Once two containers are merged, services deployed on them would execute on a single container as two processes. These two processes communicate over the container's loopback interface. The merging process converts the inter-container communications into intra-container communications.

Spread strategy deployment of Docker Swarm is considered as the baseline to this study. This deployment strategy distributes services evenly among the hosts, resulting in a minimum number of services per host. Thus, we consider the deployment of service instance per host as the baseline since there are not any optimizations present in that strategy. Change of the number of containers and hosts throughout the whole process of colocation and merging can be shown as in Table I.

TABLE I
CHANGE OF THE NUMBER OF CONTAINERS AND HOSTS THROUGH THE PROCESS

Initial deployment	**After colocation**	**After merging**
n services	n services	n services
n containers	n containers	m containers (m<n)
N hosts	M hosts (M<N)	M hosts (M<N)

A. *Container Colocation*

The purpose of the colocation phase is to identify the high-affinity containers and colocate them in a single host machine in order to change the overlay network to a bridge network to reduce the communication overhead. The amount of data exchanged between services, from the point of view of the application, does not change when the network type is changed. But the actual data volume exchanged between the services over the network, as seen from the network, contains an additional overhead and it depends on the type of the container network. Hence, in this study, affinity is referred to the actual data volume exchanged between the services. Let *o*, *b*, and *l*, denote overlay, bridge, and loopback networks respectively. If K_N represents the overhead on communications by a specific container network type *N*, below inequality can be obtained from the facts that discussed in section I and II.

$$K_l < K_b < K_o \tag{1}$$

Hence the overhead added in inter-service communication is a variable which can be changed depending on the container placement. Thus, in this study, for a particular deployment, affinity is measured by the total traffic volume exchanged between the services, including the network encapsulation overheads, during the application execution time. Therefore the communication dependency between the two services, *i* and *j* is defined as the communication affinity of services *i* and *j* ($A_{i,j}$) and it is measured as the total data volume ($volume_{i,j}$) exchanged between the two services.

$$A_{i,j} = volume_{i,j} \tag{2}$$

Note that $A_{i,j}$ consists of two components: application level data volume and the network encapsulation overheads. Application-level data volume does not change if the network type is changed. However, the encapsulation overhead depends on the container network type. Therefore, affinity reduction is possible by changing the network type from overlay to bridge and to loopback networks. This notion of affinity allows it to be estimated without instrumenting microservices. Volume of data exchanged can be passively observed by placing monitoring tools in the network. A host machine has its own limitations with respect to resources which limits the number of containers that can be colocated. In this study the researchers consider only the CPU usage as a constraint that limits the colocation. This colocation problem can be formally stated as below:

Given a set of microservices as p_1, p_2, ..., p_n and maximum CPU usage of each service as $cpu(p_1)$, $cpu(p_2)$, ..., $cpu(p_n)$ hosted in H_1, H_2, ..., H_n host machines according to Spread strategy (one service per host), where p_i is linked to p_j with the communication affinity $A_{i,j}$:

Find a set of microservices P_K; $(K = 1, ..., v)$ to deploy in host H_l, such that A_{P_K} is maximum and $cpu(P_K) \leq cpu(H_l)$ where A_{P_K} and $cpu(P_K)$ represents the total communication affinity and the total CPU usage of the set of P_K microservices respectively. Hence ultimately find an integer number of hosts H_m such that $m \leq n$ to deploy all the given p_1, p_2, ..., p_n microservices. A solution is optimal if it has the minimal m and maximum affinity score (A_{P_K}) for all P_K s.

1) Service Communication Graph: In order to solve the above formally stated problem, the service communication graph should be generated. REMaP [6] creates such a graph entirely based on the data collected at runtime by instrumenting the services. It is noted that the instrumentation overhead is significant. Therefore, the approach presented in this paper relies on passive observation of the network to collect the runtime data and the static information available in the configuration files. Static links between the services and the maximum CPU usage of each service can be extracted from the configuration files of the application. Since Docker [17] is used as the container runtime, the configuration file is a yaml file defining services, networks, and volumes for an application. Services that are defined to be in the same network can reach each other. Hence the "networks" tag which defines the networks for each service is used to extract the connections between services. This study only considers CPU usage as a constraint that limits the colocation of containers. It is noted that the runtime information about the CPU usage is heavily influenced by the other processes running on the host. Application developers are in a better position to estimate and provide hints on the CPU usage. It is possible to set various constraints to limit a given container's access to the host machine's CPU cycles by setting the "--cpus" tag in the docker compose file. Hence docker-compose file is parsed to extract this information of each service.

To measure the affinity between services ($A_{i,j}$), it is needed to monitor the application for a given time period and the total amount of data (bytes) transferred between container pairs, considered as the traffic $volume_{i,j}$ between the connected pair of services*(i,j)*. Tcpdump packet analyzer tool [18] is used to passively record the network traffic transferred between containers without instrumenting the services. Therefore, this measure includes the network overheads as well. During this monitoring time, it is assumed that anomalies have not occurred and the general behaviour of the µApp is captured. From the gathered data, a communication graph (Fig. 1) is constructed where each node represents a service. The aggregate CPU usage of two adjacent vertices/services is the weight of an edge ($max_{cpu(A)}$+$max_{cpu(B)}$ = W_{AB}) and the communication affinity is the value of an edge ($A_{A,B}$= V_{AB}).

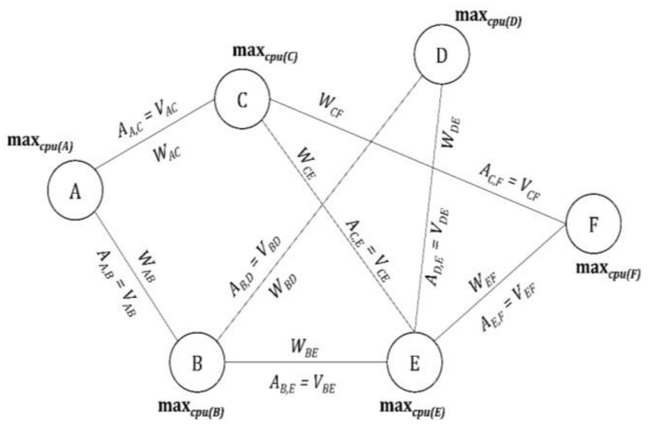

Fig. 1. Example communication graph

2) A Knapsack Problem Based Heuristic: Given the communication graph, the colocation problem can be mapped to a Binary Knapsack Problem as follows:

The set of edges are considered as objects and the communication affinities are considered as the object values. The total CPU consumption of the two services/nodes that it connects with represents the object weight. The host machine is the knapsack, and the CPU capacity is the weight limit of the knapsack. Place the objects in the knapsack to maximize the total value under the constraint that the total weight should not exceed the weight limit of the knapsack. A solution to this problem gives the set of edges with maximum communication affinity score such that the total CPU usage of the adjacent services to the selected edges do not exceed the CPU limit of the host machine. It is assumed that the CPU usage of each service is lesser than the CPU capacity of the host machine. By continuously applying this until all the services are allocated to a host machine, the process ends up grouping the services with high affinities together to be colocated in the given host. Since selecting fractions of objects (edges) is not possible, further this can be specified as a 0/1 or BKP. Hence the problem is mapped to an NP-Hard problem. Also, since this process packs as much as services into a single knapsack(host), ultimately it may return the minimum number of hosts needed to deploy all the services. After applying this process, the resulting communication graph would appear as in Fig. 2 where each colour represents a host. In this example, the number of hosts needed to deploy the six services is reduced to three.

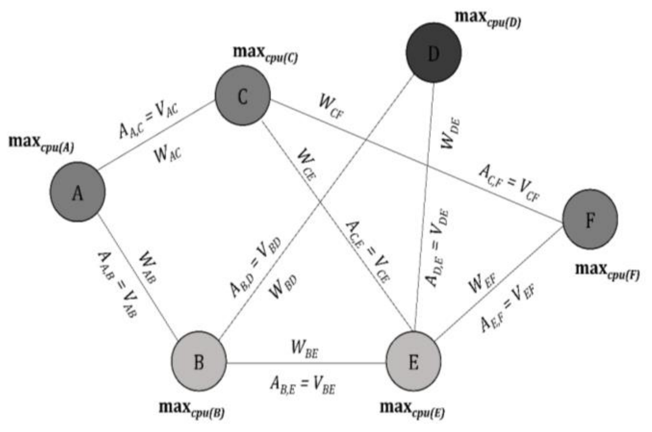

Fig. 2. A derived communication graph

Theoretical benefit or affinity reduction gained from a group of colocated services would be equal to the summation of the edge values of the colocated services. Note that in this study affinity is considered as the total volume of data exchanged and that includes the network type specific overhead as well. Considering the group of ACF in Fig. 2, theoretical affinity reduction is V_{AC}+V_{CF}. But the practical benefit would be less than the theoretical benefit since single host container networks (bridge) also add an overhead, but less than the overhead incurred by multi-host container networks (overlay). Therefore, an affinity reduction is guaranteed through this process.

To solve the knapsack problem a dynamic programming-based heuristic is used [23]. The time complexity of that approach is O(W*N) where W is the maximum weight the knapsack can carry which is mapped to the CPU limit of the host machine and N is the number of items which is the number of edges in this study. At the end of each iteration, this algorithm selects a better set of edges from the given edges (objects). Once an edge is selected that means the services that it connects with are eligible to deploy in that particular host. Hence in each iteration, it gives a better set of services which mostly reduces the communication overhead of the application to deploy in the host selected for that particular iteration. Selected edges in each iteration are removed from the initial set of edges. A selected edge carries two adjacent services which are to be deployed in the host selected for that particular iteration. All the other adjacent edges to those services which are selected to be placed in a host are also removed as considering those edges would have no point because the service is already allocated to a host. The iterations end when no edges are left to consider. As a result, there can be situations where services are left without being allocated to any host when all the adjacent edges to that particular service are removed (F and H services in Fig. 3).

However, such services do not directly communicate with each other. Hence deploying them in the same host or

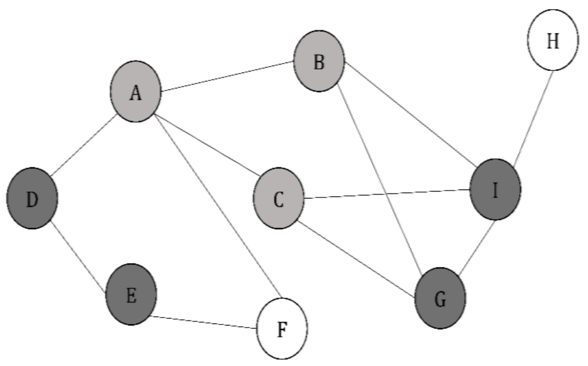

Fig. 3. Example graph to represent the services left without allocating to any host

separate hosts does not affect the overhead reduction. But considering one of the objectives of the study which is finding the minimum number of hosts to deploy the application, the Bin Packing algorithm is applied only to those remaining services that have not placed in any hosts. Bin is the host and items are the services left without allocating to any host. This is a one-dimensional bin packing problem since the only parameter it considers is the CPU usage of the service as the weight of the item. It is assumed that the host's CPU capacity is greater than each service's maximum CPU usage.

The best-fit offline algorithm is used to solve the bin packing problem since the items are available upfront. The time complexity is O(N log N) where N is the number of services input. It returns the minimum number of bins/hosts required to deploy the remaining services. Note that the complexity of the bin packing algorithm is not counted into the complexity of the main objective of this study since the bin packing algorithm only assists the objective of finding the minimum number of hosts needed to deploy the μApp. The final output from the container colocation is the final service graph that represents all the decisions made from the above algorithms by using colour coded nodes where each colour represents a host.

B. Container Merging

Colocation reduces the overlay network to a bridge network. Communication overhead can be further reduced by merging containers to eliminate the bridge network as well. Merging two containers is in the sense of having two services inside the same container which were previously executed in two separate containers. Note that after merging, service granularity remains unchanged. Hence merging preserves the design time service separation of the application. Merging only changes the communication mode of the colocated services from bridge to loopback interface.

Container merging eliminates the namespace isolation of the services that the containerization provides [24]. Therefore, containers are merged only if there is no requirement to execute the services in separate namespaces. Dependency conflicts between the services is another constraint on the container merging process. Having different versions of the same dependency is considered as a dependency conflict. Hence, before the container merging process, dependencies of the services to be merged are examined to ensure that there are no restrictions to execute them in a single container. A dependency conflict checker takes the dependency files in XML format as the input and after parsing them, it outputs the capability of merging.

To execute two services inside a single container it is essential to have one Dockerfile for both services. A script is used to combine the content of two Dockerfiles into a single Dockerfile. It is not possible to create a merged container from different base images. Therefore, both containers to be merged should extend from the same base image. Instead of directly executing the service from the ENTRYPOINT in Dockerfile, it is set to execute a shell script and that script starts the two services. Hence ultimately those services would execute in the same container as two processes.

IV. Results and Evaluation

This section elaborates the obtained results, how the results are evaluated, and the success level of the proposed solution.

A. Benchmark Application Selection

Sock-Shop μApp provided by Weaveworks [19] is selected as the benchmark application to evaluate the proposed approach. This selection is based on a study by Aderaldo et al. [20]. Further many of the research in this area [21], [22] have used this as the benchmark for their evaluations. Microservices of this application are written in different languages and dependency files of these services are in different formats. A graph processing application which is used as a page rank analytics platform to rank the Twitter profiles is also selected for evaluation again based on the recommendations by Aderaldo et al. [20]. All the services in this application are implemented in Java programming language except the databases and servers. Hence all the dependency files are in XML format. Therefore, the colocation process is evaluated for both these applications and the merging process is evaluated only using the Page-Rank application.

B. Evaluation of the Proposed Approach

The selected benchmark applications were evaluated under the following deployment strategies.

- Spreaded Deployment: Initial deployment which is the baseline.
- Colocated Deployment: Deployment after colocation.
- Merged Deployment: Deployment after merging.

Encapsulation overhead is a component of the communication affinity. Therefore, affinity depends on the type of the network. Due to different encapsulations of data packets exchanged in the above three deployment strategies, inter-service communication affinities of the application comprise several components (Table II).

Table II.

Composition of Communication Affinities under Different Deployment Strategies

Spreaded deployment	**Colocated deployment**	**Merged deployment**
Inter-host communication affinity	Inter-host + Inter-container communication affinity	Inter-host + Inter-container + Intra-container communication affinity

1) Colocation Process Evaluation:

Service communication graphs generated by analysing the docker-compose files of the Sock-Shop and Page-Rank applications are in Fig. 4 and Fig. 5 respectively. The number above each node is the maximum CPU cores that service can take.

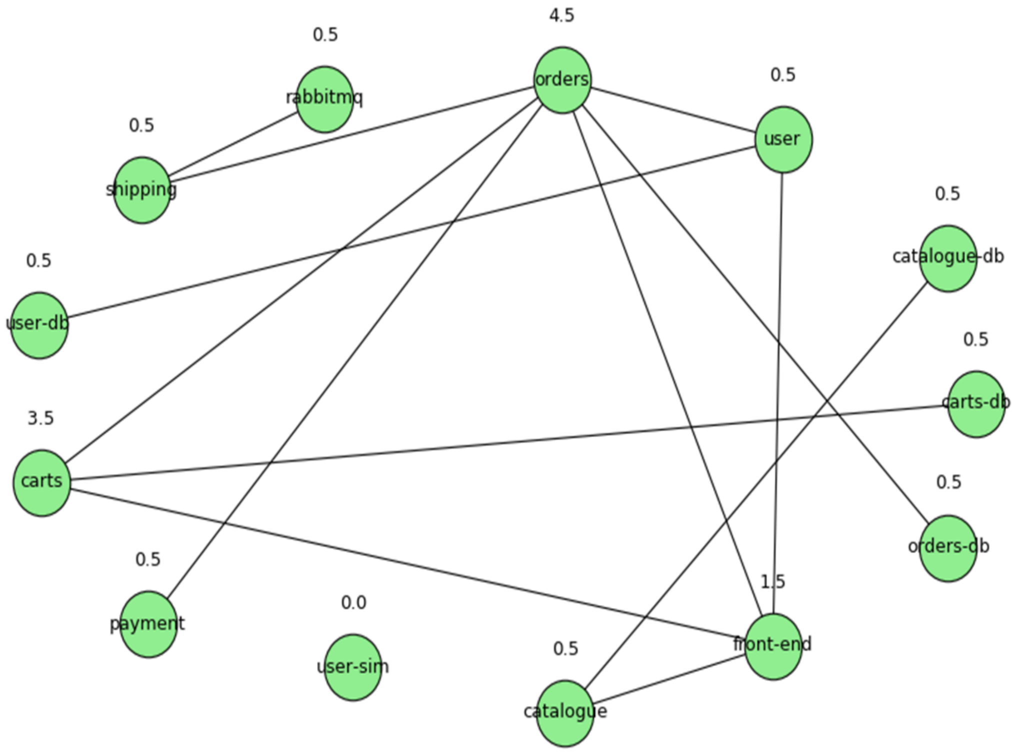

Fig. 4. Initial service communication graph for Sock-Shop μApp

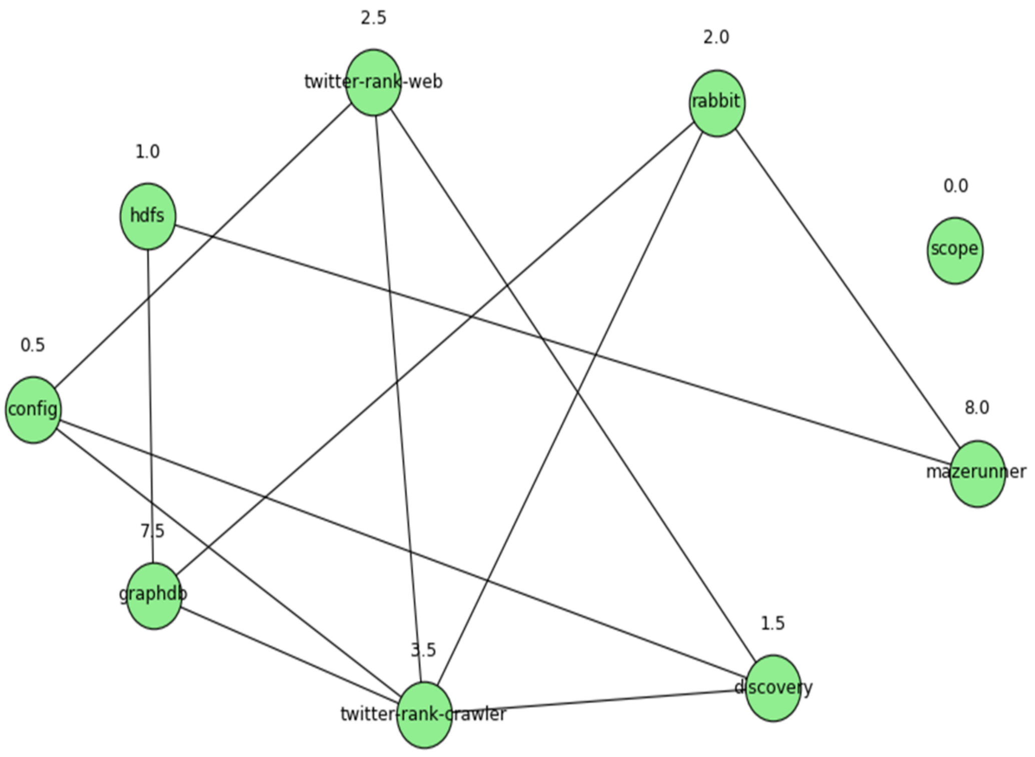

Fig. 5. Initial service communication graph for Page-Rank µApp

Since baseline is the spread strategy deployment, initial monitoring was done deploying each service/container in a separate VM. Each communication link of this configuration was monitored using tcpdump to extract the affinity values to annotate the communication graph. Fig. 6 and Fig 7 depicts the communication graphs of the Sock-Shop and Page-Rank applications respectively after this annotation. Each edge contains the total maximum CPU usage of adjacent services and the communication affinity between them. Hence this can be considered as the input to the knapsack method.

Colocation decisions in Table III were made for the Sock-Shop application for host machines with CPU cores 4,5,5, and 4. Table IV presents the colocation decisions for the Page-Rank application deployed over host machines with CPU cores 3,6,9, and 10. Fig. 8 and Fig. 9 are the final service graphs generated to depict the colocation decisions where each colour represents a host. From Fig. 4 to Fig. 9 are the direct outputs of the implemented program.

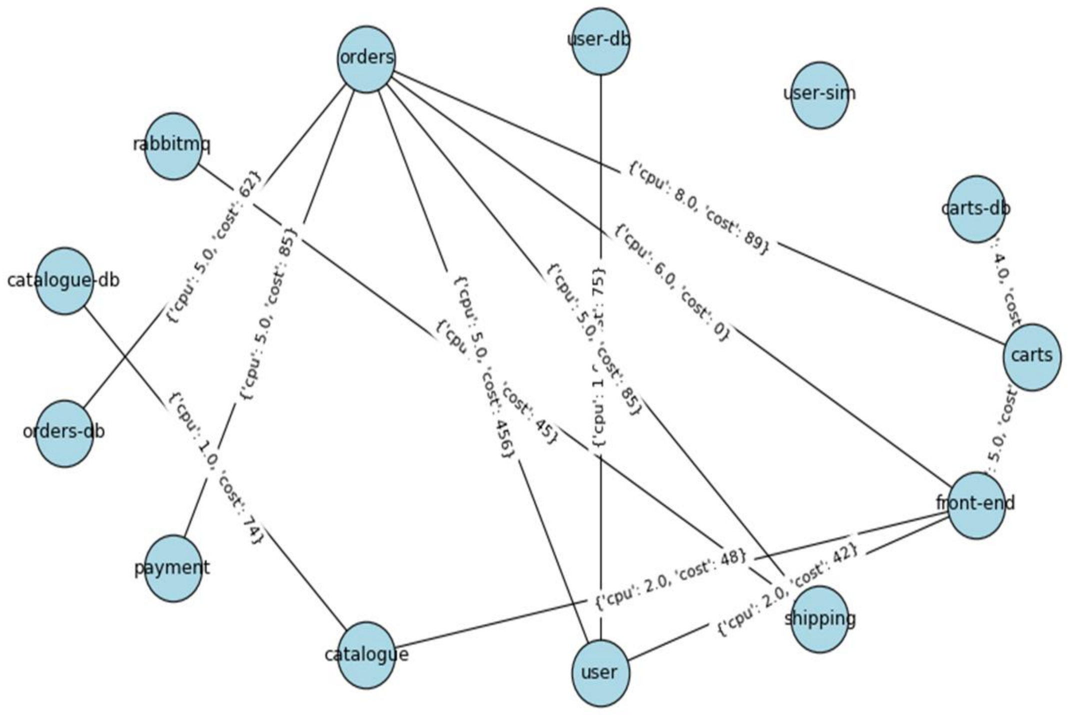

Fig. 6. Annotated service communication graph for Sock-Shop µApp

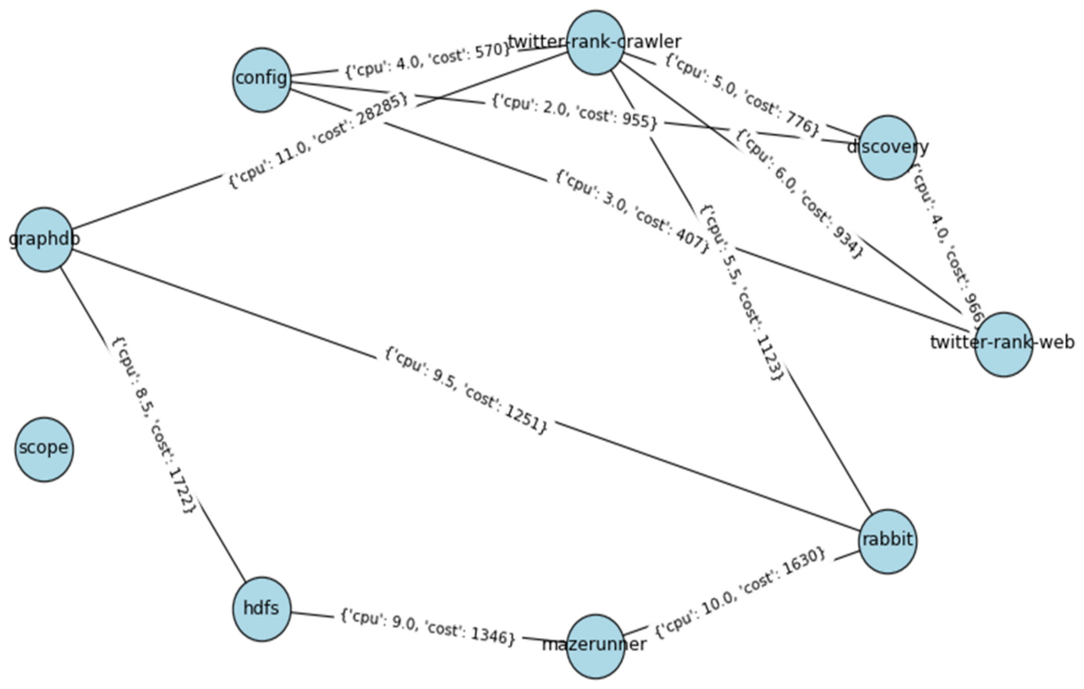
config
twitter-rank-crawler
discovery
graphdb
twitter-rank-web
scope
rabbit
hdfs
mazerunner
{'cpu': 4.0, 'cost': 570}
{'cpu': 5.0, 'cost': 776}
{'cpu': 2.0, 'cost': 955}
{'cpu': 11.0, 'cost': 28285}
{'cpu': 6.0, 'cost': 934}
{'cpu': 3.0, 'cost': 407}
{'cpu': 4.0, 'cost': 966}
{'cpu': 5.5, 'cost': 1123}
{'cpu': 9.5, 'cost': 1251}
{'cpu': 8.5, 'cost': 1722}
{'cpu': 9.0, 'cost': 1346}
{'cpu': 10.0, 'cost': 1630}

Fig. 7. Annotated service communication graph for Page-Rank µApp

TABLE III.

COLOCATION DECISIONS FOR SOCK-SHOP MAPP

Host Machine	Selected set of services to deploy
CPU cores 4	front-end, user, user-db, catalogue, catalogue-db
CPU cores 5	carts, carts-db, rabbitmq, shipping
CPU cores 5	orders, payment
CPU cores 4	orders-db, user-sim

TABLE IV.

COLOCATION DECISIONS FOR PAGE-RANK MAPP

Host Machine	Selected set of services to deploy
CPU cores 3	discovery, config
CPU cores 6	twitter-rank-web, twitter-rank-crawler
CPU cores 9	hdfs, graphdb
CPU cores 10	rabbitmq, mazerunner

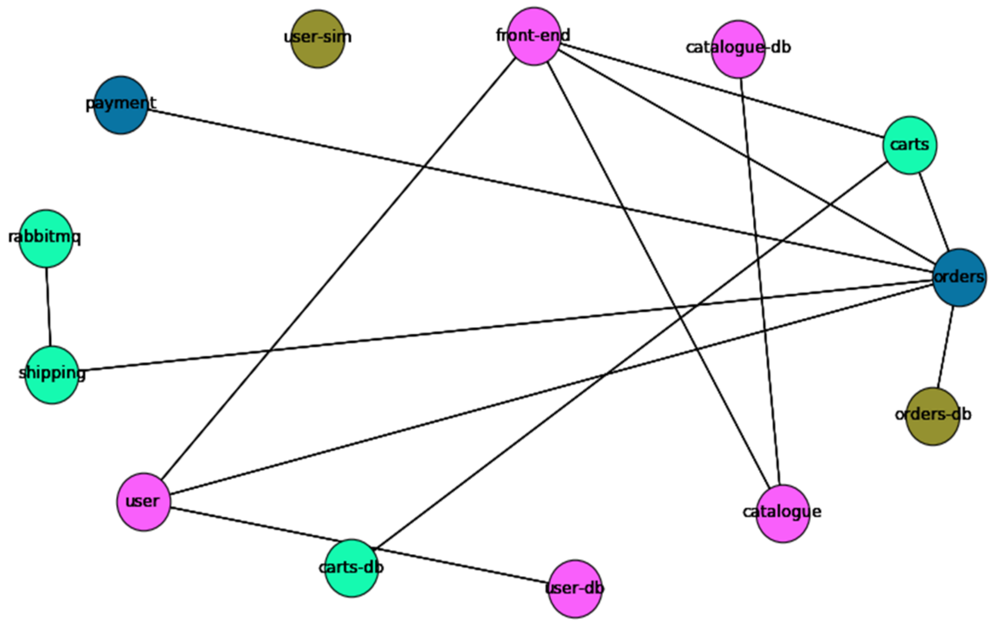

Fig. 8. Final service communication graph for Sock-Shop µApp

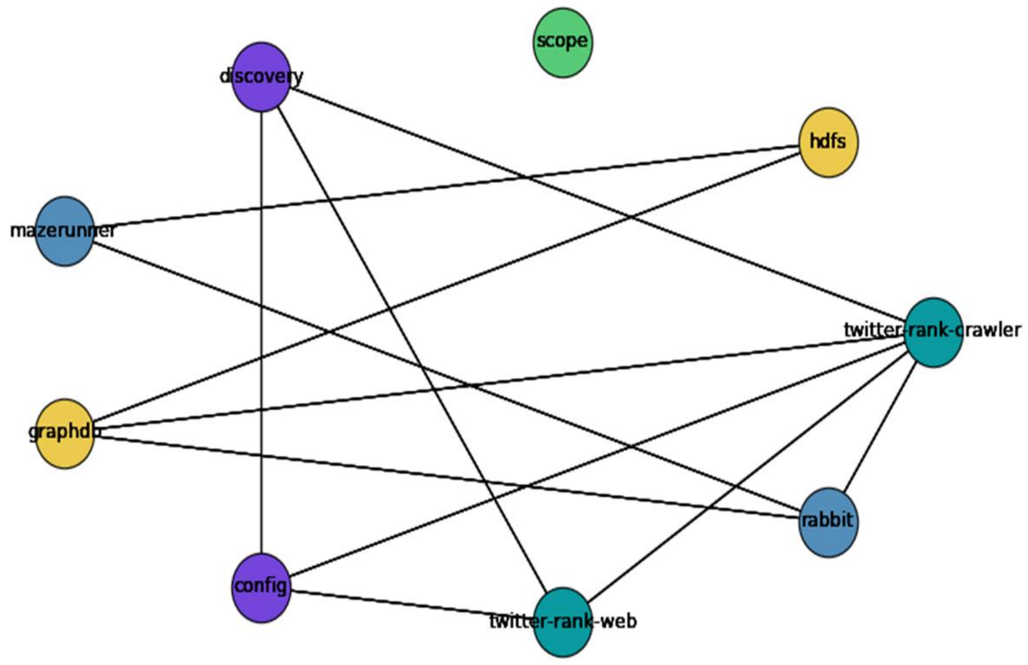

Fig. 9. Final service communication graph for Page-Rank μApp

After redeploying the applications according to the above deployment decisions, affinity values for the same connections were measured. To evaluate this process, results from the spread strategy deployment were compared with the results of the colocated deployment. Total communication affinity reduction from spreaded deployment to colocated deployment is shown in Fig. 10 and Fig. 11 for Sock-Shop and Page-Rank benchmarks respectively. As depicted in these figures, the total affinity reduction on communication over the overlay network to bridge network, are 57% and 52% for the two applications.

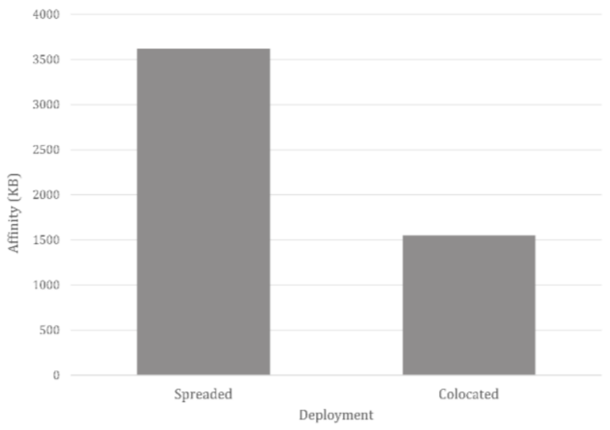

Fig. 10. Sock-Shop: Comparison of total affinities between spreaded and colocated deployments

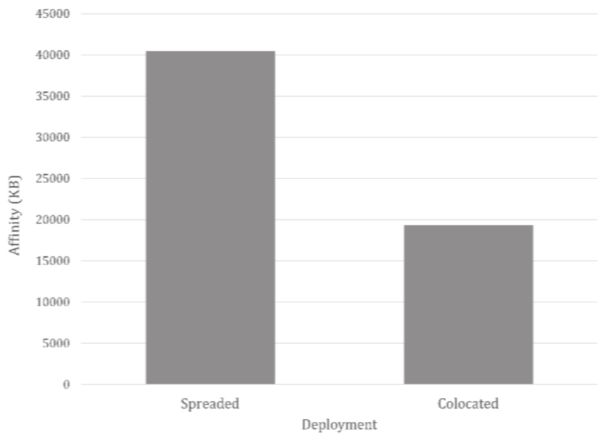

Fig. 11. Page-Rank: Comparison of total affinities between spreaded and colocated deployments

In spreaded deployment traffic is forwarded through the network interfaces of the hosts using an overlay network. This includes several layers of packet encapsulations, decapsulations, and address translations (Fig. 12). But in colocated deployment, traffic of the containers that are in the same host goes through a bridge network with less encapsulation overhead (Fig. 13). This is the reason for the affinity reduction in the colocated deployment.

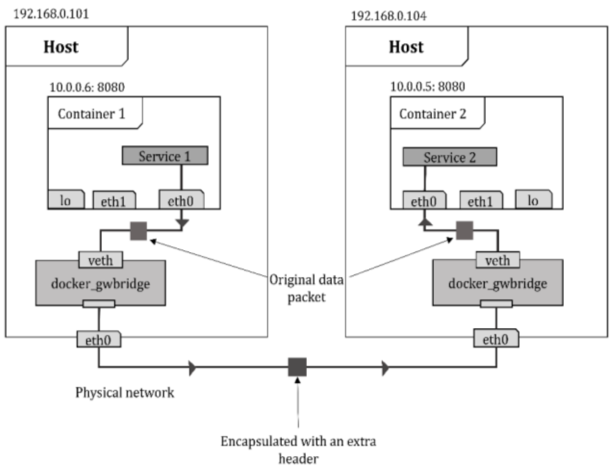

Fig. 12. Underline packet transmission in spreaded deployment

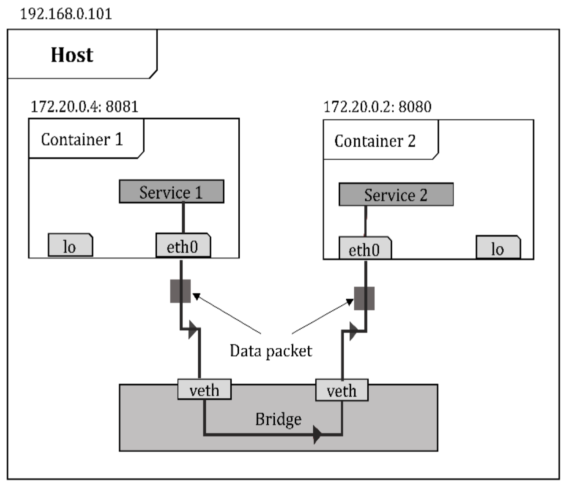

Fig. 13. Underline packet transmission of colocated containers

Stacked charts in Fig. 14 and Fig. 15 depict the contribution of the inter-host communication affinity and inter-container communication affinity to the total communication affinity in the two deployments. It is clear that the inter-host communication affinity is reduced by a significant amount through the colocation process. After colocation, inter-service communication between the colocated services change from inter-host to inter-container communication. An inter-container overhead is always guaranteed to be less than the corresponding inter-host overhead. Hence from the colocation process, if service colocation takes place, approximately 50% of an affinity reduction is guaranteed.

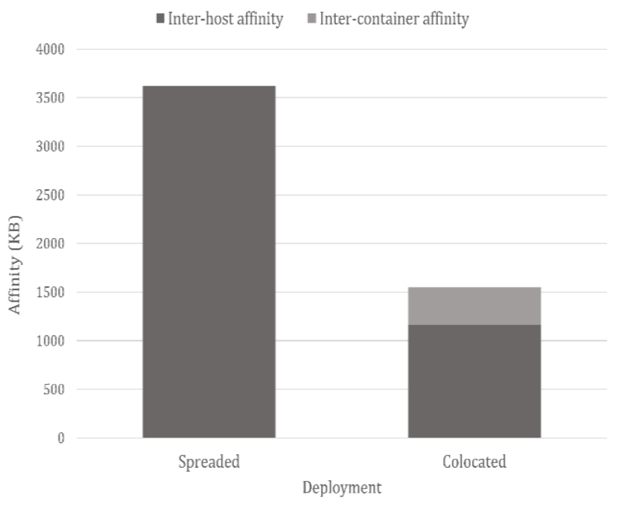

Fig. 14. Sock-Shop: Inter-host and inter-container components of the total communication affinity

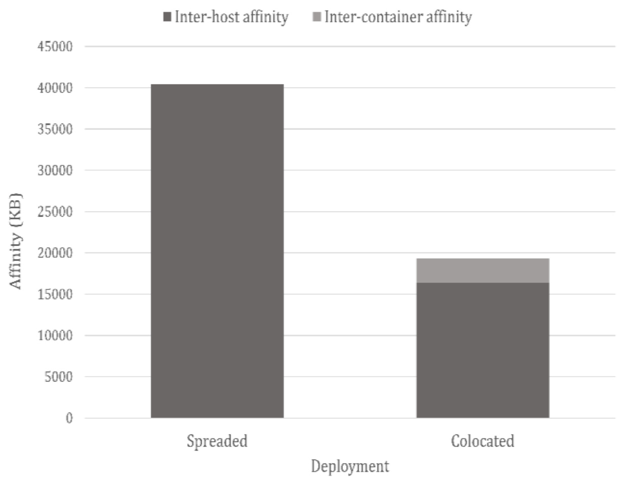

Fig. 15. Page-Rank: Inter-host and inter-container components of the total communication affinity

2) *Merging Process Evaluation:*

The services of the Page-Rank application which are colocated in each host (Table IV) can be considered for merging. However, only the discovery service with the config service and the twitter-rank-web service with the twitter-rank-crawler service fulfilled the prerequisites for merging. After merging those service containers, affinity values for the same connections were again measured. As in Fig. 16, the total affinity reduction from colocated deployment to merged deployment is approximately 13%. In this scenario, the traffic returns back to the container from the loopback interface since both services are in the same container (Fig. 17). Therefore, further affinity reduction was possible.

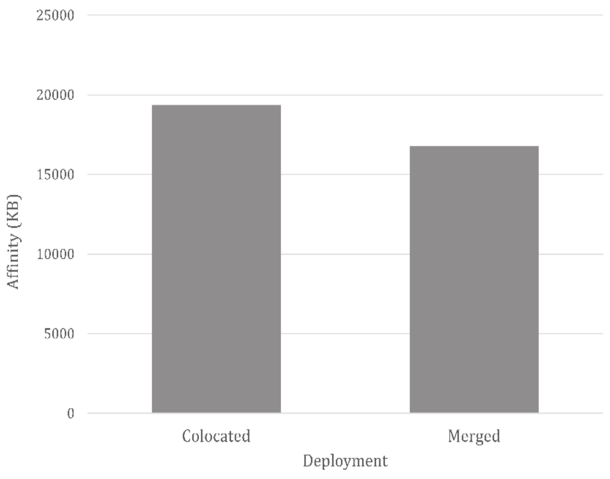

Fig. 16. Page-Rank: Comparison of total affinities between colocated and merged deployments

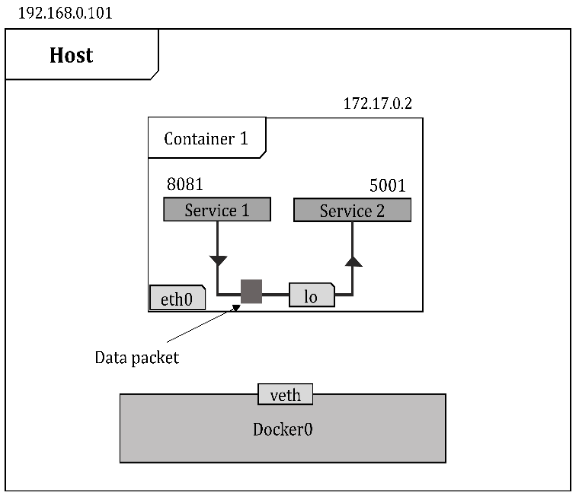

Fig. 17. Underline packet transmission of a merged container

The stacked chart in Fig. 18 shows the contribution of the inter-host, inter-container, and intra-container communication affinities to the total communication affinity in the two deployments. By analysing that, it is noticed that both inter-host and inter-container affinities have reduced. But the merging process does not affect the inter-host communication at all. Hence it is not guaranteed that the inter-host communication affinity would reduce all the time. However, once the containers are merged, inter-service communication of the services in the merged containers changes from inter-container to intra-container communication. An intra-container overhead is always guaranteed to be less than the corresponding inter-container overhead and ultimately that affects the reduction of the total affinity. Hence from the merging process, an affinity reduction is guaranteed and approximately 13% of an affinity reduction is noticed.

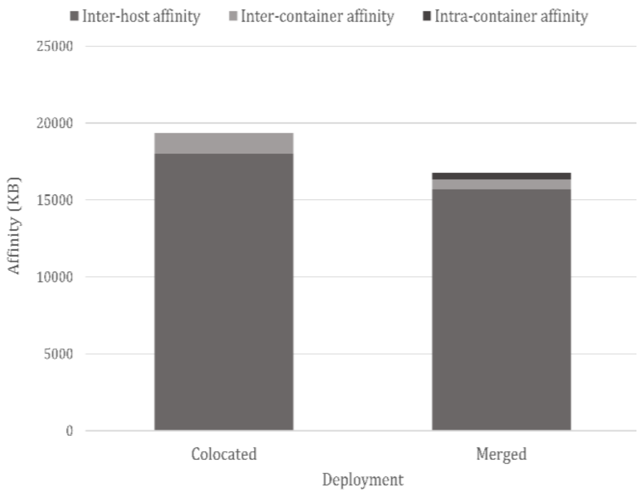

Fig. 18. Page-Rank: Inter-host, inter-container and intra-container components of the total communication affinity

3) *Overall Evaluation:*

Evaluation of the complete solution has been done in terms of communication affinity and the execution time of the application. Fig. 19 shows the affinity reduction from the initial deployment to the final deployment in the Page-Rank µApp. Total affinity reduction from the proposed approach is approximately 58.5%. The contribution of each affinity component to the total affinity for different deployments is shown in Fig. 20. The ultimate impact of this communication affinity reduction to the µApp is the execution time reduction. In Page-Rank µApp, the execution time for one operation (to update the ranks on the dashboard) reduces from 1490 seconds to 1291 seconds as shown in Table V. This represents 13.4% reduction in the execution time per operation.

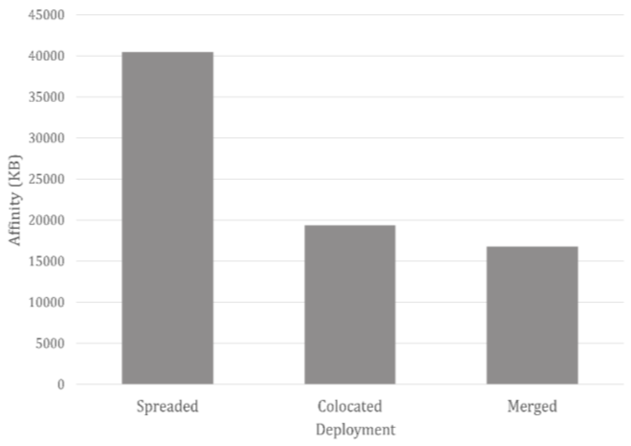

Fig. 19. Total communication affinity comparison of 3 deployments

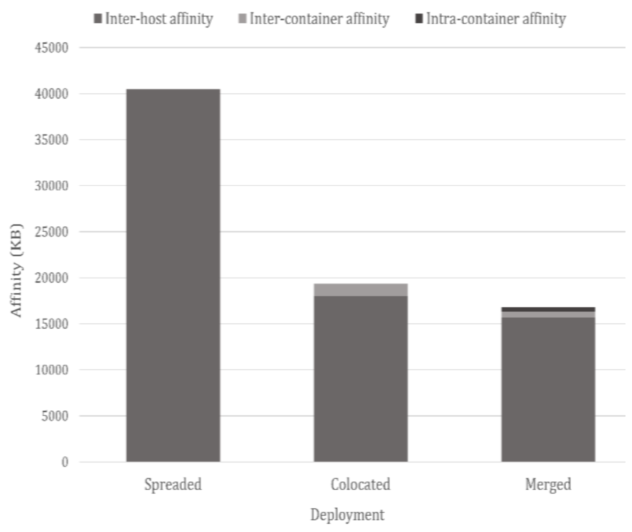

Fig. 20. Contribution to total affinity from 3 communication affinity components

TABLE V.

EXECUTION TIME PER OPERATION IN PAGE-RANK MAPP

Deployment	Time (in seconds)
Spreaded	1490
Colocated	1358
Merged	1291

μApps used to evaluate the proposed solution are not biased and they are accepted among the research community in this domain, as good applications to represent and evaluate the microservice architecture [20]. Hence the results obtained can be generalized to a wider context.

V. CONCLUSIONS

The volume of data exchanged between two microservices is a good indicator of their affinity. Typically, microservices are deployed in their own containers and different networks such as overlay, bridge and loopback, are used to interconnect these containers. The application-level data volume exchanged remains the same irrespective of the type of network used to exchange the data. However, different networks add different encapsulation overheads to communication. This overhead is significant, and it can be reduced by changing the network by intelligently placing the containers in hosts based on the communication affinities between the services. Container placement is constrained by the resources available in the hosts. This study models this problem as an instance of the BKP which is in the complexity class NP-Hard. This paper presents a novel heuristic to solve this problem and deploy the containers to significantly reduce the total communication affinity of an application. The proposed approach goes beyond the colocation and merges containers where possible. To the best of our knowledge, this is the first work that merges containers to reduce the communication cost of μApps. This study demonstrates that, by employing the combined strategy of colocation and merging, it is possible to reduce the communication overhead up to 58.5%. Also, the execution time is reduced by 13.4%.

The colocation process considers only the CPU consumption as a constraint on the colocation. But there are other constraints such as memory. There is a potential to improve these results by considering multiple such constraints. However, it is not a trivial extension to the current work. It requires extensive further study to address this problem. In this study the authors have not attempted runtime reconfiguration of the deployment. The application is redeployed after taking the colocation and merging decisions. Therefore, there can be a significant downtime during the redeployment. However, this strategy ensures the stability and the consistency of the services. Further, studies are required to safely reconfigure a deployment during the runtime.

REFERENCES

[1] "Microservices", martinfowler.com, 2021. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 28-Mar- 2021]

[2] T. Ueda, T. Nakaike and M. Ohara, "Workload characterization for microservices, " IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, pp. 1-10, doi: 10.1109/IISWC.2016.7581269, 2016.

[3] K. Suo, Y. Zhao, W. Chen and J. Rao, "An analysis and empirical study of container networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 2018, pp. 189-197, doi: 10.1109/INFOCOM.2018.8485865.

[4] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," 2015 10th Computing Colombian Conference (10CCC), 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[5] J. Chen, K. Chiew, D. Ye, L. Zhu and W. Chen, "AAGA: Affinity-aware grouping for allocation of virtual machines," 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), 2013, pp. 235-242, doi: 10.1109/AINA.2013.22.

[6] A.Sampaio, J.Rubin, I. Beschastnikh and N. Rosa, "Improving microservice-based applications with runtime placement adaptation", Journal of Internet Services and Applications, vol. 10, no. 1, 2019. Available: 10.1186/s13174-019-0104-0.

[7] N. Tziritas, T. Loukopoulos, S. Khan, C. Xu and A. Zomaya, "A Communication-Aware Energy-Efficient Graph-Coloring Algorithm for VM Placement in Clouds," 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, (SmartWorld/SCALCOM/UIC/ ATC/CBDCom/IOP/SCI), 2018, pp. 1684-1691, doi: 10.1109/SmartWorld.2018.00286.

[8] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.

[9] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri and Y. Al-Hammadi, "Performance comparison between container-based and VM-based services," 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), 2017, pp. 185-190, doi: 10.1109/ICIN.2017.7899408.

[10] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," 2015 IEEE 14th International Symposium on Network Computing and Applications, Cambridge, MA, 2015, pp. 27-34, doi: 10.1109/NCA.2015.49.

[11] K. Sun, H. Yang, Y. Park, Y. Kim, and W. Lee, Considerations for Benchmarking Network Performance in Containerized Infrastructures, 2020

[12] N. Kratzke, “About microservices, containers and their underestimated impact on network performance,” ArXiv171004049 Cs, Sep. 2017.

[13] "Production-Grade Container Orchestration", Kubernetes, 2020. [Online]. Available: https://kubernetes.io/. [Accessed: 07- Nov-2020].

[14] "Swarm mode overview", Docker Documentation, 2020. [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed: 08-Nov- 2020].

[15] J. Han, Y. Hong and J. Kim, "Refining microservices placement employing workload profiling over multiple Kubernetes clusters," in

IEEE Access, vol. 8, pp. 192543-192556, 2020, doi: 10.1109/ACCESS .2020.3033019.

[16] J. Sonnek, J. Greensky, R. Reutiman and A. Chandra, "Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration," 2010 39th International Conference on Parallel Processing, 2010, pp. 228-237, doi: 10.1109/ICPP.2010.30.

[17] "Empowering App Development for Developers | Docker", Docker, 2020. [Online]. Available: https://www.docker.com/. [Accessed: 08-Nov- 2020].

[18] "TCPDUMP/LIBPCAP public repository", Tcpdump.org, 2020. [Online]. Available: https://www.tcpdump.org/. [Accessed: 08- Nov-2020].

[19] "Microservices Demo: Sock Shop", Microservices-demo.github.io, 2017. [Online]. Available: https://microservices-demo.github.io/. [Accessed: 19- Mar- 2021].

[20] C. M. Aderaldo, N. C. Mendonça, C. Pahl and P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research," 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), 2017, pp. 8-13, doi: 10.1109/ECASE.2017.4.

[21] C. Nguyen, A. Mehta, C. Klein and E. Elmroth, "Why cloud applications are not ready for the edge (yet)", Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, 2019. Available: 10.1145/3318216.3363298 [Accessed: 19- Mar- 2021].

[22] J. Rahman and P. Lama, "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud", 2019 IEEE International Conference on Cloud Engineering (IC2E), 2019. Available: 10.1109/ic2e.2019.00034 [Accessed: 19- Mar- 2021]

[23] A. Shaheen and A. Sleit, "Comparing between different approaches to solve the 0/1 Knapsack problem", International Journal of Computer Science and Network Security, vol. 16, no. 7, 2016. [Accessed: 19- Mar- 2021].

[24] N. AGARWAL, "Understanding the Docker Internals", Medium, 2017. .