ShareOn: Shared Resource Dynamic Container Migration Framework for Real-Time Support in Mobile Edge Clouds

SHALINI CHOUDHURY, SUMIT MAHESHWARI, IVAN SESKAR, AND DIPANKAR RAYCHAUDHURI
WINLAB, Rutgers University, North Brunswick, NJ 08902, USA
Corresponding author: Shalini Choudhury (shalini@winlab.rutgers.edu)

ABSTRACT Mobile Edge Cloud (MEC) technology is envisioned to play a key role in next generation mobile networks by supporting low-latency applications using geographically distributed local cloud clusters. However, MEC faces challenges of resource assignment and load balancing to support user mobility and latency-sensitive applications. Virtualized resource reallocation techniques including dynamic service migration are evolving to achieve load balance, fault tolerance and system maintenance objectives for resource constrained edge nodes. In this work, a compute and network-aware lightweight resource sharing framework with dynamic container migration, ShareOn, is proposed. The migration framework is validated using a set of heterogeneous edge cloud nodes distributed in San Francisco city, serving mobile taxicab users across that region. The end-to-end system is implemented using a container hypervisor called LXD (Linux Container Hypervisor) executing a real-time application to detect license number plates in automobiles. The system is evaluated based on key metrics associated with application quality-of-service (QoS) and network efficiency such as the average system response time and the migration cost for different combinations of load, compute resources, inter-edge cloud bandwidth, network and user latency. A detailed migration cost analysis enables evaluation of migration strategies to improve ShareOn’s performance in comparison to alternative migration techniques, achieving a gain of 15-22% in system response time for highly loaded edge cloud nodes.

INDEX TERMS Quality of experience (QoE), container migration, mobile edge computing, virtualization, real-time applications, edge cloud, migration cost, inter-edge cloud bandwidth.

I. INTRODUCTION

Advanced mobile applications such as Augmented/Virtual Reality (AR/VR) involve intensive computation with real time response constraints. The delay requirements of these applications may not be met by the cloud-assisted techniques due to large network latency. Mobile Edge Cloud (MEC) [1], [2] has been identified as a promising approach which supports low-latency applications by bringing compute, network and storage close to the user. Edge cloud (EC) provides computation at the edge of the network using limited available resources. Furthermore, the geographically local MECs use various techniques to distribute tasks across the network edge to satisfy user service requirements [3]–[5].

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

MEC infrastructure must respond to factors such as user mobility, network outage, fluctuating load, variability in network performance/congestion, and resource heterogeneity [6]. Thus, maintaining application quality-of-service (QoS) via distributed coordination between local edge cloud clusters is a challenge [7]–[9]. MEC utilizes virtualization technologies to run and manage applications and services at the edge of the mobile network using Network Functions Virtualization (NFV) as the environment [10]–[12]. Typically, the architecture of a general MEC system consists of three computing layers: the first at the mobile client, the second at a local network attached edge cloud, and the third at a centralized data center/cloud in the core of the network, as shown in the Figure 1 [13]. The hierarchically placed, limited-resource, MEC nodes often offload compute intensive tasks to a central cloud server, thereby increasing the...
total response time of an application. Thus, this architecture does not readily support real-time applications which require round trip latency of less than ~100 ms.

Application QoS can be maintained in the MECs using a number of distinct mechanisms [14]–[17]. These include the use of either centralized or distributed resource assignment schemes which allocate compute resources to mobile user requests at nearby servers. The centralized software defined network (SDN) framework is widely adopted for data-intensive computing and easy management of the architecture [18]–[20]. However, SDN infrastructure faces issues related to data mobility in the process of container migration and might be inefficient for dispersed edge cloud nodes. Resource virtualization (virtual machines and virtual networks) can also be used to partition and control resource usage between multiple competing applications or users [21].

In the context of SDN and resource virtualization, NFV is a new network paradigm which enables interconnection of virtualized network functions to set up service function chaining (SFC) [22]–[24]. It is also possible to employ GPU’s and/or parallelize computing resources across the network to accelerate the computation [25], [26]. Further, mobile user performance can be dynamically optimized via container migration in which the cloud process is moved from one computing node to another in response to mobility events and to ease the load across the network. Migration implies moving a virtual machine (VM) or a container from one edge cloud to the other [27]–[30]. In this paper, we describe and validate a container migration framework suitable for MEC scenarios using a sample latency constrained application.

The rest of the paper is organized as follows. Section II provides the motivation for container migration and our key contributions. Section III provides the migration primitives including service containerization and its benchmarking. Section IV details the analytical and cost model of the migration system designed in this work. Section V details the migration set-up along with the evaluation on a realistic testbed. Section VI illustrates simulation set-up and parameters. Results are discussed in Section VII, related work in Section VIII with a note on limitations, challenges and our future work and Section IX concludes the paper.

II. MOTIVATION AND KEY CONTRIBUTIONS

The limited available resources in MEC, cannot efficiently serve sporadic requests due to user mobility and varied load when using a straightforward edge cloud to core cloud offloading technique. Figure 2 evaluates impact of load on an EC node. In this evaluation we execute a set of example services such as credit card number detection (Service 1) and traffic lane detection (Service 2) on an Intel i7 CPU K875 with 8GB RAM and 2.93 GHz processor. Figure 2 concludes that the rise in load degrades the service performance. For service 2, at 0.3 load level - emulated with Linux Stress tool, the EC node’s available resources are not able to sustain the application. A similar pattern is observed for service 1 at load 0.4. Therefore, newer techniques such as resource sharing by use of migration can be analyzed for improving performance. The overloaded edge cloud scenario motivates the technique of lightweight container migration, to an available resourceful destination edge node, to address the issue of compute bottleneck. With the help of flexible resource provisioning and sharing among neighboring edge cloud nodes, MEC can meet unpredictable traffic demands and quickly scale the network due to its multi-tenancy feature. In order to reduce the application latency and provide the required application QoS, service requests are handled by the virtualized environment. Container-based virtualization is finding increasing adoption in MEC systems for realizing slice isolation and fine resource control [31]. Resource isolation (especially, memory) across components of different applications is necessary for the integrity of individual applications.

Containers run at the user-space process over the kernel and have better bandwidth and CPU performance as compared to VMs. Their low running overhead and small size make them ideal candidate for the service migration. Virtualization based on containers (containerization) allow users to run an application and its dependencies in an Operating System (OS) with flexible resource allocation, easy scaling and improved efficiency [32], [33]. These are highly portable since each container image is inclusive of all the dependencies required to execute an application. Also, the host OS limits the container access to the physical resources. Containers do not require separate OS instances and operate by sharing the same OS kernel as the host machine, leading to a gain in memory, CPU and storage. The major factor in the interest of containers is that they spin up and stop faster, which is crucial for distributed application [34], [35]. Thus, lightweight containers allow resource sharing among neighboring nodes using technique such as migration [67]. The shared resource dynamic container migration approach therefore can be used to address degrading application QoS,
arising from the processing latency (system load) and/or the network latency (user mobility).

This paper aims to develop a container migration framework (ShareOn) and validates its performance via simulation of a realistic MEC scenario. An end-to-end migration framework running real-time application has been deployed to analyze the impact of resource allocation, latency, bandwidth, container size and migration time. The framework is evaluated on software-based emulation environment set-up using the ORBIT testbed [36], running LXD containers using CRIU (Checkpoint Restore in Userspace) [37]. An edge cloud network topology is studied with nine heterogeneous edge nodes placed geographically apart in San Francisco (SFO) city to match the real world scenario. The migration evaluation is done based on a real-time automated license plate recognition (ALPR) [38] application, orchestrated with the help of live scripts. ALPR analyzes images and video streams to identify license plates and the output is the text representation of the license plate characters. A comparative model is also set up for large scale simulation, in which the containers are hosted in an edge cloud network in SFO city. Real traces from taxicabs in SFO [39] are used to model user mobility.

ShareOn is a distributed migration scheme to support user mobility across dispersed edge cloud infrastructure and is scalable for networks hosting large number of edge compute nodes.

The key contributions of this paper are:

1) Design and develop a distributed, shared resource dynamic container migration framework to support low-latency applications in the MEC, by accurately deciding the migration destination depending on the user proximity and compute resource availability at the destination EC node

2) Evaluate the proposed framework on a large-scale city-based simulation model using the realistic taxicab traces and parameters such as bandwidth, load and compute. This framework is compared with different migration approaches to show that multi-parametric migration performs significantly better in handling traffic bursts in contrast to single/selective parametric approaches.

3) Solve system workload and mobility issues for different resource combinations in a container migration study and evaluate the average system response time, migration cost and effect of inter-edge bandwidth on migration.

4) Propose techniques to minimize service latency at an edge cloud by evaluating system-wide migration cost based on pre-copy, post-copy and migration time. This work further details the overall cost as local migration cost, i.e., delay incurred at each EC node and shared migration cost, i.e., collective delay incurred between pair of EC nodes.

III. CONTAINER MIGRATION PRIMITIVES IN THE MEC

In the distributed container migration approach, the edge cloud nodes can independently implement service migration strategies including the policies for suspension and migration of currently running applications. This section presents an overview of the migration and its benchmark performance in terms of memory, CPU, latency and pre-copy time.

A. SERVICE MIGRATION

Virtualization of OS has gained popularity since it allows multiple instances of OS to run on a given physical machine, leading to efficient usage of physical resources and simple management. One of the benefits of virtualization is migration, the process of transitioning from one operating environment to another without disruption in user connectivity.

Although, it is essential to accurately decide migration destination depending on the mobile users proximity and the computation resource availability at an edge cloud node, the unpredictability of user movement add challenges to an optimal decision. Furthermore, the bursty compute traffic generated by large number of devices such as UEs and IoTs, [40]–[43] causes additional load variability in the system. This degrades the network performance as the computation resources to support such traffic bursts become unavailable. Under such conditions service migration can ensure improved performance of the system.

For an effective service migration solution, parameters such as edge cloud node load, available resources (CPU, RAM, inter-edge bandwidth), distance between the edge cloud nodes, and application requirements should be considered. This work attempts to implement a migration framework taking these parameters into account, by which an application or cloud-hosted services are prevented from being locked into a single physical device.

B. SERVICE CONTAINERIZATION

The enormous services running on today’s cloud servers [44] (VM based) face challenges of dynamic resource allocation, switching (on/off) delays, and inability to scale due to their size thus requiring large chunk of host computing resources. Containers are miniaturized virtual resources exclusively consisting of computational tasks and worker processes.
Their intrinsic property of service isolation [45] and their simple design make them ideal for the services which require faster and frequent switching [46]. Therefore, containers allow splitting services by grouping simple, smaller tasks into a single package called a microservice. With the help of containers, these microservices can be managed in runtime, thereby enabling multiplicity of such services on a shared infrastructure. Containerization is also beneficial for NFV infrastructure by providing flexibility, resiliency, scalability and portability. Inherently containers provide easy deployment, development and testing of applications on a large number of servers and are a viable choice for service migration as utilized in this work.

C. SERVICE CLASSIFICATION

For achieving migration without impacting the user QoS, it is important to understand various types of services that can be containerized for migration. Services based upon their requirements and usage can be classified as follows.

- **Stateless vs Stateful**: Stateless services do not use previously processed tasks as feedback to the current task. Therefore, they can run without saving the user generated data in a session. State refers to an altering condition, inclusive of the results from the internal operations and interactions with other services or applications. Stateful application can retain information about its state after each service execution. The content of the stored data may depend on the type of service. An essential feature about statefulness is that it requires persistent storage for later availability of the information. The state management plays a key role in migrating stateful services which adds migration complexity. On the contrary, stateless migration reduces complexity, the amount of data to be sent and requires less network resource [47].

- **User-space vs Network Space**: The container migration can be used in the **user-space** for the end-user applications or in the **network space** to support NFVs such as firewall, load balancer, DNS and gateways (GWs) [48], [49]. For non-standalone services, – dependent on the other containerized services – migration is not a feasible solution as group of containers along with a large amount of state has to be migrated which is not trivial.

In our standalone system study, containerized services are independent of each other where the application’s data are attached to the respective containers; hence, migration can preserve or enhance service quality. This paper illustrates the migration approach with an example user-space application (ALPR) while the approach can be similar to the network space services.

D. CONTAINER MIGRATION BENCHMARKING

This section compares different types of services and analyzes candidates to survive migration without degrading the application QoS. Figure 3 compares the Memory and CPU utilization for real-time stateless and stateful applications for different load conditions, with stringent latency requirements. Figures 3 (a) and (b) show the stateless operations like file writing (FW), lane detection (LD) [50] and credit card number detection (CCND) [51]. These applications do not demand any state data to be stored and hence they can be deployed as user-space standalone stateless services in the containers. Figures 3 (c) and (d) show the performance of stateful applications such as, Group path planning (GPP) [52], deep speech (DS) [53] and tensor flow (TF) [54]. These standalone stateful applications rely heavily on the state data to continue running a service using the host storage volume and then mapping them to containers.

Table 1 and 2 compare the relative performance of the stateless and stateful applications described above. For the stateless applications, it is observed that at the lower load, the resource utilization and application run time in the containers are significantly low. The application session can save the data generated for usage in the subsequent sessions with volume plugin, but there is almost negligible wait time for synchronization of information from the previous session. Hence, the process is faster with reduced memory usage.

The stateful applications use a number of logical cores to determine internal performance values. This results in overhead since the number of required cores might not be available. It is observed while the stateful applications run in the container, they consume more resources at higher load than compared to stateless applications. Therefore, the required latency constraints cannot comply with the strict time requirement of real-time applications because of the limited support for data portability in containers. Stateful
TABLE 1. Stateless application resource parameters.

| App. | Memory | CPU | Latency | Pre-copy Time |
|------|--------|-----|---------|---------------|
| FW   |        |     |         |               |
| LD   |        |     |         |               |
| CCNR |        |     |         |               |

TABLE 2. Stateful application resource parameters.

| App. | Memory | CPU | Latency | Pre-copy Time |
|------|--------|-----|---------|---------------|
| GPP  |        |     |         |               |
| DS   |        |     |         |               |
| TP   |        |     |         |               |

applications requires the state to be stored in memory of the host wherein there is a frequent interaction of the application with the storage capacity and requires a full database sync for continued service.

These observations lead us to characterize the applications in terms of their suitability for containerization as well as migration to support real-time applications. In general, it is to be noted that stateless containerized applications are viable candidates for service migration owing to their limited memory and CPU utilization without impacting application performance. Further, the absence of associated states allow the destination EC node to seamlessly instantiate the container without any loss of information/data.

IV. MODELING CONTAINER MIGRATION

In this section the migration latency and cost are modeled to quantitatively analyze the system.

A. MIGRATION LATENCY MODEL

Here, the latency model for inter-edge cloud container migration is presented by considering paging, transmission, propagation and container spin-up delays. Further, the time taken to select a container and pair it with target EC node for migration using the control plane information is taken into account.

1) CONTROL PLANE LATENCY

During the process of control plane information exchange, the packets containing node’s information are exchanged among the neighbors. The time taken to process the periodic control messages, \( t_{\text{control}} \), is assumed to be a constant.

2) SELECTION LATENCY

The algorithm run time to select the containers to be migrated to a suitable destination is called \( t_{\text{select}} \). It is measured for the Algorithms 2 and 3 in Section IV-C.

3) PAGING LATENCY

The amount of time taken at the source node to create pages of the container selected to be migrated is dependent on the processing speed of the host node. Therefore, the paging latency is determined with the help of processing speed of the node and the memory access speed. The processing speed is dependent on the processor clock frequency \( f \). It is assumed that processor executes one instruction per clock cycle. Therefore, the time taken to create the container pages, each of size \( s_{\text{page}} \), can be given as

\[
t_{\text{page}} = \frac{s_{\text{con}}}{f \cdot s_{\text{page}} \cdot \alpha}
\]

where \( \alpha \) is the rate at which the page is dirtied (see Table 3).

4) TRANSMISSION LATENCY

The transmission delay for each container page is dependent on each page size and available transmission rate between the source and destination node. It is estimated as

\[
t_{\text{tx}} = \frac{s_{\text{page}}}{b_{\text{i}}}
\]

5) PROPAGATION LATENCY

The propagation latency, \( t_{\text{prop}} \), is calculated as

\[
(t_{\text{d}} - t_{\text{s}})/S,
\]

where \( S \) is dependent on the speed of the medium.

6) SPIN-UP LATENCY

The latency incurred to process all the pages received at the destination and instantiate the container is termed as spin-up latency, \( t_{\text{spin}} \). The spin-up delay is dependent on inter-page

TABLE 3. Container migration parameters.

| Param | Brief | Explanation |
|-------|-------|-------------|
| \( \alpha \) | page dirty rate (KBps) | memory pages modified per sec |
| \( s_{\text{page}} \) | page size (KB) | default size of a memory page |
| \( p_{\text{rd}} \) | memory pages (kpps) | # memory pages created per sec |
| \( s_{\text{p}} \) | processor speed (GHz) | residual bandwidth |
| \( m \) | RAM (GB) | user to cloud network delay |
| \( b_{\text{i}} \) | inter-edge bw (Gbps) | wait time of request |
| \( l_{\text{n}} \) | network latency (ms) | push packet bits into the link delay |
| \( l_{\text{q}} \) | queuing latency | latency beyond which app performance degrades |
| \( l_{\text{lx}} \) | transmission latency | range (0–1) |
| \( l_{\text{th}} \) | app threshold latency | between edge clouds at an edge cloud |
| \( k \) | load at MEC | \( t_{\text{nx}} + t_{\text{tx}} + t_{\text{prop}} \) |
| \( k_{\text{in}} \) | running containers | time to create pages of the container |
| \( k_{\text{out}} \) | containers sent | time to process a page to spin up the container |
| \( t_{\text{c}} \) | control plane delay | cost to migrate a container |
| \( t_{\text{p}} \) | computation delay | current load at a node |
| \( t_{\text{prop}} \) | propagation delay | cost to migrate a container |
| \( t_{\text{total}} \) | total response time | cost to migrate a container |
| \( l_{\text{px}} \) | page creation delay at source | cost to migrate a container |
| \( l_{\text{pxd}} \) | page processing delay at dest | cost to migrate a container |
| \( l_{\text{pxr}} \) | pages received at the destination | cost to migrate a container |
| \( l_{\text{pd}} \) | edge cloud load | cost to migrate a container |
| \( C_{\text{m}} \) | migration cost | cost to migrate a container |
| \( \beta \) | inter-page arrival delay | cost to migrate a container |
| \( s_{\text{con}} \) | container size | cost to migrate a container |
| \( \beta_{\text{min}} \) | first threshold value | cost to migrate a container |
| \( \beta_{\text{max}} \) | second threshold value | cost to migrate a container |
| \( t_{\text{nt}} \) | total migration time | cost to migrate a container |
| \( \delta T \) | cutoff time for migration | cost to migrate a container |
| \( UR \) | utilization region | utilization region of a node |
| \( \gamma_{\text{e}} \) | latency factor | associated with node util |
| \( \gamma_{\text{ad}} \) | utilization factor | node resource utilization factor |
arrival delay (β), number of received pages (page_r) and time to process a page at the destination (t_{page,d}) as: \( t_{spin} = page_r \cdot max(\beta, t_{page,d}) \).

**B. MIGRATION COST**

For a given container to be migrated, the cost is a function of pre-copy time, post-copy time and migration time. The system performance can be evaluated using metrics such as workload characteristic, container size, node computing capabilities, memory dirty rate, network transmission rate and inter-edge bandwidth. The overall cost incurred by each of these components can be defined as a function of time using the migration cost as follows:

- **Local Migration Cost (LMC):** The LMC is the total time incurred in: (a) querying local compute, storage and memory resources, (b) calculating the utilization region (UR), and (c) finding the list of suitable containers to be migrated based upon SLA violations.
- **Shared Migration Cost (SMC):** The SMC is defined as the total time incurred in: (a) receiving and processing control plane messages from neighboring MEC nodes, (b) listing containers based upon available resources including inter-edge bandwidth, and (c) creating memory pages to be migrated from each of these containers.
- **Global Migration Cost (GMC):** The maximum of the sum value of LMC and SMC for all the MEC nodes in the system if all the costs are collected at a central location.

1) LOCAL MIGRATION COST

The network latency, \( t_n \), between the user and the destination edge cloud node can be estimated as \( d_{u,e} \cdot l_d + var(l) \) where \( d_{u,e} \) is the distance between the user and the destination edge node, \( l_d \) is the network latency per unit distance and \( var(l) \) is the past moving average latency variance of the user’s geographic region and the destination node. The processing latency, \( t_p \), of a destination node can be approximated as \( t_{p,avg} + utilFac \cdot \gamma_e \) where \( utilFac \) is the current utilization factor of a node, and \( \gamma_e \) is the latency factor associated with the current utilization.

The maximum computation capability of a local MEC node is assumed to be constant in terms of its processing speed \( s_p \). The available memory \( m \), CPU load \( load \) and number of running containers \( k \) can be queried at each MEC node. The utilization region of a node can be calculated to be in High, Med or Low zone.

The total time taken to create the dirty pages for all the migrating containers is given by the Eq. 1 where \( b_l^{l,d} \) is the available bandwidth between the source and destination nodes. Eq. 1 is a combination of the compute time and the migration time due to inter-edge bandwidth, where \( N \) is the number of edge cloud nodes, and \( k_{out} \) are the number of outgoing containers from the given node.

\[
I_{dirty, total} = \sum_{l=1}^{N} \left( \frac{k_{out}^{l,d} \cdot \alpha \cdot s_{page}}{b_l^{l,d}} \right)
\]  

LMC can be calculated as shown in equation 2.

\[
LMC = k_{in} \cdot t_{spin} + t_{dirty} + c
\]  

Here, \( c \) is the constant time to query the local node.

2) SHARED MIGRATION COST

Migration gives rise to computation and network overheads. Therefore, the decision algorithm has to adequately determine whether, when and where to migrate depending on aspects such as application QoS, user mobility, inter-edge bandwidth, and resource availability at MECs. Each EC node tracks its own compute and networking resources, and is aware of the application latency of each request. These nodes have control plane connectivity to each other and hence can query the neighboring nodes periodically for the above mentioned parameters [56]. Thus, in our system, each node can decide, control and migrate its containers in a distributed manner. The SMC at each node is calculated using equation 3.

\[
SMC = t_{control} + t_{select} + l_{tx} + t_{prop}
\]  

3) GLOBAL MIGRATION COST

GMC is calculated using eq. 4

\[
GMC = \max\{LMC_i + SMC_i\}, \forall i \in N
\]  

The objective of this paper is to minimize the average system latency (application) as fulfilled by algorithm 3 to select the right container for migration. In a continuous migration process, multiple containers migration time overlaps which can be optimized by minimizing the time of maximum GMC as: \( \min\{\max\{LMC_i + SMC_i\}, \forall i \in N\} \).

**C. MIGRATION FRAMEWORK: ShareOn**

The proposed shared resource migration framework ShareOn is a collection of algorithms 1, 2 and 3 and works as follows:

**Algorithm 1** First, each node shortlists high total application latency containers that are exceeding the threshold time \( t_{th} \) and determines the primary reason of the delay — high processing latency, networking latency or both.

**Algorithm 2** For individual container, a few suitable neighbors are selected using two conditions: (a) falls in low or med Util regions and/or (b) geographically closer to the user. The High, Med or Low utilization region of a node can be calculated by capturing available processing speed and RAM, and comparing them with pre-defined values needed to run an application. The average processing per container at a node is \( s_{p,avg} = s_p/k \) and the average RAM is \( m_{avg} = m/k \). This procedure lists all the EC nodes to its corresponding utilization region.

**Algorithm 3** This approach maps all containers with their suitable neighbors. A destination is chosen based upon number of migrations from the source to the destination considering the available inter-edge bandwidth, memory and compute. In order to determine a best destination node for migrating a container, the control messages are sent and received from all the neighboring nodes. The control plane
information received from neighbor includes Listpre-select, current load, unused memory m, and UR. To be able to assign a destination the algorithm maps the container to the low or medium utilized node and also evaluates if the cutoff time for migration to that node exceeds a threshold \( \delta T \) value. This approach avoids ping-pong migration effects for containers which otherwise would result in system instability and increased migration count. The final destination is selected based on \( \min(C_m) \) value following which the memory pages are created and migration is initiated.

V. CONTAINER MIGRATION PROTOTYPE

For analyzing the impact of container migration we conduct a systematic study of the parameters such as system load, inter-edge bandwidth, compute capability, memory and storage requirements and the application performance adhering to the QoS using a realistic testbed set-up. This section describes container migration system and approach, using the emulation set-up which is a pre-cursor to the large-scale simulation performed in this work.

Figure 4 shows the software-based emulation testbed environment set-up on the SB9 (sandbox-9) of the ORBIT testbed [13], [36] for the migration experiments. The heterogeneous compute nodes with Ubuntu 16.04.3 instance are configured as edge cloud nodes running containerized services using the LXD. The emulated experiment is matched to the real-world scenario by using realistic parameters such as bandwidth, load and computation. The essential components for the experiment are described below.

Network Configuration: Edge cloud nodes are placed geographically at randomly chosen locations across the San Francisco (SFO) city as shown in Figure 5 [13]. Mobility is emulated by injecting users from the publicly available SFO cab traces.

Accessing MEC Nodes: From the ORBIT lab console, we use SSH tunneling to connect to the edge cloud nodes for the initial set-up and configuration to support service containerization and migration.

Container Creation: We use Linux based open-source, LXD container technique. The core of LXD is a privileged daemon, which provides a REST API over a local UNIX socket and the network.

Container Network Configuration: The primary sockets used in container migration are: control stream, CRIU (Checkpoint Restore in Userspace) images stream and filesystem stream. LXD supports creating/managing bridges, IPv4 address, NAT (Network Address Translation) and DHCP (Dynamic Host Configuration Protocol) range. The LXD containers are set-up for storage and networking.

Algorithm 1 Container Pre-Selection for Migration

\[\text{Input: } k, t_{th}, t_{total}\]
\[\text{Output: } \text{Pre-selected containers}\]
\[
\text{for container } \epsilon \text{ EdgeNode do}
\]
\[
\text{if } (t_{total} > t_{th}) \text{ then}
\]
\[
\text{container } \rightarrow \text{Listpre-select}
\]
\[
\text{end if}
\]
\[\text{end for}\]
\[
\text{SORT(descending) Listpre-select } (t_{total})
\]

Algorithm 2: Utilization Regions (UR) of a Node

\[\text{Input: } s_p, m_p, k, s_p, s_p, m, s_m, m_m\]
\[\text{Output: } \text{UR(L, M or H)}\]
\[
\text{Calculation}
\]
\[
s_p, m_p \leftarrow s_p/k; /*\text{node calculates average processing}*/
\]
\[
m_p \leftarrow m/p; /*\text{node calculates average memory}*/
\]
\[
\text{if } s_p, m_p \leq s_p, m_p \text{ then}
\]
\[
H \rightarrow \text{UR}
\]
\[
\text{else if } (s_p, m_p > s_p, m_p \text{ and } s_m, m_m) \text{ AND } (s_p, m_p \leq s_m, m_m) \text{ then}
\]
\[
M \rightarrow \text{UR}
\]
\[
\text{else}
\]
\[
L \rightarrow \text{UR}
\]
\[\text{end if}\]
\[\text{end if}\]
\[\text{STORE UR}\]

Algorithm 3 Final Containers and Their Destination Selection for Migration

\[\text{Input: } \text{Listpre-select, C_m, L_{est}}\]
\[\text{Output: } \text{Final selected containers for migration and their destination edge cloud nodes}\]
\[
\text{for cont } \epsilon \text{ Listpre-select do}
\]
\[
\text{for DestNode } \epsilon \text{ EdgeNodes do}
\]
\[
\text{if DestNode } \in \text{UR(L, M) AND } t_{total} - t_{est,dest} > \delta T \text{ then}
\]
\[
\text{DestNode } \rightarrow \text{Listdest}
\]
\[
\text{container } \rightarrow \text{Listselect}
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]
\[
\text{for container } \epsilon \text{ Listselect do}
\]
\[
\text{FinalDest } = \min(C_m) \forall \text{Listdest}
\]
\[
\text{FinalDest } \rightarrow \text{Listcont,dest}
\]
\[\text{end for}\]
Container Migration: The LXD containers are orchestrated for migration using the CRIU, Shell and Python scripts, in a distributed manner at each edge cloud node. All major container engines such as Docker, LXD, OpenVZ, etc already package CRIU as one of the dependencies. Using CRIU it is possible to freeze an entire or part of a running application and checkpoint it as a collection of files. In order to restore the container, all the checkpointed files are sent to the destination host over a TCP socket [55].

Service and Resource Management: The service and resource management techniques are implemented using generic functions (Figure 4, right) such as: (a) service manager: to enable a required service on the node, (b) load emulation: to increase the load at given edge cloud node, (c) container manager: to keep track of container specific resources such as type of service and its QoS, and (d) migration controller: to start and keep track of migration.

User-level Application as a Containerized Service: We use ALPR (automated license plate recognition) as a containerized service to evaluate the system performance. ALPR detects the license plates of cars from the frames obtained from user device video stream. After processing these frames in a container, the output is possible plate numbers with the set confidence level. The application phases include detection, binarization, char analysis, plate edges, character segmentation, OCR (Optical Character Recognition) and post processing. The application containers include files, environment variables and libraries necessary for running the required software.

A. MIGRATION APPROACH
The migration process has two entities, a source, which is the host that initially has the container and a sink, the container receiver. To transition from the source to destination, container migration undergoes certain phases to offer uninterrupted services to users in the new target node environment. The migration phases and processes are designed to incur negligible downtime by the end user, similar to [57]. The phases include both pre-copy and post-copy stages to minimize the total migration time. These phases are elaborated below.

1) DECISION AND CONTROL PHASE
The first step for migration is to decide if a container is under-performing relative to the service requirements. In such circumstances, these containers are listed for migration. Before the system can decide migration, control plane information exchange is performed. A low overhead control plane protocol is used to exchange both routing and computing state information between edge cloud clusters in the region. Upon selecting the container to be migrated, neighboring edge clouds are queried using an extended inter-domain protocol such as EIR (edge-aware interdomain routing). This protocol sends control message packets along with the corresponding link information via a technique called telescopic flooding. The nodes distant from the source receive the control information less frequently as compared to the nearer ones. This type of control message flooding technique ensures that every network has a global view of the topology, but with control overheads enabling quicker information exchanges [58].

2) PREPARATION PHASE
In this phase, the first base image of the container is committed at the source and the destination is confirmed by network handshakes.

3) PRE-COPY PHASE
Prior to the containers awaiting transition to a best fit destination, memory snapshots are synchronized and sent to the probable destination nodes without interrupting service. After the migration request has been triggered, pages are sent in turns and are compared with each other, at the source and target nodes. Initially all pages are marked and sent to the target node. During this process, the changes occurred at the source container pages are marked and only these pages are transferred to the destination. This process is called creating dirty pages.

4) STATE TRANSFER PHASE
The above process continues until container’s minimum number of snapshots, in the source, at different time slots, are checkpointed and sent. After this, the container is initiated at the destination. This process continues until a considerable amount of dirty memory pages have been accumulated at the target server, leading to stopping the container at the source with a final commit to the destination.
5) REDO (POST-COPY) PAGING PHASE

After stopping the container at the source, the most recent memory and files are sent over to the destination which is restored with most recent runtime states. The erroneous dirty pages are resent from the source following the iterative process in the “Pre-copy phase”.

After undergoing all the migration phases, the container is restored in the target node with the current run time states and the user starts receiving service from the target edge cloud node. This migration approach is a framework to adaptively find memory fraction in the destination to resume better functioning.

B. MIGRATION FLOW

Figure 6 shows the software stack at each MEC node for the distributed migration flow by placing control and decision logic. The containers running at the Linux user-space are orchestrated with the ability to provide resource as well as performance metrics to the underlying modules [59]. The resource tracker module queries and assesses the containers as well as the host Kernel for parameters such as compute, memory, and storage in the host as well as neighboring nodes. The resource controller is the experiment specific module to modify the inter-edge bandwidth and load parameters of the node. The migration decider module takes feedback from the resource tracker and controller to determine the possible destination edge cloud nodes for the under-performing containers at the source node. Upon accumulating this information from other nodes it sends to the migration controller block. Finally, the migration controller module coordinates all the phases described in the migration approach in Section V-A, enabling information exchange among the distributed edge cloud nodes for the number of hosted containers, and running status.

C. CONTAINER MIGRATION SYSTEM EVALUATION

Using the test-bed set-up described above, migration cost parameters: pre-copy, migration, and post-copy time, with respect to machine type, network bandwidth and container size, are measured. In order to assess migration requirement and cost, different size containers (0.6-4.6GB) running ALPR application are exchanged between two similar test nodes by varying the processing speed. Figure 7 shows the total migration time ($t_{mt}$) which includes pre-copy, migration and post-copy time. The migration time for fixed sized containers and given inter-edge bandwidth remains the same. The pre-copy and post-copy time are inversely proportional to the processing speed since it is also shared by the containers for the application specific computations.

Evaluating the impact of processing speed, by hosting fixed size containers, we measured the total migration time considering the tenure elapsed at pre-copy, migration and post-copy phases. A 0.6 GB linux LXD container thriving on the resources of an intel core i5 node with 1.40 GIPS has a lower performance in comparison to the ones catered by hosts with higher processing capabilities. The pre-copy and post-copy time vary considerably with higher processing speed (2.93 GIPS and 3.40 GIPS) while the migration time continues to be constant for all the same size containers since it is independent of the processor and is affected by the available bandwidth between the nodes. The bandwidth is kept constant in Figure 7 to analyze the compute resource effect on the pre-copy and post-copy latency. Similar is the observation for the containers sizes of 1.4 GB, 2.4 GB and 4.8 GB.

VI. SIMULATION SET-UP

We scale-up the emulation to carry out a large scale simulation modeling the SFO topology (Figure 5) consisting of nine edge cloud nodes and 536 taxicab users as shown in Figure 8. This section details the simulation scenario and the numerical values for the parameters described in Table 3 and 4. The simulation is carried out in the following phases: (a) without migration technique, (b) migration without using our proposed approach, and (c) migration with the proposed approach. Finally, the migration cost is presented for the system.
A. SIMULATION SCENARIO

Initially, the system performance is evaluated without migration using following baseline approaches.

**Constrained Load (CL):** The user requests are equally divided among the heterogeneous edge cloud nodes, routing a fixed number of requests based upon user vicinity. The remaining requests are routed to the next closest node and so on.

**Nearest-Edge (N):** The user requests are always routed to the closest node irrespective of the node’s current load. It can be noted that this creates a load imbalance on the system.

Figure 9 compares the above two load scenarios for the deployed ECs. The EC response time consists of the processing delay and the queuing delay at a node, which rises with the load in both the cases. Contrary to the constrained load case, nearest edge case has high variability in response time due to the different load conditions at each EC node. For instance, EC2 is lightly loaded, EC1 and EC6 are medium loaded whereas EC8 is highly loaded as noted in Figure 8, when UEs are connected as per the nearest-edge scheme. In both the cases, the response time delay arising due to the highly load condition necessitates migration.

Container migration is analyzed using various parametric variation as explained below.

**Bandwidth-only:** The users are connected at the nearest edge cloud node and the migration is triggered based on the application QoS and decided based upon the available inter-edge cloud bandwidth.

**Processing-only:** Similar to the Bandwidth-only case, here the migration is triggered based on the application QoS and decided based upon the destination node resources (compute capability).

The approach presented in Section IV using the algorithms (1, 2, and 3) is collectively referred to as ShareOn. The following cases which is based upon initial user allocation at the EC are evaluated for ShareOn.

**ShareOn-CL:** Here, system users are initially allocated according to the Constrained-Load scheme and later migrated using ShareOn.

**ShareOn-N:** Here, the system users are initially allocated according to the Nearest-Edge scheme and later migrated using ShareOn.

In all these cases, user mobility is introduced and load is varied by injecting multiple requests per taxicab. Our proposed approach, ShareOn, can be instantiated from the nearest or the constrained load scenario considering their first connected request as the initial state.

B. SIMULATION PARAMETERS

The numerical values for the simulation are listed in Table 4. The mobility pattern of all the users is known and the load is varied from 0.1 to 1 by initiating multiple requests from a taxicab. Where, 0.1 load implies 536 requests being serviced by the nine edge cloud nodes, 0.2 load implies 2x requests and subsequent load requests are similarly incremented.

VII. RESULTS AND DISCUSSION

The results obtained from the simulation model introduced in the previous section are given here.

A. SYSTEM PERFORMANCE

Figure 10 compares our migration approach, ShareOn with no migration cases. In the Constrained Load case, the average system response time at load 0.1 is low as compared to the other approaches since the service requests have been equally distributed among nine edge cloud nodes. However, as load gradually increases, the average system response time starts degrading. Without any optimization this approach cannot handle the volume of requests with the available resources at each node. While in the case of the Nearest Edge (users connect to the closest MEC) the average system response time is prominent. The reason for this rise is that in the real-time mobile taxicab trace, most of the users connect to the North-East edge cloud node due to their close physical proximity to that node. The limited resources at that node

---

**TABLE 4. Simulation parameters.**

| Parameter | Value | Parameter | Value |
|-----------|-------|-----------|-------|
| $s_p$     | 2.2-3.9 Gbps | $b_i$     | 10-100 Gbps |
| $r_{rd}$  | 2.5-4 kpps  | $s_{page}$ | 4-64 KB |
| $m$       | 8-32GB      | $load$    | 0-1   |
| $s_{con}$ | 0.5-4.6 GB | #taxi     | 536   |
are unable to support all the connected users and hence the system enters into overload resulting in a large average system latency.

Using ShareOn − CL, initially there are not many migrations since all the requests are well distributed among the nine EC nodes. On increasing the load, the system response is better than the non-migration Constrained-Load approach, as the load gets efficiently distributed across the geographical regions. This is due to considering processing capability, inter-edge bandwidth and network latency of each node while deciding resource sharing. In the case of ShareOn − N, the users are initially (low load condition) connected at the nearest EC where the available resources are exhausted resulting in a slight increase in the system response time.

B. ShareOn VS. OTHER MIGRATION APPROACHES

Figure 11 compares ShareOn with bandwidth-only and processing-only migration methods. The former performs better at lower loads as the migration time dominates the pre-copy and post-copy time for the fewer container migrations. As the bandwidth-only approach keeps track of bandwidth before initiating migration, the average system latency does not suffer from the migration time factor. For the higher load scenario, only considering bandwidth is not sufficient as the migration time dominates the pre-copy and post-copy time, thereby increasing the total system latency. In either case, ShareOn performs significantly better than these single metric approaches.

C. ShareOn PERFORMANCE WITH INITIAL USER DISTRIBUTION

Figure 12 presents the cumulative distribution function (CDF) comparison for different initial user distributions with and without ShareOn. At the higher load values, ShareOn − CL provides better performance than the Constrained Load as the former redistributes the containers across the edge network depending upon the available inter-edge bandwidth and load at the destination EC node. This redistribution using Constrained Load scheme is similar to the containers initially distributed using the Nearest Edge scheme for both the load values shown. In either case, ShareOn − N redistributes and finds optimal locations for the containers and thus at both the load values, ShareOn − N performs better.

D. IMPACT OF CONTAINER RESOURCE REQUIREMENT ON MIGRATION

In our simulation, the average processing requirement is reduced to depict the heterogeneity of the application running in the containers. At the lower load region, this reduction do not affect the average system response because there are less number of migrations being undertaken at the nodes due to sufficient resource availability. Hence, from Figure 13 we observe that the system response for ShareOn − CL and Shareon − CL(reducedprocessing) are initially the same, till load instance 0.2. When the load of the system increased it can be noted that the number of containers listed to be migrated also increased because the resource requirement for each container is less and resource availability at each node is more giving rise to more containers qualifying to be migrated. In the load region 0.3 onward ShareOn − CL system performance degrades and the observed latency rises. On the contrary Shareon − CL(reducedprocessing) improves to the extent that a gain of 22% system response can be observed.

In Shareon − N(reducedprocessing), the average system response time is low for the lower load scenario in comparison to Shareon − N, because the users connect to their closest respective EC nodes and number of users serviced at their source EC node are more due to reduced processing requirement. Shareon − N(reducedprocessing)
performs close to Shareon − N from load 0.6 onwards because the scope of finding resources at the source EC node is reduced and the resources at the other destination node are also exhausted. For Shareon − CL, at lower load, containers with reduced processing requirements are sustained at the source EC node and thus avoiding the need of migration. However, as the load increases, the reduced resource requirement of containers is fulfilled by migrating them to a better destination. Thus, the container processing requirement has a substantial impact on the performance of migration schemes.

Alongside the reduced processing requirements, the RAM requirement is increased by 40% to analyze system performance resembling the stateful application which demands more RAM to store the previous session’s data. Figure 14 shows that at load 0.1 (536 user requests) Shareon − CL has better average system response time because most of the containers are serviced at the source node due to availability of the required resources and thus numbers of containers to be migrated at bandwidth 0.1 Gbps is way less than Shareon − N. On the contrary, for Shareon − N initially the users are connected to the nearest edge cloud, due to resource constraints at the source edge node, the list of containers to be migrated at 0.1 Gbps is large, which increases the average system response time to 76 ms. As the bandwidth increases the average system response time reduces (noticeably more for Shareon − N) since more and more listed containers gets the opportunity to migrate to a better destination EC node. It is observed that bandwidth 1 Gbps onwards, the average system response time stabilizes, this is because all the containers enlisted for migration are sent to their respective best fit destination node overcoming the bandwidth constraints. Similar is the observation at load region 0.5, the inter-edge bandwidth of (> =1 Gbps) is sufficient to allow migration of all the containers. It is noticed that the average system response time of Shareon − CL is almost equal to Shareon − N because all the nodes are overwhelmed with 5× more requests as compared to load 0.1.

F. MIGRATION COST

Figure 16 depicts the local migration cost of individual node and shared migration cost between the pair of nodes, along with the average inter-edge bandwidth. It is observed that the LMC depends only on a node’s local resources and therefore it grows significantly with the number of containers (incoming as well as outgoing) for their pre-copy and post-copy requirements. The SMC, along with the number of containers, also considers the network parameters such as inter-edge bandwidth. For instance, while EC7 has higher LMC and lower BW thereby rising its SMC, EC3 has lower SMC owing to its higher inter-edge bandwidth and lower LMC.

Global Migration Cost: In a distributed system such as the one presented in this paper, measuring the global system cost is infeasible due to lack of centralized entity. Nevertheless, from the simulation, we observed that the GMC, which is the maximum sum of all the pairwise local and shared
cost, provides us intuitions for system-level optimization. For example, a node which has highest LMC + SMC is the one which is either highly loaded or has least average available inter-edge bandwidth or both. Similarly, for a node whose LMC + SMC is low can be a good candidate EC to migrate containers. Thus, the GMC analysis can assist in capturing the global system parameters and optimize the process of container destination selection. The discussion about global optimization is out of scope of this paper. Our project code is available at [60].

VIII. RELATED WORK

Nadgowda et al. discusses the Voyager framework, a live container migration service designed according to Open Container Initiative (OCI) principles. Voyager combines CRIU based memory migration along with data federation to reduce migration downtime [61]. This work focuses on optimal state migration solution for a containerized application and ensures that the application’s runtime state is correctly restored. A comparative study of Container (LXD [62] / Docker [63]) and VM migration is presented in [61], similar to [64]. An overview of state-of-the-art migration techniques including cold, pre-copy, post-copy, and hybrid migrations is presented in [65]. A comprehensive performance evaluation characterizes migration techniques with prolonged total migration time. The authors in [27] propose a migration framework by leveraging layered storage on Docker platforms for mobile clients to achieve low end-to-end latency. The limitation is that this paper does not consider the load status of the destination edge server for the service handoff via migration. Hawilo et al. proposed a solution centered on a specific type of VMs performing NFV functionalities, assuming all VMs are housed within a data center [66]. This study focuses on an integer programming (IP) optimization model orchestrator. The framework facilitates the placement of the virtual network functions (VNFs) considering constraints such as inter-container relations and service function chain (SFC) delays. Barbalace et al. focuses on heterogeneous container migration method for natively-compiled containerized applications across compute nodes with differing instruction Set Architectures [67]. The migration schemes address the issues of stateful services and incurs negligible overhead during migration. [68] presents a classic linear assignment algorithm using computational nodes and optimizes the VM placement so that the impact of data access latencies on completion times for data intensive cloud applications is minimized. Kaur et al. addressed the optimization problem by considering SDN-based edge-cloud interplay. This scheme based on delay and energy decide whether or not to offload the flow to the edge. The framework operates by assigning workload based on weights and computation capacities of the VMs [69]. This approach of maintaining the application QoS by cooperation of cloud and edge did not take into account network operation, failures, and containers’ dependencies.

The related work and other existing literature either studies VM or implements container migration without explicitly taking the container specific parameters e.g. dynamic resource allocation (available processing speed, RAM and bandwidth) and container size, into account. Some studies focus on techniques of migration without analysing the effect of EC node resources, load or network variability which directly impacts the decision of efficient migration. The migration cost of a heterogeneous system is a complex combination of local and remote computing and network transport resources. System load, available processing resource and inter-node bandwidth affect the total migration time. Each node can decide, initiate and control migration of its containers in a distributed manner. Furthermore, considering the above mentioned parameters to simulate distributed container migration to test feasibility in a city-scale scenario is still unexplored.

A. LIMITATIONS

In order to select best fit destination this work did not focus on power consumption of the EC node as a parameter for the migration decision making. Considering power as a metric would allow power estimation of CPU cores before migrating the container allowing simultaneous optimization of energy consumption, migration counts, and SLA violation.

B. CHALLENGES

For migration of containers this work attempted to keep the downtime significantly low. In order to achieve reduced container downtime, we adopted logging and replay approach. In some scenarios the dirty page producing rate was higher than the transfer and update rate, leading to a prolonged iteration stage. To overcome this challenge pre-copying approach based on time series was adopted which included (i) copy the image file, (ii) iterative log and replay, and (iii) stop and resume.

C. FUTURE WORK

We plan to evaluate ShareOn on alternative networks and compute-aware algorithms in an edge-cloud enabled realistic outdoor testbed such as COSMOS [71]. We plan to evaluate ShareOn on alternative networks and compute-aware algorithms in an edge-cloud enabled realistic outdoor testbed such as COSMOS [65]. For future work we aim to
adopt an SDN infrastructure using OpenFlow controllers to evaluate migration benefits for 5G URLLC use cases and analyze the system response time, network and computational requirements.

IX. CONCLUSION

This paper has proposed – ShareOn – a distributed shared resource framework for container migration, enabling edge cloud nodes to support system load, heterogeneity, and network fluctuations. ShareOn is evaluated for the real-time services deployed at ORBIT radio grid testbed, and is then simulated for San Francisco city based large-scale edge cloud network and taxicab’s mobility. The system response time and migration cost are evaluated by executing a low-latency application (ALPR) using different algorithms running at individual edge cloud nodes. In this work we determine number of migrations initiated at each edge cloud node to dynamically redistribute the resources. Also, the system performance is analyzed for various load conditions and container resource requirements.

ShareOn is optimized with parameters such as network latency, edge cloud resources, edge cloud load, and inter-edge bandwidth for better application QoS. The system performance of ShareOn is compared with two non-migration approaches: equal-load and nearest-edge, and two migration-based approaches: bandwidth-only and processing-only. From our experiments, we make following observations: (a) migration is a viable approach when sufficient computation (at source and destination) and inter-edge bandwidth are available, (b) processing-only and bandwidth-only approaches fail to lower the average system response time at higher load as compared to the the multi-parameter ShareOn, (c) varying resource requirements for a container, affects the number of migrating containers thereby impacting application performance, and (d) the migration cost analysis provides an overview of the system overhead and selecting destinations for migration.

REFERENCES

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, “Mobile edge computing: A survey,” IEEE Internet Things J., vol. 5, no. 1, pp. 450–465, Feb. 2017.
[2] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, “Scalability and performance evaluation of edge cloud systems for latency constrained applications,” in Proc. 3rd IEEE/ACM Symp. Edge Comput. (SEC), Oct. 2018, pp. 286–299.
[3] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, “Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges,” IEEE Commun. Mag., vol. 55, no. 4, pp. 54–61, Apr. 2017.
[4] M. Chen and Y. Hao, “Task offloading for mobile edge computing in software defined ultra-dense network,” IEEE J. Sel. Areas Commun., vol. 36, no. 3, pp. 587–597, Mar. 2018.
[5] T. Zhao, S. Zhou, X. Guo, and Z. Niu, “Tasks scheduling and resource allocation in heterogeneous cloud for delay-bounded mobile edge computing,” in Proc. IEEE Int. Conf. Commun. (ICC), May 2017, pp. 1–7.
[6] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, “Edge computing: A survey,” Future Gener. Comput. Syst., vol. 97, pp. 219–235, Aug. 2019.
[7] F. E. F. Samann, S. R. M. Zeebaree, and S. Askar, “IoT provisioning QoS based on cloud and fog computing,” J. Appl. Sci. Technol. Trends, vol. 2, no. 1, pp. 29–40, Mar. 2021.
[8] C. Cicconetti, M. Conti, and A. Passarella, “Low-latency distributed computation offloading for pervasive environments,” in Proc. IEEE Int. Conf. Pervasive Comput. Commun. (PerCom), Mar. 2019, pp. 1–10.
[9] J. Zhao, Q. Li, Y. Gong, and K. Zhang, “Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks,” IEEE Trans. Veh. Technol., vol. 68, no. 8, pp. 7944–7956, Aug. 2019.
[10] H.-C. Hsieh, C.-S. Lee, and J.-L. Chen, “Mobile edge computing platform with container-based virtualization technology for IoT applications,” Wireless Pers. Commun., vol. 102, no. 1, pp. 527–542, Sep. 2018.
[11] H. Hawilo, M. Jammal, and A. Shami, “Exploring microservices as the architecture of choice for network function virtualization platforms,” IEEE Netw., vol. 33, no. 3, pp. 202–210, Mar./Apr. 2019.
[12] B. Liu, M. A. Gregory, and S. Li, “Multi-access edge computing fundamentals, services, enablers and challenges: A complete survey,” J. Netw. Comput. Appl., vol. 199, Mar. 2021, Art. no. 103308.
[13] S. Maheshwari, S. Choudhury, I. Seskar, and D. Raychaudhuri, “Traffic-aware dynamic container migration for real-time support in mobile edge clouds,” in Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS), Nov. 2018, pp. 1–6.
[14] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, “Edge-centric computing: Vision and challenges,” ACM SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, 2015.
[15] L. Loven, E. Peltonen, E. Harjula, and S. Pirttikangas, “Weathering the reallocation storm: Large-scale analysis of edge server workload,” in Proc. Int. Joint Eur. Conf. Netw. Commun. 6G Summit (EuCNC/6G Summit), Jun. 2021, pp. 317–322.
[16] C. Westphal, “Challenges in networking to support augmented reality and virtual reality,” in Proc. Int. Conf. Comput. Netw. Commun. (ICNC), 2017.
[17] H. T. Din, C. Lee, D. Niyato, and P. Wang, “A survey of mobile cloud computing: Architecture, applications, and approaches,” Wireless Commun. Mobile Comput., vol. 13, no. 18, pp. 1587–1611, Dec. 2011.
[18] A. Alomarni, S. K. Subramaniam, N. Samian, R. Latip, and Z. Zakarain, “Resource management in SDN-based cloud and SDN-based fog computing: Taxonomy study,” Symmetry, vol. 13, no. 5, p. 734, Apr. 2021.
[19] H. Wu, J. Chen, C. Zhou, W. Shi, N. Cheng, W. Xu, W. Zhang, and X. S. Shen, “Resource management in space-air-ground integrated vehicular networks: SDN control and AI algorithm design,” IEEE Wireless Commun., vol. 27, no. 6, pp. 52–60, Dec. 2020.
[20] M. A. Assef, M. M. Mohamad, and A. A. Al-Roubaiye, “Toward adaptive and scalable OpenFlow-SDN flow control: A survey,” IEEE Access, vol. 7, pp. 107346–107379, 2019.
[21] F. Bronzino, S. Maheshwari, I. Seskar, and D. Raychaudhuri, “NOVN: Named-object based virtual network architecture,” in Proc. 20th Int. Conf. Distrib. Comput. Netw., Jan. 2019, pp. 90–99.
[22] M. Gharbouei, C. Contoli, G. Cavofigli, G. Cuffaro, B. Martini, F. Paganelli, W. Cerroni, F. Cappanera, and P. Castoldi, “Demonstration of latency-aware and self-adaptive service chaining in 5G/SDN/NFV infrastructures,” in Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN), Nov. 2018, pp. 1–2.
[23] J. Zhang, Z. Wang, N. Ma, T. Huang, and Y. Liu, “Enabling efficient service function chaining by integrating NFV and SDN: Architecture, challenges and opportunities,” IEEE Netw., vol. 32, no. 6, pp. 152–159, Nov. 2018.
[24] M. M. Ahmed, A. E. Ebrahim, and W. R. Anis, “Efficient placement of service function chains in cloud computing environments,” Electronics, vol. 10, no. 3, p. 323, Jan. 2021.
[25] A. Abelfattah, V. Barra, N. Beams, R. Bleie, J. Brown, J. S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, and P. Fischer, “GPU algorithms for efficient exascale discretizations,” Parallel Comput., vol. 108, Dec. 2021, Art. no. 102841.
[26] B. Kasap and A. J. van Opstal, “Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations,” Neurocomputing, vol. 302, pp. 55–65, Aug. 2018.
[27] L. Ma, S. Yi, and Q. Li, “Efficient service handoff across edge servers via Docker container migration,” in Proc. 2nd ACM/IEEE Symp. Edge Comput., Oct. 2017, pp. 1–13.
[28] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K. Choo, and M. Dlodlo, “From cloud to fog computing: A review and a conceptual live VM migration framework,” IEEE Access, vol. 5, pp. 8284–8300, 2017.
E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," Concurrency Comput. Pract. Exp., vol. 32, no. 17, p. e5668, Sep. 2020.

B. Iyram, "Principles of container-based application design," Redhat Consulting, Raleigh, NC, USA, White Paper, 2017. [Online]. Available: https://d3hjxemb1n7eqz.cloudfront.net/user/c1-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf

K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog/nano data centers," IEEE Wireless Commun., vol. 24, no. 3, pp. 48–56, Jun. 2017.

Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT). Accessed: Jan. 4, 2022. [Online]. Available: http://www.orbit-lab.org/

CRIU. Accessed: Jan. 20, 2022. [Online]. Available: http://www.criu.org/
opentrace/opentrace. Accessed: Jan. 20, 2022. [Online]. Available: https://github.com/openotrace/openotrace/wiki/OpenALPR-Design

M. Piorkowski. CRAWDAD Dataset Eplt/Mobility. Accessed: Jan. 20, 2022. [Online]. Available: https://crawdad.cs.dartmouth.edu/eplt/mobility/20090224

R. S. Sinha, Y. Wei, and S.-H. Hwang, "A survey on LPWA technology: LoRa and NB-IoT," IET Exp., vol. 3, no. 1, pp. 14–21, Mar. 2017.

F. Al-Turjman. "Information-centric framework for the Internet of Things (IoT): Traffic modeling & optimization," Future Gener. Comput. Syst., vol. 80, pp. 63–75, Mar. 2018.

S. H. Rastegar, A. Abbasfar, and V. Shah-Mansouri, "Rule caching in SDN-enabled base stations supporting massive IoT devices with bursty traffic," IEEE Internet Things J., vol. 7, no. 9, pp. 8917–8931, Sep. 2020.

H. S. Jang, H. Jin, B. C. Jung, and T. Q. S. Quek, "Resource-optimized recursive access class barring for bursty traffic in cellular IoT networks," IEEE Internet Things J., vol. 8, no. 14, pp. 11640–11645, Jul. 2021.

A. Ali-Eldin, J. Westin, B. Wang, P. Sharma, and P. Shenoy, "SpotWeb: Running latency-sensitive distributed web services on transient cloud servers," in Proc. 28th Int. Symp. High Perform. Parallel Distrib. Comput., Jan. 2019, pp. 1–12.

Y. Li, J. Zhang, C. Jiang, J. Wan, and Z. Ren, "PINE: Optimizing performance isolation in container environments," IEEE Access, vol. 7, pp. 30140–301422, 2019.

C. Pahl, "Containerization and the PaaS cloud," IEEE Cloud Comput., vol. 6, no. 3, pp. 24–31, May/Jul. 2016.

Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu, "Lessons learned from migrating complex stateful applications onto serverless platforms," in Proc. 12th ACM SIGOPS Asia-Pacific Workshop on User Space Application, Aug. 2021, pp. 89–96.

Space User Application. User Space Application—An Overview ScienceDirect Topics. Accessed: Jan. 20, 2022. [Online]. Available: https://www.sciencedirect.com/topics/computer-sciences/user-space-application

SDXCentral. What Is a Virtual Network Function or VNF? Accessed: Jan. 20, 2022. [Online]. Available: https://www.sdxcentral.com/networking/vnf/dfinitions/virtual-network-function/

Github—Finding Line Lunes Using Python and OpenCV. Accessed: Jan. 7, 2022. [Online]. Available: https://github.com/nakshibhuaya/car-finding-line-lined

Github Credit Card Number Recognition. Accessed: Jan. 11, 2022. [Online]. Available: https://github.com/tonyquiry/Credit-Card-Number-Recognition

Z. Yu, L. Jinhai, G. Guochang, Z. Rubo, and Y. Haiyan, "An implementation of evolutionary computation for path planning of cooperative mobile robots," in Proc. 4th World Congr. Intell. Control Autom., 2002, pp. 1798–1802.

R. Stoyanov and M. J. Kellinghaupt, "Efficient live migration of Linux containers," in Proc. Int. Conf. High Perform. Comput., in Lecture Notes in Computer Science (LNCS), vol. 11203. Springer, 2018, pp. 184–193.

S. Maheshwari, P. Netalkar, and D. Raychaudhuri, "DISCO: Distributed control plane architecture for resource sharing in heterogeneous mobile edge cloud scenarios," in Proc. IEEE 48th Int. Conf. Distrib. Comput. Syst. (ICDCS), Nov. 2020, pp. 519–529.

B. Xu, S. Wu, J. Xiao, H. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards efficient live migration of Docker containers," in Proc. IEEE 15th Int. Conf. Cloud Comput. (CLOUD), Oct. 2020, pp. 321–328.

S. Mukherjee, S. Striam, T. Yu, and D. Raychaudhuri, "EIR: Edge-aware inter-domain routing protocol for the future mobile internet," Comput. Netw., vol. 127, pp. 13–30, Nov. 2017.

D. Bernstein, "Containers and cloud: From XLC to Docker to Kubernetes," IEEE Cloud Comput., vol. 1, no. 3, pp. 81–84, Sep. 2014.

S. Maheshwari. Sumitee87/shareon. Accessed: Jan. 15, 2022. [Online]. Available: https://github.com/sumitee87/shareon

S. Nadgouda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS), Jun. 2017, pp. 2137–2142.

The LXD Container Hypervisor. Accessed: Feb. 2, 2022. [Online]. Available: https://www.linuxcontainers.org/lxd

Docker. Accessed: Jan. 25, 2022. [Online]. Available: https://www.docker.com/

A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salondis, "Live service migration in mobile edge clouds," IEEE Wireless Commun., vol. 25, no. 1, pp. 140–147, Feb. 2018.

C. Pulfaito, C. Vallati, E. Mingozza, G. Merlini, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," Sensors, vol. 19, no. 7, p. 1488, Mar. 2019.

H. Havilio, M. Jammal, and A. Shami, "Orchestrating network function virtualization platform: Migration or re-instantiation?" in Proc. IEEE 6th Int. Conf. Cloud Netw. (CloudNet), Sep. 2017, pp. 1–6.

A. Barbalace, M. L. Karouei, W. Wang, T. Xing, P. Olivier, and B. Ravindran, "Edge computing: The case for heterogeneous-ISA container migration," in Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environment, Mar. 2020, pp. 73–87.

M. Alichey and T. V. Lakshman, "Optimizing data access latencies in cloud systems by intelligent virtual machine placement," in Proc. IEEE INFOCOM, Apr. 2013, pp. 647–655.

K. Kaur, S. Garg, G. S. Aujla, N. Kumar, J. J. P. C. Rodrigues, and M. Guizani, "Edge computing in the industrial Internet of Things environment: Software-defined-networks-based edge-cloud interplay," IEEE Commun. Mag., vol. 56, no. 2, pp. 44–51, Feb. 2018.

R. Ramaswamy, N. Weng, and T. Wolf, "Characterizing network processing delay," in Proc. IEEE Global Telecommun. Conf. (GLOBECOM), Sep. 2004, pp. 1629–1634.

D. Raschchadni, F. Seskar, G. Zussman, T. Korakis, D. Kilper, T. Chen, J. Kolodziejksi, M. Sherman, Z. Kostic, X. Gu, H. Krishnaswamy, S. Maheshwari, P. Skrimponis, and C. Gutterman, "Challenge: COSMOS: A city-scale programmable testbed for experimentation with advanced wireless," in Proc. 26th Annu. Int. Conf. Mobile Comput. Netw., Apr. 2020, pp. 1–13.

SHALINI CHOUDHURY received the B.Tech. degree in electrical engineering from Nagpur University, India, in 2011, and the M.S. degree from CUNY, The City College of New York, USA, in 2016. She is currently pursuing the Ph.D. degree in electrical and computer engineering at Rutgers University, New Jersey, USA. She has previously worked with the Interdigital’s Research and Innovation Team and the Accenture Laboratory. Her research interests include wireless communications and networking specifically low latency mobility support for emerging 5G and beyond networks.
SUMIT MAHESHWARI (Member, IEEE) received the bachelor’s degree in electronics and communication engineering from Dr. M. G. R. University, India, the master’s degree in wireless communications from IIT Kharagpur, India, and the Ph.D. degree in electrical and computer engineering from WINLAB, Rutgers University, in 2020. He is currently a Software Engineer at Microsoft. He also held various positions at Affirmed Networks, Nokia Bell Laboratories, AT&T, and Samsung earlier. His research interests include computer networks and wireless communications, with specific focus on edge clouds and virtual networks. He is an ACM Member.

IVAN SESKAR (Senior Member, IEEE) is currently the Chief Technologist of the WINLAB, Rutgers University, for experimental systems and prototyping projects. He is also the Program Director of COSMOS Project responsible for the New York City NSF PAWR Deployment, the PI of the NSF GENI Wireless Project, which resulted in campus deployments of LTE/WiMAX base stations at several U.S. universities, and the PI for the NSF CloudLab deployment at Rutgers. He has been the Co-PI and the Project Manager for all three phases of the NSF-supported ORBIT mid-scale testbed project at WINLAB, successfully leading technology development and operations since the testbed was released as a community resource, in 2005, for which the team received the 2008 NSF Alexander Schwarzkopf Prize for Technological Innovation. He is a Co-Chair of the IEEE Future Networks Testbed Working Group, a member of ACM, and the Co-Founder and the CTO of Upside Wireless Inc.

DIPANKAR RAYCHAUDHURI (Life Fellow, IEEE) received the B.Tech. degree (Hons.) from IIT Kharagpur, in 1976, and the M.S. and Ph.D. degrees from SUNY, Stony Brook, in 1978 and 1979, respectively. He is currently a Distinguished Professor in electrical and computer engineering and the Director of the Wireless Information Network Laboratory (WINLAB), Rutgers University. As the WINLAB’s Director, he is responsible for an internationally recognized industry-university research center specializing in wireless technology. He has worked as the Principal Investigator for several large multi-institutional U.S. National Science Foundation grants, including the “ORBIT” wireless testbed and the “MobilityFirst” future Internet architecture (FIA) project, and is also leading the “COSMOS” project under the NSF Platforms for Advanced Wireless Research (PAWR) Program. He has previously held corporate research and development positions, including the Chief Scientist of Iospan Wireless (2000–2001), the Assistant General Manager and the Department Head of the NEC Laboratories (1993–1999), and the Head of Broadband Communications at Sarnoff Corporation (1990–1992). He was a recipient of several professional awards, including the Rutgers School of Engineering Faculty of the Year Award, in 2017, the IEEE Donald J. Fink Award, in 2014, the Indian Institute of Technology, Kharagpur, the Distinguished Alumni Award, in 2012, and the Schwarzkopf Prize for Technological Innovation, in 2008.