Received July 18, 2021, accepted August 15, 2021, date of publication August 17, 2021, date of current version August 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3105583

FAST: Flexible and Low-Latency State Transfer in Mobile Edge Computing

TUNG V. DOAN[1], GIANG T. NGUYEN[2,3], MARTIN REISSLEIN[4], (Fellow, IEEE), AND FRANK H. P. FITZEK[1,3], (Senior Member, IEEE)

[1]Deutsche Telekom Chair of Communication Networks, Technische Universität Dresden, 01062 Dresden, Germany
[2]Chair of Haptic Communication Systems, Technische Universität Dresden, 01062 Dresden, Germany
[3]Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, 01062 Dresden, Germany
[4]School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287, USA

Corresponding author: Martin Reisslein (reisslein@asu.edu)

This work was supported in part by German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy–EXC 2050/1–Cluster of Excellence, Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, under Project 390696704, and in part by German Ministry for Research and Education (BMBF, Bundesministerium für Bildung und Forschung) as part of Germany's High-Tech Strategy–FKZ under Grant 16KIS0956K.

ABSTRACT Mobile Edge Computing (MEC) brings the benefits of cloud computing, such as computation, networking, and storage resources, close to end users, thus reducing end-to-end latency and enabling various novel use cases, such as vehicle platooning, autonomous driving, and the tactile internet. However, frequent user mobility makes it challenging for the MEC to guarantee the close proximity to the users. To tackle this challenge, the underlying network has to be capable of seamlessly migrating applications between multiple MEC sites. This application migration requires the quick and flexible migration of the application states without service interruption, while minimizing the state transfer cost. In this article, we first study the state transfer optimization problem in the MEC. To solve this problem, we propose a metaheuristic algorithm based on Tabu search. We then propose Flexible and Low-Latency State Transfer in Mobile Edge Computing (FAST), the first programmable state forwarding framework. FAST flexibly and directly forwards states between source instance and destination instance based on Software-Defined Networking (SDN). Both simulation results and practical testbed results demonstrate the favorable performance of the proposed Tabu search algorithm and the FAST framework compared to the state-of-the-art schemes.

INDEX TERMS Application state transfer, multi-access edge computing (MEC), network function virtualization (NFV), software-defined networking (SDN).

I. INTRODUCTION

Mobile Edge Computing (MEC) brings the flexibility and elasticity of cloud computing to run the applications in a close proximity of the end users, e.g., at a base station [2]–[8]. If the MEC can efficiently support the user mobility, then the MEC enables novel use cases, such as vehicle platooning, autonomous cars, and immersive media [9]–[14]. A key challenge for the MEC is to continuously maintain running applications in close proximity of the end users according to their movements across the coverage areas of different base stations. Service migration, i.e., the process of transferring an application from one place to another, requires a consistent operational state before and after the migration. Service migration becomes especially challenging when service downtime has to be minimal so as to maintain seamless operations.

The downtime is caused by the process of saving, transferring, and recovering the application's data during the service migration. Early migration schemes relied entirely on virtualization technologies, such as virtual machines (VM) and containers. In order to accomplish the service migration, the *VM and container migration* schemes transfer: (*i*) the *application states*, i.e., the state information that the application itself generated (mainly the variable values of the counters and timers and other intermediate data that characterize the current functioning status of the application at a given time), and (*ii*) the *VM or container states*, i.e., the state information that the VM or container generated (e.g., the states of the operating system). The VM or container state information is

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

typically an enormous amount of data, resulting in substantial delays for the MEC service migration with VM and container migration schemes [15], [16]. Furthermore, these VM and container migration schemes are not suitable for novel MEC use cases, such as vehicle platooning and tactile internet, which may require manipulations (e.g., splitting or merging) of the application states (e.g., when a truck leaves or rejoins a vehicle platoon). In contrast, with application state transfer in the MEC, which we refer to as *MEC state transfer* for brevity, only the application states are extracted and transferred on demand during the migration. MEC state transfer, which is the focus of this study, can flexibly support novel MEC use cases that require application state manipulations.

A key challenge of MEC state transfer is to minimize the total cost of the state transfer due to the limited MEC resources [17]. In order to constantly maintain the low-latency communication between the MEC applications and the users, frequent service migrations are performed. However, these frequent migrations result in high computation cost (i.e., the hardware resources consumed to host the migrated applications) and high bandwidth consumption (i.e., the network bandwidth used for the state transfers). The existing service migration optimization studies, e.g., [18]–[21], have considered VM or container migration schemes. To the best of our knowledge, the optimization of MEC state transfer has not been examined in detail to date. In this paper, we optimize the MEC state transfer during the service migration. Due to the scarcity of MEC resources, our goal is to minimize the total cost of the state transfer under various constraints, such as the communication delay and network bandwidth. We develop a metaheuristic algorithm based on Tabu search to solve the MEC state transfer optimization problem. The results obtained from the proposed Tabu search algorithm are used to make optimized migration decisions, e.g., where to place the migrated applications and which links to use for transferring the application states.

In order to execute the optimized migration decision, an underlying framework supporting fast state transfer is needed. Several prior studies, e.g., [22]–[24], mainly rely on a centralized controller in an SDN-enabled infrastructure [25], [26] to receive all states from one application instance and to forward the states to another application instance. While the centralized controller simplifies the orchestration of the service migration processes, the controller introduces significant delays and can potentially become a bottleneck when migrating numerous states, resulting in long downtimes during service migration. Some state management schemes directly forward states between application instances, without the support from the underlying infrastructure [27], [28]. However, this direct state forwarding requires heavy modifications of the application behavior and logic so that the applications can manage and transfer the states on their own. Moreover, transferring states without network support is inefficient, especially under high fluctuations of network traffic.

We advocate for the programmable state transfer by introducing FAST, a state forwarding scheme that is both flexible and achieves low latency. FAST achieves low latency by directly forwarding states between application instances. FAST increases flexibility by providing applications with a library implementing operations on states. Furthermore, FAST leverages SDN for state transfer, allowing multi-backup and on-the-fly state management without affecting the rest of the network. Our evaluation results from a practical testbed implementation indicate that FAST typically reduces the migration time to less than half compared to state-of-the-art schemes. Furthermore, FAST is built atop a widely used SDN controller, so as to facilitate quick deployment as an add-on functionality in SDN-enabled networks. To the best of our knowledge, FAST is the first programmable state forwarding scheme. With the focus on the network perspective, FAST can be easily integrated into existing state management frameworks, such as OpenNF [22] and FTM [29].

This paper makes five main contributions:

- In Section III, we formulate the MEC state transfer as an optimization problem that minimizes the migration cost, including the computation cost, communication cost, and buffering cost under various constraints.
- In Section III-G, we propose a low-complexity Tabu search algorithm to minimize the migration cost.
- In Section IV, we propose the Flexible And low-latency State Transfer (FAST) framework for the programmable state transfer in large-scale networks. FAST is readily portable to various frameworks, such as Split/Merge, S6 [30], and Fault-Tolerant MiddleBox (FTMB) [31]. Also, the FAST framework can be utilized to implement the Tabu search minimization of the migration cost from Section III-G in practical MEC systems. An emulation performance evaluation of the FAST framework in a small MEC system (with virtualization-based application instances and software switches) is presented in Section IV-D.
- In Section V, we implement a light-weight simulator (available from https://github.com/openMECPlatform/fast-simulator) to simulate the optimization problem of the service migration, allowing network operators and researchers to quickly evaluate their algorithms on large-scale network topologies. The simulation results indicate that our proposed Tabu algorithm significantly outperforms elementary heuristic algorithms and performs close to the optimum, while incurring much lower computational complexity compared to the optimal solution.
- In Section VI, we implement the FAST framework in a real MEC testbed (with bare metal application execution and hardware switches). The migration cost minimization solutions obtained from the Tabu search algorithm (from Section III-G) can be fed into FAST for the practical deployment of optimized MEC service migration. We make the FAST framework source code publicly available at https://github.com/openMECPlatform/fast.git.

II. BACKGROUND AND RELATED WORK

This section first presents background on state management frameworks and then reviews existing studies on state transfer optimization as well as state management frameworks.

A. BACKGROUND ON STATE MANAGEMENT FRAMEWORKS

The application states represent data generated from handling user requests. The application states are diverse, ranging from simple states, such as connection information in a firewall and address mappings in a network address translator, to complex states, such as scores in a gaming application, user settings in a streaming application, and mobility trajectory information in a self-driving vehicle application. The frequent changes of the application states require a mechanism to control the application state consistency. This consistency requirement should allow an application at a destination instance to resume at the exact running state that it had at the source instance.

Since service migration [27] and failure recovery [31] rely on the application states, differentiating them from each other necessitates two distinct state management schemes. For a service migration, a *move* scheme requires that the state transfer must have been completed before the application is resumed at the destination instance. For a failure recovery, a *copy* scheme requires the source instance to continuously transfer the states to the destination instance until a failure occurs.

The application state management [22] carries out the service migration and failure recovery with four state operations: *export*, *forward*, *import*, and *guarantee*. First, the *export* operation extracts the states from the source instance of a running application. Afterwards, the *forward* operation transfers the application states to a destination instance. Then, the *import* operation inserts the application states into the destination instance. The *guarantee* operation enforces different policies for state transfer and orchestration between state and packet flows. By default, the state transfer is performed without any guarantee policy. The *parallel optimization (PL)* and *loss free (LF)* policies accelerate the state transfer process without the loss of state packets. Meanwhile, an *order preserving (OP)* policy ensures the orderly receipt of the state packets at the destination instance. An *early release (ER)* policy allows the transfer of user data packets to the destination instance during the state transfer, eliminating the need for buffering the user data packets in the network devices.

Due to the wide application diversity, centralized state management frameworks, e.g., [22]–[24], are widely adopted to manage the application states, i.e., receiving the states from one application instance and then forwarding the states to other application instances. A centralized framework provides an abstract interface for external entities (e.g., network operators) to easily perform the state operations. After the state migration has been completed, the user traffic is re-directed to the new application instance. Since this process requires a flexible network, the implementation of the state management typically relies on SDN.

SDN makes forwarding decisions in a separate entity, the so-called SDN controller, which is decoupled from the underlying network devices. Thus, SDN provides a programming interface to automate network configuration. Building on this standard SDN design [32], a state management architecture generally consists of an application plane, a control plane, and a data plane, as illustrated in Fig. 1.

The data plane consists of application instances and SDN-enabled network devices. The application instances are commonly deployed on VMs or containers. An application instance consists typically of a source instance and a destination instance. The source instance contains a running application that serves user requests and generates the states that are later used to resume the application at the destination instance. Meanwhile, the destination instance is the backup of the source instance; the destination instance takes over the role of the source instance (i.e., to serve the user requests) in case of a migration or recovery. The SDN-enabled network devices, e.g., OpenFlow switches, forward the user traffic to the running application instance based on the forwarding rules issued by the SDN controller.

The application plane includes a state control application entity that provides a set of commands to request the state operations, e.g., moving the states from one application instance to another instance.

The control plane interfaces the application plane with the data plane. The control plane provides a northbound interface for the state control application to request the state operations and a southbound interface for data plane elements to perform the operations. A state management framework consisting of a flow manager and an application state manager is built on top of the SDN controller, see Fig. 1. The application state manager processes the requests from the state control application, exchanging messages (e.g., JSON) with the application instances in the data plane to indicate the state operations. The flow manager installs forwarding rules on the network devices via OpenFlow messages to re-direct the user traffic to the new application instance.

A key challenge of MEC state management is to further reduce the latency in order to reduce the service downtime, while performing the state operations across multiple MEC sites. Generally, the application state management is expected to meet four vital MEC requirements [33]:

R1: Low latency. MEC applications are typically latency-sensitive. Therefore, the application state management should achieve fast recovery to avoid service degradation.

R2: Programmability. The dynamic change of the network requires that the application states are flexibly transferred between a running application and its backup. Thus, the state forwarding must be programmable to apply different forwarding rules to the underlying network devices. The MEC can take advantage of the programmability to permit the users to move freely across different networks.

R3: Flexibility. The MEC state management should be able to add or remove backups on the fly without interfering the rest of the network. This plays an important role in the MEC due to the user mobility. Specifically, it is critical to remove the backups in the current serving area as well as to add new backups in the new serving area.

R4: High availability. MEC applications may be prone to failures; thus, providing high availability is important. The state management should allow a running application to have multiple backups, which are distributed over the network.

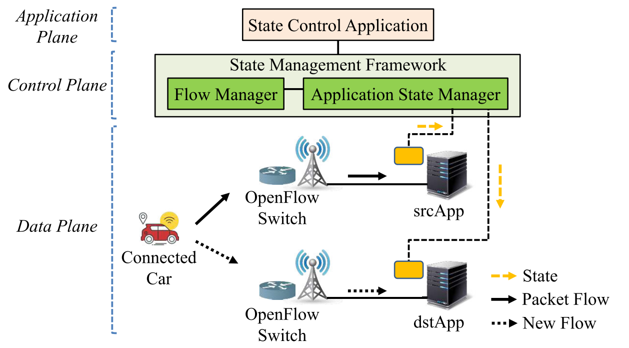


FIGURE 1. Architecture for centralized state management: the application state manager acts as a proxy for the state transfer between source application instance ***srcApp*** and destination application instance ***dstApp***. After all states have been received at ***dstApp***, the flow manager tells the OpenFlow switch to redirect the user traffic (packet flow) to ***dstApp***.

B. RELATED WORK ON STATE TRANSFER OPTIMIZATION

This sections reviews the literature related to the MEC state transfer optimization in Section III. Aside from being beneficial for high-mobility users, service migration introduces considerable costs. Most studies [34], [35] show the impact of the service migration on the migration time, downtime, and the amount of migration data. The migration time is measured from the time instant when the migration starts at the source server to the time instant when the application is resumed at the destination server. The downtime is the time duration during which the service is unreachable during the migration. The amount of migration data is the data volume transferred from the source server in order to recover the application at the destination server.

Due to the complexity of the application design, existing service migration studies have mostly focused on the two virtualization technologies, namely virtual machine (VM) and container, i.e., the existing studies have optimized VM and container migration schemes, see e.g., [36]–[40]. A comprehensive comparison between the migration approaches for VM and container has been conducted in [15]. The migration data is generated by suspending the entire VM or container and saving the memory content into files. Numerous studies have tried to reduce the amount of migration data to decrease the migration time and downtime. Jin *et al.* [41] introduced a compression technique that exploits the high similarity of memory pages and contained zero bytes to minimize network traffic during migration. Svärd *et al.* [42] proposed a dynamic page transfer ordering strategy that minimizes the number of retransmitted pages. Machen *et al.* [43] and Ma *et al.* [44] re-designed the migration data structure based on layering technology, eliminating the duplicated memory content. A strategy to replicate virtual network functions in multiple servers to reduce the need for migrations has been explored in [45], while [46] examined the load threshold that should trigger a migration. An optimized multi-cluster overlay network for the transfer of software instances has been designed in [47], while related hypervisor configuration for the reconfiguration of virtual networks are examined in [48].

To the best of our knowledge, application state transfer (without the VM or container states) has previously only been studied in the data center context in [49]. In particular, the study [49] optimized the migration of flows (application states) among network function instances in a data center. In contrast, we optimize the application state transfer in the MEC context, i.e., we optimize the MEC state transfer, considering MEC servers that are distributed over a network graph and mobile users that connect through distributed access nodes.

C. RELATED WORK ON STATE TRANSFER FRAMEWORKS

This section reviews the literature related to the FAST state transfer framework introduced in Section IV. The existing state transfer (management) studies have typically focused on Network Function Virtualization (NFV), i.e., on managing and transferring the states of virtual network functions, e.g., firewall and load balancer. We review the state transfer frameworks that have been developed for conventional cloud computing and assess whether these frameworks can fulfill the requirements of the MEC state transfer. Also, we review the existing state transfer frameworks for the MEC state transfer.

Recent frameworks allow the application states to be directly transferred between the application instances. The frameworks [31], [50], [51] reduced the network overhead by introducing a compact format for the application states. These studies strive to fulfill the low-latency (R1) requirement, but their designs couple packet processing and state processing when handling user requests, i.e., the application states are stored locally at the source instance. The concurrent transfer of all application states results in significant downtime and thus cannot further optimize R1.

One strategy to further reduce the downtime is to completely decouple the packet processing from the state processing. StatelessNF [52] places the application states in a remote data store, thus rendering the application stateless. In the context of the general stateless application paradigm, the recent MIGRATE study [53] has examined the state transfer of the application state data that is stored in a local database in the MIGRATE framework. In contrast to the MIGRATE study [53], we consider the state transfer for stateful applications in this study. Also, the SDN controller in MIGRATE only directs the user traffic; whereas, we employ the SDN controller to make forwarding decisions for the direct state transfer.

Nevertheless, this StatelessNF approach introduces end-to-end latency between the application instances and remote storage systems. Consequently, StatelessNF poses the new challenge of optimally placing the storage system. To avoid the challenge of optimally placing storage systems, D3 [54] and E-state [55] leverage key-value stores for state distribution among the application instances. These studies significantly reduce the migration time, but require on-demand state lookup that implies a reactive strategy. Consequently, the proposed solutions introduce high latency for the destination instance to obtain the application states when there are frequent updates. To overcome this issue, Doan *et al.* [27] proposed a proactive migration framework that allows state synchronization between multiple application instances. Owing to the use of the data store, multiple instances could access the application states and thus enable these studies to achieve the high-availability (R4) requirement, while further optimizing R1.

However, the above studies only concentrate on the state transfer while leaving other state operations unaddressed. Specifically, they lack an abstract interface to indicate (signal) the state transfer to the application instance. These studies are locked in specific applications, hindering them from supporting multiple applications, which are ubiquitous in the network. Furthermore, there is no guarantee that the destination instance will be correctly recovered, and the user traffic will be re-directed to it after the state transfer is complete.

To address this problem, SDN features can be exploited for managing the application states. Split/Merge [23] is built atop an SDN controller, providing APIs to split the states when an instance becomes overloaded and merging the states when the processing load decreases. Split/Merge thus achieves elastic scaling. However, Split/Merge has low fault tolerance because the original states are not actually copied; instead, the original states are separated from the original instance and then imported into other instances. To overcome this limitation, Pico Replication [24] extends Split/Merge to replicate the states between application instances. However, the two aforementioned approaches did not consider state loss and state reordering that could make the application unrecoverable after the migration. OpenNF [22] and its enhanced versions [29], [56], [57] have been introduced to tackle these critical issues by offering APIs over the controller for guaranteed methods, such as order-preserving and loss-free processing.

The above studies rely on the controller to forward the application states from one instance to others, thus satisfying R4. However, they introduce a potential bottleneck in that the controller acts as a proxy for the state transfer between the application instances. Thus, they are still far from satisfying R1 while the support of programmability (R2) and flexibility (R3) is still an open question.

In summary, the prior studies on the state management mainly tried to fulfill R1 and R4. Specifically, these studies show that an abstract scheme is needed to efficiently manage application states. Moreover, the inflexibility of the state transfer made them ill-suited for large-scale networks and consequently failed to achieve R2 and R3. In this paper, we propose to adopt SDN for the direct state transfer. In particular, unlike OpenNF which relies on the SDN controller as the intermediate node for the state transfer, we take advantage of the SDN controller to make the forwarding *decisions* for the direct state transfer. The use of SDN for this decision-making purpose ensures a high level of programmability (R2) and flexibility (R3) [58]–[60].

Overall, the prior studies mainly focused on minimizing the migration time, the downtime, and the amount of migration data. Meanwhile, the computation cost and buffering cost have rarely been considered. The computation cost is determined by choosing the destination server from a set of servers so as to save hardware resources; in particular, the selected server should minimize the migration frequency since frequent migrations require extensive hardware resources to host the migrated application. The computation cost has not been widely considered in the literature since most studies have been for data centers, which have abundant hardware resources. Considering the limited MEC hardware resources, the computation cost becomes an important metric. The buffering cost is defined as the amount of incoming traffic that is buffered during the migration. Most existing studies have ignored the buffering cost by assuming that the packet loss during the migration has a negligible impact on the recovery of the application after the migration. However, many MEC applications could be seriously degraded due to packet losses. For instance, for a video caching application, packet losses could falsify the caching ratio, which may incorrectly trigger content requests from the origin data center (instead of correctly streaming MEC cached content). As another example, packet losses can corrupt intrusion detection systems [22].

III. OPTIMIZING THE MEC STATE TRANSFER

A. SYSTEM MODEL

Consider a network graph $G = (V, E)$ consisting of a set V of network nodes and a set E of edges. Each of the $|V|$ network nodes is an SDN (OpenFlow) switch. Each edge $e \in E$ is a link connecting two switches. Let B_e and D_e denote the available bandwidth and delay of link e, respectively. We denote $A \subset V$ for the set of access nodes connected to base stations. Each of the $|A|$ $(A \subset V)$ access nodes is an SDN switch. We denote $M \subset V \backslash A$ for the set of MEC servers that serve user requests. We assume that each MEC sever $m \in M$ is directly attached to (i.e., coupled with) a network node in $V \backslash A$.

Let F denote the (overall) set of user traffic flows from the access nodes to the MEC servers. Specifically, let $F_{a,m} \subset F$ denote the set of flows from access node $a \in A$ to MEC server $m \in M$. Each flow $f \in F$ has a given traffic rate R_f and delay constraint D_f. The access node $a \in A$ is the ingress node of a user traffic flow $f \in F_{a,m}$ into the network and forwards the flow to the MEC server $m \in M$, which is considered as egress node of the network model.

TABLE 1. Summary of key notations.

Network Topology	
$G = (V, E)$	Graph G with nodes set V and links set E
$A \subset V$	Set of access nodes
$M \subset V \backslash A$	Set of MEC servers
F	Set of network flows $f \in F$
R_f	Traffic rate of flow $f \in F$
$F_{a,m} \subset F$	Set of flows from access node $a \in A$ to MEC server $m \in M$
D_e	Propagation delay of link $e \in E$
B_e	Bandwidth of link $e \in E$
D_f	Delay constraint of flow $f \in F$
State Migration	
$A_a^{mig} \subset A$	Set of given target access nodes for the flow migration from source access node $a \in A$
$F_a^{mig} \subset F$	Set of flows from source access node $a \in A$ that are migrated
$F_{a,a'}^{mig} \subset F_a^{mig}$	Set of flows migrated from access node $a \in A$ to access node $a' \in A_a^{mig}$
t_f^{ctr}	Time for a controller to indicate the state operations to the application instances
t_f^{u}	Time for a controller to update flow rules after the migration
t_f^{state}	Time to transfer states during the migration
t_f^{total}	Total migration time
$N_{a,m'}$	Number of migrated flows from access node $a \in A$ that are migrated to MEC server $m' \in M$
$x_{f,a',m'}$	Binary variable to indicate whether a migrated flow $f \in F_{a,a'}^{mig}$ traverses MEC server $m' \in M$ after the migration
$y_{a,m'}$	Binary variable to indicate whether MEC server $m' \in M$ is adopted to serve $N_{a,m'}$ migrated flows

For modeling the state transfer, for the flows $f \in F_{a,m}$ from access node a to MEC server m, let $A_a^{mig} \subset A$ denote the set of target access nodes and let $F_a^{mig} \subset \{\cup_{m \in M} F_{a,m}\}$ denote the set of flows that are migrated away from access node $a \in A$. We denote $F_{a,a'} \subset F_a^{mig}$ for the set of flows migrated from access node $a \in A$ to access node $a' \in A_a^{mig}$. Let $x_{f,a',m'}$ be a binary decision variable, whereby $x_{f,a',m'} = 1$ indicates that the migrated flow $f \in F_{a,a'}$ adopts access node $a' \in A_a^{mig}$ as the ingress node and the MEC server $m' \in M$ as the egress node after the migration; $x_{f,a',m'} = 0$ otherwise.

In summary, the change of the old access (ingress) node a to a new access (ingress) node a' models the user mobility; the change from the old MEC server (egress node) m to a new MEC server (egress node) m' models the flow migration. In order to avoid clutter in the notation definitions, we do not explicitly distinguish between user mobility (access node change a to a') and flow migration (MEC server change m to m'). Instead, we define the term "flow migration" to encompass the change of the access node from a to a' due to user mobility, which will trigger a flow migration optimization execution; this flow migration optimization execution may or may not result in a change of the MEC server depending on the outcome of the optimization.

B. COMPUTATION COST

Let $N_{m'}$ denote the number of migrated flows that an MEC server $m' \in M$ needs to serve after the migration of the flows $f \in F_{a,a'}^{mig}$, i.e., $N_{a,m'}$ is the number of migrated flows that adopt an MEC server $m' \in M$ as the egress node. $N_{a,m'}$ can be represented as:

$$N_{a,m'} = \sum_{a' \in A_a^{mig}} \sum_{f \in F_{a,a'}^{mig}} x_{f,a',m'}. \tag{1}$$

We only migrate the flows as required due to user mobility. For example, consider an access node a_1 and MEC server m_1 with five flows before migration. Suppose that three flows need to migrate due to user mobility (i.e., 2 other flows are retained due to lack of mobility). In the migration, one migrated flow chooses MEC server m_2 and the two other migrated flows choose MEC server m_3; thus, $N_{a_1,m_2} = 1$ and $N_{a_1,m_3} = 2$.

We denote $y_{a,m'}$ as a decision variable to indicate whether an MEC server $m \in M$ is adopted to serve the $N_{a,m'}$ migrated flows, i.e.,

$$y_{a,m'} = \begin{cases} 1, & N_{a,m'} \geq 1 \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

We let the computation cost C_a^{comp} represent the number of VMs created to serve the migrated flows; specifically, we evaluate C_a^{comp} as:

$$C_a^{comp} = \sum_{m' \in M \backslash \{m\}} y_{a,m'}. \tag{3}$$

The computation cost C_a^{comp} essentially counts the number of new VMs that are required at the various MEC nodes $m' \in M \backslash \{m\}$ to accommodate the migrated flows $f \in F_{a,a'}^{mig}$ from access node a. We acknowledge that our computation model is coarse in the sense that any positive number $N_{a,m'} \geq 1$ of migrated flows is counted as requiring one VM. We also note that we do not count a computation cost for flows that stay at their original MEC server m, i.e., no computation cost is counted for flows that are not migrated. We adopt this simple coarse computation cost model as our study focus is on the costs that arise due to user mobility that requires flow migration decision making. A more sophisticated computation cost model, e.g., a model that considers flow rates and specific computation demands, is an interesting direction for future research.

C. COMMUNICATION COST

We denote $C_{a',m'}^{comm}$ as the communication cost of all flows $f \in F_{a,a'}^{mig}$ traversing from access node $a' \in A_a^{mig}$ to MEC server $m' \in M$ after the migration:

$$C_{a,a',m'}^{comm} = \sum_{f \in F_{a,a'}^{mig}} x_{f,a',m'} \sum_{e \in L_{a',m'}} R_f, \tag{4}$$

where $L_{a',m'}$ denotes the network path (set of links) between a' and m', including all the links through intermediate network nodes, i.e., the conventional packet switches that forward the data packets (without conducting any significant computing on the data packets). Thus, the communication cost C_a^{comm} is:

$$C_a^{comm} = \sum_{a' \in A_a^{mig}} \sum_{m' \in M} C_{a,a',m'}^{comm}. \tag{5}$$

D. BUFFERING COST

The state migration process is performed as follows. First, a controller tells an application instance *srcApp* on the source MEC server to export states. Simultaneously, the controller informs the application instance on the destination MEC server *dstApp* to import the states from *srcApp*. We denote t_f^{ctr} for the total time for the message exchanges between the controller, *srcApp*, and *dstApp* to migrate a flow $f \in F$. Next, *srcApp* transfers the states of flow f to *dstApp* through the underlying network. We denote t_f^{state} for the state transfer time. Finally, the controller updates the forwarding rules to redirect the user traffic of flow f to *dstApp*. We denote t_f^u for the time to perform this update process. Thus, the total migration time t_f^{total} is:

$$t_f^{total} = t_f^{ctr} + t_f^{state} + t_f^u. \tag{6}$$

In Eq. (6), t_f^{ctr} and t_f^u are typically negligible. Specifically, t_f^{ctr} only introduces the latency before and after the migration. Meanwhile, t_f^u is only added once to t_f^{total} before or after the state transfer. There are numerous approaches for reducing t_f^u, such as controller placement [61], traffic prediction [62], and path aggregation [63], which are outside the scope of this paper. Overall, t_f^{state} is the main contributor to t_f^{total}. The state transfer time t_f^{state} includes the transmission delay, link propagation delay, queuing delay, and processing delay of the state information (packets). We assume that the queue is empty and thus neglect the queuing delay. Furthermore, we consider fast-forward packet switches with negligible processing delay and negligible store-and-forward transmission delay in the network.

Thus, only the transmission delay $t_{trans,f}^{state}$ out of the source MEC node and the link propagation delay $t_{prop,f}^{state}$ contribute significantly to the state transfer time t_f^{state}, i.e.,

$$t_f^{state} = t_{trans,f}^{state} + t_{prop,f}^{state}. \tag{7}$$

Let $\eta_{a,a'}$ be number of flows whose access node is changed from $a \in A$ to $a' \in A_a^{mig}$ due to the user mobility. We evaluate the total state transfer time $t_{a',m'}^{state}$ for all flows $f \in F_{a,a'}^{mig}$ with their MEC server changed from m to m' $(m, m' \in M)$ as:

$$t_{a',m'}^{state} = \mathcal{T}_{trans} + \mathcal{T}_{prop}. \tag{8}$$

The transmission delay $\mathcal{T}_{trans}$ of the state information of all flows $f \in F_{a,a'}^{mig}$ is generally proportional to the number of migrated flows [49], [64]. Thus, $\mathcal{T}_{trans}$ can be expressed as:

$$\mathcal{T}_{trans} = \sum_{f \in F_{a,a'}^{mig}} t_{trans,f}^{state} \tag{9}$$

$$= \left(\rho + \omega\eta_{a,a'}\right), \tag{10}$$

where ρ and ω are constants that characterize the different MEC applications.

With $L_{m,m'}$ denoting the network path, including all the links through intermediate packet switches that forward the data packets between the original MEC server m and the target MEC server m' $(m, m' \in M)$, the propagation delay $\mathcal{T}_{prop}$ is:

$$\mathcal{T}_{prop} = \sum_{e \in L_{m,m'}} D_e. \tag{11}$$

With Eq. (10) for the transmission delay and Eq. (11) for the propagation delay, Eq. (8) for the total state transfer time becomes

$$t_{a',m'}^{state} = \rho + \omega\eta_{a,a'} + \sum_{e \in L_{m,m'}} D_e. \tag{12}$$

The buffering cost $C_{a',m'}^{buff}$ for all flows $f \in F_{a,a'}^{mig}$ can be evaluated as

$$C_{a',m'}^{buff} = \sum_{f \in F_{a,a'}^{mig}} x_{f,a',m'} t_{a',m'}^{state} R_f. \tag{13}$$

Thus, the total buffering cost C_a^{buff} is

$$C_a^{buff} = \sum_{a' \in A_a^{mig}} \sum_{m' \in M \setminus \{m\}} C_{a',m'}^{buff}. \tag{14}$$

E. CONSTRAINTS

We proceed to introduce the constraints for the communication cost and the state transfer cost.

1) MEC SERVER SELECTION CONSTRAINT

For the sake of simplicity, we assume that all flows $f \in F_{a,a'}^{mig}$ that are migrated from access node a to access node $a' \in A_a^{mig}$ utilize the same MEC server $m' \in M$, i.e.,

$$\sum_{m' \in M} x_{f,a',m'} = 1 \quad \forall f \in F_{a,a'}^{mig}, a' \in A_a^{mig}. \tag{15}$$

The extension to utilizing different MEC servers can be accommodated with the cost models of the preceding subsections in a straightforward manner. In brief, the communication cost model in Section III-C would have to account for the communication costs from the new access node to the multiple destination MEC servers, and the buffering cost model in Section III-D would have to account for the state transfer between the source MEC server and the multiple destination MEC servers. Also, the computation cost model in Section III-B would have to be updated to the adoption of multiple destination MEC servers for the migration. In order to avoid clutter that would obscure the main concepts of this article, we omit the details of this extension to multiple destination MEC servers.

2) DELAY CONSTRAINT

For all migrated flows with their previous access node $a \in A$, i.e., all flows $f \in F_a^{mig}$, the delay constraint must not be violated, i.e.,

$$\sum_{e \in E} x_{f,e} D_e \leq D_f, \quad \forall f \in F_a^{mig}, \tag{16}$$

whereby $x_{f,e}$ is a binary decision variable: $x_{f,e} = 1$ indicates that the flow $f \in F_a^{mig}$ adopts link $e \in E$ in its flow path after the migration; $x_{f,e} = 0$ otherwise.

Algorithm 1: Greedy Initial Algorithm

```
   Input  : $G = (V, E), A_a^{mig}, M, F$
   Output: $List_a^{MEC}$
1  $List_a^{MEC} = [\ ]$
2  for $a' \in A_a^{mig}$ do
3      $List_{a'}^{cost} = [\ ]$
4      for $m' \in M$ do
5          $Path_{data} = Dijkstra(a', m')$
6          if $Path_{data} == \emptyset$ then
7              continue
8          end
9          Calculate $C_{a,a',m'}^{comm}$ using Eq. (4)
10         if $m' == m$ then
11             $C_{a',m'}^{buff} = 0$
12         else
13             $Path_{state} = Dijkstra(m, m')$
14             if $Path_{state} == \emptyset$ then
15                 continue
16             end
17             Calculate $C_{a',m'}^{buff}$ using Eq. (13)
18         end
19         Calculate $C_{a,a',m'}^{total}$ using Eq. (18) with $\alpha = 0$
20         $List_{a'}^{cost}.append(C_{a,a',m'}^{total})$
21     end
22     $candidate = arg_{m'} \min(List_{a'}^{cost})$
23     $List_a^{MEC}.append(candidate)$
24 end
```

3) LINK CAPACITY CONSTRAINT

The total traffic rate of all flows $f \in F$ traversing link $e \in E$ must not exceed the link capacity, i.e.,

$$\sum_{e \in E} \sum_{f \in F} x_{f,e} R_f \leq B_e. \tag{17}$$

F. OBJECTIVE FUNCTION

We aim to minimize the total migration cost C^{total}, which incorporates the computation cost, the communication cost, and the buffering cost for all flows migrated from access node $a \in A$, i.e.,

$$\min C_a^{total} = \min \left(\alpha C_a^{comp} + \beta C_a^{comm} + \gamma C_a^{buff} \right), \tag{18}$$

where α, β, and γ are tunable weights that control the relative importance (priority) of computation cost, communication cost, and buffering cost subject to the constraints of Section III-E.

The problem in Eq. (18) is NP-hard: we simplify the problem in Eq. (18) by setting β and γ to 0 (only for the purpose of this proof). The problem is transformed to the problem that tries to find an MEC server that can serve as many migrated flows as possible without violating the constraints of Section III-E. This problem is equivalent to the knapsack problem which is NP-hard.

Algorithm 2: Tabu Search

```
   Input  : $G = (V, E), A, M, F, i_{stop}, i_{tabu}$
   Output: $S_{best}$
1  $S_0 = List_a^{MEC}$ obtained from Algorithm 1
2  $i = 0, S_{curr} = S_0, S_{best} = S_0$
3  $List_{tabu} = [\ ]$
4  while $i < i_{stop}$ do
5      $move = MoveStrategy(S_{curr})$
6      $Update(S_{curr})$ from the move
7      Calculate $f(S_{curr})$ using Eq. (18)
8      $i = i + 1$
9      if $move$ is not in $List_{tabu}$ then
10         if $len(List_{tabu}) < i_{tabu}$ then
11             $List_{tabu}.append(move)$
12         else
13             break
14         end
15     else
16         if $f(S_{curr}) < f(S_{best})$ then
17             $List_{tabu}.remove(move)$
18         end
19     end
20     if $f(S_{curr}) < f(S_{best})$ then
21         $S_{best} = S_{curr}$
22         $i = 0$
23     end
24 end
```

G. PROPOSED TABU SEARCH ALGORITHM

Generally, heuristic algorithms are tightly coupled to a specific problem, while meta-heuristic algorithms leverage general techniques [65] that have been proven to efficiently solve a broad range of hard problems [66]. Popular meta-heuristic algorithms include ant colony optimization [67], genetic algorithms [68], and Tabu search [69]. Tabu search tends to have lower complexity than other meta-heuristics [70]. Since low complexity is well-suited for the low-latency requirement in MEC, we employ the Tabu search algorithm to solve our problem in Eq. (18).

Tabu search is an iterative search process used for mathematical optimization. Tabu search starts to explore the search space from an initial solution and then iteratively moves from its neighborhoods to transit from a current solution to a better solution until a stopping condition is met. Tabu search relies on a Tabu list of the previously visited solutions to avoid re-visiting previous solutions. We summarize the major elements of the Tabu search algorithm in the context of our problem setting as follows:

- We propose to start the Tabu search with an initial greedy solution obtained with Algorithm 1. Algorithm 1 seeks the best MEC server corresponding to each target access node a' in terms of communication cost and buffering cost. Thus, we set $\alpha = 0$ in Eq. (18). To calculate the communication cost, we adopt the shortest path with respect to the latency from the target access node a' to the MEC server candidate m' based on Dijkstra's algorithm (Lines 5–9). For the buffering cost, we also employ Dijkstra's algorithm to compute the shortest path from the original MEC server m to the MEC server candidate m' (Lines 10–19). We then calculate the total cost $C^{total}_{a,a',m'}$ and add it to a list $List^{cost}_{a'}$ (Line 20). The target MEC server is selected with minimum total cost from $List^{cost}_{a'}$ and then added to a list $List^{MEC}_{a}$ (Lines 22–23).
- Tabu search explores the neighborhood of the current solution to seek a better solution as outlined in Algorithm 2. A neighborhood solution is determined by applying a single move from the current solution: A move is defined as a mapping between an access node and an MEC server. We first randomly select a target access node from A^{mig}_{a}. We then move its corresponding MEC server to another MEC server that satisfies the constraints described in Section III-E.
- To prevent cycling the previously visited solutions, a move that satisfies the constraints in Section III-E is added to the Tabu list $List_{tabu}$. If the Tabu list exceeds its pre-defined length i_{tabu}, then the algorithm is stopped (Line 13). This facilitates reducing the complexity of the search process. Notably, we set an *aspiration criterion*: if the aspiration criterion is met, then the move is released (removed) from the Tabu list (Lines 16–18). The aspiration criterion is defined for the case when a better solution than the current best solution is found.

In summary, our proposed Tabu search Algorithm 2, operates as follows. First, an initial solution S_0 containing a list of target MEC servers is obtained from Algorithm 1 (Line 1). Afterwards, S_0 is assigned to the current solution S_{curr} and the best solution S_{best}, while a Tabu list $List_{tabu}$ is initialized as empty (Lines 2–3). A while loop is performed to iteratively seek a solution until the number of iterations exceeds i_{stop} (Lines 4–24). Specifically, the move strategy is applied and S_{curr} is updated from the move (Lines 5–6). The total cost $f(S_{curr})$ is re-calculated using Eq. (18) and then the number of iteration is increased by one (Lines 7–8). In Lines 9 to 19, the algorithm determines whether the move is in $List_{tabu}$. If the move has not been previously visited, then the move is added to $List_{tabu}$ (unless the length of $List_{tabu}$ exceeds i_{tabu}); otherwise, the algorithm is stopped (Lines 10–14). If the move is in $List_{tabu}$, the algorithm checks whether the aspiration condition is met (Lines 16–18). The move is released from $List_{tabu}$ if the total cost of S_{curr} is lower than the cost of S_{best}. In Lines 20 to 23, if a solution is found, the algorithm assigns S_{curr} to S_{best} and restarts the while loop.

IV. FAST: FLEXIBLE AND LOW-LATENCY STATE TRANSFER

The MEC framework deploys applications and a backup scheme and instructs FAST to transfer states.

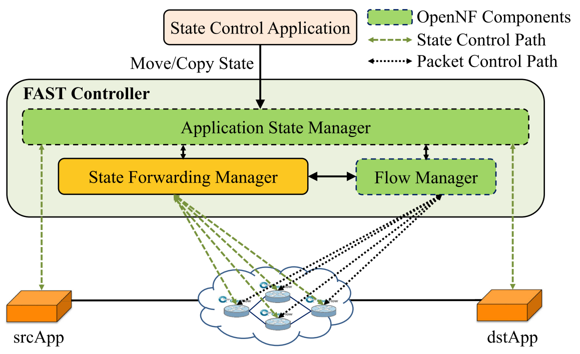


FIGURE 2. FAST framework for the state management in MEC between *srcApp* and *dstApp*: The OpenNF application state manager handles the requests of the state operations (i.e., move, copy, and share) received from the state control application. The novel state forwarding manager manages the forwarding rules for the state transfer between *srcApp* and *dstApp*. Meanwhile, the OpenNF flow manager coordinates the redistribution of the user traffic.

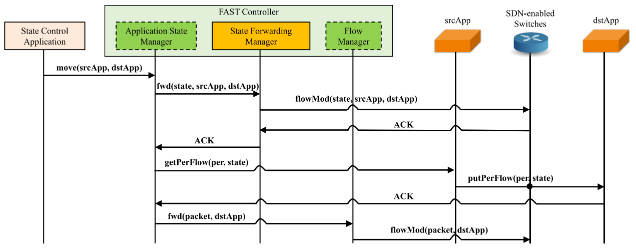


FIGURE 3. FAST state management procedure: The FAST controller directly transfers the application states from *srcApp* via SDN-enabled switches to *dstApp*.

A. FAST: DESIGN CONCEPT

As shown in Fig. 2, the FAST framework consists of three components: an application state manager, a state forwarding manager, and a flow manager. FAST is extended from OpenNF [22], leveraging the OpenNF application state manager and flow manager to perform the state operations and the forwarding decisions of the user traffic. The novel state forwarding manager manages the forwarding rules for the application states. Based on the global view of the network, the state forwarding manager applies different forwarding rules to the underlying network devices, thus enabling the programmability of the application state transfer. FAST relies on the SDN design and utilizes the SDN functionalities. To allow multiple concurrent application backups, the state forwarding manager performs multicast communication using a group table provided by OpenFlow. To add or delete the backups on-the-fly, FAST simply enables or blocks the switch ports, which lead to the corresponding backup instances. This reduces the effort compared to using a conventional network for the state forwarding.

FAST shares the same data plane for both the state forwarding and the data packet forwarding. The state forwarding manager and the flow manager cooperate to avoid flow rule conflicts, which could result in unrecoverable backup instances after a state transfer. There is a case that state flows and packet flows go through the same SDN-enabled switch with different priorities. Without having a consensus about the forwarding policies, the states could not reach the backup instance. Thus, before installing the forwarding rules for the application states on the network devices, the forwarding manager and the flow manager should exchange their forwarding policies to avoid potential conflicts.

B. FAST: STATE MANAGEMENT PROCEDURE

The goal of FAST is to directly forward applications states, thus mitigating the burden of the centralized state management. As shown in Fig. 3, FAST allows the application states to be directly forwarded from a source application instance *srcApp* via the SDN-enabled switches to a destination application instance *dstApp*. First, the state control application sends a request to the application state manager indicating a *move* scheme. The application state manager then informs the state forwarding manager to set the flow rules in the SDN-enabled switches for the state transfer. Next, the switches send an acknowledgment message (e.g., 200 OK response) to the state forwarding manager, indicating that the flow rules have been successfully installed. The state forwarding manager then informs the application state manager about the completion of the flow rule installation. Afterwards, the application state manager issues the *export* operation to *srcApp*. Next, *srcApp* extracts its states and then directly forwards the states via the forwarding path set up by the state forwarding manager to *dstApp*. After the state transfer is complete, *dstApp* sends an acknowledgment message to the application state manager, confirming that all states have been received. Finally, the application state manager invokes the flow manager to instruct the SDN-enabled switches to re-direct the user traffic to *dstApp*.

C. FAST: IMPLEMENTATION

Among the well-known state management frameworks, such as OpenNF, FTMB, and Pico the OpenNF framework is well suited to implement FAST because OpenNF is open-source and programmable. However, FAST can also be ported to other frameworks, facilitating their ability to enable flexible and low-latency state transfer. We first leverage the existing fundamental OpenNF functions to process states (e.g., import or export states) inside applications. We then extend OpenNF to enable the state forwarding via the underlying network devices. Specifically, we implement a novel state forwarding manager that works in cooperation with the OpenNF application state manager and flow manager.

To integrate FAST into OpenNF, our implementation operates as follows. First, OpenNF allows applications to communicate with the controller via a shared library that provides generic application interfaces. Specifically, OpenNF uses two network ports, namely a *stateControl* port (7790) and an *eventControl* port (7791) for exchanging the state packets and event packets between the application instances and the controller, respectively. Following this port-based design, we create a new forwarding path for the application states. In order to separate out the state forwarding functionalities from the controller while ensuring adaptability, we propose to use a *stateTransfer* port (7792) dedicated to the state transfer.

Second, after the state packets have been received from *srcApp*, the OpenNF controller converts their message format to the *putPerFlow* format and then sends them to *dstApp*. Such an operation is performed to avoid the misconfiguration caused by the centralized management in OpenNF. If the state packets were directly forwarded from *srcApp* to *dstApp*, then they would obviously be unknown and would immediately be discarded by the state handler at *dstApp*. To overcome this issue while retaining the same design logic, we implement a method to construct the *putPerFlow* structure right at *srcApp*. Consequently, after the state packets have been generated, they are repackaged and translated from the *getPerFlow* structure to the *putPerFlow* structure. Afterwards, the state packets are transferred to *dstApp* through the network devices. The final challenge arises at *dstApp*, whereby OpenNF originally uses the state handler at the *stateControl* port. Similarly, we replicate this method at the *stateTransfer* port to handle the states received from *srcApp*.

In summary, the state transfer proceeds as follows. First, *srcApp* and *dstApp* request to establish connections with the FAST controller (i.e., via the *stateControl* port and via the *eventControl* port) and simultaneously start a thread to listen to the *stateTransfer* port. When receiving a *move* request, the application state manager invokes the state forwarding manager to install the forwarding rules for the application states on the network devices. The state manager then indicates an *export* operation to *srcApp* via the *eventControl* port. The state packets are generated and repackaged at *srcApp*. Next, based on the forwarding path installed by the state forwarding manager, these packets are directly forwarded to *dstApp* through the *stateTransfer* port. The state packets are then received by the state handler at *dstApp*. After all state packets have been correctly received, *dstApp* sends an acknowledgment message to the controller via the *stateControl* port. The controller finally informs the flow manager to direct the user traffic to *dstApp*.

D. FAST: EMULATION PERFORMANCE EVALUATION

In this section, we evaluate FAST and compare it with related schemes. We assess the efficiency of FAST by measuring the total move time, which reflects the service downtime and is widely considered in the literature [22], [31]. The total

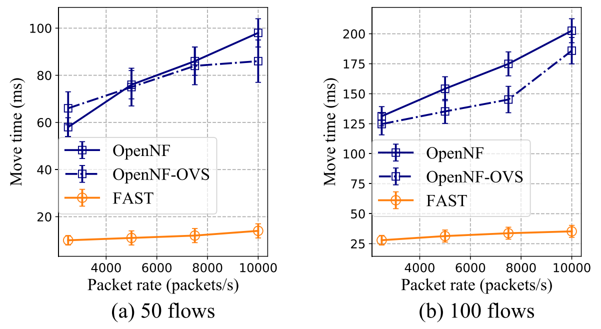


FIGURE 4. Emulation evaluation: Total move time as a function of packet rate for 50 packet flows (a) and at 100 packet flows (b).

move time is the time duration from the time instant when the export of the application states from the source application instance commences to the time instant when the user traffic is re-routed to the destination application instance. Since the packet rate and the number of flows govern how many states are generated, we examine their impact on the total move time. Moreover, we investigate the impact of the state management on the total move time for existing guarantee policies in OpenNF, including no guarantee (NG), parallel optimization (PL), loss-free (LF), order-preserving (OP), and early release (ER), as described in Section II-A.

1) EVALUATED SCENARIOS

We compare the performance of FAST against the following two schemes:

- **OpenNF**: The OpenNF controller acts as a proxy, receiving the states from the source application instance and then forwarding the states to the destination application instance. The state transfer is performed with a layer-2 switch.
- **OpenNF-OVS**: The state transfer is similar to the one in OpenNF, but the states are transferred via an Open vSwitch (OVS) instance.

2) EMULATION SETUP

We conducted the evaluations on a machine configured with Intel(R) Core(TM) i7-7500U CPU at 2.50 GHz, 16 GB memory, and Ubuntu 14.04. We used Mininet [71], which is a virtualized tool to emulate hosts, network switches, and links, to set up a network topology consisting of two software switches and three hosts. FAST and OpenNF are implemented atop a Floodlight SDN controller. The three hosts are used as follows: one host acts as a packet generator and the other two hosts are the source application instance *srcApp* and the destination application instance *dstApp*. These application instances operate in a Linux namespace to emulate a virtualized environment, which is typical for light-weight cloud computing. The Passive Real-time Asset Detection System (PRADS) [72], which has been widely used to examine state management solutions [22], [29], is deployed as the source application instance *srcApp* and the destination application instance *dstApp*. We reused existing functionalities in OpenNF to export/import states from/to PRADS. Since our main focus is on the state transfer, we modified PRADS for the direct state forwarding as mentioned in Section IV-C. One switch is used for the packet forwarding and another switch is used for the state forwarding. The use of two separate switches prevents the mutual impact of the state forwarding and the packet forwarding. The two switches in FAST and OpenNF-OVS are OVS instances. The switches in the OpenNF scheme are used as follows: one switch acts as the OVS instance for the user traffic and another switch is a normal layer-2 switch for the state forwarding.

We first ran the Floodlight controller and then created the Mininet topology. Afterwards, we started the packet generator to forward the user traffic to the PRADS instance on *srcApp*. After observing a prescribed number of flows processed by *srcApp*, the controller initiated a move scheme. For instance, the moves in Fig. 4 were initiated after there were 50 flows (Fig. 4 (a)) or 100 flows (Fig. 4 (b)) processed by *srcApp*. After the movement has been completed, the PRADS instance on *srcApp* is stopped and taken over on *dstApp*. We report the 95% confidence intervals obtained from 10 independent replications.

3) EMULATION RESULTS

We first investigate the total move time for varying packet rates from 2500 packets/s to 10000 packets/s, which we generated with the tcpreplay tool [73] following the evaluation strategy in [22]. All evaluated schemes applied the PL and LF policies. We considered the LF policy so that we can observe the impact of both the packet rate and the number of flows on the total move time. More specifically, to guarantee the LF move, the SDN controller has to buffer the incoming packets during the state transfer (otherwise *srcApp* will discard these packets to ensure state consistency during the state transfer). These buffered packets will be released to *dstApp* after the state transfer is completed.

The increasing number of flows results in more flow states that need to be transferred, and thus longer state transfer times, which contribute significantly to the total move times. Subsequently, the higher state transfer times result in more buffered packets, since the buffered packets are released after the state transfer is completed. Meanwhile, a higher packet rate increases the number of buffered packets during the state transfer. Since the application itself cannot control the packet rate of the incoming packets, to reduce the number of buffered packets, the total move time needs to be minimized. The joint use of the PL and LF policies allows the concurrent execution of the export of states at *srcApp* and the import of states at *dstApp*, thus accelerating the state transfer.

As expected, Fig. 4 shows an upward trend in the total move time with increasing packet rate. This upward trend is caused by the increasing number of arriving packets, resulting in more buffered packets which need to be released. Notably, we observe from Fig. 4 that FAST achieves much shorter total move times than OpenNF and OpenNF-OVS. For a packet rate of 10000 packets/s, FAST reduces the total move time down to less than a quarter of the total move times of OpenNF and OpenNF-OVS.

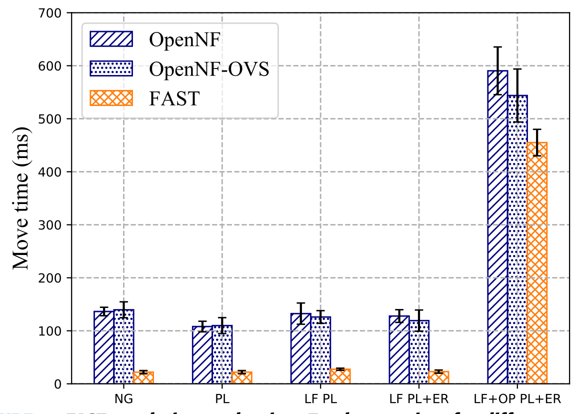


FIGURE 5. **FAST emulation evaluation: Total move time for different guarantee policies for 100 packet flows, each with 2500 packets/s compared to no guarantee (NG) policy. The parallel optimization (PL) and early release (ER) policies are grouped (PL+ER) to indicate the policies for reducing the total move time; the loss-free (LF) and order-preserving (OP) policies are grouped (LF+OP) to indicate the policies for the correct recovery of the application.**

Moreover, we observe from Fig. 4 that FAST has only a very slight upward trend of the total move time with increasing packet rate compared to the significantly steeper upward trends of OpenNF and OpenNF-OVS. The flatter upward trend of FAST is achieved by directly forwarding the states between the application instances. The steeper upward trends of OpenNF and OpenNF-OVS are mainly due to the involvement of the OpenNF controller in the state transfer. Specifically, we observe from Fig. 4 that OpenNF-OVS incurs less total move time than OpenNF in most settings. When the number of flows increases, the gap between OpenNF-OVS and OpenNF tends to increase (except for 10000 packets/s for 100 flows in Fig. 4 (b), when the OVS software switch starts to become saturated). Fig. 4 (b) shows that for 100 flows and 10000 packets/s, OpenNF-OVS reduces the total move time by roughly 15% compared to OpenNF. This is because OpenNF-OVS used OVS for the state forwarding, while OpenNF simply adopted a normal switch; whereby OVS tends to perform better than the normal switch when the packet rate increases (as long as the packet rate does not saturate the switch).

To investigate the impact of the guarantee policies on the total move time, we examined five schemes including NG, PL, LF PL, LF PL+ER, and LF+OP PL+ER for 2500 packets/s per flow and 100 flows. Fig. 5 shows that FAST achieves significantly shorter move times than OpenNF-OVS and OpenNF. For the first four schemes, FAST has an 80% shorter total move time than the other approaches. For the NG scheme, OpenNF incurs 140 ms total move time, including 17 ms for exporting the states from *srcApp*, 13 ms for importing them into *dstApp*, and the remaining 110 ms for the execution time at the OpenNF controller. Meanwhile, the total move time in FAST drops to 27 ms, including 16 ms for extracting the states at *srcApp* and the remaining 11 ms for importing them into *dstApp*, while the state transfer delay is negligible in the considered Mininet setting. The results for the LF+OP PL+ER scheme indicate that FAST reduces the total move time by 20% compared to OpenNF. We observe that the OP policy significantly increases the total move time. As shown in Fig. 5, OpenNF completes a move for the LF+OP PL+ER scheme in 590 ms, nearly a 4-fold increase compared to the NG scheme. This is mainly because the OP policy takes a significant amount of time to control the order of the states generated by *srcApp* before sending them to *dstApp* through an elaborate two-phase forwarding state update procedure (see [22] for details). Thus, the OP policy is only considered for applications that are degraded by orderless states, such as intrusion detection systems [22]. Nevertheless, FAST reduces the move time by 135 ms compared to OpenNF and by 89 ms compared to OpenNF-OVS.

It is worth noting that the total move time of the FAST framework is mainly incurred for five main steps: *i*) the SDN controller for the flow rule installation of the state transfer, *ii*) the applications for enabling the connections for the direct state transfer, i.e., for opening the ports for the state transfer, *iii*) the *srcApp* for extracting the states, *iv*) the SDN switches as the data plane for the state transfer for forwarding the states (incurring the state transfer delay), and *v*) the *destApp* for importing the states. The time required by the SDN controller and the applications for opening the ports for the state transfer in the FAST framework is negligible (i.e., each performed only once for the state transfer, incurring typically less than 1 ms delay). FAST extracts, transfers (via the intermediate SDN switches), and imports the states in the data plane. In contrast, for the OpenNF-related schemes, the SDN controller is the intermediate node for the state transfer. The OpenNF-related schemes transfer the application state information via Java Script Object Notation (JSON) that the SDN controller receives from the source application instance, unpacks, and then re-packages before forwarding to the destination application instance, incurring significant time delays in the SDN controller. Furthermore, the FAST framework is a "topology-aware" scheme that relies on the underlying data-plane network devices to transfer the application states. Whereas, the OpenNF-related schemes are "topology-unaware" in that they rely on the SDN controller, which may be located far from the source and destination MEC servers, for the forwarding of the application states.

The emulation evaluation has established that the FAST framework outperforms the OpenNF-related schemes at the level of the overall state transfer strategy. Specifically, the state transfer through the data plane (with decision making in the SDN controller) in the FAST framework achieves generally shorter move times than the state transfer through the SDN controller in the OpenNF-related schemes. Section V, follows up on this overall result, by examining different optimization (decision making) mechanisms that operate in the SDN controller in the FAST framework.

V. SIMULATION EVALUATION OF MEC STATE TRANSFER OPTIMIZATION

This section focuses on the performance comparison of different optimization (decision) mechanisms within the FAST framework and addresses the following two questions with extensive simulation evaluations: *i*) How efficient is the proposed Tabu search MEC state transfer optimization in terms of the computation cost, communication cost, and buffering cost?, and *ii*) How much complexity does the proposed Tabu search MEC state transfer optimization introduce?

A. BENCHMARK SCHEMES

We compare our proposed FAST-Tabu algorithm, which we refer to as "Tabu" for brevity, against the following five benchmark schemes. We note that all compared schemes, i.e., Tabu and the five benchmark schemes, operate within the context of FAST, i.e., operate with the direct state forwarding feature in FAST.

- **Optimal**: We find an optimal solution using Gurobi (version 9.02) [74], which is a widely used commercial optimization tool.
- **Latency-aware state transfer (LAST)**: LAST chooses the MEC server with the lowest end-to-end latency from the new access node a' to the new MEC server m'.
- **Communication-aware state transfer (CAST)**: CAST selects the MEC server with the least bandwidth usage, i.e., the MEC server with least communication cost as defined in Section III-C.
- **Random**: The algorithm uniformly randomly selects an MEC server that satisfies the constraints in Section III-E.
- **No migration**: If the constraints are satisfied, the old MEC server m remains unchanged when flows move to a new access node a'. Otherwise, a flow deadline violation is recorded.

B. SIMULATION SETUP

We used the Python NetworkX [75] library, a Python library for creating, manipulating, and studying complex networks, to generate an Erdös-Rényi (ER) [76] random graph model with $|V| = 50$ nodes, including $|A| = 10$ access nodes and $|M| = 40$ MEC servers (i.e., each of the network nodes in $V \backslash A$ has an attached MEC server) with the connection probability $p = 0.6$. The ER network graph is widely considered in the literature [77]–[79] to simulate real-world network topologies. We consider more MEC servers ($|M| = 40$) than access nodes ($|A| = 10$), which is reasonable because there should at least be one MEC server corresponding to (close to) each access node [80], [81]. Meanwhile, considering compute-intensive applications that offload their heavy computation tasks to MEC [82], a large set of MEC servers ensures sufficient computation resources for the user requests. Also, our study examines the optimization of the state transfer between the MEC servers. A large number of MEC servers implies that the optimization problem has to consider more options and thus makes the optimization more complex. Nevertheless, a different regime of scare resources, in which there are only few MEC servers, could be investigated in future research with the general optimization model introduced in Section III. The link delay between nodes is randomly distributed between 10 ms and 50 ms, mimicking the physical link delay. This setting is also used by Mouradian *et al.* [83] and Yang *et al.* [84]. The link bandwidth is randomly distributed in the set of [1, 10] Gbps [83], [85]. For the objective function in Eq. (18), since the communication cost typically dominates the total cost, we set $\beta = 1$ and $\alpha = \gamma = 10$.

Service requests (traffic flow requests) have a maximum delay constraint chosen randomly between 50 ms and 100 ms [86]. The arrival of the service requests follows a Poisson distribution with a rate of 1 request per 30 seconds. Each flow request has an exponentially distributed lifetime with an average of 3 hours [87]. These settings mimic real network traffic and allow for generating large numbers of requests. The traffic rate is randomly selected in the range of 10 Mbps to 50 Mbps per flow, reflecting the typical MEC network traffic [84]. The requests are uniformly randomly distributed to the access nodes $a \in A$. We consider three open-source applications for setting the application parameters that determine the transmission time of the states [49], namely PRADS, Zeek [88] (formerly Bro), and Iptables [89], which have been widely considered for the performance evaluation of the state transfer [22], [29], [30], [49], [54], [56], [84]. We investigate the performance of the evaluated schemes on a desktop computer with Intel Core i7-6700T CPU and 16 GB RAM.

The simulation evaluation proceeds as follows. First, flow (service) requests are generated into the network. Each flow has an access node $a \in A$ as the network ingress node and an MEC server $m \in M$ as the network egress node, whereby the ingress and egress nodes of each flow are randomly selected. To simulate the condition to trigger a migration, mobility models, such as SUMO [90] or the ONE simulator mobility models [91], could be used. However, most MEC studies that adopted these mobility models, e.g., [92], [93], simply assumed that one MEC server is attached to each base station. Thus, the use of a mobility model in these studies becomes an important factor for the experiment. In contrast to the aforementioned studies, we do not assign an MEC server to each base station. Instead, in this paper, we deploy an edge cloud network architecture [94] to provide a set of $|M| = 40$ MEC servers at the edge of the network. The edge cloud network architecture increases the flexility (i.e., different choices to deploy MEC applications because there are many MEC servers) and availability (i.e., more hardware resources). Aligned with the considered edge cloud network architecture, we consider a mobility model in which the migration is triggered based on the number of flows, which is an essential metric to measure the performance of the migration, as mentioned in Section IV-D. More specifically, after observing a certain number of flows (e.g., 100 flows or 500 flows) generated in the network, a migration at a uniformly randomly selected access node $a \in A$ is triggered.

More specifically, the simulation proceeds as follows. The network starts up initially empty. Flows are generated one by one and they are randomly assigned to an ingress (access) node and an egress (MEC) node. In particular, the access node and MEC node are uniformly randomly selected among the access and MEC nodes that meet the latency constraint of the flow and the link capacity constraint. The access and MEC nodes "fill up", i.e., get more and more ongoing flows; we count the total number of ongoing flows in the entire network (not at a specific access node). When the total number $|F|$ of flows in the network reaches a prescribed threshold in the range of 100 to 500 flows, we randomly select an access node $a \in A$ whose flows need to be migrated. The reason for considering the total number $|F|$ of flows in the network is that it will significantly impact the migration costs. For instance, more flows in the network result in higher link bandwidth consumption for the entire network, thus influencing the choice of the MEC server after the flow migration. For this selected access node a, we uniformly randomly select the set of target access nodes A_a^{mig} to which the flows can be migrated among the other access nodes (the number of target access nodes is pre-defined; in Section V-D1, we consider $|A_a^{mig}| = 2$ target access nodes; in Section V-D2, we consider $|A_a^{mig}| = 1$ to 6 target access nodes). Then, each flow will either be retained (for a user without mobility) in the original access node or randomly assigned (for a user with mobility) to one of the $|A_a^{mig}|$ target access nodes ($1/(|A_a^{mig}| + 1)$ probability for each of these $|A_a^{mig}| + 1$ options) to form the sets $F_{a,a'}^{mig}$ of flows that are migrated from access node a to the target access nodes $a' \in A_a^{mig}$. The simulation iteration (replication) is stopped after the migration is completed. We conducted 100 independent replications for each evaluation and present the resulting 95% confidence intervals.

C. PERFORMANCE METRICS

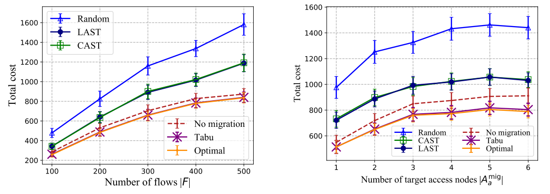
We evaluate the effectiveness of our proposed scheme with six performance metrics: computation cost [Eqn. (3)], communication cost [Eqn. (5)], buffering cost [Eqn. (14)], total cost [Eqn. (18)], total state transfer time, and execution time. The total state transfer time is the time needed to transfer all states of the migrated flows, whereby the time to transfer the states of each flow is calculated by Eqn. (7) and the parameters to evaluate the transmission delays of the state information of the three considered applications follow the approach in [49]. The execution time is the time needed to execute each evaluated scheme.

D. SIMULATION RESULTS

1) IMPACT OF NUMBER OF FLOWS

We first compare the performance of the proposed FAST-Tabu algorithm against the benchmark algorithms operating within the FAST framework by varying the number $|F|$ of flows at which the migration is triggered from 100 to 500 with a step size of 100 flows. The number of target access nodes is set to $|A_a^{mig}| = 2$.

Fig. 6(a) plots the total cost of the evaluated schemes as a function of the number $|F|$ of flows that triggers the flow migration. As expected, Fig. 6(a) shows an upward trend in the total cost with increasing number of flows at which the migration is performed. In particular, the optimal and Tabu schemes achieve lower total costs than the LAST and CAST schemes, which in turn achieve lower total costs than the random scheme.



(a) Total cost vs. Number of flows (b) Total cost vs. Number of target access nodes

FIGURE 6. Total cost as a function of the total number of flows in the network that trigger flow migration and the number of target access nodes.

In particular, the results in Fig. 6(a) indicate that the optimal and Tabu schemes incur only about half the total cost of the random scheme. The optimal and Tabu schemes have a tendency to achieve slightly lower total cost than the no migration scheme. Importantly, the no migration scheme introduced a significant flow deadline violation probability of 42% for the scenarios considered in Fig. 6(a); the flow deadline violation probability is constant with respect to the number of flows that trigger a migration. A flow deadline violation occurs after a migration if the new ingress access node a' (selected uniformly randomly among the target access nodes in A_a^{mig}) has a propagation delay from the (non-migrated) MEC server m of the flow that exceeds the flow deadline D_f. The five evaluated migration schemes never caused a flow deadline violation in our simulations. Unlike the no migration scheme, which retains the migrated flows at the original (non-migrated) MEC server (egress node) m, the migration schemes freely adopt an MEC server (egress node) based on their respective strategies.

We next investigate the communication cost while varying the number $|F|$ of flows that trigger flow migration in Fig. 7(a). Fig. 7(a) verifies our expectation that a higher number of required flows to trigger a migration, results in a higher communication cost. This is because the increase in the number of flows in the network results in more migrated flows, thus increasing the communication cost. Fig. 7(a) indicates that the proposed Tabu search scheme achieves similar performance compared to the optimal scheme, while outperforming the no migration and random schemes. As CAST and LAST adopt the MEC servers according to the computed shortest paths, CAST and LAST achieve the lowest communication cost.

We proceed to examine the buffering cost while varying the number $|F|$ of flows that trigger flow migration in Fig. 7(b). Fig. 7(b) also points out the upward trend in terms of the buffering cost. The results indicate that the optimal scheme and the proposed Tabu search scheme achieve similar performance while performing much better than the other schemes. Notably, the proposed Tabu search scheme incurs only approximately one third of the buffering cost of the LAST, CAST, and random schemes.

We study the computation cost of the evaluated schemes as a function of the number $|F|$ of flows that trigger the

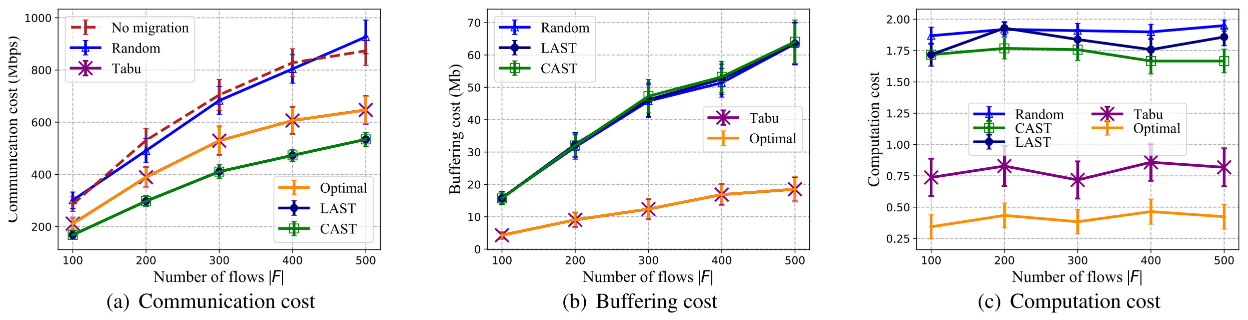


FIGURE 7. Communication cost, buffering cost, and computation cost as a function of the number of flows in the network that trigger flow migration.

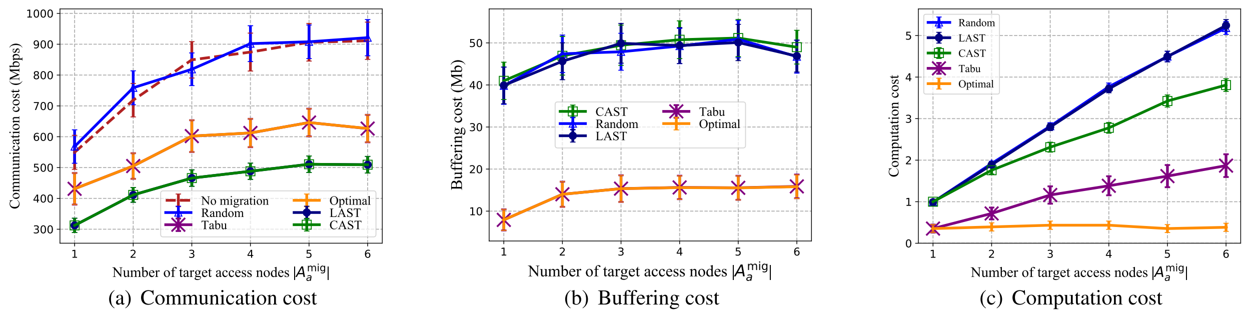


FIGURE 8. Communication cost, buffering cost, and computation cost as a function of the number of target access nodes.

migration in Fig. 7(c). The flat curves in Fig. 7(c) indicate that the number of flows has an insignificant impact on the computation cost. This is mainly because we considered only $|A_a^{mig}| = 2$ target access nodes and the flows from a given target access node a' are typically associated with a close-by MEC server to meet the flow deadlines. The results show that the proposed Tabu search scheme incurs higher computation cost than the optimal scheme, but outperforms the other schemes. This is because the optimal scheme and the proposed Tabu search scheme strive to find the MEC servers that serve as many migrated flows as possible, i.e., strive to minimize the extra required VMs to serve the migrated flows. Meanwhile, the LAST, CAST, and random schemes typically require a new VM for the MEC server that is close-by to each target access node. LAST and CAST choose the best MEC server for each flow for their respective optimization objectives. They do not take into consideration whether that MEC server also serves other flows or not (i.e., do not try to minimize the extra number of required VMs at MEC servers to serve the migrated flows). Similarly, the random scheme chooses a random MEC server without considering whether that MEC server also serves other flows or not. In this sense, the LAST, CAST, and random schemes are "memoryless" as they do not keep track of the MEC servers that flows are migrated to.

We next investigate the total state transfer time as a function of the number $|F|$ of flows that trigger migration in Fig. 9(a). Fig. 9(a) indicates that the optimal scheme and the proposed Tabu search scheme significantly shorten the total state transfer time; down to roughly one third of the other schemes. This is because LAST and CAST ignore the state transfer cost, preferring the shortest paths from the target access nodes to the target MEC servers, which may be far away from the original MEC servers. Thus, the LAST and CAST schemes can introduce very large total transfer times. Similarly, the random scheme simply chooses a target MEC server that satisfies the constraints (without considering the state transfer cost).

2) IMPACT OF NUMBER OF TARGET ACCESS NODES

In this section, we evaluate the performance as a function of the number $|A_a^{mig}|$ of target access nodes in the range from one to six, while setting the number $|F|$ of flows in the network that trigger the migration to 300.

Fig. 6(b) plots the total cost as a function of the number of target access nodes. The results confirm the superiority of the optimal scheme and the proposed Tabu search scheme in all cases. It is worth noting that the gap between the optimal and Tabu schemes and the no migration scheme tends to increase with more target access nodes. This implies that the migration offers more cost benefit than the no migration scheme when the number of target access nodes is large. We note for completeness that the no migration scheme had a flow deadline violation probability of 47.2% for the scenarios in Fig. 6(b) and is therefore not further considered.

Fig. 8(a) shows the communication cost for different numbers of target access nodes. The results are similar to the results in Fig. 7(a). However, the optimal scheme and the proposed Tabu search scheme increase the gap to the no migration scheme and the random scheme, especially for large numbers of target access nodes. This increased gap for the communication cost significantly contributes to the widening gap in the total cost in Fig. 6(b). For an increasing number of target access nodes, the communication cost is mainly governed by the effectiveness of the schemes in finding the optimal paths.

Fig. 8(b) plots the buffering cost while varying the number of target access nodes. The flat curves in Fig. 8(b) indicate the insignificant impact of the number of target access nodes on the buffering cost. The reason is that the buffering cost mainly depends on the number of migrated flows (and not on the number of target access nodes). The optimal and the Tabu schemes consistently require less than roughly a third of the buffering cost of the LAST, CAST, and random schemes.

We investigate the computation cost while varying the number of target access nodes in Fig. 8(c). Fig. 8(c) shows the upward trend in the computation cost for increasing numbers of target access nodes. This is mainly because the larger number of target access nodes typically requires new VMs for the migrated flows at more MEC servers. Fig. 8(c) indicates that our proposed Tabu search scheme incurs higher computation cost than the optimal scheme, while incurring substantially lower (nearly only half) computation cost compared to the LAST, CAST, and Random schemes. This is mainly because the optimal and proposed Tabu search schemes strive to find the MEC servers that can serve the migrated flows most effectively, i.e., while requiring the least numbers of VMs to be added to the MEC servers.

We investigate the total state transfer time for increasing numbers of target access nodes in Fig. 9(b) (while the total number of access nodes in the network remains fixed at $|A| = 10$). Fig. 9(b) shows that the larger the number of target access nodes, the longer the total state transfer time. The reason is that more target access nodes result in more target MEC servers. Thus, the number of state transfers to new MEC servers is increased. Notably, the results show that the optimal scheme and the proposed Tabu search scheme clearly outperform the other schemes in terms of the total state transfer time.

We proceed to examine the computational complexity (execution time) of the proposed Tabu search scheme in comparison with the other schemes for a varying number of target access nodes in Fig. 10. Notably, Fig. 10 shows that the execution time of the proposed Tabu search scheme is much less than for the optimal scheme for two or more target access nodes. For 6 access nodes, the proposed Tabu search scheme reduces the execution time down to nearly one eighth compared to the optimal scheme. The complexity of the optimal scheme is drastically increased when the number of target access nodes is large. Meanwhile, the number of target access nodes has a negligible impact on the execution time of the proposed Tabu search scheme because the Tabu search algorithm controls the complexity by using the Tabu list, see Section III. The LAST, CAST, and random schemes are low-complexity heuristic algorithms that quickly find the target MEC servers.

In summary, our proposed Tabu search scheme achieves mostly similar performance but introduces much less complexity compared to the optimal scheme. Thus, the proposed Tabu search scheme appears well suited for MEC state transfer optimization.

VI. FAST EXPERIMENTS ON A REAL TESTBED

Unlike the data center, the MEC is a novel cloud paradigm that has not been widely considered in real-world implementations. Therefore, there is no accepted reference network topology for the MEC available yet. Thus, we consider a testbed that consists of six MEC servers, one traffic generator, one controller server, and one Aruba 2930F hardware switch with SDN capabilities. The MEC servers and the traffic generator are equipped with Intel(R) Core(TM) i7-6700 CPU at 3.40 GHz and 32 GB memory, running on Ubuntu Bionic 18.04. The MEC servers are used to deploy the examined applications, whereby each of the two MEC servers will host the source instance and destination instance of an application, respectively. Tcpreplay [73] is used as the traffic generator, replaying the network traffic to the MEC servers from a recorded pcap file. We implemented FAST and OpenNF on the controller server equipped with Intel(R) Xeon(R) CPU E5-2630 v3 (32 cores) at 2.40 GHz and 128 GB memory, running on Ubuntu Xenial 16.04. Our experimental results represent the averages and 95% confidence intervals of 10 measurements.

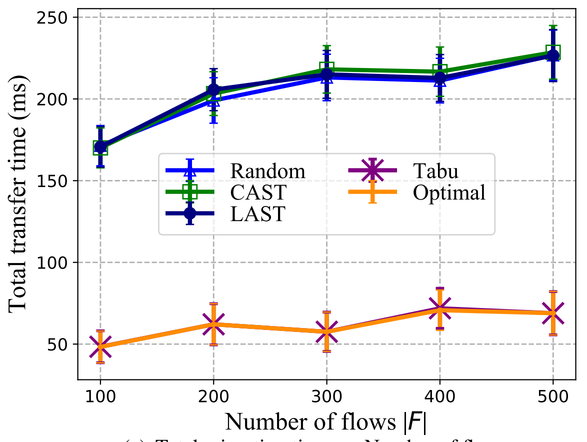
We examined three applications, namely PRADS, Iptables [89], and Zeek [88] (formerly Bro), which are deployed in bare metal fashion [95], [96] on the MEC servers. The major benefit of these applications is to improve the network security. As stated in [17], security and privacy issues introduce critical MEC challenges. Thus, the considered applications could potentially be deployed to secure the user traffic at the network edge. Since PRADS has been introduced in Section IV-D2, we briefly introduce Iptables and Zeek as follows.

Iptables is a well-known user-space utility program for configuring IP packet filter rules of the Linux kernel firewall. Iptables defines tables containing chains of rules for packet processing, whereby packets are processed by sequentially traversing the rules in chains. Iptables uses a connection tracking module (conntrack) to manage the state entries.

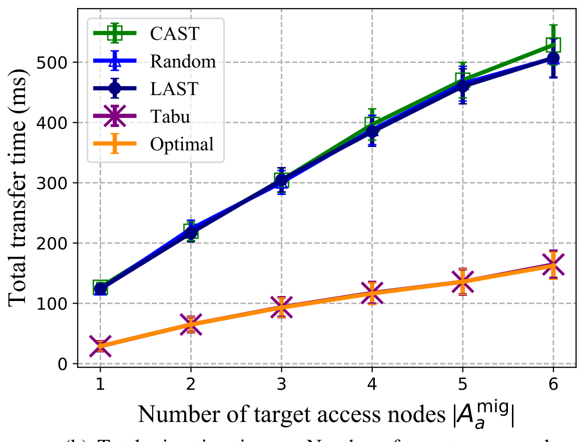
Zeek is a popular open-source platforms for network security monitoring. Zeek inspects all traffic on a network interface in depth for signs of suspicious activity. Zeek can perform a wide range of analysis and detection functions, e.g., detect malware via external registries, and validate SSL certificate chains. To perform these functions, Zeek has built-in modules that track and manage the network states.

Since our main focus is on the state transfer, we first reused existing functionalities in OpenNF to allow exporting/importing the states from/to PRADS, Iptables, and Zeek. We then modified these applications to enable FAST, see Section IV-C.

Fig. 11 plots the total move time for the different applications at 100 packet flows. Importantly, the results indicate that FAST achieves significantly shorter total move times than OpenNF. Specifically, compared to OpenNF, FAST reduces the total move times for Iptables, PRADS, and Zeek as follows: 48.6% (from 68.33 ms down to 35.13 ms) for Iptables,



(a) Total migration time vs. Number of flows



(b) Total migration time vs. Number of target access nodes

FIGURE 9. Total migration time as a function of the number of flows in the network that trigger a migration and the number of target access nodes.

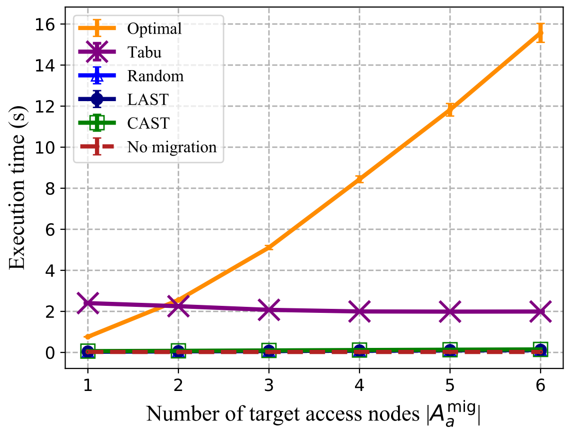


FIGURE 10. Execution time as a function of the number of target access nodes.

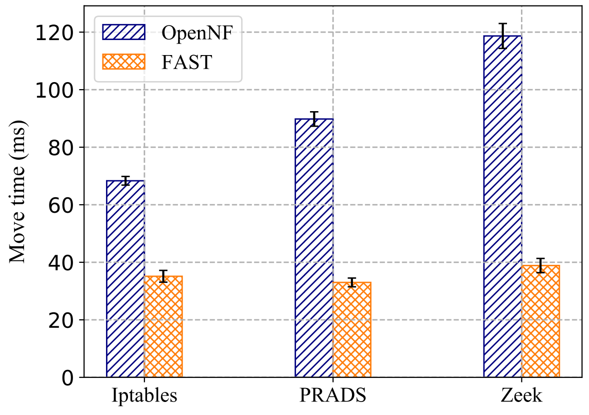


FIGURE 11. Total move time in real testbed for different applications at 100 packet flows.

63.3% (from 89.8 ms down to 33 ms) for PRADS, and 67.3% (from 118.67 ms down to 38.87 ms) for Zeek. With OpenNF, the total Zeek move time is significantly longer than the Iptables move time. This is mainly because Zeek has a larger state size than Iptables. The involvement of the OpenNF controller in the state transfer leads to a significant impact of the state size on the total move time.

VII. CONCLUSION

Frequent user mobility necessitates quick and flexible migration of application states without service interruption, while minimizing the state transfer costs. We have proposed a meta-heuristic algorithm based on Tabu search to solve the MEC state transfer optimization problem. The simulation results indicate favorable performance characteristics of the proposed Tabu search algorithm in terms of the total transfer cost. Notably, the proposed Tabu search scheme incurs significantly lower computational complexity than the optimal scheme, thus confirming its applicability in the MEC.

To migrate application states quickly, reliably, and in a flexible manner, we have introduced FAST, a state management scheme running at the SDN controller to coordinate the state transfer process. FAST allows for the forwarding of application states directly between the MEC application instances (without transferring the VM or container states). Furthermore, FAST provides an API that allows for further extensions. We make the full FAST framework source code publicly available at https://github.com/openMECPlatform/fast.git. We conducted a FAST performance measurement for various real-world applications on a real testbed. The measurement results demonstrate that FAST reduces the total move time by over 60% compared to the existing OpenNF approach.

We identify three future research directions for our FAST framework. First, since FAST offers direct state transfer between the source and destination MEC servers that host the MEC application before and after the migration, the support of a mechanism in FAST to predict the user mobility could be helpful to facilitate the process of finding the destination MEC server i.e., the target access node and destination MEC server could be determined in advance and thus the application states could be tentatively transferred before a final migration decision is made; upon making the final migration decision, we would then only need to re-direct the user's traffic to the new destination MEC server. To achieve this goal, machine learning techniques, such as deep learning and reinforcement learning, can be employed. However, the use of the existing machine learning techniques in MEC is known to introduce significant computation complexity [97], thus possibly prolonging the migration process. Future research should develop low-complexity machine learning techniques for this mobility prediction.

A second future research direction is the support of a programmable data plane in FAST. Programmable data planes have gained tremendous interest in recent years [98], allowing the network devices to perform complex operations on packets. Since FAST advocates the programmable state forwarding, the use of programmable data planes, such as POF (Protocol-Oblivious Forwarding) [99] and P4 (Programming protocol-independent packet processors) [100], in FAST can potentially greatly improve the flexibility of the state transfer. However, the programmable data planes are still in early-stage development, and may give rise to critical issues, such as security violations and performance degradation [98]. For the security issue, the application state (which might contain user-sensitive data) can be vulnerable. Meanwhile, the performance degradation can lead to state loss and introduce long migration times. Therefore, the trade-offs arising from programmable data planes for the application state migration in FAST need to be thoroughly examined in future research.

This study primarily targeted the reduction of the MEC state transfer latency, which is an important MEC QoS metric [2]. A third future research direction is to broaden the network model to incorporate other QoS metrics, e.g., the priority of the incoming service requests. Additionally, future research could consider the efficient use of storage resources and the reduction of the power consumption for the MEC state transfer. As the examined state transfer only captures the data needed to recover the application after the migration, our proposed solution could potentially save storage resources and reduce the power consumption (i.e., require less energy to recover the application) compared to the traditional container or VM migration.

ACKNOWLEDGMENT

This article was presented at the 2020 IEEE Global Communications Conference: Next-Generation Networking and Internet (Globecom2020, NGNI), Taipei, Taiwan. A preliminary abridged version of this study has appeared in [DOI: 10.1109/GLOBECOM42002.2020.9322117].

REFERENCES

[1] T. V. Doan, C. Ding, G. T. Nguyen, D. You, and F. H. P. Fitzek, "FAST: Flexible and low-latency state transfer in mobile edge computing," in *Proc. GLOBECOM IEEE Global Commun. Conf.*, Dec. 2020, pp. 1–6.

[2] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, and M. Guizani, "Multi-access edge computing: A survey," *IEEE Access*, vol. 8, pp. 197017–197046, 2020.

[3] M. Liu and Y. Liu, "Price-based distributed offloading for mobile-edge computing with computation capacity constraints," *IEEE Wireless Commun. Lett.*, vol. 7, no. 3, pp. 420–423, Jun. 2018.

[4] Y. Mansouri and M. A. Babar, "A review of edge computing: Features and resource virtualization," *J. Parallel Distrib. Comput.*, vol. 150, pp. 155–183, Apr. 2021.

[5] Q.-V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W.-J. Hwang, and Z. Ding, "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, vol. 8, pp. 116974–117017, 2020.

[6] S. D. A. Shah, M. A. Gregory, S. Li, and R. D. R. Fontes, "SDN enhanced multi-access edge computing (MEC) for E2E mobility and QoS management," *IEEE Access*, vol. 8, pp. 77459–77469, 2020.

[7] S. D. A. Shah, M. A. Gregory, and S. Li, "Cloud-native network slicing using software defined networking based multi-access edge computing: A survey," *IEEE Access*, vol. 9, pp. 10903–10924, 2021.

[8] H. Zhang, Y. Yang, X. Huang, C. Fang, and P. Zhang, "Ultra-low latency multi-task offloading in mobile edge computing," *IEEE Access*, vol. 9, pp. 32569–32581, 2021.

[9] T. V. Doan, D. You, H. Salah, G. T. Nguyen, and H. P. Frank Fitzek, "MEC-assisted immersive services: Orchestration framework and protocol," in *Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcast. (BMSB)*, Jun. 2019, pp. 1–6.

[10] T. V. Doan, Z. Fan, G. T. Nguyen, D. You, A. Kropp, H. Salah, and F. H. P. Fitzek, "Seamless service migration framework for autonomous driving in mobile edge cloud," in *Proc. IEEE 17th Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2020, pp. 1–2.

[11] H. Ma, S. Li, E. Zhang, Z. Lv, J. Hu, and X. Wei, "Cooperative autonomous driving oriented MEC-aided 5G-V2X: Prototype system design, field tests and AI-based optimization tools," *IEEE Access*, vol. 8, pp. 54288–54302, 2020.

[12] T. Hoeschele, C. Dietzel, D. Kopp, F. H. P. Fitzek, and M. Reisslein, "Importance of internet exchange point (IXP) infrastructure for 5G: Estimating the impact of 5G use cases," *Telecommun. Policy*, vol. 45, no. 3, Apr. 2021, Art. no. 102091.

[13] J. Nakazato, M. Nakamura, T. Yu, Z. Li, K. Maruta, G. K. Tran, and K. Sakaguchi, "Market analysis of MEC-assisted beyond 5G ecosystem," *IEEE Access*, vol. 9, pp. 53996–54008, 2021.

[14] E. Pencheva, D. Velkova, and I. Atanasov, "Edge based mission critical session control," in *Proc. Int. Conf. Inf. Technol. (InfoTech)*, Sep. 2020, pp. 1–4.

[15] T. V. Doan, G. T. Nguyen, H. Salah, S. Pandi, M. Jarschel, R. Pries, and F. H. P. Fitzek, "Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud," in *Proc. IEEE 2nd 5G World Forum (5GWF)*, Sep. 2019, pp. 46–52.

[16] P. Shantharama, A. S. Thyagaturu, A. Yatavelli, P. Lalwaney, M. Reisslein, G. Tkachuk, and E. J. Pullin, "Hardware acceleration for container migration on resource-constrained platforms," *IEEE Access*, vol. 8, pp. 175070–175085, 2020.

[17] 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. 2018.

[18] W. Lu, X. Meng, and G. Guo, "Fast service migration method based on virtual machine technology for MEC," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4344–4354, Jun. 2019.

[19] K. Qu, W. Zhuang, Q. Ye, X. Shen, X. Li, and J. Rao, "Dynamic flow migration for embedded services in SDN/NFV-enabled 5G core networks," *IEEE Trans. Commun.*, vol. 68, no. 4, pp. 2394–2408, Apr. 2020.

[20] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao, "PLAN: Joint policy- and network-aware VM management for cloud data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1163–1175, Apr. 2017.

[21] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, and Q. Liang, "Communication-aware container placement and reassignment in large-scale internet data centers," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 540–555, Mar. 2019.

[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. and Akella, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.

[23] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *Proc. USENIX (NSDI)*, 2013, pp. 227–240.

[24] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. 4th Annu. Symp. Cloud Comput.*, Oct. 2013, pp. 1–15.

[25] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, Apr. 2019.

[26] M. Silva, P. Teixeira, C. Gomes, D. Dias, M. Luís, and S. Sargento, "Exploring software defined networks for seamless handovers in vehicular networks," *Veh. Commun.*, vol. 31, Oct. 2021, Art. no. 100372.

[27] T. V. Doan, Z. Fan, G. T. Nguyen, H. Salah, D. You, and F. H. P. Fitzek, "Follow me, if you can: A framework for seamless migration in mobile edge cloud," in *Proc. IEEE INFOCOM Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 1178–1183.

[28] S. Pandi, R. S. Schmoll, P. J. Braun, and F. H. P. Fitzek, "Demonstration of mobile edge cloud for tactile internet using a 5G gaming application," in *Proc. 14th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2017, pp. 607–608.

[29] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian, "Transparent flow migration for NFV," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[30] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. USENIX NSDI*, vol. 2018, pp. 299–312.

[31] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, 2015.

[32] *OpenFlow-Enabled SDN and Network Functions Virtualization*, Brief, ONF Solution, Open Networking Foundation, Menlo Park, CA, USA, 2014.

[33] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—A key technology towards 5G," ETSI, Sophia Antipolis, France, ETSI White Paper 11, 2015.

[34] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 2, pp. 1206–1243, 2nd Quart., 2018.

[35] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A survey on service migration in mobile edge computing," *IEEE Access*, vol. 6, pp. 23511–23528, 2018.

[36] Z. Liang, Y. Liu, T.-M. Lok, and K. Huang, "Multi-cell mobile edge computing: Joint service migration and resource allocation," *IEEE Trans. Wireless Commun.*, early access, Apr. 12, 2021, doi: 10.1109/TWC.2021.3070974.

[37] O. Oleghe, "Container placement and migration in edge computing: Concept and scheduling models," *IEEE Access*, vol. 9, pp. 68028–68043, 2021.

[38] Q. Yuan, J. Li, H. Zhou, T. Lin, G. Luo, and X. Shen, "A joint service migration and mobility optimization approach for vehicular edge computing," *IEEE Trans. Veh. Technol.*, vol. 69, no. 8, pp. 9041–9052, Aug. 2020.

[39] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on Markov decision process," *IEEE/ACM Trans. Netw.*, vol. 27, no. 13, pp. 1272–1288, May 2019.

[40] X. Zhou, S. Ge, T. Qiu, K. Li, and M. Atiquzzaman, "Energy-efficient service migration for multi-user heterogeneous dense cellular networks," *IEEE Trans. Mobile Comput.*, early access, Jun. 8, 2021, doi: 10.1109/TMC.2021.3087198.

[41] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Aug. 2009, pp. 1–10.

[42] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2011, pp. 111–120.

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

[44] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Trans. Mobile Comput.*, vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[45] F. Carpio, A. Jukan, and R. Pries, "Balancing the migration of virtual network functions with replications in data centers," in *Proc. NOMS IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2018, pp. 1–8.

[46] Y. Wang, X. Zhu, and X. Qiu, "A quick adaptive migration algorithm for virtual network function," in *Proc. Int. Conf. Wireless Satell. Syst.* Cham, Switzerland: Springer, 2019, pp. 333–347.

[47] R. Bruschi, F. Davoli, P. Lago, and J. F. Pajo, "A multi-clustering approach to scale distributed tenant networks for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 499–514, Mar. 2019.

[48] N. Deric, A. Varasteh, A. Basta, A. Blenk, R. Pries, M. Jarschel, and W. Kellerer, "Coupling VNF orchestration and SDN virtual network reconfiguration," in *Proc. Int. Conf. Netw. Syst. (NetSys)*, Mar. 2019, pp. 1–3.

[49] C. Sun, J. Bi, Z. Meng, T. Yang, X. Zhang, and H. Hu, "Enabling NFV elasticity control with optimized flow migration," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2288–2303, Oct. 2018.

[50] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless NFV state transfer," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 4, pp. 964–977, Dec. 2017.

[51] M. Ghaznavi, E. Jalalpour, B. Wong, R. Boutaba, and A. J. Mashtizadeh, "Fault tolerant service function chaining," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 198–210.

[52] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX NSDI*. Boston, MA, USA, Mar. 2017, pp. 97–112.

[53] J. Santa, A. F. Skarmeta, J. Ortiz, P. J. Fernandez, M. Luis, C. Gomes, J. Oliveira, D. Gomes, R. Sanchez-Iborra, and S. Sargento, "MIGRATE: Mobile device virtualisation through state transfer," *IEEE Access*, vol. 8, pp. 25848–25862, 2020.

[54] Z. Cao, A. Abujoda, and P. Papadimitriou, "Distributed data deluge (D3): Efficient state management for virtualized network functions," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 782–787.

[55] M. Peuster and H. Karl, "E-state: Distributed state management in elastic network function deployments," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 6–10.

[56] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr," in *Proc. USENIX NSDI*, Mar. 2016, pp. 239–253.

[57] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, "U-HAUL: Efficient state migration in NFV," in *Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst.*, Aug. 2016, pp. 1–8.

[58] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer, "Flexibility in softwarized networks: Classifications and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2600–2636, 3rd Quart., 2019.

[59] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba, "How to measure network flexibility? A proposal for evaluating softwarized networks," *IEEE Commun. Mag.*, vol. 56, no. 10, pp. 186–192, Oct. 2018.

[60] C. Trois, M. D. Del Fabro, L. C. E. de Bona, and M. Martinello, "A survey on SDN programming languages: Toward a taxonomy," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 4, pp. 2687–2712, 4th Quart., 2016.

[61] M. He, A. Basta, A. Blenk, and W. Kellerer, "Modeling flow setup time for controller placement in SDN: Evaluation for dynamic flows," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–7.

[62] A. AlGhadhban and B. Shihada, "Delay analysis of new-flow setup time in software defined networks," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2018, pp. 1–7.

[63] R. Khalili, Z. Despotovic, and A. Hecker, "Flow setup latency in SDN networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2631–2639, Dec. 2018.

[64] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, Aug. 2015, pp. 43–48.

[65] N. Mladenović, J. Brimberg, P. Hansen, and J. A. Moreno-Pérez, "The *p*-median problem: A survey of metaheuristic approaches," *Eur. J. Oper. Res.*, vol. 179, no. 3, pp. 927–939, 2007.

[66] K. Hussain, M. N. M. Salleh, S. Cheng, and Y. Shi, "Metaheuristic research: A comprehensive survey," *Artif. Intell. Rev.*, vol. 52, no. 4, pp. 2191–2233, Dec. 2019.

[67] M. Dorigo and T. Stützle, "The ant colony optimization metaheuristic: Algorithms, applications, and advances," in *Handbook Metaheuristics*. Boston, MA, USA: Springer, 2003, pp. 250–285.

[68] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, Jun. 1994.

[69] F. Glover and M. Laguna, "Tabu search," in *Handbook of Combinatorial Optimization*. New York, NY, USA: Springer, 1998, pp. 2093–2229.

[70] J. Silberholz and B. Golden, "Comparison of metaheuristics," in *Handbook Metaheuristics*. Boston, MA, USA: Springer, 2010, pp. 625–640.

[71] *Mininet, an Instant Virtual Network on Your Laptop (or Other PC)*. Accessed: Aug. 17, 2021. [Online]. Available: http://mininet.org

[72] *PRADS, Passive Real-time Asset Detection System*. Accessed: Aug. 17, 2021. [Online]. Available: http://gamelinux.github.io/prads
[73] *Tcpreplay, an Open Source Network Security Monitoring Tool*. Accessed: Aug. 17, 2021. [Online]. Available: https://tcpreplay.appneta.com/
[74] *Gurobi Optimization*. Accessed: Aug. 17, 2021. [Online]. Available: https://www.gurobi.com
[75] *NetworkX, Network Analysis in Python*. Accessed: Aug. 17, 2021. [Online]. Available: https://networkx.github.io
[76] P. Erdős and A. Rényi, "On random graphs I," *Pub. Math. (Debrecen)*, vol. 6, pp. 290–297, Jan. 1959.
[77] M. Bouet and V. Conan, "Mobile edge computing resources optimization: A Geo-clustering approach," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 2, pp. 787–796, Jun. 2018.
[78] M. A. T. Nejad, S. Parsaeefard, M. A. Maddah-Ali, T. Mahmoodi, and B. H. Khalaj, "vSPACE: VNF simultaneous placement, admission control and embedding," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 542–557, Mar. 2018.
[79] N. Tastevin, M. Obadia, and M. Bouet, "A graph approach to placement of service functions chains," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 134–141.
[80] Z. Nezami, K. Zamanifar, K. Djemame, and E. Pournaras, "Decentralized edge-to-cloud load balancing: Service placement for the Internet of Things," *IEEE Access*, vol. 9, pp. 64983–65000, 2021.
[81] B. Gao, Z. Zhou, F. Liu, F. Xu, and B. Li, "An online framework for joint network selection and service placement in mobile edge computing," *IEEE Trans. Mobile Comput.*, early access, Mar. 9, 2021, doi: 10.1109/TMC.2021.3064847.
[82] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan, "Toward computation offloading in edge computing: A survey," *IEEE Access*, vol. 7, pp. 131543–131558, 2019.
[83] C. Mouradian, S. Kianpisheh, M. Abu-Lebdeh, F. Ebrahimnezhad, N. T. Jahromi, and R. H. Glitho, "Application component placement in NFV-based hybrid cloud/fog systems with mobile fog nodes," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1130–1143, May 2019.
[84] S. Yang, F. Li, S. Trajanovski, X. Chen, Y. Wang, and X. Fu, "Delay-aware virtual network function placement and routing in edge clouds," *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 445–459, Feb. 2021.
[85] S. Song, C. Lee, H. Cho, G. Lim, and J.-M. Chung, "Clustered virtualized network functions resource allocation based on context-aware grouping in 5G edge networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 5, pp. 1072–1083, May 2020.
[86] J. Pei, P. Hong, K. Xue, and D. Li, "Resource aware routing for service function chains in SDN and NFV-enabled network," *IEEE Trans. Services Comput.*, vol. 14, no. 4, pp. 985–997, Jul. 2021.
[87] M. Ghaznavi, N. Shahriar, S. Kamali, R. Ahmed, and R. Boutaba, "Distributed service function chaining," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2479–2489, Nov. 2017.
[88] *Zeek (Formerly Bro), an Open Source Network Security Monitoring Tool*. Accessed: Aug. 17, 2021. [Online]. Available: https://zeek.org/
[89] *Linux Iptables*. Accessed: Aug. 17, 2021. [Online]. Available: https://linux.die.net/man/8/iptables
[90] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO—Simulation of urban mobility," *Int. J. Adv. Syst. Meas.*, vol. 5, nos. 3–4, pp. 128–138, Dec. 2012.
[91] A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE simulator for DTN protocol evaluation," in *Proc. 2nd Int. ICST Conf. Simulation Tools Techn.*, 2009, pp. 1–10.
[92] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, Oct. 2018.
[93] G. Luo, H. Zhou, N. Cheng, Q. Yuan, J. Li, F. Yang, and X. Shen, "Software-defined cooperative data sharing in edge computing assisted 5G-VANET," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 1212–1229, Mar. 2021.
[94] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang, "Mobile edge cloud system: Architectures, challenges, and approaches," *IEEE Syst. J.*, vol. 12, no. 3, pp. 2495–2508, Sep. 2018.
[95] E. Chirivella-Perez, J. M. A. Calero, Q. Wang, and J. Gutiérrez-Aguado, "Orchestration architecture for automatic deployment of 5G services from bare metal in mobile edge computing infrastructure," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–18, Nov. 2018.
[96] P. Shantharama, A. S. Thyagaturu, and M. Reisslein, "Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies," *IEEE Access*, vol. 8, pp. 132021–132085, 2020.
[97] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey," 2019, *arXiv:1908.00080*. [Online]. Available: http://arxiv.org/abs/1908.00080
[98] R. Bifulco and G. Retvari, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–7.
[99] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 127–132.
[100] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

TUNG V. DOAN is currently pursuing the Ph.D. degree with the Deutsche Telekom Chair of Communication Networks, TU Dresden, Germany. From 2016 to 2018, he actively contributed to OpenStack, a popular open-source cloud-computing platform. His research interests include cloud/edge computing, software-defined networking (SDN), and network function virtualization (NFV).

GIANG T. NGUYEN received the Ph.D. degree in computer science from Technical University Dresden (TU Dresden), Germany, in 2016. From 2016 to 2019, he was a Postdoctoral Researcher with the Deutsche Telekom Chair of Communication Networks (ComNets), TU Dresden. After that, he worked at Wandelbots GmbH. Since July 2021, he has been an Assistant Professor with TU Dresden.

MARTIN REISSLEIN (Fellow, IEEE) received the Ph.D. degree in systems engineering from the University of Pennsylvania, Philadelphia, PA, USA, in 1998. He is currently a Professor with the School of Electrical, Computer, and Energy Engineering, Arizona State University (ASU), Tempe, AZ, USA. He is also an Associate Editor of IEEE ACCESS, IEEE TRANSACTIONS ON EDUCATION, and IEEE TRANSACTIONS ON MOBILE COMPUTING.

FRANK H. P. FITZEK (Senior Member, IEEE) received the Dipl.-Ing. degree in electrical engineering from the Rheinisch-Westfälische Technische Hochschule (RWTH), Aachen, Germany, in 1997, and the Ph.D. (Dr.-Ing.) degree in electrical engineering from Technical University of Berlin, Berlin, Germany, in 2002. He is currently a Professor and the Head of the Deutsche Telekom Chair of Communication Networks, Technical University Dresden, Germany, coordinating the 5G Lab Germany.