MEFS: Mobile Edge File System for Edge-Assisted Mobile Apps

Domenico Scotece*, Nafize R. Paiker†, Luca Foschini*,
Paolo Bellavista*, Xiaoning Ding†, and Cristian Borcea†
*Department of Computer Science and Engineering, University of Bologna, 40126 Bologna BO, Italy
†Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
Email:*{domenico.scotece, luca.foschini, paolo.bellavista}@unibo.it,
†{nrp48, xiaoning.ding, borcea}@njit.edu

***Abstract*—Computation offloading is employed by mobile apps running over resource-constrained devices to leverage the cloud in overcoming their resource limits. The advent of the Multi-access Edge Computing (MEC) paradigm further extends the potential opportunities of mobile-cloud offloading, allowing new service provisioning scenarios, such as mobile gaming and multimedia, where responsiveness of mobile devices at the network edge significantly benefits from low latency interactions. However, state-of-the-art offloading platforms for MEC architectures have not addressed the technical challenge of supporting specific file systems for this MEC-enabled class of applications, with components running at three hosting environments, i.e., mobile, edge, and cloud. This paper proposes the Mobile Edge File System (MEFS), an application-level distributed file system designed to be highly resilient and able to efficiently maintain consistency among the mobile, edge, and cloud entities. MEFS supports application handoff through live migration as end devices move between edges. The cloud transparently helps with recovery from faulty edge nodes or in the case of unavailability of edges in the user's proximity. We implemented a MEFS prototype in Android along with MEFS-based MEC-enabled mobile apps. The experimental results show how MEFS can achieve low latency and low overhead.**

Keywords—MEC, cloud, mobile devices, task offloading, file system, handoff, fault-tolerance

I. Introduction

During the past decade, the users' requirements on data rates and Quality of Service (QoS) have increased substantially. Furthermore, the technological evolution of smart phones has led to mobile apps requiring huge processing power, while the battery life and power consumption still pose significant technical challenges toward achieving optimal users' Quality of Experience (QoE). This motivates the idea and development of Mobile Cloud Computing (MCC) platforms [1], which allow mobile users to seamlessly leverage powerful resources available in the cloud. MCC has already demonstrated several advantages in terms of QoS, QoE, and energy consumption; for instance, it enables computation offloading from mobile users to the cloud [2-6]. However, MCC solutions often exhibit the drawback of increased latency due to mobile-to-cloud communication. One promising research direction is to leverage edge computing for computation offloading in a mobile-edge-cloud architecture in order to reduce the communication latency and the overall app response time.

The edge computing paradigm enables the deployment of middleboxes for supporting and enhancing service provisioning at the locations of mobile users. This allows improved scalability and reactivity in the interaction with mobile nodes when control decisions are applicable. As an architectural model and specification for edge computing, Multi-access Edge Computing (MEC) [7] proposal by the European Telecommunications Standards Institute (ETSI) evolves the traditional two-layer MCC model with the introduction of a third intermediate middleware layer that is executed at the network edge and allows users to access the services at the edge of the network, e.g., computation offloading service for mobile apps.

Compared to MCC, MEC can offer significantly lower latency and jitter; moreover, since MEC can be deployed in a fully distributed manner, it can improve the overall scalability for mobile apps. In the MEC paradigm, a MEC-enabled application (referred to as edge-assisted app) can have components running at three hosting environments, i.e., the mobile device, the cloud, and the edge servers that are selected based on the current location of the mobile device and may change when the mobile device moves.

Despite the potential advantages of the MEC paradigm, MEC platforms have not addressed yet the technical challenge of supporting specific file systems for the envisioned MEC-enabled class of applications. Therefore, computation offloading cannot be employed for most mobile apps because they need concurrent accesses to files from their hosting environments across mobile devices, dynamically selected edge servers, and the cloud.

With the growing amount of data available to mobile apps, the problem of a MEC-enabled file system could not be ignored anymore. Unfortunately, existing network file systems, such as NFS [8] and Dropbox, are not effective in handling file access for offloaded tasks of mobile apps due to their limited support for remote file access. Recently, the Overlay File System (OFS) [9] proposed support for concurrent and consistent file access on both the mobile and the cloud by working in conjunction with its task offloading middleware. Despite its advancements, OFS and similar solutions designed for MCC do not fit the MEC scenarios primarily for two reasons: 1) they cannot handle the switch of edge nodes when mobile users move; and 2) they are not resilient to the faults in MEC edges.

By carefully considering these relevant gaps, this paper proposes Mobile Edge File System (MEFS), a file system that

runs on mobile devices, edge nodes, and the cloud to efficiently and seamlessly handle file accesses for edge-assisted mobile apps. To the best of our knowledge, this is the first work that presents the design, implementation, and prototyping of such a system solution. MEFS supports file accesses with low-latency for the components of a mobile app that possibly offloads some tasks to either the edge or the cloud (*mobile-to-edge or mobile-to-cloud offloading*), and guarantees strong data consistency between these components. It is fully compatible with the MEC standard specifications. With MEFS, an entirely new class of mobile apps (i.e., apps that need access to files) can take advantage of the MEC infrastructure for faster response time and lower energy consumption on mobiles.

There are two major challenges in MEFS design. One is how to manage automatically session handoff between edge nodes; and the other is how to tolerate the faults in edge nodes and prevent data loss. MEFS manages automatic session handoff between edge nodes by following the user's node mobility. To minimize service downtime, it monitors and predicts the mobility path of each mobile node, and uses a mechanism similar to VM live migration to migrate file data and metadata, as well as session state, between the edge nodes along the mobility path.

To prevent the data loss caused by the faults in edge nodes (e.g., node crashes or becoming unreachable from the mobile node), the fault-tolerance design in MEFS uses a log-based approach: edge nodes send write operations to the cloud; when an edge node goes down or there are no available edge nodes in the mobile user's area, the cloud is able to restore the edge session by exploiting its log.

MEFS contributes to the literature in the field in multiple ways. First, the paper proposes and designs MEFS to make it possible that the components in edge-assigned apps can access files concurrently and consistently from cloud, edge, and mobile nodes. Second, we have implemented a MEFS prototype based on Android. This paper also presents the insight of MEFS implementation. Third, in order to test the MEFS design, we have implemented MEC-enabled mobile apps, one of which is a real-time video analytics mobile app. Each app can utilize the assistance from either the edge or the cloud, such that we can use the app to compare the performance of MEFS with that of a mobile-cloud file system and demonstrate the benefits of MEFS. Forth, we have conducted extensive experiments with a test app and real mobile user traces, to validate the functionality and performance of MEFS. The collected experimental results demonstrate that 1) the total time of handoff is about 300ms for more than 100 read/write operation messages processed, and 2) the cost to recover data and system state when an edge node fails does not exceed 130KB for more than 100 operation messages. Therefore, we conclude that MEFS works well in practice, achieving low latency with a small overhead.

The remainder of this paper is organized as follows. Section II provides the necessary background material, and shows an overview of the most important related literature. The requirements and the architecture of MEFS are described in Section III. Section IV presents the implementation highlights of our prototype. Section V reports the performance results. The paper concludes in Section VI.

II. Related Work

Existing solutions for MCC offloading. Computational offloading for MCC is addressed, among the others, in CloneCloud [6], MAUI [5], ThinkAir, [10] and COMET [4], which propose task offloading onto a centralized surrogate in the cloud. CloneCloud migrates threads to application-level VMs. MAUI and ThinkAir adopt the offloading approach with method granularity. COMET provides distributed shared memory to support thread offloading. Even though all of them have shown the benefits of offloading for speedup and energy savings, this only holds for cases in which little shared state synchronization between mobile devices and the cloud is necessary. MEFS solves this problem by providing strong consistency for concurrent file access.

Exploiting network edges. As an evolution to the above solutions for MCC, a few frameworks have been already proposed with a MEC or Fog oriented approach.

MECO (Mobile-edge computation offloading) [11] is a technique proposed for prolonging the battery life and enhancing the computational capacity of mobile nodes. It aims at minimizing mobile energy consumption by considering the computation overhead and the available resources at MEC. The proposed algorithm calculates an offloading priority for each user. In addition, MECO is focused on model aspects and does not consider MEC challenges such as app portability and resiliency. MEFS, on the other hand, overcomes these challenges and guarantees consistency and low latency as well. CloudAware [12] presents a programming model and a framework that directly fit the common app developer's mindset to design elastic and scalable MEC-based mobile applications with extremely low response time, e.g., multimedia applications. But these applications are typically stateless since the server is agnostic of the client state. Therefore, if the applications need to keep server-side state, a mobility management technique is needed to manage service/state migration. Unlike CloudAware, MEFS provides support for mobility management with state migration, which is transparent for developers, and for a consistent low-latency file system.

Distributed and network file system. Various distributed and network file systems [8, 13-20] allow users to access/share their files. However, these solutions are not ideal for offloaded tasks at the edge due to several reasons. First, the conventional distributed and network file systems do not provide support for tasks that have opened files during offloading. Second, these file systems do not provide an appropriate level of consistency with low latency: some of them [13-15] do not provide strong consistency; others [17-19] provide the appropriate level of consistency, but either have high latency or require mobile apps to specifically call new APIs. OFS [9] and by extension MEFS, which leverages OFS in its design, provide the required level of strong consistency with relatively low overhead. Furthermore, apps are not required to use new API calls for reading or writing files (i.e., apps are unmodified). In addition, traditional cloud storage, network, and distributed file systems usually employ a file server: this adds extra overhead for sending any communication via the file server. Also, in case of network failures, these systems need to initiate an extra VM in the cloud for resuming the computation, which will result in additional latency for completing offloaded tasks. Fourth, traditional distributed file systems usually require client software to be installed and configured beforehand, which is

highly inconvenient for offloading tasks to the edge. Through its design, MEFS avoid all these problems.

Storage at the network edge. Few recent works focused on storing data at the edge of the network [21-23]. Similar to our work, several notable efforts have been proposed the usage of computational resources at the network edge for limited latency, as well as in order to exploit the usual high-bandwidth between mobile devices and edge nodes. For instance, Lujic et al. [21] present a three-layer architecture for efficient data storage management in edge analytics: their algorithm performs real-time forecast by saving parts of data at the edge. Furthermore, the work is targeted to time series coming from IoT sensors and therefore does not face out other general problems of mobile devices, such as consistency and mobility. EdgeCourier [22] provides a personal storage system at the edge for personal office documents. To overcome the limitations due to high network traffic on mobile devices, the authors proposed an incremental sync approach at the edge of the network. In addition, they have implemented Edge-hosted personal services (EPS). An EPS runs on a network edge node, and performs a specific functionality for mobile users. A mobile user can start/stop his own EPS instance(s) on the edge node. vStore [23] is a framework that chooses the best storage location based on user context. In vStore, several rules are defined to take storage decisions: for instance, the authors proposed to store data at the edge during large-scale events. Unlike MEFS, aforementioned works do not attempt to provide a general and flexible filesystem interface for apps that can run concurrently at the mobiles and the edge. Moreover, they do not resolve the typical challenges of MEC networks, such as application portability and resilience.

Summarizing the related work, although a few solutions have been proposed to contribute to the field of MEC by addressing challenges in storage and computation offloading, there is no ready-to-use solution to satisfy the requirements of supporting file system access for edge-assisted apps. As a consequence, we present MEFS to make computational offloading practical in the MEC environment. Compared to existing file system designs, MEFS takes into consideration the mobility of the users and the failures of MEC servers.

III. MEFS Requirements and Architecture

MEFS aims at supporting edge-assisted mobile apps for MEC networks. MEFS provides support for mobility management and fault-tolerance, while offering strong consistency and low latency for concurrent file accesses. This section describes the main requirements for MEFS and presents its architectural model.

A. MEFS Requirements

From the user perspective, a critical use case regarding MEC is computation offloading, as this can save energy and/or speed up the computation. One recognized concern of computation offloading is the proper management of the associated latency: for applications with stringent response time constraints (e.g., gaming, multimedia, augmented reality), the high latency between mobile devices and the cloud is not tolerable and could be a significant obstacle to the users' QoE. To better emphasize this concept, let us consider a typical real-time video analytics scenario: law-enforcement agencies may need to perform face recognition in real-time across large areas to identify potentially dangerous people. This scenario requires very low latency because the output of the analytics is used to interact with users (i.e., law-enforcement officers), requires high bandwidth for high-definition video streaming, and requires computation at the edge to enable low usage of the cloud. With this scenario in mind, we created MEFS to provide file system support required by edge-assisted mobile apps, which can offload components to edge nodes. In this way, offloading becomes practical for a larger class of applications by including apps that need access to files.

MEFS provides full infrastructure support for MEC environments and addresses three main technical requirements:

- **Strong consistency:** Platforms that offload resource-demanding tasks of mobile apps to the cloud or the edge [4, 5, 6, 24] lead to a scenario were computation tasks run concurrently on both the mobile and the cloud/edge. These tasks may need to access files on both these entities. However, the offloading platforms either do not support offloading of tasks with file I/O [4, 24] or allow access only to the files that are available locally [4, 10]. MEFS employs our OFS system [9], which is an application level file system that sits between mobile-cloud apps and the offloading middleware. OFS allows mobile-cloud apps to access files from both mobile and cloud concurrently, while providing strong consistency and low latency.
- **Application portability**: MEFS can portably transfer apps between MEC servers. It overcomes the application portability challenge by creating a set of APIs useful for developers to manage user mobility. Once the handoff is started, MEFS automatically communicates with its MEFS module at the new edge node and transparently moves the app. Handoff is the process of switching one connection end-point from one edge node to another in the midst of communication. More specifically, MEFS transfers the file system state and associated metadata, while the offloading middleware transfers the app state (i.e., app variables).
- **Resilience**: To protect against node or communication failures, MEFS leverages the cloud, as a controller entity, to provide fault-tolerance in two cases. First, if a MEC node fails, the cloud is in charge of restoring the affected app either in the cloud or at a new MEC node. Second, if the user moves away from the current MEC node and there is no other MEC node available in her proximity (single-hop coverage range), the app is again restored in the cloud. To this end, MEFS provides a transparent mechanism that synchronizes the file system state and associated metadata between edge nodes and the cloud.

B. MEFS Architecture

MEFS leverages OFS [9], our previous mobile-cloud file system, to manage remote file access and file sharing among the distributed components of edge-assisted mobile apps. Furthermore, it provides support for application portability and resilience.

Figure 1 depicts the general architecture of our solution, with MEFS and the offloading middleware deployed on mobiles, edges nodes, and the cloud; in this scenario, a mobile app can offload its computation to a nearby edge node. The cloud is used as a controller that helps with fault-tolerance, but is not generally involved in app computation. When the user

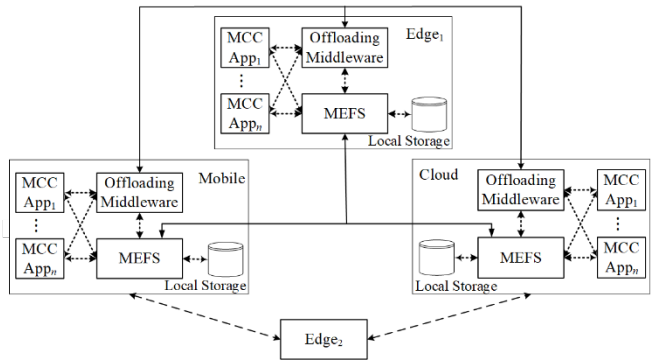


Fig. 1 Overall Architecture of MEFS for MEC Environment

moves from one edge node to another (e.g., from Edge1 to Edge2 in the figure), MEFS is able to seamlessly perform handoff in order to maintain communication locality and low latency. The figure also shows the interaction between MEFS and the employed offloading middleware, which is kept independent of the MEC-enabled file system design and implementation. In our prototype, we assumed the Avatar offloading middleware [25,26]. Unlike aforementioned proposals [11-13], in this work we rely on the MEC architecture to not only host our framework at the edge, but also to manage the mobility management and the fault-tolerance. In fact, MEFS contains support modules to overcome these MEC challenges.

Figure 2 details the MEFS architectural components. To provide *strong consistency*, MEFS uses OFS, which is designed and implemented as an event-driven middleware. The components of OFS are shown at the bottom of Figure 2. The events handled by OFS can be divided into two categories: Control events generated by OFS and the offloading middleware, and messages that represent file I/O operations generated by the apps. OFS has four major components: native/OFS switch, session management, buffer management, and consistency management. The *native/OFS switch* is included in the mobile-cloud app as a support library. It decides whether the file can be accessed locally or from OFS. The *session management* manages file states by maintaining file sessions. The *buffer manager* oversees the block buffer that contains file blocks that are currently being accessed through OFS. It also maintains metadata for each file block. Finally, the *consistency manager* maintains consistency between file I/O operations from tasks running on both mobile and cloud. It implements a *delayed-update* consistency algorithm which combines write-update and write-invalidate consistency policies [9].

To ensure that MEFS works on 5G MEC architecture, we need to provide support for mobility and for fault-tolerance. Thus, MEFS, in addition to OFS and the offloading middleware, includes two other main modules: the *mobility manager module* and the *fault-tolerance manager module*. These modules are shown in the middle part of Figure 2.

The *mobility manager module* guarantees the application portability required by the MEC standard. Specifically, in a MEC environment, the mobile users may change their location, which makes the system location dependent. That is why this module has to manage the handoff process. The module is composed of three components: monitoring, trigger, and management. The monitoring component monitors users' location in order to predict their movement. Several monitoring strategies have been proposed in the literature, and

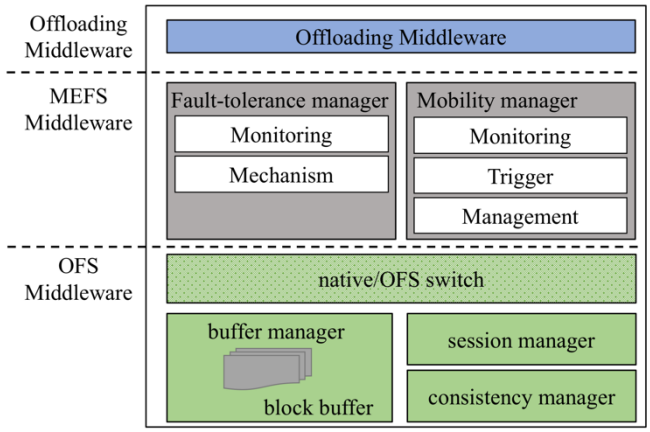


Fig. 2 MEFS Architectural Components

we designed the mobility manager module to work with any such strategy. The trigger component is in charge of determining the appropriate time to initiate the handoff. Particularly, this component collects information about users and calculates several metrics to start the handoff process. Management is the component that executes the handoff process between edge nodes. Moreover, this component defines the information flows between the system entities, which guarantee the correct and efficient execution of handoff among them.

The *fault-tolerance manager module* is in charge of managing the recovery from faults that may occur at the edge nodes. According to the MEC standard, the system resilience is a mandatory requirement. For this purpose, we have implemented this module with two main components: monitoring and mechanism. The monitoring component needs to figure out when faults happened. In the literature, several strategies have been proposed in order to overcome system failures. MEFS is designed to be agnostic to these strategies. The mechanism component defines the algorithm used for maintaining system consistency and for restoring the session during the recovery process.

IV. MEFS Implementation Highlights

We have implemented an MEFS prototype in Java on Android. However, it can be adapted to other mobile OSs. We integrated the MEFS stub in the existing native/OFS module switch using AspectJ [27]. MEFS uses an IPC service to communicate with other apps, a network service to communicate with the edge and the cloud, and runs the mobility manager module and fault-tolerance manager module as Android application services, which run perform long-running operations in the background. Lastly, we used Android's Binder IPC mechanism for IPC and a NIO-based TCP library named Kryonet [28], which provides high network throughput and low latency, for the network service.

A. Mobility Management

The MEFS mobility management is targeted to 5G networks and is fully compliant with the ETSI MEC technical requirements for managing end-to-end mobility aspects between edge nodes [29]. ETSI MEC has defined the requirements for mobility such as: continuity of service, mobility of application, and mobility of application-specific user-related information. Moreover, it specifies the standard end-to-end information flows between edge nodes that

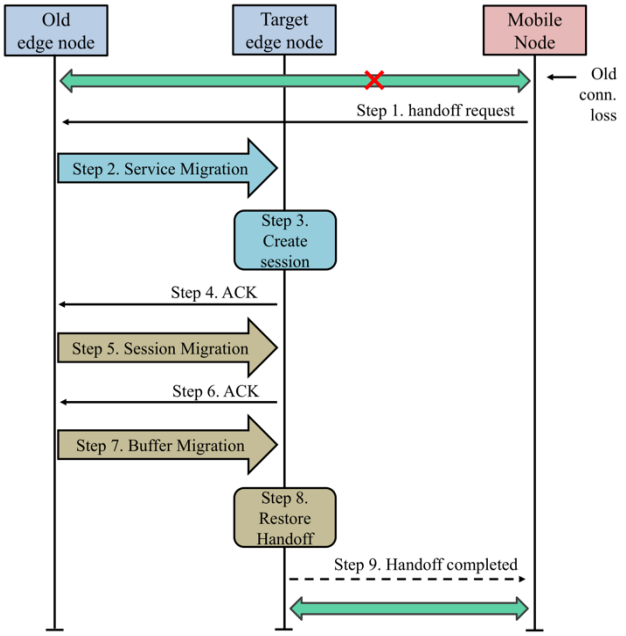


Fig. 3 Basic Handoff Protocol

systems have to manage. The flow is composed of five macro functionalities: (1) user bearer change detection, (2) service relocation management, (3) application instance relocation, (4) updating traffic rules, (5) terminating the source service. The MEFS handoff protocol was built on these specifications.

For stateful apps, such as the apps that use files, it is beneficial migrating user's data and state from edge_node1 to edge_node2 as a consequence of the associated handoff, in order to support the efficient continuation of the offloaded computation with reduced latency. For this purpose, MEFS defines and implements a reactive handoff protocol, as depicted in Figure 3. In general, the handoff can be triggered in two ways: one is infrastructural-dependent, where the system starts the handoff; the other is triggered by the mobile node. In previous works, we faced the problem of handoff triggering from infrastructure [30, 31]. In MEFS, we present a solution for triggering the handoff from the mobile node based on the user's mobility path (explained in the next paragraph). Thus, the mobile node starts the handoff process by sending a specific handoff message to the old edge node (step 1). After that, the old edge node sends the service to the target edge node (steps 2, 3). Let us note that we use the MEC term service to denote the offloaded app tasks that run at the MEC nodes.

Given that the overhead of step 2 might be high, to reduce the associated cost, we implemented a proactive migration strategy, which involves the cloud that proactively installs services on the target edge node. Figure 4 explains how our user mobility path strategy works. When possible, mobile nodes provide their expected route to the MEFS infrastructure at the starting of a new session. For example, if the mobile node has to go from A to B, the cloud knows that in the path there are edge_node2 and edge_node3 and it can proactively install the needed app components there. In principle, the target edge nodes can be reasonably well predicted based on the Received Signal Strength Indication (RSSI) or TCP throughput on the expected user's path. In the current implementation of MEFS, the prediction is based on user history of previously explored paths and on application-specific path constraints that may be defined at configuration time. We can also base the construction of the user's path on

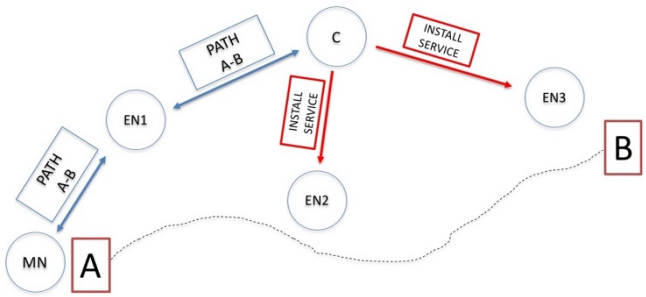


Fig. 4 User Mobility Path Strategy

our previous project on mobile crowd sensing [32]; the prediction logics can be easily improved and extended in the future without changing the remainder of MEFS and its APIs, which are prediction logic-independent. An incorrect prediction can result in extra-overhead: either the state is transferred to an edge node unnecessarily or the edge node does not benefit from proactive migration and needs to retrieve the state from the cloud on-demand, as described in Section IV.B for failures.

Steps 5-7 identify the core of the migration phase. In MEFS, this phase involves both user's data and state. In OFS, data about computation are contained in the **BufferManager**, while the state is stored in the **SessionManager**. To ensure that the target edge can restore the computation offloading process after handoff, MEFS moves the Buffer and Session objects from the old edge to the target edge. This is implemented via an abstraction, called **Handoff**, which contains the Buffer, the Session, and an associated manager class. The primary APIs of the manager class are:

TABLE I. LIST OF METHODS FOR SUPPORTING USER MOBILITY

METHOD	Description
prepareHandoff	This method is in charge of converting the handoff object to a Parcelable Android object. Parcelable is a class used in Android for more efficient object serialization.
sendHandoff	This method allows to send the handoff object from the old edge node to the target edge node. The object is sent via the OFS event support. In particular, we have defined a new event, called HANDOFF, that is useful for managing the entire handoff process.
restoreHandoff	This method runs at the target edge node and is in charge of receiving the handoff object and restoring the session and buffer. After that, the method sends a HELLO message to the mobile node in order to establish a new connection and sends the HANDOFF_FINISHED message to the old edge to inform it that the handoff is completed.

Thus, when a handoff request occurs, the old edge node invokes the *prepareHandoff* method to create the proper handoff object. To send it, the edge uses the *sendHandoff* method. Lastly, when the target edge receives the handoff object, it can invoke *restoreHandoff* to restore the session and to start a new connection with the mobile node.

Another important aspect of any MEC-related handoff process is which type of migration is performed. In the literature on VM/container migration, two different migration types are emerging as dominant: kill/restart and live migration. In particular for containers, one can simply kill the targeted container at the source edge and spawn a new one at the target edge. This implies that, in the case a container/VM

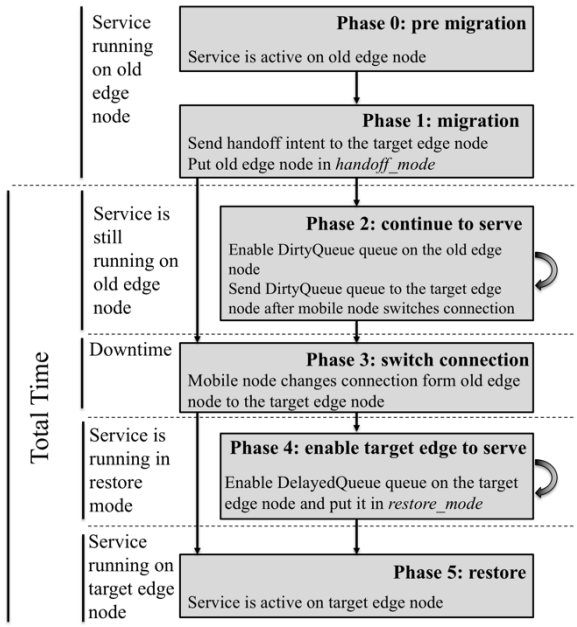


Fig. 5 Live Migration Overview

is running at source edge, during the handoff process there is a non-null slice of time when the associated service is unavailable for final end users (*downtime*).

The kill/restart type typically works well if the aim is to minimize the total handoff time [33]. In contrast, live migration has become recently the most common type for VM migration [34]. In live migration, the execution and memory states need to be copied, storage IO redirected, and network connections re-established. Among these, the memory copy usually takes the longest time.

MEFS implements its specific mechanism for live migration in the case of task offloading. Figure 5 shows an overview of the phases of our live migration implementation. When the handoff process is started, we put the MEFS middleware into a special state ("*handoff_mode*"); in this mode, the edge node stores each file access request sent by the mobile user in a specific queue called **DirtyQueue**, without performing the associated request. When the mobile node performs the connection handoff from the old edge to the target edge, the DirtyQueue is sent to the target edge. After that, the target edge can restore all the requests contained in DirtyQueue. In this way, we guarantee a short service downtime, practically almost the same with the short time interval when the mobile node is temporarily with no connection. This mechanism works coupled with a symmetric one running at the target edge.

After the mobile node has completed its handoff from the old edge node to the target edge node, it cannot perform any request at the target edge before the DirtyQueue is restored. MEFS has two types of requests: READ and WRITE. It is easy to understand that one cannot perform a consistent READ operation if there are WRITE operations in the DirtyQueue. To overcome this problem, we create a new special state for our offloading middleware called "*restore_mode*": when the old edge notifies the target edge of the intent to perform the handoff, the target edge enables the "*restore_mode*": once the mobile node connects to the target edge and the "*restore_mode*" is active, all requests performed by the mobile node are stored to a specific queue called **DelayedQueue**. The requests are still stored into the DelayedQueue until the DirtyQueue is restored. Finally, the target edge node can restore the DelayedQueue and can deactivate the "*restore_mode*".

Migration can also be done by saving the latest data and state from one edge node into the cloud and restoring them onto the other edge node. Such a saving and restoring mechanism has already been designed and implemented in MEFS for fault-tolerance (Sec. IV-B). MEFS chooses to not involve the cloud in migration management for three reasons: 1) the network latency is usually lower between edge nodes than that between an edge node and the cloud; 2) a decentralized design has better scalability; 3) the way that edge nodes work autonomously and separately from the cloud provides additional reliability.

B. Fault-tolerance

Failures may happen in a MEC environment due to various reasons. One of the common reasons is network coverage. For example, a mobile node is connected with an edge node in a certain location and then is moving to a location where there are no new edge nodes. Another common reason for failures is a crash of an edge node.

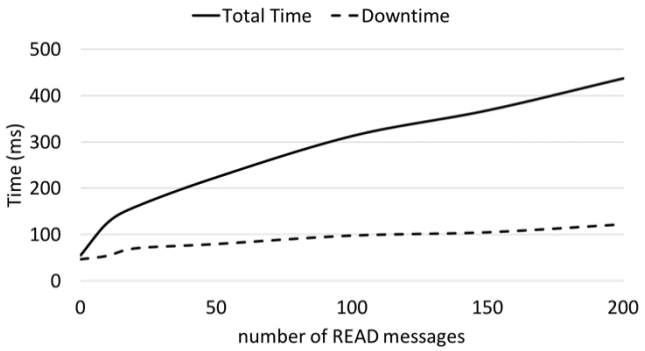
In the case of a failure, MEFS works to prevent the latest states of the files at the edge nodes from being lost or becoming inaccessible, such that MEC-enabled mobile apps can continue to run correctly on the smart phone alone or via offloading to another edge. As shown in Figure 1, MEFS exploits the cloud for this solution. At the beginning of the session, the mobile node connects both with the cloud and the edge and starts the session normally with the edge. The basic idea is to maintain data/state consistency between the edge and the cloud; this can be modeled as a traditional problem of coherency between storage at different network layers. Given that it is recognized that there is no best solution for every deployment environment and application domain to detect which data to move from the edge to the cloud and how frequently to do it, we have decided to implement a solution based on the Log-structured file system [35]. In particular, our MEFS support for fault tolerance sends to the cloud each WRITE operation performed by the edge; note that WRITE operations are smaller and faster than a backup of the whole data. In the case of a failure, the cloud will perform all the received WRITE operations in order to restore the same data/state conditions at the edge.

By delving into finer implementation details, we have implemented a **FaultToleranceManager** that offers several methods for handling failures, which may be invoked during the four phases of our fault tolerance protocol:

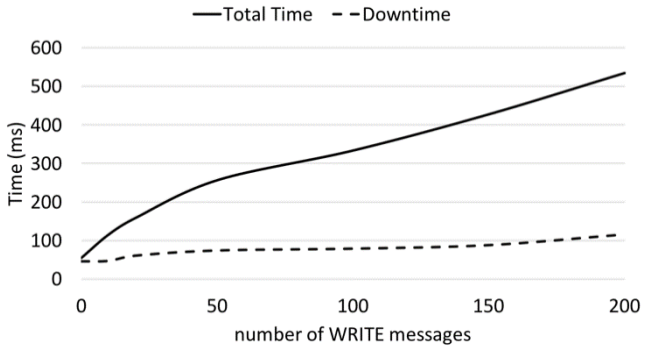
1) Send WRITE operation: This is a functionality in the FaultToleranceManager module that sends each WRITE operation from the edge to the cloud. We have also created a new event named FAULT_TOLERANT that contains the WRITE operation. Each WRITE operation sent to the cloud is stored in a queue named *operationQueue*.

2) New block: When the edge node creates a new file block (this may happen when the edge node tries to write more than 8KB, which is the standard block size) there is a functionality that sends that block to the cloud. A new event is created to support this: FAULT_TOLERANT_BLOCK.

3) PUSH message: Each time the edge node sends a PULL request (i.e., the edge node retrieves the latest blocks

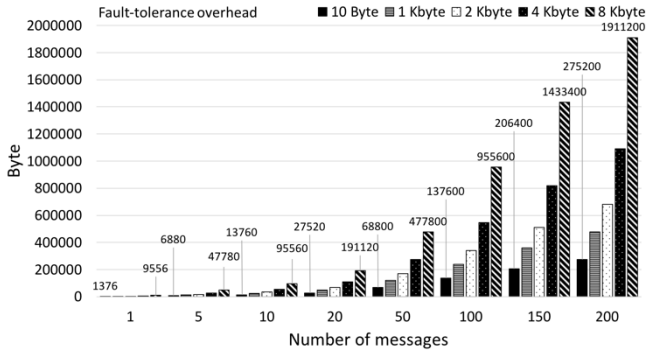


(a) The edge-assisted app performs READ operations

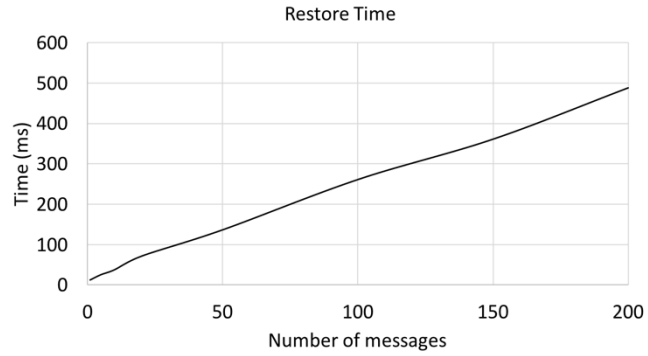


(b) The edge-assisted app performs WRITE operations

Fig. 6 Service downtime and total time of migration when the number of file operations performed by an edge-assisted app is varied from 0 to 200 operations/second.



(a) Overhead of fault tolerance mechanism



(b) Restore time at the cloud

Fig.7 Fault-tolerance performance evaluation. The overhead was calculated with several payload sizes. The restore time depends linearly on the number of messages to restore.

from the mobile node) to the mobile node, or the mobile node sends a PUSH operation (i.e., the mobile node sends the latest blocks to the edge node) to the edge node, we have to propagate these operations via an associated PUSH operation that sends all interested blocks to the cloud. Hence, the cloud must clear the operationQueue (associated with the PUSH_CLEAR event) because with this event it already has the latest version of the blocks.

4) Restore: To detect failures at edge nodes, we have implemented a simple mechanism based on ACK message exchange between the edge node and the cloud. If the cloud does not receive ACK messages from the edge after a configurable time threshold, the cloud is triggered to restore the session by performing all the operations in the operationQueue; after that, the cloud starts a new connection with the targeted mobile node.

V. MEFS Performance Evaluation

The goals of our experiments are three-fold: (1) evaluate the MEFS mobility management performance, with a focus on service downtime due to migration; (2) evaluate the MEFS fault tolerance performance, with a focus on overhead; and (3) compare the performance of MEFS on MEC vs. OFS on MCC.

For experiments, we built two mobile apps: an edge-assisted test app that replays the file access traces of real mobile users and a video analytics app, assisted by the edge or the cloud. We use the first app to evaluate the performance of mobility management and test the overhead incurred by the fault-tolerance mechanism in MEFS, and use the second app to compare MEFS with OFS.

The experiments use a prototype implementation of MEFS running on Android smart phones and Android x86 virtual machines (VMs). The phones act as mobile nodes, and the VMs act as edge nodes and cloud nodes. The VMs are hosted in a Linux OS. Each VM runs a 64-bit Android-x86 OS version 6.0, and has 2 virtual CPUs and 2 GB of RAM. The phones communicate with the edge nodes and cloud nodes using a secure WiFi network. The communication between edge nodes and cloud nodes is through wired connections.

A. Mobility Management Performance

To verify that the migration process of an app component from one edge node to another does not impose significant impact to the quality of user experience, we run several experiments, in which an edge-assisted app performs different file I/O operations when the mobile device switches the edge node it uses. We measured the service downtime and the total time used for migration in two scenarios: (1) the app performs READ operations on the mobile device at different rates: (2) the app performs WRITE operations on the mobile device at different rates. Selecting these two scenarios is to examine the impact of migration separately for READ and WRITE operations. The service downtime is the time period during which MEFS cannot respond to any file access requests from the edge component of the app. The total time used for migration is the time period between the creation of a handoff request and the time when the mobile node is connected to a new edge node and the latter finishes the restore phase.

The results in Figure 6 show that MEFS works well during migrations, and thus it is practical in real-life scenarios. Migrations impose minimal service downtime, which is usually lower than 150ms. For user QoE, 300ms is considered

as an acceptable delay [36, 37]. The service downtime of MEFS incurred by migration is lower than this value. The total time used for migration is larger than the service downtime. It is longer than 300ms when the app performs more than 100 READ or WRITE operations per second. However, this does not reduce QoE. Since migrations happen in the background and are transparent to apps, except for the service downtime, users may not experience any degraded service for most of the time.

We also notice that the total time to finish a migration is higher when the app performs WRITE operations than that with READ operations. This is because the total time used for migration is mainly determined by the amount of data that MEFS must copy from one node to the other. When the app performs WRITE operations, there will be more data to be copied to the destination edge node.

B. Fault-tolerance Performance of MEFS

MEFS sends WRITE operations from an edge node to the cloud to tolerate faults at the edge layer. The cost of this mechanism is determined by the amount of data to be transferred between the edge node and the cloud (i.e., the total number of messages and the data's payload). To evaluate the cost, we have measured the amount of data transfer in a few experiments.

Figure 7 shows the amount of data transfer when the number of messages is varied from 1 to 200, and the payload size is varied from 10B to 8KB. The highest overhead (2MB) is incurred when the number of messages is 200 and payload is 8KB. Since this overhead happens over the wired network, we consider it acceptable for fault-tolerance. The overhead is not proportional to the payload sizes. This is because the overhead is determined by message sizes, which are not proportional to the payload, as shown in Table II.

TABLE II. SIZE (IN BYTE) OF THE WRITE MESSAGES

PAYLOAD	*SIZE OF MESSAGE*
10	1376
1024	2388
2048	3412
4096	5460
8192	9556

We also measured the time to restore the state of MEFS in an edge node based on the data saved in the cloud, and show the results in Figure 7. The restore time is also determined by the amount of data to be copied between the cloud and the edge node (i.e., the total number of messages and the data's payload). The restore time increases with the number of messages, as shown in Figure 7 (the size of each message is 1376 Byte). When more than 100 messages are needed, the restore time increases to more than 300ms. Thus, we noted that to limit the restore time within 300ms, the amount of data transferred between the cloud and the edge node could be smaller than 200KB.

C. Comparison of MEFS on MEC vs. OFS on MCC

To evaluate the benefits of using MEC and MEFS over MCC and OFS, we have developed a video analytics application for face recognition purpose. According to Ananthanarayanan et al [38], large-scale video analytics may well represent the killer application for edge computing. The app captures real-time video streams received by a mobile user, analyzes the video streams for faces, recognize people from these faces, and displays the faces. The core component of this app for analyzing the streams, including face recognition with a machine-learning algorithm, runs at in the edge/cloud. The progressive knowledge is also kept in the edge/cloud. On the mobile side, the app component is mainly to send real-time video streams to the edge/cloud, and receive the photos generated by the analysis.

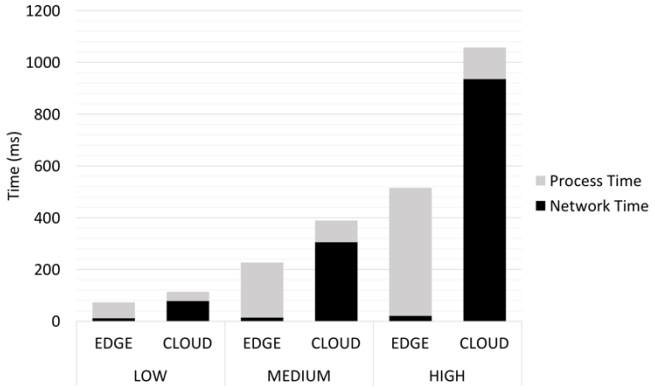


Fig. 8 Average response time of the video analytics app

In our tests, the cloud entity is a virtual machine hosted on Amazon Web Services Cloud equipped with 8 GB RAM and 4 virtual cores; the edge is a Linux box (Ubuntu 16.04 distribution) with 3.1 GHz Intel Core i5 and 4 GB RAM. The mobile node is connected to the edge node via WiFi, and the edge node is connected to the cloud via Ethernet. We have recorded continuous videos for 10 minutes with three different video qualities: a low quality video with 720x480 resolution, a medium quality video with 1280x720 resolution, and a high quality video with 1920x1080 resolution.

We have measured the response time of the app from the mobile component of the app streaming the video to its edge/cloud component until it receives the photos. The response times are shown in Figure 8. Compared to offloading to the cloud, offloading to the edge can significantly reduce the time spent on data transfers and thus reduce response time. For the videos with different qualities, when offloading tasks to the cloud, the response times are dominated by network communication. Due to the high network overhead, the powerful computation capabilities at the cloud cannot effectively reduce the response times. When offloading tasks to the edge, the communication bottleneck can be effectively mitigated. Though the edge node is not as powerful as the cloud node, and the edge node spends more time on computation than the cloud node, the overall response times are lower when offloading tasks to the cloud. The advantage of using edge is more pronounced for the video with the highest quality. The response time with the edge is 50% lower than that with the cloud.

These tests highlight the necessity of using MEC for data-intensive apps and justify the design of MEFS.

VI. CONCLUSION

This paper presented MEFS, the first mobile edge file system for edge-assisted mobile apps. MEFS provides strong consistency with low latency, and it overcomes MEC challenges such as mobility management and fault-tolerance. Furthermore, MEFS is completely transparent for edge-assisted mobile apps developers. We have implemented the

MEFS in Android, and we evaluated it under several experimental scenarios based on real apps and real mobile user traces. The experimental results demonstrated that MEFS can effectively support user's mobility and fault-tolerance at the edge nodes. Moreover, we proved that MEFS works with low latency at the edge nodes. Therefore, MEFS can be used for many types of context-aware mobile apps, including apps that have tight real-time constraints such as video streaming, augmented reality, and mobile gaming.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1409523, SHF 1617749, and DGE 1565478. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," Wireless communications and mobile computing, vol. 13, no. 18, pp. 1587–1611, 2013.

[2] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in OSDI'14, 2014, pp.97–112.

[3] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in MobileCloud '15, 2015.

[4] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in OSDI '12, 2012, pp. 93–106.

[5] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in MobiSys '10, 2010, pp. 49–62.

[6] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in EuroSys 2011, 2011, pp. 301–314.

[7] "Mobile-Edge Computing – Introductory Technical White Paper": https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf, [Online; accessed 23-Jan-2019].

[8] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS version 4 protocol," in SANE 2000, 2000.

[9] N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola and X. Ding, "Design and Implementation of an Overlay File System for Cloud-Assisted Mobile Apps," in IEEE Transactions on Cloud Computing, 2017.

[10] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in Infocom '12, 2012, pp. 945–953.

[11] C. You and K. Huang, "Multiuser Resource Allocation for Mobile-Edge Computation Offloading," 2016 IEEE Global Communications Conference (GLOBECOM), 2016, pp. 1-6.

[12] G. Orsini, D. Bade and W. Lamersdorf, "Computing at the Mobile Edge: Designing Elastic Android Applications for Computation Offloading," 2015 8th IFIP Wireless and Mobile Networking Conference (WMNC), Munich, 2015, pp. 112-119.

[13] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," ACM TOCS, vol. 10, no. 1, pp. 3–25, 1992.

[14] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, "Exploiting weak connectivity for mobile file access," SIGOPS Oper. Syst. Rev., vol. 29, no. 5, pp. 143–155, 1995.

[15] R. Tobbicke, "Distributed file systems: Focus on andrew file system/distributed file service (AFS/DFS)," in MSST'94, 1994, pp. 23–26.

[16] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, "RFS: A network file system for mobile devices and the cloud," SIGOPS Oper. Syst. Rev., vol. 45, no. 1, pp. 101–111, 2011.

[17] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, consistent, and efficient data sync for mobile apps," in FAST'15, 2015, pp. 359–372.

[18] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: Tunable end-to-end data consistency for mobile apps," in EuroSys '15, 2015, pp. 7:1–7:16.

[19] B. Atkin and K. P. Birman, "MFS: an adaptive distributed file system for mobile hosts," in Cornell University Technical Report, 2003.

[20] E. B. Nightingale and J. Flinn, "Energy-efficiency and storage flexibility in the blue file system," in OSDI '04, 2004, pp. 363–378.

[21] I. Lujic, V. D. Maio and I. Brandic, "Efficient Edge Storage Management Based on Near Real-Time Forecasts," 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC), Madrid, 2017, pp. 21-30.

[22] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "Edgecourier: an edge-hosted personal service for low-bandwidth document synchronization in mobile cloud storage services," ACM/IEEE Symp. Edge Computing (SEC '17).

[23] J. Gedeon, N. Himmelmann, P. Felka, F. Herrlich, M. Stein, M. Mühlhäuser, "vStore: A Context-Aware Framework for Mobile Micro-Storage at the Edge," Mobile Computing, Applications, and Services. MobiCASE 2018.

[24] A. Zanni et al., "Automated Offloading of Android Applications for Computation/Energy-usage Optimizations," in Infocom Demo Papers, 2017.

[25] M. A. Khan et al., "Moitree: A Middleware for Cloud-Assisted Mobile Distributed Apps," 2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), Oxford, 2016, pp. 21-30.

[26] H. Debnath et al., "Collaborative Offloading for Distributed Mobile-Cloud Apps," 2018 6th IEEE Int. Conf. Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2018, pp. 87-94.

[27] "Aspectj," https://www.eclipse.org/aspectj/, [Online; accessed 23-Jan-2019].

[28] "Kryonet," https://github.com/EsotericSoftware/kryonet, [Online; accessed 23-Jan-2019].

[29] "ETSI GR MEC 018, " https://www.etsi.org/deliver/etsi_gr/MEC/001_099/018/01.01.01_60/gr_MEC018v010101p.pdf, [Online; accessed 23-Jan-2019].

[30] P. Bellavista, A. Corradi and C. Giannelli, "A Unifying Perspective on Context-Aware Evaluation and Management of Heterogeneous Wireless Connectivity," in *IEEE Communications Surveys & Tutorials*, vol. 13, no. 3, pp. 337-357.

[31] P. Bellavista, A. Corradi and C. Giannelli, "Differentiated Management Strategies for Multi-Hop Multi-Path Heterogeneous Connectivity in Mobile Environments," in *IEEE T. Network and Service Management*, vol. 8, no. 3, pp. 190-204, September 2011.

[32] G. Cardone, A. Corradi, L. Foschini and R. Ianniello, "ParticipAct: A Large-Scale Crowdsensing Platform," in *IEEE T. Emerging Topics in Computing*, vol. 4, no. 1, pp. 21-32, Jan.-March 2016.

[33] Y. C. Tay, K. Gaurav and P. Karkun, "A Performance Comparison of Containers and Virtual Machines in Workload Migration Context," 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), Atlanta, GA, 2017, pp. 61-66.

[34] C. Clark , K. Fraser , S. Hand , J. G. Hansen , E. Jul , C. Limpach , I. Pratt , A. Warfield, "Live migration of virtual machines," Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, p.273-286, May 02-04, 2005.

[35] M. Rosenblum and J. K. Ousterhout. 1992. "The design and implementation of a log-structured file system," ACM Trans. Comput. Syst. 10, 1 (February 1992), 26-52.

[36] "Quality of Service Design Overview," http://www.ciscopress.com/articles/article.asp?p=357102, [Online; accessed 27-Feb-2019].

[37] S. J. Thorpe and M. Fabre-Thorpe, "Seeking Categories in the Brain," American Association for the Advancement of Science, vol. 291, no. 5502, pp. 260-263, January 2001.

[38] G. Ananthanarayanan *et al*., "Real-Time Video Analytics: The Killer App for Edge Computing," in *Computer*, vol. 50, no. 10, pp. 58-67, 2017.