Simulation Modelling Practice and Theory 101 (2020) 102062

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

MobFogSim: Simulation of mobility and migration for fog computing

Carlo Puliafito[*,1,2,a], Diogo M. Gonçalves[1,b], Márcio M. Lopes[b], Leonardo L. Martins[b], Edmundo Madeira[b], Enzo Mingozzi[a], Omer Rana[c], Luiz F. Bittencourt[b]

[a] Department of Information Engineering, University of Pisa, Italy
[b] Institute of Computing, University of Campinas, Brazil
[c] School of Computer Science, Cardiff University, UK

ARTICLE INFO

Keywords:
Fog computing
Edge computing
Simulator
iFogSim
Mobility
Container migration

ABSTRACT

Fog computing is an extension of the cloud towards the network edge that brings resources and services of computing in closer proximity to end users. This proximity provides several benefits such as reduced latency that improves user experience. However, user mobility may limit such benefits in practice, as the distance to a fog service may vary as a user moves from one location to another. Migration of a fog service may be one possible mitigation strategy, enabling the service to always be *close enough* to a user. Although many simulators exist for evaluating application behaviour and performance within a fog computing environment, none allows evaluation of service migration solutions to support mobility. MobFogSim is presented in this work to overcome this limitation. It extends iFogSim to enable modelling of device mobility and service migration in fog computing. MobFogSim is validated by comparing simulation results with those obtained from a real testbed where fog services are implemented as containers. Additional experiments are carried out in MobFogSim taking account of various mobility patterns of a user, derived from Luxembourg SUMO Traffic (LuST). We use an experiment-based approach to study the impact of user mobility on container migration in fog computing.

1. Introduction

The Internet has become the largest and most popular mechanism for communication among people, corporations, and governments around the world. Furthermore, a number of devices are connected to the Internet, consuming and generating data as well as offering a variety of computing services. Such devices can be fixed or mobile, for example carried by their users or attached to a vehicle. Nowadays, many applications and services running on fixed or mobile devices rely on remote services, as in cloud computing servers [1,2], to store data and/or perform data processing. One of the limitations of using remote devices for storage and computing is the potentially large latency experienced by users. In general, data centres can be far away from the end device running the

* Corresponding author.
E-mail addresses: carlo.puliafito@ing.unipi.it (C. Puliafito), diogomg@lrc.ic.unicamp.br (D.M. Gonçalves), marcio@lrc.ic.unicamp.br (M.M. Lopes), L220120@dac.unicamp.br (L.L. Martins), edmundo@ic.unicamp.br (E. Madeira), enzo.mingozzi@unipi.it (E. Mingozzi), ranaof@cardiff.ac.uk (O. Rana), bit@ic.unicamp.br (L.F. Bittencourt).

[1] Carlo Puliafito and Diogo Gonçalves share the first author role in this paper.

[2] Part of this research was conducted while Carlo Puliafito was also affiliated to the University of Florence (DINFO), Italy, and visiting student at Cardiff University, UK.

https://doi.org/10.1016/j.simpat.2019.102062
Received 19 July 2019; Received in revised form 9 December 2019; Accepted 19 December 2019
Available online 20 December 2019
1569-190X/

application, causing increased delays to access data and higher turnaround times for processing.

Fog computing [3] was proposed to overcome these limitations. Fog computing extends the cloud towards the network edge, distributing resources and services of computing as close as the user's access point, thus only one hop away from end devices. It is worth highlighting that fog computing does not replace the cloud but rather complements it. Fog computing therefore does not only reduce network delays but also: (i) lowers bandwidth consumption; (ii) improves security and privacy; (iii) provides better context awareness; and (iv) enables uninterrupted services in case of intermittent connectivity to the cloud [4]. Devices hosting fog computing services are known as fog nodes, cloudlets, or micro data-centres (MDC).

User mobility may impair fog computing performance. This is because mobility causes a change of the access points, which may increase the delay to the fog service hosted in the original cloudlet/MDC [5]. When a mobile user changes access point, ideally their data and current applications being processed should migrate to the cloudlet at the new access point to minimise access delay. To accomplish this, we assume that the user has a virtual machine (VM) or a container, similarly to cloud computing services, that contains her/his processes and data. Understanding where each application (or its components) should run and where data should be kept involves a multi-layered infrastructure with heterogeneous devices and networks as well as applications that have heterogeneous requirements and can move along the infrastructure. Simulation can be a time- and cost-effective way to evaluate VM/container migration solutions in fog computing environments with mobile users.

In a previous work [6], we introduced mobility concepts to iFogSim in a preliminary support for mobility. In this paper, we present *MobFogSim*, an open-source[3] simulator that extends the preliminary work from [6] to model more generalised aspects related to device mobility and VM/container migration in the fog, e.g., user position and speed, connection handoff, migration policies and strategies, to name a few. The contribution of this paper is threefold:

- We discuss the design considerations and the implementation details of MobFogSim, highlighting aspects required to model device mobility and VM/container migration in fog computing;
- We validate MobFogSim by comparing its container migration results with those obtained from a real testbed;
- We study the impact of user mobility on container migration in fog computing. This is achieved by running simulations in MobFogSim where users move with different mobility patterns taken from Luxembourg SUMO Traffic (LuST). The approach can be generalised to other (similar) mobility patterns.

The rest of the paper is organised as follows. Section 2 provides a general background on VM/container migration in fog computing and describes the events that occur during connection handoff and VM/container migration. Section 3 briefly outlines the features available in iFogSim and describes the design considerations for VM/container migration modelling in MobFogSim. In Section 4 we provide implementation details of MobFogSim, comparing it with iFogSim. Section 5 describes the experiments that we carried out over a real testbed to obtain seed values for supporting simulation in MobFogSim. Results of container migration over the testbed are also described in this section. In Section 6 we validate MobFogSim by comparing its results against those from the testbed. Section 7 reviews existing fog computing simulators, describing difference with MobFogSim. Finally, Section 8 provides the key conclusions we can draw from the simulations.

2. Basic concepts

In this section, we briefly introduce fog computing, virtualisation and migration concepts as well as the migration model considered in the proposed simulator.

2.1. Fog computing

Fog infrastructures can consist of one or more layers, hierarchically organised, located between the (potentially mobile) users and the cloud. The number and composition of such layers depends on the specific application domain and its requirements [4]. Typically, the topmost layer is represented by the cloud, while the lowest fog layer is expected to be close (geographically) to the user, e.g., hosted at the first hop access point. This infrastructure is illustrated in Fig. 1.

Fog-enabled applications can use cloudlets to store and process data. Ideally, this should occur in the closest cloudlet (e.g., the one at the user's access point). When considering a mobile user, to keep this ideal configuration, data and processing pertaining to that user should move along with her/him to maintain low latency. How this data/processing migration is implemented depends on the nature of the underlying software. In this paper, we consider that fog computing services are available in the form of a virtualised software environment, such as VMs or containers, that encapsulate access to computing and network resources.

To support VM/container migration in fog computing environments, a migration architecture involves multiple mechanisms while a VM/container is moving from a source cloudlet to a destination cloudlet. Differently from the centralised cloud computing paradigm, in this paper we consider fog nodes as the decision makers. Moreover, in a real environment, the start, duration, and end of the migration process will depend on circumstances that involve the users' mobility characteristics, as well as the handoff process of the access point to which the mobile device is connected. Thus, a combination of factors should be considered during user mobility, such as the VM/container migration between the source and the destination cloudlets and the handoff process of mobile device

[3] See https://github.com/diogomg/MobFogSim. Last accessed: 15 October 2019.

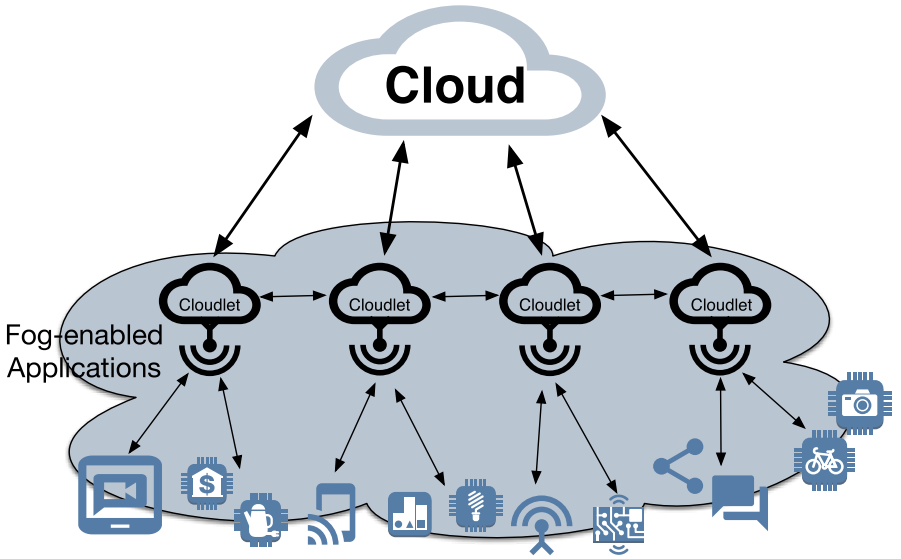


Fig. 1. Fog and its cloudlets/MDCs are between the users and the cloud.

connections. To describe the aforementioned circumstances, we discuss some scenarios that may occur during a migration process [6].

2.2. Migration model

Three relevant aspects that should be considered during the migration and handoff processes are:

1. Cloudlets and their geographical location.
2. Network connections (i) between the user and the access point; (ii) between the access point and the cloudlet running the user's VM/container; and (iii) between the source and the destination cloudlets.
3. Mobility timeline: a user's direction and speed.

Although all these aspects are interdependent, each one of the components of the scenario can be evaluated on its own. Thus, handoff- and migration-related events can happen at the same time or in different orders. We identify eight such events, which are here numbered in order to logically separate them and emphasise that each of them may occur independently or in a different order. The following is a high-level description of these events; in Section 4.2, we describe the sequence of events occurring in a migration and handoff scenario, as per the implementation in MobFogSim.

Event 1 - E1 occurs when the fog system decides that it is time to perform a migration (i.e., When-to-Migrate decision [7]). Many different approaches exist to support this decision, depending on the strategies being adopted. For instance, VM/container migration may be triggered before the connection handoff starts (i.e., before E4), based on known information on a user's current mobility (e.g., position, direction, speed) and/or based on known mobility patterns for categories of users (e.g., public transport buses) [8]. This is called *proactive migration*. Alternatively, migration can be triggered once connection handoff has already started, and the new access point of the user is known in a deterministic way. This is instead called *reactive migration*. Details on the When-to-Migrate approach that is currently implemented in MobFogSim can be found in Section 3.2.2. However, we highlight that developers can implement their own migration decisions in the simulator.

Event 2 - E2 is the set of necessary procedures associated with preparing the VM/container for migration (e.g., checking its size and page dirtying rate, checkpointing its state, monitoring network speed to estimate the migration time, etc.) and establishing a network connection between the source and the destination cloudlet for migration. Preparation for data migration depends on the actual migration technique being used. Please, refer to the end of Section 3.2.2 (i.e., Before Migration) for details on the migration techniques that are currently implemented in the simulator.

Event 3 - E3 refers to the beginning of the process of sending data from the source to the destination cloudlet, i.e., the VM/container migration process actually starts here.

Event 4 - E4 is when the connection handoff starts. Handoff decision-making is defined by parameters of the network and/or the device being utilised. In our ideal case, this event occurs after the migration starts (i.e., after E3), so that it can be contained within the VM/container migration phase. The duration of the connection handoff is very likely to be lower than that of a VM/container migration, indeed. However, it is worth noting that some migration policies may consider also a reactive migration, i.e., one where VM/container migration starts after the beginning of connection handoff.

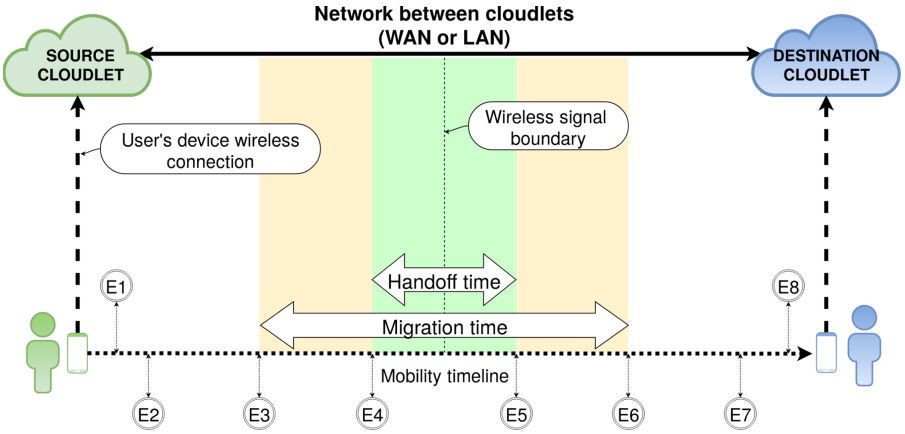


Fig. 2. Proactive migration scenario: migration starts before the handoff.

Event 5 - E5 is the end of the connection handoff process. Now, the user's device is connected to the new access point. Meanwhile, VM/container migration is still occurring.

Event 6 - E6 is the end of the VM/container migration process. The data transfer is completed and the user's VM/container is now running on the destination cloudlet.

Event 7 - E7 represents the first access from the user to the VM/container running on the new cloudlet, which means that at this point networking connections must be re-established to reflect the new location.

Event 8 - E8 represents the point in time when the whole process is complete. We highlight that E7 and E8 coincide if VM/container migration finishes after the connection handover (i.e., E6 occurs after E5). However, if VM/container migration finishes before the connection handover (i.e., E6 occurs before E5), E7 takes place before E8.

Based on the above events, we discuss different scenarios that can take place when users move, illustrating: (i) the source cloudlet, which is hosting and running the user's VM/container before migration; (ii) a wireless link that connects the user's device to the access point where the source cloudlet is connected; and (iii) the destination cloudlet connected to the destination access point (i.e., that after the wireless handoff). The two cloudlets are connected through a cabled network (WAN or LAN), which is used to transfer the VM/container. The connection between cloudlets can present different topologies depending on how they are deployed.

Scenario 1 - Proactive migration Fig. 2 considers that the migration process starts (i.e., E1) before the user reaches the point of performing the wireless connection handoff. This way, when E1 occurs, the VM/container is migrated without an abrupt connection interruption from the wireless handoff. Ideally, the migration and the handoff should end at the same time to reduce the delay to the user: note that the longer the migration takes to finish after the handoff, the longer the user will have to access the source cloudlet through the new access point (which means more than one network hop). On the one hand, proactive migration may improve overall performance, since the user's VM/container is ideally available at the destination cloudlet once the wireless connection handoff is finished. On the other hand, though, it might worsen performance if user's mobility prediction is erroneous.

Scenario 2 - Reactive migration As the migration and handoff decision-making are assumed to be independent from one another, these can occur at any time along the user's path. Therefore, the events timeline can change, for example, with the migration process starting (i.e., E3) after the handoff process has begun (i.e., E4). Fig. 3 shows the case in which the whole migration process takes place once the handoff process is finished. However, the two processes can partially overlap; this depends on the actual policies and algorithms adopted for migration. Reactive migration may lead to increased delays and potential drops in the quality of experience (QoE) observed by the user. This is because, when the handoff process is finished, the user is still accessing the source cloudlet resources through the new access point: the application goes through more than one hop to access the source cloudlet. VM/container migration towards the destination cloudlet terminates after a while. Nevertheless, reactive migration may prove beneficial in situations where user's mobility pattern is highly unpredictable, and it is therefore better to know the user's new position prior to migrating the VM/container.

Scenario 3 - Concurrent migration A third possible scenario is such that both handoff and VM/container migration start and end simultaneously (Fig. 4). Although this is feasible, it is unlikely, since VM/container migration takes longer than a handoff process. We consider this scenario as a complement to Scenario 1: the shorter the time difference between migration and handoff, the better, as the application will always have access to the VM/container on the closest cloudlet during the migration process. In this scenario, differently from Scenarios 1 and 2, the VM/container does not need to be accessed through another access point. In this case, the downtimes of the handoff and of VM/container migration coincide, minimising the delays experienced by the user.

3. Design considerations

MobFogSim is a simulator that extends iFogSim to model aspects related to device mobility and VM/container migration in fog computing. We had already incorporated new features into iFogSim in a previous work [6]. In this paper, we extend that preliminary version to cover a wider set of parameters as well as to support mobility by integrating our simulator with the mobility tool *Simulation*

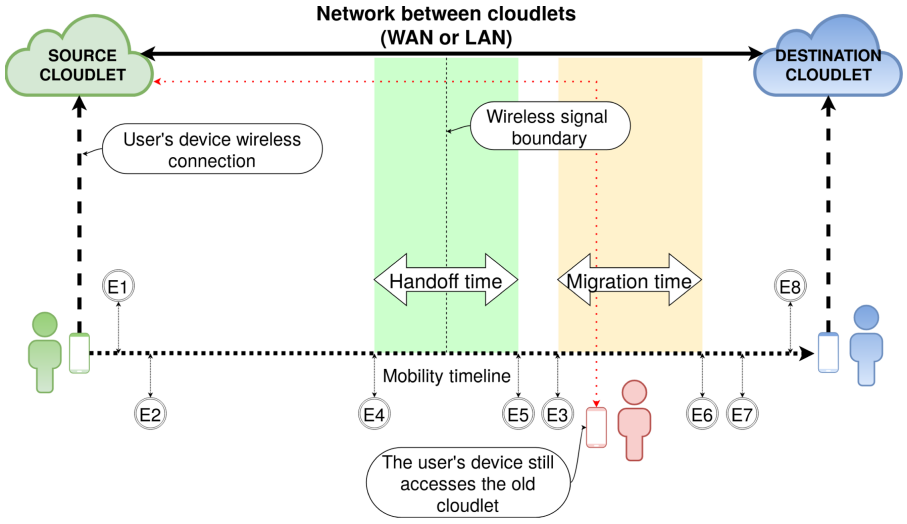


Fig. 3. Reactive migration scenario: migration starts after the handoff.

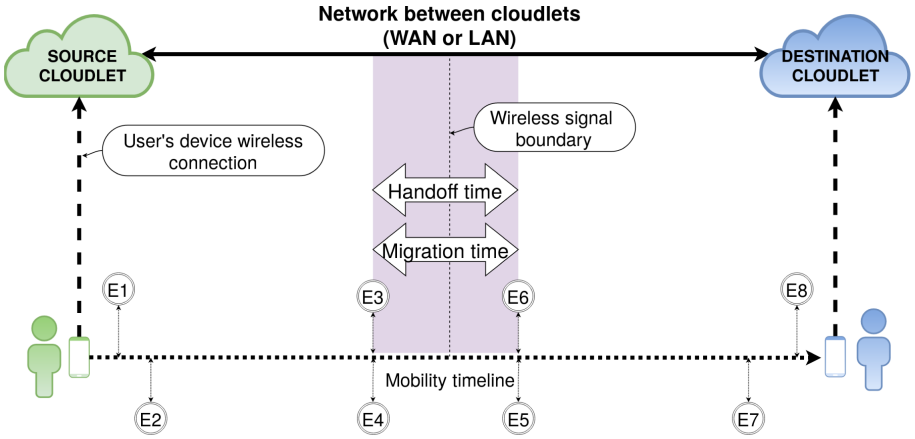


Fig. 4. Concurrent migration scenario: handoff and migration start and end simultaneously.

of Urban Mobility (SUMO) [9]. These modifications, which allow more generalised mobility simulations in fog computing, are described later in this paper. In what follows, we discuss the design considerations for VM/container migration modelling in MobFogSim (see Section 3.2) after a brief overview of iFogSim in Section 3.1.

3.1. iFogSim overview

iFogSim [10] is an extension of CloudSim [11] that implements simulation support for fog computing. It supports the configuration of a fog/cloud hierarchy by defining the connection between edge devices, fog devices (cloudlets or fog gateways), and a cloud data centre. The main classes of iFogSim, which are implemented in Java, are briefly described below in order to introduce the simulator and present the necessary background to describe the modifications of MobFogSim in the next section. iFogSim has kept the CloudSim core implementation to realise the processing of events among fog components. Besides, iFogSim has created new classes and methods to run a fog simulation. Its main components are:

- *FogDevice:* It presents the hardware features of a fog or IoT device. It extends *PowerDatacenter* from CloudSim. RAM, MIPS, storage size, and bandwidth (uplink and downlink) are the main class attributes. The methods of this class perform specific tasks of a fog or IoT device to process the received tuples.
- *Tuple:* Extends *Cloudlet*[4] of CloudSim. This represents a tuple (or task) created by an IoT device that is sent to an AppModule to be processed.
- *Application:* It is designed according to a Directed Acyclic Graph (DAG). The vertices are the arrival execution modules from a

[4] In CloudSim, a cloudlet is an application running in the cloud.

```
1:  while User u is within the migration zone do
2:    if MigrationDecision() is TRUE then
3:      if User u is at the migration point established by the Migration Policy then
4:        Before migration: prepare_migration()
5:        while Migration is being performed do
6:          if Cannot migrate then
7:            Fail and leave VM/container at the source cloudlet
8:          end if
9:        end while
10:       After migration: reconfigure_network()
11:     end if
12:   else
13:     Leave VM/container at the source cloudlet
14:   end if
15: end while
```

Algorithm 1. Migration overview.

tuple. The edges are data dependencies between the modules (vertices). The following classes are used to instantiate applications:

- *AppModule:* Vertex that processes tuples. This class extends *PowerVm* from CloudSim.
- *AppEdge:* Edge to link a pair of vertices (AppModules), thus to create a dependency between these entities.
- *AppLoop:* It is the DAG flow from the initial vertex (entry node) to the last vertex (exit node).
- *Sensor:* Objects of this class are responsible for creating the application tuples.
- *Actuator:* Objects of this class receive the tuples processed by an AppModule at the end of the AppLoop.

iFogSim already implements important components for simulating fog computing. However, this simulator does not model: (i) mobile devices; (ii) geographical position; (iii) wireless base stations; (iv) VM/container migration; (v) evaluation of VM/container migration and related policies. Motivated by these lacks, in this paper we propose MobFogSim, which builds upon iFogSim to support device mobility and VM/container migration in fog simulations.

3.2. Migration overview

An architecture for VM/container migration in the fog has been presented in [12] and is composed of three main layers: (i) cloud computing, (ii) fog computing (with a set of cloudlets), and (iii) the fog-enabled/IoT devices layer. Based on that, we devised the necessary steps for performing VM/container migration between two cloudlets in this fog architecture, as presented in Algorithm 1.

When the user gets close to crossing the wireless network boundary and, consequently, is close to a possible handoff process from the current access point to the next one, the fog infrastructure should start the migration process. The *Migration Decision* process (line 2 in Algorithm 1) is the one that makes the migration decision based on policies and strategies that are established by the migration system. Migrations occur when the system identifies a better cloudlet to place the user's VM/container (it is based on some metric like lowest latency, lowest distance, or another metric, as we discuss further in the paper). In scenarios where there are no cloudlets available offering better conditions to the user, the current cloudlet remains the only alternative to run the userâs VM/container (line 13 in Algorithm 1). If the policy/strategy judges that it is necessary to perform a migration process, the migration point, which defines how close the user is to the wireless network boundary (explained in details in Section 3.2.1), defines when the migration starts. When the user crosses the migration point, the system is allowed to start the migration. Once these conditions (where and when) are satisfied, the system must prepare the VM/container data for migration. This process happens in the *Before Migration* phase. When the Before Migration phase finishes, the migration system has the necessary information to start the migration. In the *During Migration* phase, the system must monitor, manage, and synchronise the process, depending on the type of migration being used (e.g. cold/non-live or live). If anything impairs VM/container migration, either the management system tries to solve the problem at runtime, or the migration might be aborted, and the migration process finishes leaving the VM/container on the source cloudlet. On the other hand, if everything runs as expected, the migration process goes to the *After Migration* phase to liaise with the user's mobile device to close the connection with the old cloudlet and use the new cloudlet.

The migration algorithm, and especially the migration decision-making, needs input from a set of parameters to implement a migration policy and a migration strategy, as detailed below.

1. Migration policy: A migration policy can consider the geographical context of a user and her/his device in order to trigger a migration:
   - User's position on the map
   - User's speed and direction
   - Migration zone: area where the migration decision is computed.

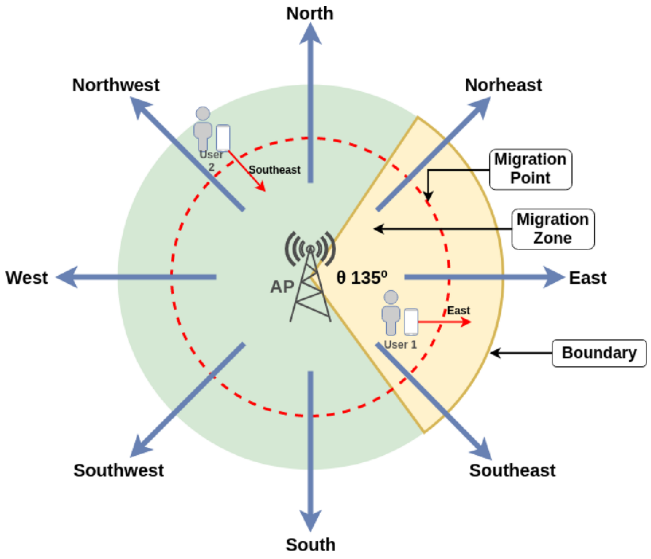


Fig. 5. Migration model and parameters.

- Migration point: a location in the map where the computed migration can be performed.

2. Migration strategy: A migration strategy is a placement algorithm that decides which is the next cloudlet to receive a VM/container that is to be migrated. This can be modelled in several ways, using different optimisation techniques and input data from the system and from the users/applications. Currently, MobFogSim implements three different migration strategies based on network condition and user location-aware metrics. These migration strategies are provided as examples since developers can implement their own algorithms, which can take into consideration specific, and more or less flexible, Quality of Service (QoS) levels.

3.2.1. *Migration policy*

The *migration policy* defines the migration zone and a migration point in the map. The current model implemented in MobFogSim is based on the migration policy discussed in [6]. An illustration of that model can be seen in Fig. 5, which shows some of the parameters to be monitored.

Users 1 and 2 have different speeds, directions, and geographical positions in the map. These attributes are verified during the MigrationDecision process. The system monitors the users and decides whether their VMs/containers should be migrated or not.

The *migration zone* is an area where migration decisions are constantly computed. Looking at Fig. 5, this is the area limited within the migration point (dashed red line). Once a MigrationDecision returns TRUE, it is actually carried out only when the user reaches the migration point, which is any point along the dashed red line in figure.

The destination cloudlets considered in a migration decision are limited by the migration cone (light yellow area in Fig. 5). This cone is extended to the next access points in the map, thus limiting the amount of destination cloudlets for the user's VM/container and also contributing to a speedup in the underlying optimisation process performed by the migration strategy. This cone is defined by:

1. The two directions adjacent to the current direction of the user (e.g., the direction of user 1 is East, then the adjacent edges are Northeast and Southeast).
2. An angle θ that defines the relative region between the access point and the user (e.g., 135˚ in Fig. 5).

This cone is always constructed on the same side and direction of the moving user. Note that user 2 does not have a cone because she/he is already connected to the AP and is moving towards this AP, i.e., her/his direction is Southeast, but her/his position relative to the AP is Northwest. On the other hand, the cone for user 1 is shown because her/his direction is East, and her/his position relative to the AP is also East.

The *Migration Point* is a point on the map where VM/container migration should be started before the handoff mechanism occurs. The migration point can be set depending on characteristics of the infrastructure and the wireless connection, e.g., taking into account how wireless handoff policies behave. A migration point can be either static or dynamic. A static migration point is fixed on the map regardless of other parameters. For example, Fig. 6 shows an example of a static migration point that is defined as when the user has already travelled 70% of the radius of the coverage area, thus still remaining 30% of the distance to travel before the expected handoff occurrence point (boundary). Instead, a dynamic migration point can take into consideration other parameters, as for example the size of the data being migrated as well as the user's speed. Fig. 7 shows a scenario with two examples that combine data size and user's speed. User 1 has a larger volume of data to migrate and has a higher speed; user 2 has a smaller volume of data and a slower speed.

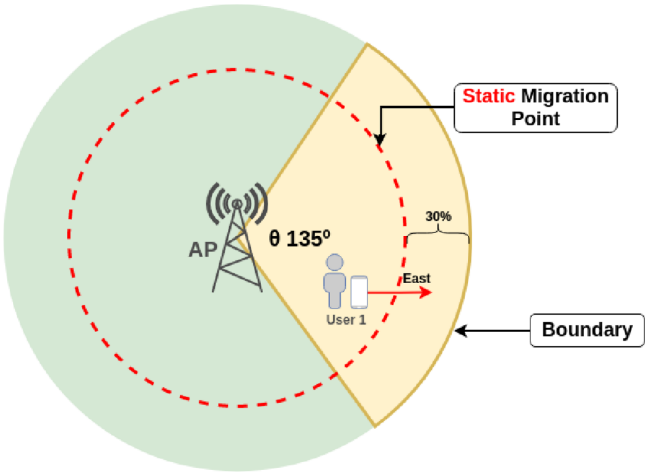


Fig. 6. Static Migration point.

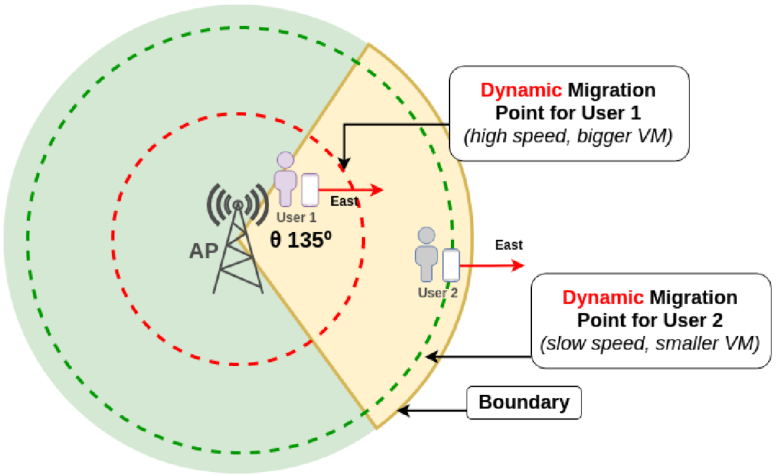


Fig. 7. Dynamic Migration point.

3.2.2. Migration strategy

This section describes the migration strategy that is designed to be applied to Algorithm 1.

Migration Decision

In the first part of Algorithm 1, *MigrationDecision* decides whether the user's VM/container should be migrated or not. The MigrationDecision flow is illustrated in Fig. 8. First, the algorithm verifies if the user is moving, and if so, the migration decision process will start by discovering the relative user location in relation to the access point, aiming at verifying if the user is in the migration zone. If the user is in the migration zone, the algorithm chooses a new cloudlet to receive the user's VM/container. This choice depends on the *Migration Strategy* (described next in this paper). After the algorithm chooses a new cloudlet, it checks if the chosen cloudlet is available. If the cloudlet is available, the migration is scheduled to start when the user reaches the migration point. If there are no cloudlets available and the user performs a handoff, the VM/container will remain in the same cloudlet, and a new migration decision can take place when the user is within the migration zone in the new access point. In the simulator, the migration decision can also consider other characteristics (e.g., type of service offered by this cloudlet - public or private - and if SLAs exist or not).

Three different migration strategies are considered when choosing the next cloudlet for a VM/container:

1. *The lowest distance between the user and the access point* - This strategy chooses the cloudlet connected to the access point that is closest to the user.
2. *The lowest distance between the user and the new cloudlet* - This strategy chooses the cloudlet that is closest to the user.
3. *The lowest latency* - This strategy chooses the cloudlet with the lowest latency to the user.

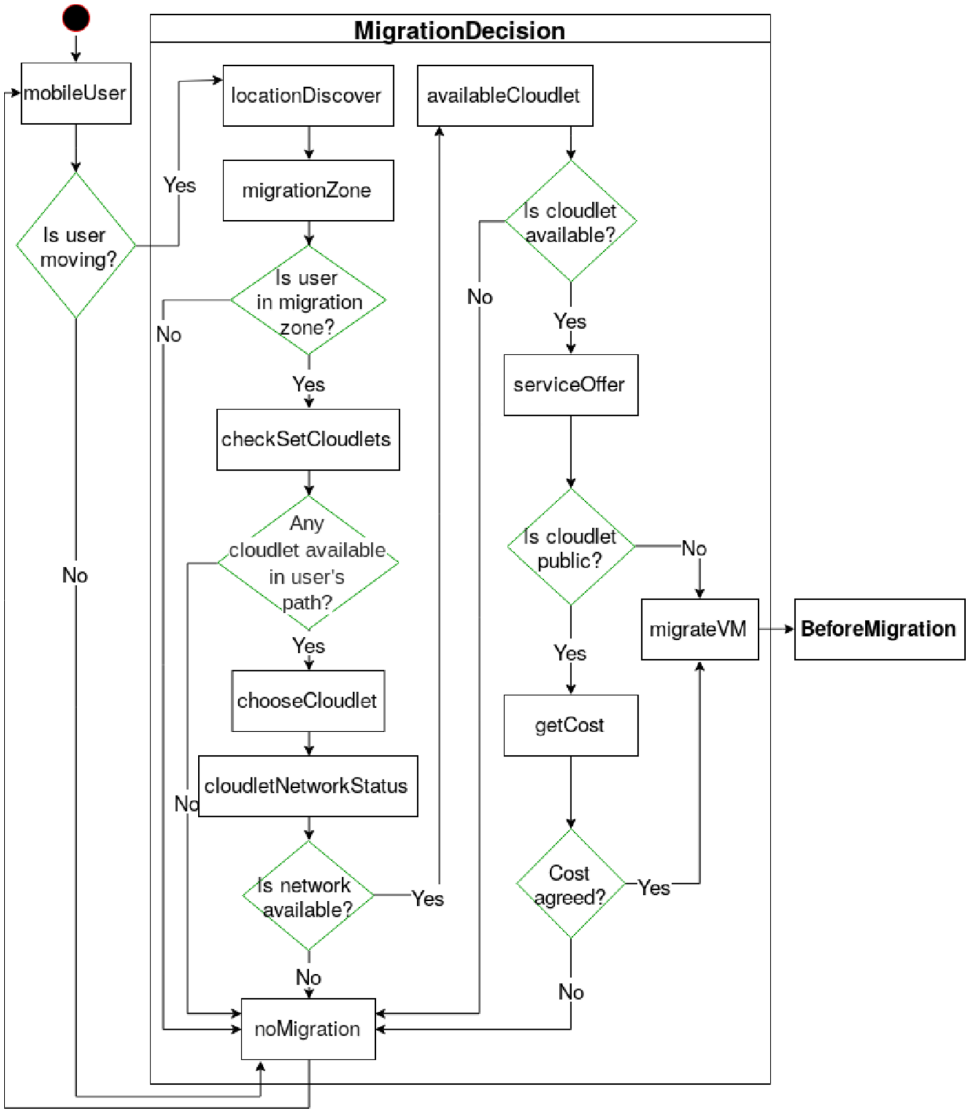


Fig. 8. MigrationDecision component.

Migration strategies 1 and 2 are calculated based on the geographical distance between the user and the access point/cloudlet. We highlight that the fog computing paradigm suggests to provide computing and network resources as close as possible to the users. However, many aspects may impact the quality of the connection between the users and their applications placed in the fog. The closest cloudlet and access point may not necessarily offer the lowest latency to the user. By implementing these strategies, MobFogSim allows to study this correlation between geographical distance and latency.

Migration strategy 3, instead, takes into consideration the end-to-end latency between the user and the destination cloudlet. This value is calculated as the sum between the latency from the user to the access point and that from the access point to the destination cloudlet.

Developers can extend these strategies and implement their own algorithms, for instance by choosing the destination cloudlet that offers the lowest migration time or energy consumption, or by including load balancing mechanisms.

3.2.3. *Before Migration*

Once the MigrationDecision component has decided that it is time to migrate the VM/container, the *BeforeMigration* component intervenes. Fig. 9 illustrates the steps performed by this component.

The first two steps of BeforeMigration are *dataPrepare* and *replicaVM*. They are respectively used to define the way through which VM/container data are prepared for migration and the way used to actually migrate them. After this, the third step opens the connection between the source and the destination cloudlet. Once the connection is established, data transfer starts through the network. Note that data preparation and transmission are performed according to a certain VM/container migration technique. In

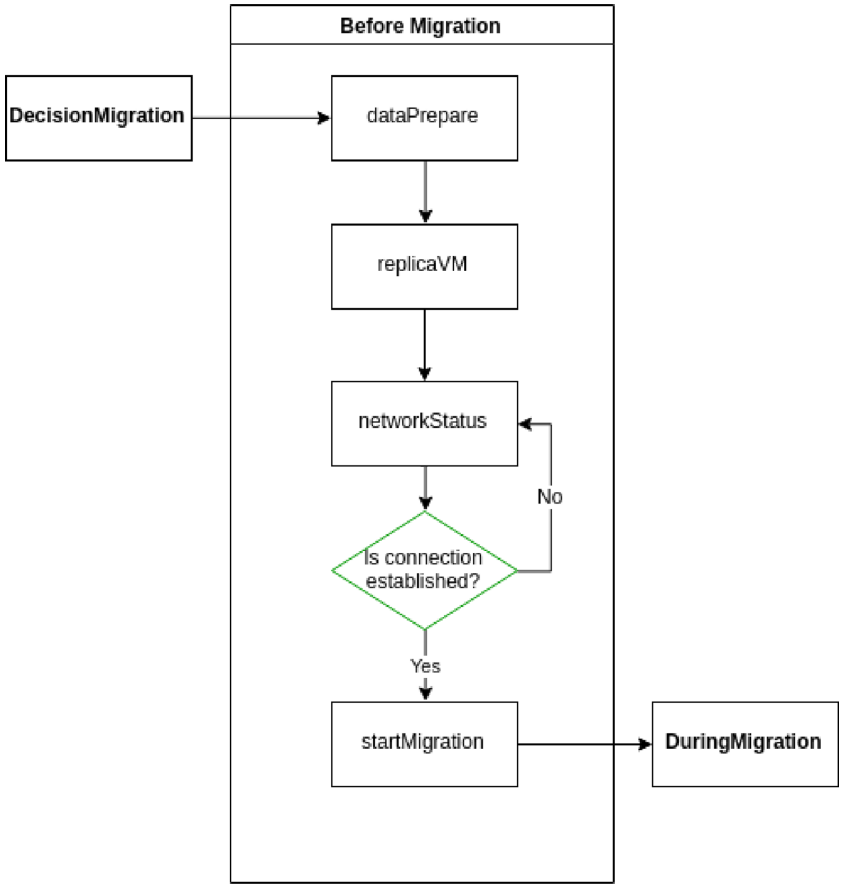


Fig. 9. BeforeMigration component.

literature, there exist four migration techniques, i.e., cold, pre-copy, post-copy, and hybrid migration [13]. At the moment of writing, MobFogSim models two of these techniques: cold and post-copy migration, which are described next. However, we highlight that developers are welcome to implement the remaining techniques and include them in the simulator.

Cold migration This migration technique is said to be "cold" because it: (i) first freezes/stops the VM/container to ensure that it no longer modifies its state; (ii) then checkpoints the whole state and transfers it while the VM/container is stopped; and (iii) finally resumes the VM/container at the destination cloudlet only when all the state is available. As such, cold migration features a very long *downtime*, namely the time interval during which the VM/container is not up and running. Downtime even coincides with the *total migration time*, which is the overall time required to complete the migration.

Post-copy migration This is a "live" migration technique, which means that the VM/container keeps on running while most of its state is being transferred to the destination cloudlet. The VM/container is stopped only for the transmission of a minimal amount of the overall state, after which the container runs at destination. This leads to a lower downtime compared to that of cold migration. Going into detail, post-copy migration first suspends the VM/container on the source cloudlet and copies its execution state (i.e., the CPU state, the content of registers) to the destination so that the VM/container can resume its execution there. Only after that (i.e., *post*) and while the VM/container is running, it copies all the remaining state, namely all the memory pages, which represent the vast majority of the whole state. Actually, there exist three variants of post-copy migration, which differ from one another on how they perform this final step. MobFogSim implements the "lazy migration" variant. With lazy migration, the resumed VM/container tries to access memory pages at destination, but, since it does not find them, it generates page faults. As a result, the *lazy pages daemon* at destination contacts the *page server* on the source cloudlet. This server then "lazily" (i.e., only upon request) forwards the faulted pages to the destination.

4. Implementation details

In this section, we discuss the most noteworthy aspects relative to the implementation of MobFogSim. In Section 4.1, we report the main Java classes that make MobFogSim an extension of iFogSim with support to device mobility and VM/container migration. Then, in Section 4.2, we focus on the simulation events and their flow within the migration and handoff procedure. Finally, Section 4.3 illustrates how we extended the preliminary version of MobFogSim to support realistic user's mobility patterns.

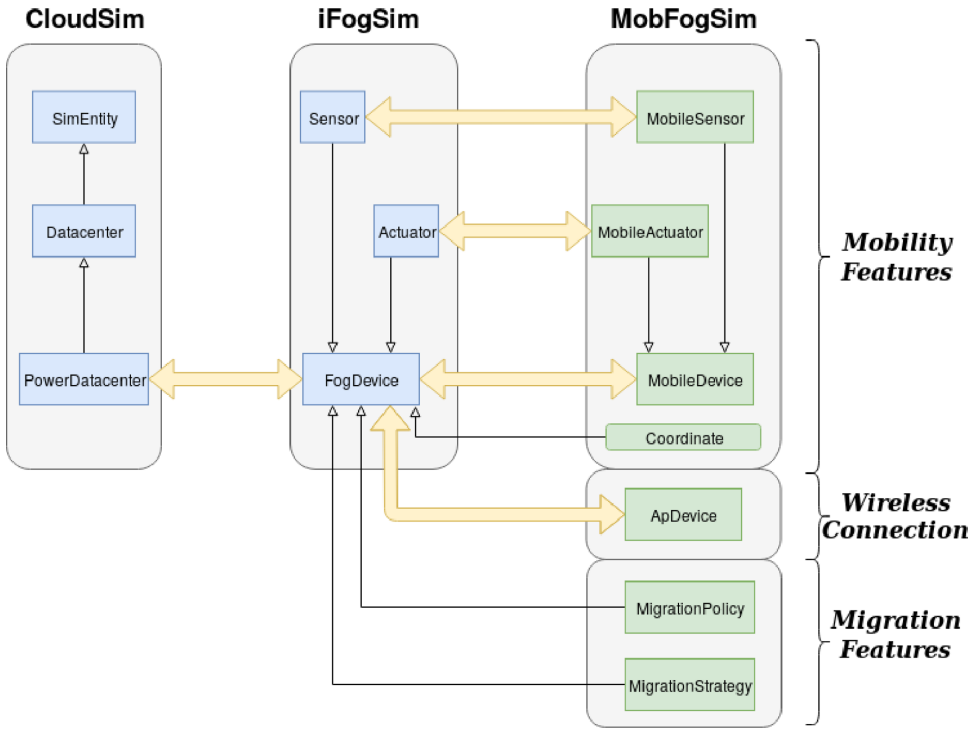


Fig. 10. Overview of MobFogSim as an extension of iFogSim and CloudSim.

4.1. MobFogSim as an extension of iFogSim

Fig. 10 shows the main Java classes present in the simulator. The leftmost part includes PowerDatacenter, which is the class from CloudSim that is important for the creation of the relevant entities in iFogSim and MobFogSim. The classes from iFogSim reside in the middle of the picture. The rightmost part, instead, shows the main classes introduced with MobFogSim. These allow to model device mobility, network handoff, and VM/container migration according to the description from the previous section. In what follows, we describe such classes:

- *Coordinate:* This class acts as a Cartesian Plan map (*X, Y*) to have all entities position during the simulation. These entities can be the fog servers (serverCloudlet), the wireless base stations (access points - APs), and the users' (IoT) devices. The developer can configure the map boundaries.
- *ApDevice:* This class extends FogDevice and has the access point responsibility in a wireless network. This class manages the handoff mechanisms and connections/disconnections of end devices.
- *MobileDevice:* This class also extends FogDevice. Its main goal is to allow a separation between fog servers and IoT devices, since iFogSim implements any device (servers and IoT) with the same features. With this separation, it is possible to have specific features for different devices.
- *MobileSensor:* This class extends the Sensor class. In iFogSim, to build a scenario with multiple sensors, the developer needs to instantiate several Sensor objects. In MobFogSim, MobileSensor already has a set of sensors, and the developer can instantiate only one object and add, when necessary, more sensors into the same hardware. A MobileSensor is associated to a MobileDevice.
- *MobileActuator:* This class is at the same level of abstraction of MobileSensor and extends the Actuator class from iFogSim.
- *MigrationStrategy:* This class implements the migration strategy to be applied in the simulation.
- *MigrationPolicy:* This class implements the migration policy to be applied in the simulation.

These classes implement the migration model presented in Section 3.2, thus creating a mobile simulation environment in MobFogSim that includes the decision-making algorithms described earlier in this paper, the events related to the migration, and the localisation system.

4.2. Implementation of events in MobFogSim

To summarise the core of MobFogSim, Fig. 11 shows part of the main events generated in a simulation. On the left-hand side, AppExample is a class where a developer is building all configurations for the simulation. In this class, one can perform steps to, for example: (i) create all ServerCloudlets and their features; (ii) create all SmartThings and their VMs/containers, MobileSensors,

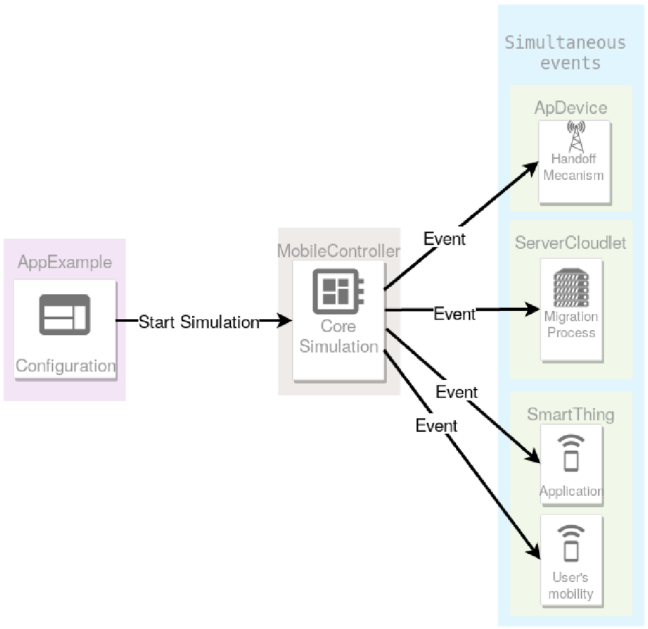


Fig. 11. The main events generated in a simulation.

MobileActuators; (iii) create the broker configuration; (iv) create tuple application; (v) create the network; and (vi) schedule all the initial simulation events. After all settings are in place, the MobileController class controls and schedules all the events in the simulator that are run by the different types of devices (ServerCloudlet, ApDevice, and SmartThing). A ServerCloudlet is responsible for executing all the events related to migration (e.g., verify if the fog service should migrate according to some migration policy). An ApDevice is responsible for executing all the events associated with the handoff mechanism (e.g., make the disconnection from the source AP and make the connection to the destination AP). A SmartThing is responsible for executing the user's end application and for implementing mobility (e.g., start processing tuples provided by the hardware sensor / move the user's device according to the new geographical position). An essential aspect of the simulation events is that each device can schedule events at the same simulation time, since those devices are independent entities.

For the sake of comprehensiveness, in Fig. 12, we illustrate the flow of events that occur in MobFogSim during a migration and handoff procedure, along with the classes involved. The purpose is to show how the sequence of events E1-E8 from Section 2.2 is implemented in the simulator. The scenario considered in Fig. 12 is such that: (i) VM/container migration is proactive; (ii) it is based on the post-copy (live) technique; and (iii) uses "lowest latency" as migration strategy. However, we highlight that the events would be the same also under different scenarios, though with a different flow and/or different classes involved. Note that elements in the sequence diagram are coloured differently according to their nature. Specifically, purple boxes represent Java classes. Blue and green arrows represent events relative to migration and handoff, respectively. The brown arrow represents an event in MobFogSim that is neither explicitly associated to migration nor handoff. Finally, black arrows are invocation of methods (and their eventual return values).

By looking at Fig. 12, it is possible to note that the first event is (periodically) scheduled by MobileController to the source FogDevice. This event is called *MAKE_DECISION_MIGRATION*, and it corresponds to E1 from Section 2.2 if the decision is to migrate the VM/container. In order to make this decision, FogDevice executes the *invokeDecisionMigration()* method, which in turn invokes the *shouldMigrate()* method of the LowestLatency class. The latter triggers the *verifyPoints()* method of LiveMigration, which inspects the relative position of the mobile device with respect to the current access point to find out if the user is in the migration zone and in the migration point. Moreover, verifyPoints() calculates the migration time of the VM/container. With the information set by verifyPoints(), shouldMigrate() is able to decide that it is time to migrate the VM/container (i.e., it returns *true* to invokeDecisionMigration()) and also selects the destination cloudlet based on the migration strategy. Once the decision to migrate is made, the source FogDevice sends a *TO_MIGRATE* event (i.e., E2 from Section 2.2) to itself, which results in the invocation of *invokeBeforeMigration()*. This method triggers the *dataPrepare()* method of the PrepareLiveMigration class, which calculates the time required to checkpoint the state of the VM/container as well as that to open the connection towards the destination cloudlet. Both these times are returned, as delayProcess, to the source FogDevice class. VM/container migration is then started by the source FogDevice by issuing the *START_MIGRATION* event (i.e., E3 from Section 2.2), with the consequent invocation of *invokeStartMigration*. This method disassociates the VM/container from the source cloudlet and associates it to the destination one. Then (and while the VM/container is being migrated), the handoff part of the procedure is carried out. Going into detail, MobileController (periodically) issues the CHECK_NEW_STEP event to itself and executes the checkNewStep() method. This method verifies that the mobile device is in the handoff occurrence point (i.e., at the boundary between the coverage areas of two access points) and

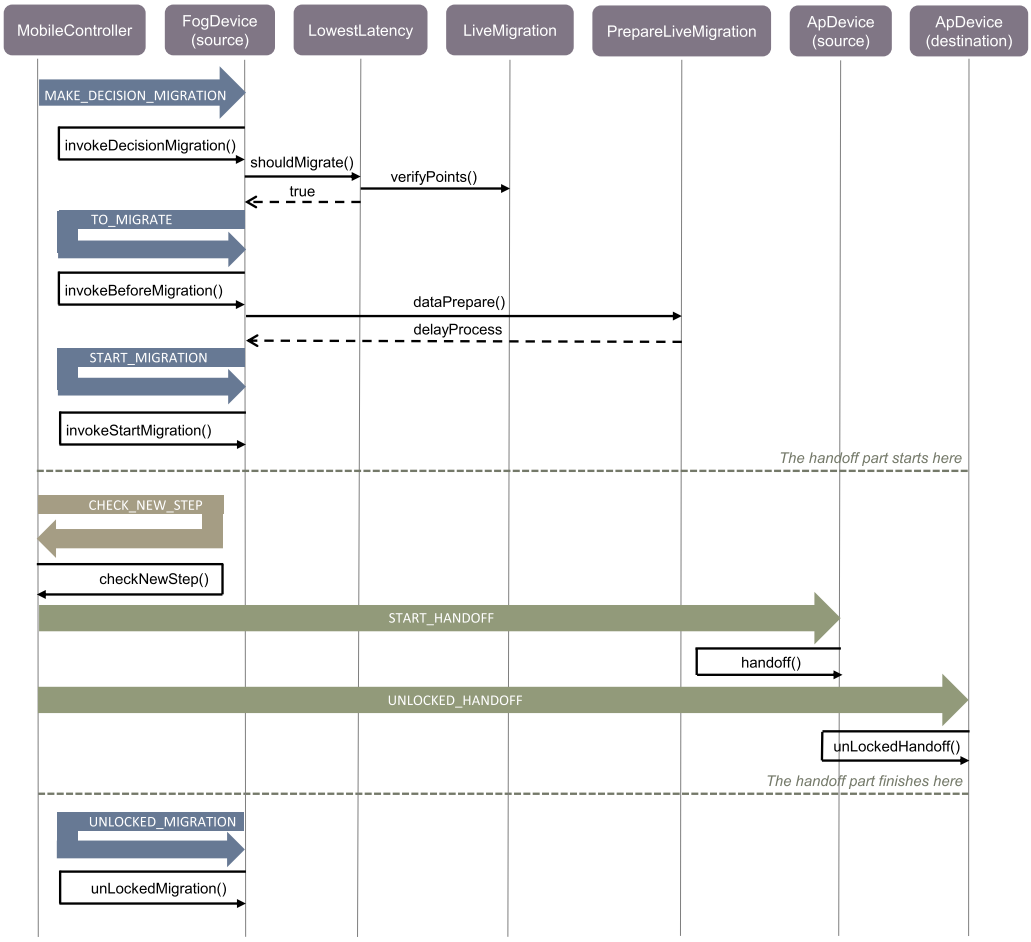


Fig. 12. Overview of the migration and handoff process as implemented in MobFogSim.

hence calculates the destination access point and the handoff time. Next, MobileController sends the *START_HANDOFF* event (i.e., E4 from Section 2.2) to the source access point, which invokes the *handoff()* method to disconnect the mobile device from the source access point and associate it to the destination one. Then, MobileController sends the *UNLOCKED_HANDOFF* event (i.e., E5 from Section 2.2) to the destination access point, which triggers the *unLockedHandoff()* method to set the end of the connection handoff process. Finally, the migration process concludes similarly to the handoff one, namely through the invocation of the *UNLOCKED_MIGRATION* event (i.e., E6 from Section 2.2) and of the *unLockedMigration()* method. Thus, the whole process is finished, and the user, who is now connected to the destination access point, can access the VM/container on the destination cloudlet (i.e., E7/E8 from Section 2.2).

4.3. Support to realistic user's mobility

The modifications made to iFogSim, which resulted in MobFogSim, introduced support to mobile devices. However, in its preliminary version, MobFogSim built only basic mobility patterns (i.e., constant speed in a straight line) for its users. Aiming to enrich the mobility scenarios, we have made some modifications in the simulator to support customised user's mobility patterns as input data.

Fig. 13 presents an example of the data flow used in a simulation. In this example, the mobility tool *Simulation for Urban Mobility* (SUMO) [9] (b) interprets the source mobility database saved in, for example,.XML format (a). Many realistic mobility databases are available in.XML format, such as the project LuST [14], which presents realistic data from vehicles in Luxembourg. The result of SUMO is then saved in.csv format (c).

Each.csv file represents the mobility of a vehicle in the simulation. Each line in this file contains data from the vehicle at one point in the simulation. The data are: (i) position x and y on the map; (ii) speed in metres per second; (iii) the direction in radiants; and (iv) the simulation time in which these data were collected.

This new database is hence used as a basis to define the users' mobility in MobFogSim. The simulator interprets this database and makes some modifications to adapt these data to its mobility model. Among these changes, there are, for example, the conversion of

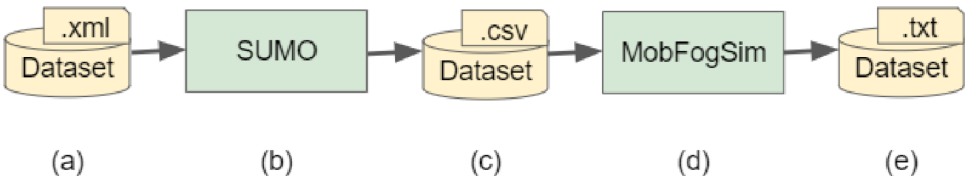


Fig. 13. Example of the data flow in a simulation.

the vehicle speed from metres per second to kilometres per hour as well as the conversion of the direction from radiants to the 8 main cardinal points, which are the basis of the mobility model in MobFogSim.

After the simulation in MobFogSim (d), a new database is built (e), which presents the results of user's behaviour for local resource management. Among these results, there are: (i) the average and the maximum latency presented by the application along the user's path; (ii) the migrations performed; (iii) the packages requested and attended; and (iv) the number of handovers.

The simulator described up to now was used to perform illustrative simulations, which are detailed later in this paper. Before presenting these simulations, in the next section we describe how some simulation parameters were measured from experimentation over a real testbed to: (i) validate the simulator behaviour; and (ii) prepare simulation scenarios using real data.

5. Simulator calibration and container migration over a real testbed

The objective of this section is twofold. Firstly, in Sections 5.1–5.3, we describe the experiments that we carried out over a real testbed to calibrate MobFogSim (i.e., to obtain realistic input values for the next simulations). Such a testbed, which is depicted in Fig. 14 and is detailed in the following sections, is the same that was used in [13] to evaluate container migration techniques in fog computing. Secondly, in Section 5.4, we report from [13] the main container migration results over the testbed. These results will be compared in Section 6.1 with those from MobFogSim in order to validate it. Note that, in Sections 5 and 6, we only deal with container migration. The reason for this is that Section 5 is about the calibration of the simulator based on the considered testbed, where fog services are implemented as containers, and about container migration results over that testbed. Section 6, instead, simulates container migration in MobFogSim to: (i) compare the results with those from this section, for validation purposes; (ii) evaluate the impact of different usersâ speeds on container migration, which is modelled based on the data extracted from the testbed. However, we restate the possibility to simulate also VM migration in MobFogSim.

5.1. Characterisation of maximum MIPS rating

There exist multiple ways to estimate a computer speed; one of these is to measure speed in Million Instructions Per Second (MIPS). MobFogSim, as iFogSim, takes MIPS ratings as inputs to indicate the maximum computation speed of devices in an IoT-Fog environment and to define the execution speed of tasks. The purpose of this first group of preliminary experiments is to obtain the maximum MIPS ratings of the devices in the real testbed.

The end device in the testbed is an ASUS Zenbook UX331UN notebook whose specifications are reported in Table 1. To calculate

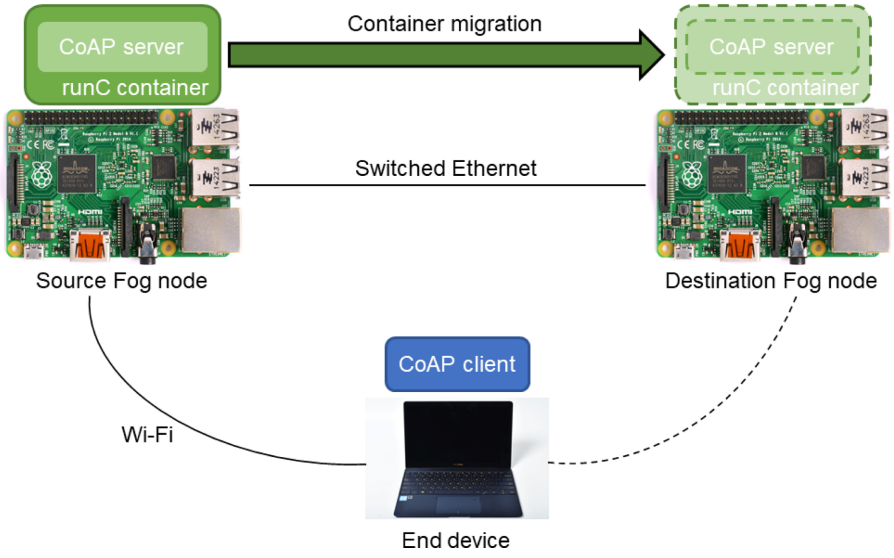


Fig. 14. Overview of the real testbed.

Table 1
Device specifications.

Device	CPU	RAM	Storage	Architecture	OS	Kernel	Technology
ASUS Zenbook UX331UN	Quad-Core 1.8 GHz (Turbo at 4.0 GHz)	16 GB	512 GB	x86_64	Ubuntu 18.04.1	Linux 4.15.0	SMT
Raspberry Pi 3 Model B	Quad-Core 1.2 GHz	1 GB	16 GB	aarch64	Debian 9.5	Linux 4.14.73	SMP

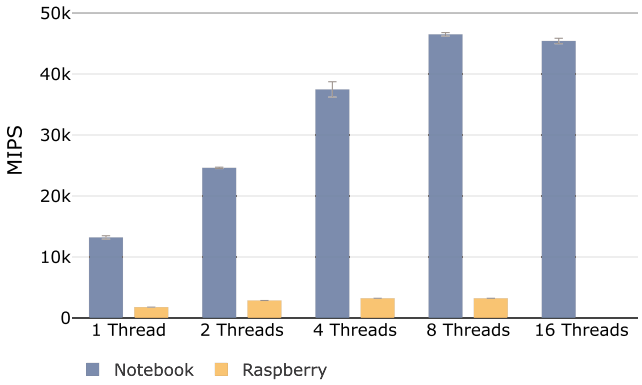


Fig. 15. Maximum MIPS ratings.

the maximum MIPS rating of this device, we leveraged the *cpumaxmp64* executable, which is one of the Roy Longbottom's Linux MultiThreading benchmarks [15]. This benchmark performs 64bit integer add instructions over 64bit registers via assembly language. It is possible to specify the number of threads, between 1 and 64, as a command-line parameter. Each thread executes independent code. The assembly code loops execute two billion add instructions each. We carried out experiments with 1, 2, 4, 8, and 16 threads, running the benchmark five times for each number of threads. The black bar charts in Fig. 15 represent the obtained results for the notebook, with a 95% confidence level. As shown, MIPS rating increases with the number of threads, reaching an average value of 46533.80 MIPS with 8 threads. With 16 threads, however, MIPS rating slightly decreases to 45441.20 MIPS. We were expecting this outcome, as the considered notebook features an Intel i7-8550U CPU, which is a Quad-Core processor with Hyper-Threading technology[5]. Hyper-Threading is the Intel implementation of Simultaneous MultiThreading (SMT), which makes a physical core appear to the OS as two logical processors [16]. Therefore, this notebook has eight logical processors, which explains why performance with 8 threads is the highest.

We then calculated the maximum MIPS rating of a Raspberry Pi 3 Model B, whose specifications are detailed in Table 1. In the real testbed, both the source and the destination cloudlets are Raspberry Pis. This time, we exploited the *MP-DHRYPi64* executable, which belongs to the Roy Longbottom's Raspberry Pi benchmark collection [17]. Each run of MP-DHRYPi64 executes 1, 2, 4, and 8 threads, with each thread executing a copy of the Dhrystone benchmark. Dedicated data arrays are used for each thread, but there are numerous other variables that are shared. The Dhrystone benchmark provides a measure of integer performance and has been the key standard benchmark since 1984. Its code, which is written in C, includes simple integer arithmetic, string operations, logic decisions, and memory accesses. Between 21% and 65% of the overall execution time is spent on string operations (i.e., string assignments and comparisons) [18]. Speed was originally measured in Dhrystones per second. This was later changed to VAX MIPS by dividing Dhrystones per second by 1757, which is the number of Dhrystones per second of the first 1 MIPS minicomputer, namely the DEC VAX 11/780 [19]. We run MP-DHRYPi64 five times. Results for the Raspberry Pi are the orange bar charts in Fig. 15 and are shown with a 95% confidence level. As for the notebook, MIPS rating increases with the number of threads. However, differently from the notebook, the maximum MIPS rating of the Raspberry Pi is reached with 4 threads rather than 8. Average computation speed is 2873.50 MIPS with two threads, 3234.33 MIPS with 4 threads, and 3231.95 MIPS with 8 threads. This is because the Raspberry Pi 3 Model B has a Broadcom BCM2837 CPU, which is a Quad-Core processor with Symmetric MultiProcessing (SMP) technology rather than SMT. As a result, the OS in a Raspberry Pi sees only four logical/physical processors, and this is why the highest performance is reached with 4 threads. Moreover, looking at Fig. 15, it is possible to notice how a common notebook is significantly more powerful than a Raspberry Pi: the maximum MIPS rating of the former is nearly 15 times higher than that of the latter.

5.2. Characterisation of latency and throughput

The objective of this second set of preliminary experiments is to characterise the Round Trip Times (RTTs) and the throughputs among the devices in the testbed. As shown in Fig. 14, the notebook is connected through Wi-Fi to the source cloudlet, which hence

[5] See https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html. Last accessed: 5 April 2019.

also behaves as a Wi-Fi access point. In order to let the Raspberry Pi behave as a Wi-Fi access point, we installed and configured *hostapd* (version 2.4) and *dnsmasq* (version 2.76). RTTs over Wi-Fi were measured using the *ping* command in Linux over 10 runs, with 20 measurements per run. We then considered the average RTT per run to obtain 9.55 ± 0.88 ms at a 95% confidence level. In order to get the throughput from the notebook to the Raspberry Pi, we performed 10 measurements using the *iperf3* tool, sending 50 MB each time. The resulting throughput value is 13.32 ± 0.97 Mbps, with a 95% confidence level. In a similar way, we calculated the throughput from the Raspberry Pi to the notebook, obtaining 13.05 ± 1.40 Mbps.

In [13], container migration was performed with two different couples of RTT and throughput values between the cloudlets. Each couple identifies a specific network condition between the cloudlets. The first condition (i.e., Condition A) has a RTT of 122.95 ± 5.57 ms and a throughput of 11.34 ± 2.31 Mbps. To obtain these values, we considered a computer connected through Ethernet to the University of Pisa network as source cloudlet and a smartphone connected to the Internet through 4G/LTE as destination. The second condition (i.e., Condition B) has a RTT of 6.94 ± 0.61 ms and a throughput of 72.41 ± 3.87 Mbps. These values were instead calculated by employing two fixed computers belonging to a bridged LAN of the University of Pisa and installed in two different buildings placed about 1 km far apart. We obtained both RTT and throughput values with the same procedure adopted to get values between the notebook and the Raspberry Pi. Since, in our testbed, the two Raspberry Pis are located in the same office and communicate through a switched Ethernet network (see Fig. 14), we had to emulate the aforementioned throughput and RTT values. To this purpose, we leveraged *Linux Traffic Control Network Emulator* (tc-netem[6]) to artificially set RTT values between the Raspberry Pis and *Linux Traffic Control Hierarchy Token Bucket* (tc-htb[7]) to limit the throughput. Note that legend entries in all the following figures with results report the one-way latency rather than the RTT between cloudlets. This is because MobFogSim takes one-way latencies as input.

5.3. Characterisation of the application

The purpose of this final set of preliminary experiments is to characterise the application that was used in [13] to evaluate container migration techniques. By characterisation of the application, we mean the measurement of all those values that MobFogSim requires as inputs to detail the application.

The considered application is a client-server one where both the client and the server are written in Java using the *Californium Constrained Application Protocol (CoAP)*[8] framework. This required the installation of *openjdk8* on all the devices of the testbed. CoAP is a specialised Web transfer protocol for use with constrained devices and constrained networks in the IoT. CoAP is inspired by HyperText Transfer Protocol (HTTP) and is therefore based on the Representational State Transfer (REST) paradigm. According to this paradigm, servers expose resources under a Uniform Resource Locator (URL), and clients access these resources using one of the following methods: GET, PUT, POST, and DELETE [20]. Let us now describe the considered client-server application in more detail. Before migration starts, the CoAP server runs within a runC[9] container on the source cloudlet. After migration ends, the container runs on the destination cloudlet. The CoAP client, instead, always runs on the notebook. RunC version 1.0.1 was used as container runtime on the Raspberry Pis. Once started, the server allocates 75 MB of RAM for random data. Once per second, the client sends a POST request to the server to represent sensor data. Upon reception of the request, the server modifies some of the memory pages with new random values and, in case of success, answers the client with response code 2.01 (*Created*). In [13], the server could either modify 10 kB or 500 kB every time. The purpose for this was to evaluate the effect of different page dirtying rates on container migration techniques. We define page dirtying rate as the speed at which the server modifies memory pages. As already explained in Section 3.2.2, MobFogSim currently implements migration according to the cold and post-copy techniques. As shown in [13], none of these techniques is influenced by the page dirtying rate of the service. Therefore, in this paper we consider 500 kBps as the only page dirtying rate featured by the CoAP server.

The first parameter that we measured to characterise the application were the MIPS ratings with which both the client and the server are executed. To this purpose, we run both the client and the server specifying the *stat* command of the *Linux perf*[10] utility (version 4.14.87 on the Raspberry Pi and 4.15.18 on the notebook). Linux perf is a user-space application that, making use of the *perf_events* interface of the Linux kernel, accesses all the CPU internal counters for performance monitoring. We run the application five times and, based on the performance metrics from Linux perf, we calculated MIPS ratings as follows:

$$MIPS = \frac{f \cdot ipc}{10^6} \tag{1}$$

where f is the CPU frequency (Hz) with which the task is executed, while *ipc* (i.e., Instructions Per Cycle) is the average number of instructions that are executed per clock cycle. The resulting task execution speeds are 2901 ± 119.26 MIPS and 281 MIPS for the client and the server, respectively. Results are shown with a 95% confidence level. With Linux perf, we also obtained the number of instructions that were executed each time by both the client and the server. The client executes 966.01 ± 13.32 million of instructions, while the server executes 2438.62 ± 22.72 million of instructions, with a 95% confidence level.

[6] See https://www.systutorials.com/docs/linux/man/8-tc-netem/. Last accessed: 30 December 2018.
[7] See https://linux.die.net/man/8/tc-htb. Last accessed: 30 December 2018.
[8] See https://www.eclipse.org/californium/. Last accessed: 10 April 2019.
[9] See https://github.com/opencontainers/runc. Last accessed: 10 April 2019.
[10] See http://www.brendangregg.com/perf.html. Last accessed: 10 April 2019.

We also measured the RAM requirements of both the client and the server. We consider RAM requirements in MobFogSim to be expressed as the maximum Resident Set Size (RSS) of a process during its lifetime, namely the maximum portion of RAM memory occupied by that process. To measure it, we specified the command[11] */usr/bin/time -f "RSS=%M"* when starting both the client and the server from terminal. After five runs, we obtained 49.05 ± 0.14 MB and 128.09 ± 0.16 MB as RAM requirements of the client and the server respectively, with a 95% confidence level. Besides, we leveraged the *du* command in Linux to measure the disk usage of both the client, namely its JAR file, and the server, namely the Open Container Initiative (OCI) bundle[12] for the runC container. Results are of 4 MB for the client and 412 MB for the server.

Finally, we also measured the size of each CoAP request (i.e., from the client to the server) and each CoAP response (i.e., from the server to the client). To do so, we launched Wireshark (version 2.6.6) on the notebook and found that the CoAP request was 87 B while the CoAP response was 54 B.

5.4. Container migration over the testbed

In this section, we report the main container migration results that were obtained from the experimentation over the testbed. Further details may be found in [13]. In that paper, we evaluated all the existing container migration techniques: cold, pre-copy, post-copy, and hybrid. However, MobFogSim currently allows to simulate container migration only according to the cold and post-copy techniques. As a result, these are the only two techniques that we consider in this paper. We performed five runs for each combination of migration technique (i.e., cold, post-copy) with network condition (i.e., A, B). We exploited Checkpoint/Restore In Userspace (*CRIU*[13]) version 3.10 for checkpointing and restoring the container, and *rsync* version 3.1.2 as file transfer mechanism to migrate the checkpointed container state. In what follows, we analyse results in terms of: (i) total migration time, i.e., the overall time required to complete container migration; (ii) downtime, i.e., the time interval during which the container is stopped for migration, and the service is therefore not available to the client; and (iii) total volume of data transmitted during migration. We leveraged the Linux *time* command to measure total migration times and downtimes, while we used the rsync *–stats* option to collect statistics on the amount of data transferred through rsync. All the following results are presented with a 95% confidence level.

Fig. 16 depicts the total migration times. These times for post-copy migration are always longer than those for cold migration, even though these two techniques overall transmit similar amounts of data, as discussed later. This result can be explained as follows. In cold migration, memory pages are transmitted from the source to the destination node without the need for any request from the latter. Instead, post-copy migration was implemented in [13] according to the "lazy migration" variant, which means that memory pages are transferred by the page server on the source node only upon request of the lazy pages daemon running at destination. Thus, the time to perform such requests, which are not present in cold migration, increases the overall total migration time of the post-copy technique. This difference between cold and post-copy migrations is particularly noticeable under network Condition A, where RTT between the cloudlets is considerably higher than that of Condition B.

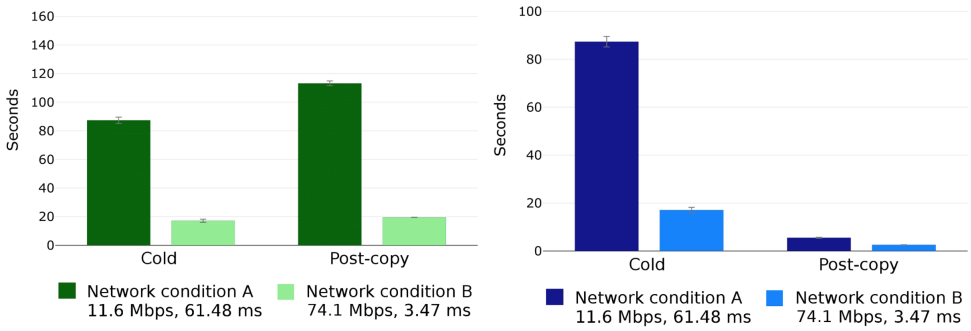
We now focus on downtimes. As shown in Fig. 16b, downtimes are the main difference between the two migration techniques. Cold migration is so called because the container is frozen/stopped throughout the whole migration procedure. As a result, downtime for cold migration is the highest among the migration techniques and even coincides with the total migration time. Especially under low-throughput conditions (e.g., Condition A), cold migration downtimes may be unacceptable for most applications. Post-copy migration, instead, is one of the "live" migration techniques. This means that the container is running while most of its state is being migrated to the destination node. In particular, post-copy migration first stops the container and transfers at destination only the execution state (i.e., the CPU state, the content of registers), whose size is negligible with respect to that of memory pages. Then, it resumes the container at destination and transmits all the memory pages while the container is up and running. Therefore, downtimes for post-copy migration are significantly lower than those for cold migration, as depicted in Fig. 16b. Note that the size of the execution state for the considered container is 1.2 MB, which is significantly smaller than the overall volume of data transferred to the destination cloudlet, as we show through the next figure.

Fig. 17 illustrates the volumes of data transferred during migration. Cold and post-copy techniques transfer similar amounts of data, namely about 115 MB. This is due to the fact that both these techniques transfer each memory page only once, unlike pre-copy and hybrid migrations, which indeed transfer higher volumes of data [13]. Moreover, the amount of data transferred by cold and post-copy techniques is irrespective of the network conditions between cloudlets.

6. MobFogSim evaluation

In this section, we evaluate MobFogSim. Specifically, Section 6.1 validates our simulator by comparing its results of container migration with those from the real testbed. Then, in Section 6.2, we employ MobFogSim to assess how user's mobility impacts on container migration in the fog. To this purpose, we run simulations where users move with different realistic mobility patterns taken from urban buses of Luxembourg.

[11] See http://man7.org/linux/man-pages/man1/time.1.html. Last accessed: 10 April 2019.
[12] See https://github.com/opencontainers/runtime-spec/blob/master/bundle.md. Last accessed: 02 May 2019.
[13] See https://www.criu.org/Main_Page. Last accessed: 26 April 2019.



(a) Total migration time. (b) Downtime.

Fig. 16. Migration time and downtime under conditions A and B.

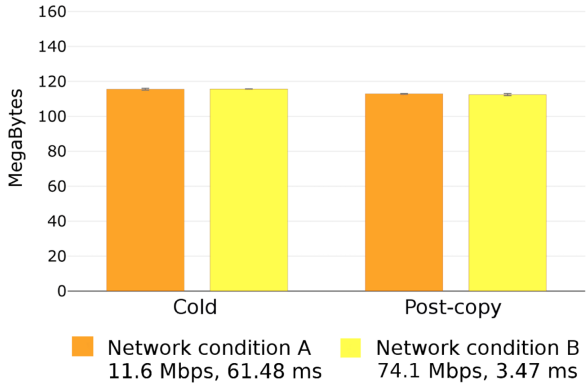


Fig. 17. Amount of data transferred.

6.1. Simulator validation

In this section, we validate our simulator by comparing its results of container migration with those discussed in Section 5.4. The preliminary experiments described in Sections 5.1–5.3 allowed us to select realistic values for the input parameters in MobFogSim, thus to replicate the network and service conditions of [13] during simulation. Aiming to simulate an environment as similar as possible to the testbed one, we assume cloudlets with homogeneous computing and network resources, as presented in the testbed. Although we did not perform simulations in heterogeneous scenarios in terms of cloudlets' resources, this is supported by the simulator. For convenience, Table 2 reports these parameters in alphabetical order along with their values. Names in brackets are the actual variable names used in MobFogSim to indicate those parameters. We run 30 simulations for each combination of migration technique (i.e., cold, post-copy) with network condition (i.e., A, B). Similarly to Section 5.4, simulation results are analysed and compared with those from the real testbed in terms of (i) total migration time; (ii) downtime; and (iii) network usage in terms of (a) total volume of data transmitted during migration and (b) link usage. The results are presented with a 95% confidence interval.

In addition to the input parameters provided by testbed experiments, some complementary input values were assumed as part of the simulated environment. These simulation settings are described as follows. In the simulated scenario, we assumed a square 10 x 10 km map with 144 uniformly distributed cloudlets. Each cloudlet is connected to one access point which reaches up to 500 m of signal coverage. Each simulation assumes a uniformly distributed random direction for user's mobility. The user is supposed to cross the map in a constant speed (20 kmph) until she/he reaches the opposite map edge. However, the simulations end once the user finishes her/his first container migration process. The migration point policy was defined as a static point. The migration process starts once the user reaches the migration point, which is defined at 40 m from the access point coverage boundary. The destination of the container in the migration process is chosen based on a greedy approach. The migration strategy assumed in this evaluation selects the cloudlet with the Lowest Latency among a set of 10 candidate cloudlets that are present in the user's path. Aiming to evaluate a more flexible environment, we assumed three different execution state sizes transmitted during migration: 1.2 (which is the actual size from the testbed results), 6.0, and 12.8 MB. Table 3 summarises the above settings of the simulated environment.

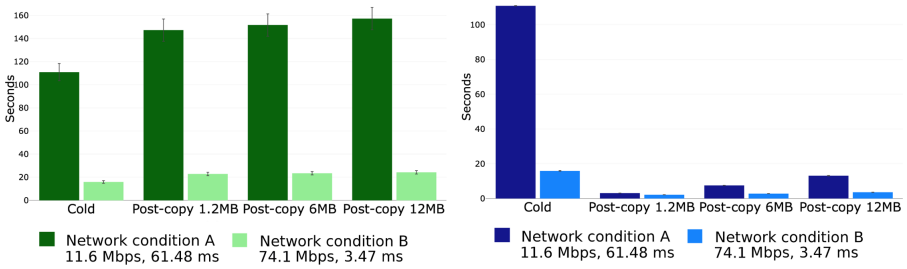
Fig. 18 a presents the total migration times in the simulated scenarios. Similarly to the testbed results presented in Fig. 16a, post-copy migration, in all its variants, presents higher values than cold migration under the correspondent network conditions. As already explained in Section 5.4, this result is due to the fact that post-copy migration transfers memory pages only upon request, and the time for such requests increases the total migration time. Going into detail, under condition A, both the simulated and the testbed post-copy scenarios present a migration time about 30% longer than that of cold migration. However, the simulated environment

Table 2
List of input parameters and their values based on the testbed experiments for simulation in MobFogSim.

Parameter	Value
Client execution speed (*mips*)	2901 MIPS
CoAP request size (*tupleNwLength*)	87 B
CoAP response size (*tupleNwLength*)	54 B
Disk usage of client (*size*)	4 MB
Disk usage of server (*size*)	412 MB
Maximum speed of notebook (*mips*)	46534 MIPS
Maximum speed of Raspberry Pi (*mips*)	3234 MIPS
Number of instructions executed by client (*tupleCpuLength*)	966 million
Number of instructions executed by server (*tupleCpuLength*)	2439 million
One-way latency between notebook and Raspberry Pi (*UplinkLatency*)	4.78 ms
One-way latency under condition A between Raspberry Pis (*lat*)	61.48 ms
One-way latency under condition B between Raspberry Pis (*lat*)	3.47 ms
RAM requirement of client (*ram*)	49 MB
RAM requirement of server (*ram*)	128 MB
Server execution speed (*mips*)	281 MIPS
Throughput from notebook to Raspberry Pi (*upBw*)	13640 kbps
Throughput from Raspberry Pi to notebook (*downBw*)	13363 kbps
Throughput under condition A between Raspberry Pis (*bw*)	11612 kbps
Throughput under condition B between Raspberry Pis (*bw*)	74148 kbps

Table 3
List of input parameters and their values assumed for the settings of the validation scenario in MobFogSim.

Parameter	Value
Container's execution state size in live migration process	1.2 MB, 6.0 MB, and 12.8 MB
Access point coverage (radius)	500 m
Number of cloudlets	144
Density of cloudlets per access points	1:1
Migration strategy	Lowest latency
Migration point policy	Static (40 m)
User's speed	Constant (20 kmph)



(a) Migration time. (b) Downtime.

Fig. 18. Migration time and downtime under conditions A and B in MobFogSim.

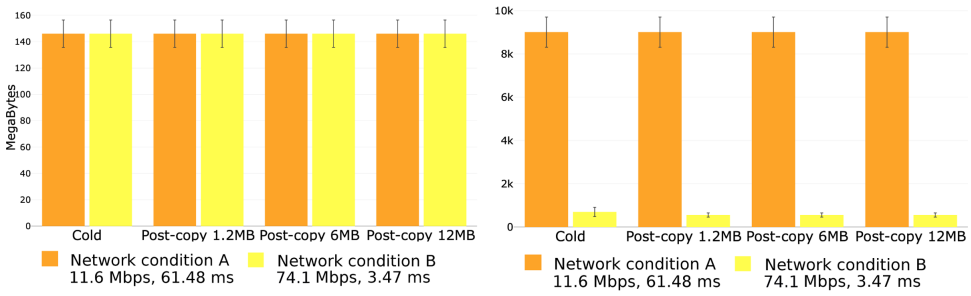
presents average values that are 25% higher than those in the correspondent testbed scenario. Under condition B, both the simulated cold and post-copy migrations present results that are close to the values from the testbed.

In the particular case of post-copy migration, the size of the execution state transmitted in that process does not present a significant impact on the migration time. Increasing the execution state size from 1.2 to 12.8 MB resulted in an increment of about 6% in the migration time under both network conditions A and B.

Based on these simulation scenarios used as a starting point for comparisons, in general, the total migration time presented by MobFogSim tends to be consistent with the testbed results for both cold and post-copy migrations.

Another relevant metric for the evaluation of the migration techniques is the downtime, which is presented in Fig. 18b. Similarly to the testbed results presented in Fig. 16b, the downtime of cold migration is higher than that of post-copy and even coincides with the migration time presented in Fig. 18a, under both conditions A and B. Even though the execution state size does not have a significant impact on the migration time, this parameter strongly influences post-copy migration in terms of downtime. Assuming an execution state size of 12.8 MB, the simulations suggest a downtime about 420% higher under condition A and 66% higher under condition B, if compared to migrations with an execution state size of 1.2 MB.

Aiming to complement the evaluation of the migration techniques used in the scope of this work, we also present performance in



(a) Amount of data transferred. (b) Network use.

Fig. 19. Data transferred and network use under conditions A and B in MobFogSim.

terms of: (a) total volume of data sent between two cloudlets over the migration process; and (b) link usage based on the amount of data to be sent, the throughput, and the latency between the cloudlets. Fig. 19a presents the amount of transferred data between the cloudlets. The simulation scenario presented a volume of transferred data close to 150 MB, which is about 20 MB higher than the testbed environment. As presented in the testbed results (see Fig. 17), the volume of data sent is stable under both the network conditions. Besides, as in the testbed, both the migration techniques transfer a similar volume of data (nonetheless, total migration time for post-copy is higher due to the time necessary to request memory pages). As a complementary metric, Fig. 19b presents the time of link usage, which we define as the amount of data sent multiplied to the link latency between the source and destination cloudlets. This metric clarifies the difference between the two network conditions, A and B, in terms of efficiency. In average, condition A presents an efficiency of around 1200% that of condition B.

6.2. *Different users' mobility patterns*

Due to the wide range of potential users of the fog computing infrastructure, each of them with their requirements and characteristics, the architecture must be able to deal with their peculiarities. Some users' devices like smartphones or devices embedded in vehicles present different mobility patterns in terms of route and speed.

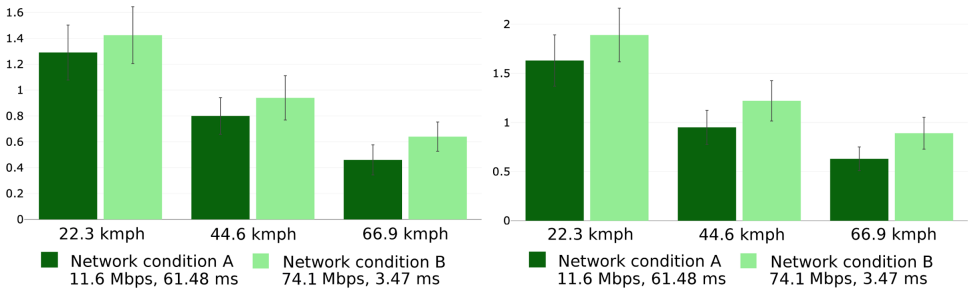
Aiming to increase the covered scenarios presented in Section 5, the objective of this section is to evaluate MobFogSim in scenarios with different users' mobility patterns and to assess how users' mobility impacts on container migration in the fog. To achieve it, we considered a more realistic scenario by selecting 100 different urban buses mobility patterns from LuST [14] as the input for users' mobility in the simulations. The set of selected buses moves, on average, at 22.3 kmph in a route of, on average, 26.44 min.

Each bus has a different route and speed. This increased the mobility scenarios presented in Section 5 but did not allow us to isolate variables to evaluate the impact of different mobility aspects, such as speed, in the container migration process. Aiming to improve the covered scenarios in terms of speed, we built two more datasets, keeping the same routes of the original one described above but using one multiplication factor for users' speeds. Specifically, these two new datasets were built by increasing the users' speeds by two and three times. These three datasets allowed us to compare the same realistic route, though using different users' speeds. For convenience, we used these datasets to describe the simulation scenarios in this section. Scenario 1 used the original users' speeds (22.3 kmph, on average); scenario 2 considered the doubled users' speeds (44.6 kmph, on average), while scenario 3 was that with the tripled users' speeds (66.9 kmph, on average). These new scenarios allowed us to use realistic mobility patterns to evaluate the impact of user's speed on the container migration process.

The simulation scenarios presented in this section were built based on the settings presented in Section 6.1, except from the execution state size and, as mentioned, the user's mobility pattern. The execution state size for post-copy migration was fixed at 1.2 MB, which is the closest simulation setup to the testbed results.

A number of metrics can be used to understand the impact of user's mobility on container migration in fog computing. The first metric is the number of migrations made for the users along their routes. Fig. 20a presents the average number of cold migrations performed in scenarios 1, 2, and 3, considering network conditions A and B for each scenario. Fig. 20b presents these results for post-copy migration. For both the migration techniques, the users' speeds presented a determinant impact on the number of migrations. Under both the network conditions and for both the migration techniques, increasing the users' speeds tends to decrease the total number of migrations. Specifically, for both cold and post-copy migrations, scenarios 3 show almost 60% (under condition A) and 55% (under condition B) fewer migrations than scenarios 1. Scenarios with Cold migrations present, on average, close to 15% fewer migrations than those with post-copy migrations, although they could be seen as technically equivalent if considering the confidence intervals.

Even though users' speed has a significant impact on the number of migrations, it does not affect the migration time and the downtime. These indeed remain consistent with the baseline simulations described in Section 6.1. Fig. 21a and 21 b show the results of migration time using cold and post-copy migrations, respectively. As discussed in Section 6.1, the results of this section show that better network conditions provide shorter migration times to the users. In general, condition B presents migration times that are 85% shorter than those under condition A. Simulations with cold migration under condition A present a migration time of 110 s on



(a) Average number of migrations – cold. (b) Average number of migrations – live.

Fig. 20. Average number of migrations for cold and live techniques with different users' speeds in MobFogSim.

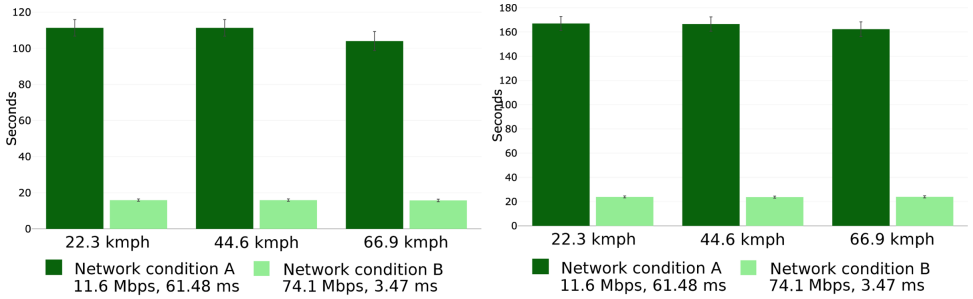
average, while, under condition B, the average migration time is 15 s. The post-copy technique presents instead migration times close to 150 s under condition A and 25 s under condition B. The post-copy technique requires, on average, 20% more time to finalise each migration than the cold technique.

Fig. 22 a shows the downtime in case of cold migrations, while Fig. 22b presents that in case of post-copy migrations. As depicted, users' speeds have no significant impact on this metric in both migration approaches. On average, condition B produces a downtime 85% lower than that in scenarios under condition A. Similarly to the migration times, the results for downtime are consistent with the baseline simulations discussed in Section 6.1 in most of the scenarios. The downtime of cold migrations is between 10% (condition B) and 20% (condition A) higher in the simulation scenarios than the results presented in the testbed. Live migration presented a consistent result for downtime assuming condition B. Instead, the testbed results assuming condition A are 80% lower than the values in the simulation environment. Even though absolute values differ, we observe the same pattern in the behaviour between the testbed and the simulated results.

The amount of data transmitted between cloudlets in each migration process is not related to the user's speed. However, the average number of migrations, presented in Figs. 20a and 20 b, affects the cumulative amount of data transmitted in the migration process along the user's path. Figs. 23a and 23 b present the total amount of data transmitted through the user's path for cold and live migrations, respectively, between cloudlets under network conditions A and B. In the case of cold migrations, fast users (66.9 kmph, on average) under condition A transfer about 170 MB of data, which is 55% of what is transferred in scenario 1 under the same conditions. The post-copy technique presents similar results in terms of the influence of users' speeds but transfers between 12% and 21% more data than the cold migration approach.

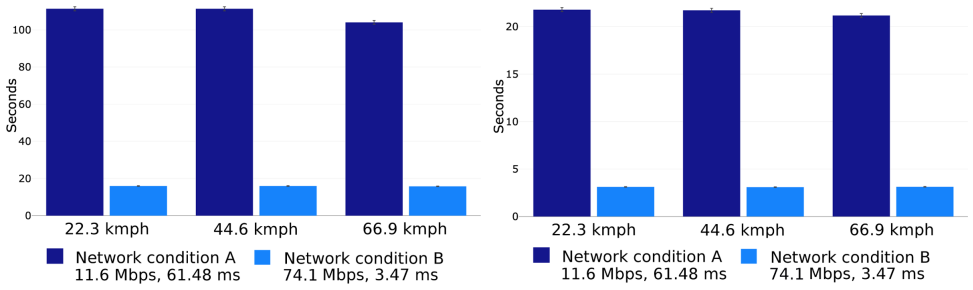
One of the advantages of fog computing is to provide user's devices with low latency access to remote resources. MobFogSim returns some results related to the delay between the user's devices and her/his container running at the fog layer. Fig. 24a shows the average delay in case of cold migrations. As expected, the worst network condition, namely condition A, presents the highest results. Fig. 24b depicts instead the results in case of post-copy migrations. This technique produces average delays that are between 60% and 70% higher than those for the cold migration under condition A and about 30% higher under condition B. Such values are justified by the difference between the number of computed packets in these two migration approaches. The average delay computed by MobFogSim does not include dropped packets. The migration process starts when the system identifies a better cloudlet to serve the user and finishes when the user is closer to her/his destination cloudlet. In the cold migration approach, packets that arrive during this time interval are dropped, which makes the packets be accepted when the user is closer to her/his source or destination cloudlets and, consequently, with a lower latency. In the post-copy approach, instead, the user's container still serves her/him during the migration process, which allows the container to accept packets when latency is not as low as it could be.

These results show that post-copy migration decreases the downtime but may increase the delay during the migration process. For some use cases, decreasing the downtime may be paramount; for other delay-sensitive applications, a significant increase in the delay



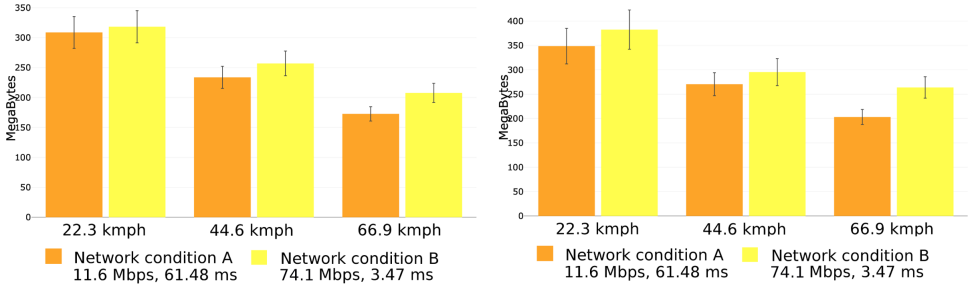
(a) Average migration times – cold. (b) Average migration times – live.

Fig. 21. Average migration times for cold and live techniques with different users' speeds in MobFogSim.



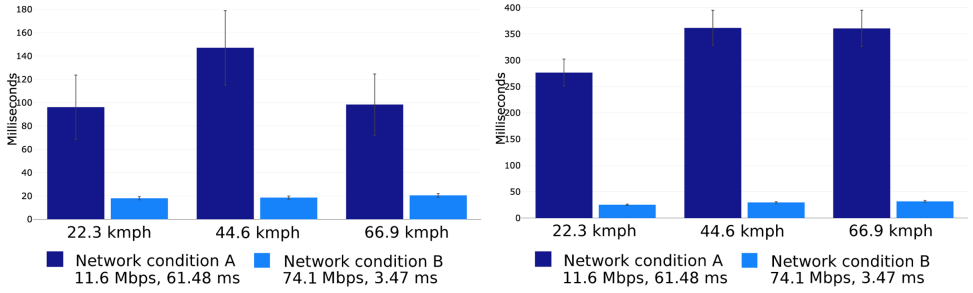
(a) Average downtimes – cold. (b) Average downtimes – live.

Fig. 22. Average downtimes for cold and live techniques with different users' speeds in MobFogSim.



(a) Data transferred – cold. (b) Data transferred – live.

Fig. 23. Average amount of data transferred for cold and live techniques with different users' speeds in MobFogSim.

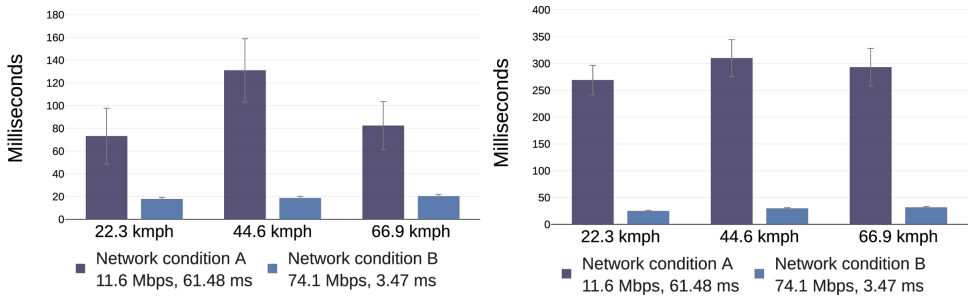


(a) Average delays – cold migration with static migration point. (b) Average delays – live migration with static migration point.

Fig. 24. Average delays for cold and live techniques with different users' speeds and static migration point in MobFogSim.

to the container may compromise user's experience.

As discussed in Section 3.2.1, MobFogSim is designed to support both static and dynamic migration. In static migration, the migration process starts at a known point in time and may be dependent on the distance to the edge of the area covered by a network access point. With dynamic migration, the start of the migration process is decided based on the following parameters: (i) a user's speed of movement; (ii) network connection between source and destination cloudlets; and (iii) data volume to transmit. This section concludes with an analysis of the impact of dynamic migration on migration performance. We carried out additional experiments under the same conditions described above at the beginning of this section, with the only difference being the use of a dynamic migration time point rather than a static one. Results describing the number of migrations, migration times, downtimes, and data transferred during the migration proccess are in line with those obtained considering a static migration point. This suggests that the type of migration point negligibly influences these metrics. However, a dynamic migration event may significantly reduce the delay experienced by a mobile user, as shown in Fig. 25. Static migration decisions are of limited benefit when a user has a high speed of movement and when there are poor network conditions between cloudlets. In these situations, by the time that the container is migrated to a new cloudlet, the mobile user might have moved further away and may be connected to another access point which is again far away from the new cloudlet. A dynamic migration decision starts (and therefore terminates) the migration process earlier for fast users and/or poor network conditions between cloudlets. This leads in general to an increase in duration over which a mobile



(a) Average delays – cold migration with dynamic migration point.

(b) Average delays – live migration with dynamic migration point.

Fig. 25. Average delays for cold and live techniques with different users' speeds and dynamic migration point in MobFogSim.

Table 4
Comparative table summarising the main characteristics of the fog computing simulators.

Simulator	Mobility/Handoff	VM/container Migration	Programming Language
VirtFogSim [21]	✓	-	MATLAB
YAFS[22]	✓	-	Python
FogNetSim++ [23]	✓	-	C++
FogTorch [24]	-	-	Java
iFogSim[10]	-	-	Java
EdgeCloudSim [26]	✓	-	Java
MobFogSim	✓	✓	Java

user is connected to a *close enough* cloudlet and consequently leads to an overall reduction of network delay. Such improvements are not so evident under network condition B, where throughput between cloudlets allows migration to terminate quickly. Instead, under network condition A, average delay with dynamic migration are about 85% of those with a static migration point for both cold (see Fig. 25a) and live (see Fig. 25b) migration techniques.

7. Related work

In this section, we review the state-of-the-art simulators for fog computing environments and highlight the comprehensiveness and novelty of MobFogSim. Table 4 summarises the main characteristics of these simulators, with a focus on mobility support.

VirtFogSim [21] does not model aspects such as VM/container migration, energy consumption, or pricing. However, it dynamically tracks the energy-delay application performance against abrupt changes due to failures or device mobility, e.g., mobility-induced changes of the available up/down bandwidth. The most distinctive functionality of VirtFogSim is that it allows to model cellular network access, which is useful when simulating 4G/5G scenarios. VirtFogSim is currently the only simulator that explicitly provides such a feature. Besides, this simulator includes a Graphical User Interface (GUI) that shows the simulation results in tabular, bar-chart, and coloured map graph formats.

Yet Another Fog Simulator (*YAFS*) [22] is a Python simulator for fog computing environments. It is particularly good at modelling network failures and therefore allows to evaluate service placement solutions in failure cases or to design robust networks. Network failures may be modelled in two possible ways: (i) through the runtime creation/deletion of cloudlets and network links; (ii) through *custom processes*, namely functions invoked at runtime for the implementation of real events. YAFS models mobility, sensors, and actuators but does not model aspects such as energy consumption or VM/container migration. Finally, although it does not include a GUI for the description of fog network topology, YAFS allows to import a simulation scenario as a JSON file.

The main objective of *FogNetSim++* [23] is to overcome the limitations of the other simulators in network modelling. They do not (or only partially) take into account real-network properties and therefore simulate idealistic networks where no packet loss, congestion, or channel collision happen. Instead, FogNetSim++ extends OMNeT++[14], which is a well known framework for building network simulators, to model all these aspects. Moreover, it includes popular communication protocols for simulation, such as TCP, UDP, MQTT, and CoAP. Furthermore, FogNetSim++ models several other aspects, such as energy consumption, pricing, mobility, and handoff mechanisms.

FogTorch [24] is a Java tool that outputs all the possible deployments of application modules over a fog computing infrastructure, provided: (i) the specification of the application requirements; (ii) the description of the infrastructure, in terms of devices and network links; and (iii) the definition of a deployment policy. The FogTorch user then selects the best deployment out of the proposed alternatives. *FogTorch*Π [25] is an extension of FogTorch that uses Monte Carlo simulations to model variations over time of the QoS

[14] See https://omnetpp.org/.

of network links, which is expressed in terms of latency and bandwidth.

iFogSim [10] was the first fog computing simulator. It is implemented in Java as an extension of the most popular cloud computing simulator, *CloudSim*[15]. iFogSim allows to test resource management and service placement strategies in terms of: (i) service latency; (ii) service throughput; (iii) network usage; (iv) energy consumption; (v) operational costs; and (vi) pricing. iFogSim provides a GUI to describe fog network topologies (i.e., sensors, actuators, cloudelts, cloud data centre, and interconnections among them). Topologies can then be exported as a JSON file and imported again in a later moment. iFogSim presents several strengths, which are the reason why we chose to implement MobFogSim as its extension. Firstly, iFogSim provides the widest range of functionalities, which span from network and resource management modelling to energy consumption and operational costs modelling. Secondly, it is by far the most used fog simulator in literature, which makes it the best candidate to compare own solutions with the related work. Thirdly, iFogSim extends CloudSim, which is the most popular cloud computing simulator. Therefore, it is relatively easy to use for all those who have already had experience with CloudSim. However, iFogSim also has some limitations. The most evident is probably the lack of a detailed and consistent documentation, which might make iFogSim hard to use for those who have never worked with CloudSim before. Besides, network modelling in iFogSim is rather simplistic, as this simulator does not deal with real-networks aspects such as packet loss, network congestion, and channel collisions. Furthermore, iFogSim does not model mobility scenarios and VM/container migration among cloudlets.

EdgeCloudSim [26] is another prominent simulator for fog computing environments. Also EdgeCloudSim is an extension of CloudSim. EdgeCloudSim models network delays more accurately than iFogSim, which considers network delays to be always fixed. EdgeCloudSim, instead, includes a networking module that calculates network delays, based on the current network load, when data need to be sent over the network. Even though EdgeCloudSim provides a better network modelling, it does not provide the same range of functionalities that are available in iFogSim. For example, energy consumption, operational costs, and pricing modellings are all missing in EdgeCloudSim. Device mobility is modelled, but VM/container migration is not. Hence, EdgeCloudSim may be useful to those who work in Java and are mostly focused on evaluating network metrics (e.g., service latency, network usage).

None of the above simulators models VM/container migration, which however is an important resource management and service provisioning aspect. VM/container migration may enable both traditional (i.e., already existing in cloud-only networks) and novel (i.e., introduced with fog computing) scenarios. For instance, VMs/containers may need to be migrated between the cloud and the fog to accommodate application requirements that change over time. Besides, VM/container migration may be used for load balancing purposes among cloudlets. Finally, VM/container migration is the most used approach in literature to support device mobility, which is a typical issue of fog computing networks. We implemented *MobFogSim* with the objective to fill this gap in literature. At the moment of writing, MobFogSim is the only simulator that models VM/container migration in fog computing environments. As a result, it represents a useful tool to test VM/container migration solutions when setting up a real testbed is too cost- and time-consuming. MobFogSim is implemented as an extension of iFogSim and thus inherits its wide range of functionalities, enriching them with VM/container migration and mobility modelling.

8. Conclusions

The need to support VM/container migration in fog computing (particularly for mobile users) has been outlined in this paper. Migration can take account of geographical location of cloudlets with reference to the user, the direction and speed of travel of the user, and the network characteristics between the user and the cloudlet. The migration process has been described as a set of events (identified in a number of different scenarios), taking account of both migration and handoff strategies. Container migration using both cold and post-copy mechanisms has been outlined and subsequently simulated using the *MobFogSim* toolkit – an extension to the iFogSim simulator. A lab-based testbed has also been developed (using Raspberry Pi nodes) and used to seed (initialise) parameters for the simulation environment and also to check the validity of the generated results from simulation. MobFogSim is able to take account of user's mobility, wireless connectivity, and the VM/container migration process (as outlined in the scenarios mentioned above). The simulator is able to utilise a user-defined migration strategy. Our results demonstrate that MobFogSim provides a useful basis for supporting fog computing for applications where users are mobile and where a migration strategy is needed to move state/data across cloudlets.

Acknowledgments

The authors would like to thank the following agencies for partially supporting this research: Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq); the São Paulo Research Foundation (FAPESP), grant #2015/16332-8; the INCT of the Future Internet for Smart Cities funded byCNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de NÃvel Superior - Brasil (CAPES) - Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9; the Italian Ministry of Education and Research (MIUR) in the framework of the Crosslab project (Departments of Excellence).

References

[1] M. Aazam, E.-N. Huh, M. St-Hilaire, C.-H. Lung, I. Lambadaris, Cloud of Things: Integration of IoT With Cloud Computing, Springer International Publishing,

[15] See https://github.com/Cloudslab/cloudsim. Last accessed: 18 July 2019.

2016, pp. 77–94.
[2] N. Fernando, S.W. Loke, W. Rahayu, Mobile cloud computing: A Survey, Future Generat. Comput. Syst. 29 (1) (2013) 84–106, https://doi.org/10.1016/j.future.2012.05.023.
[3] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog Computing and Its Role in the Internet of Things, Proceedings of the 1st MCC Workshop on Mobile Cloud Computing, (2012), pp. 13–16, https://doi.org/10.1145/2342509.2342513.
[4] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, O. Rana, Fog computing for the internet of things: a survey, ACM Trans. Int. Technol. 19 (2) (2019), https://doi.org/10.1145/3301443.
[5] L.F. Bittencourt, J. Diaz-Montes, R. Buyya, O.F. Rana, M. Parashar, Mobility-Aware application scheduling in fog computing, IEEE Cloud Comput. 4 (2) (2017) 26–35, https://doi.org/10.1109/MCC.2017.27.
[6] M.M. Lopes, W.A. Higashino, M.A.M. Capretz, L.F. Bittencourt, MyiFogSim: A simulator for virtual machine migration in fog computing, Proceedings of the 6th International Workshop on Clouds and (eScience) Applications Management (CloudAM). Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, (2017), pp. 47–52, https://doi.org/10.1145/3147234.3148101.
[7] C. Puliafito, E. Mingozzi, G. Anastasi, Fog Computing for the Internet of Mobile Things: Issues and Challenges, Proceedings of the IEEE 3rd International Conference on Smart Computing (SMARTCOMP), (2017), pp. 1–6, https://doi.org/10.1109/SMARTCOMP.2017.7947010.
[8] D. Gonçalves, K. Velasquez, M. Curado, L. Bittencourt, E. Madeira, Proactive Virtual Machine Migration in Fog Environments, Proceedings of the IEEE Symposium on Computers and Communications (ISCC), (2018), pp. 742–745, https://doi.org/10.1109/ISCC.2018.8538655.
[9] M. Behrisch, L. Bieker, J. Erdmann, D. Krajzewicz, SUMO - Simulation of Urban Mobility: An Overview, Proceedings of the 3rd International Conference on Advances in System Simulation (SIMUL), (2011).
[10] H. Gupta, A. Vahid Dastjerdi, S.K. Ghosh, R. Buyya, iFogSim: A Toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments, Software: Pract. Exper. 47 (9) (2017) 1275–1296, https://doi.org/10.1002/spe.2509.
[11] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, R. Buyya, Cloudsim: A Toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. Practice Exper. 41 (1) (2011) 23–50, https://doi.org/10.1002/spe.995.
[12] L.F. Bittencourt, M.M. Lopes, I. Petri, O.F. Rana, Towards virtual machine migration in fog computing, Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), (2015), pp. 1–8, https://doi.org/10.1109/3PGCIC.2015.85.
[13] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, A. Puliafito, Container migration in the fog: A Performance evaluation, MDPI Sensors 19 (7) (2019), https://doi.org/10.3390/s19071488.
[14] L. Codeca, R. Frank, T. Engel, Luxembourg SUMO Traffic (LuST) Scenario: 24 Hours of Mobility for Vehicular Networking Research, Proceedings of the IEEE Vehicular Networking Conference (VNC), (2015), pp. 1–8, https://doi.org/10.1109/VNC.2015.7385539.
[15] R. Longbottom, Roy Longbottom's PC Benchmark Collection - Linux MultiThreading Benchmarks, 2010, Last accessed: 5 April 2019., http://www.roylongbottom.org.uk/linux%20multithreading%20benchmarks.htm.
[16] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-Threading technology architecture and microarchitecture, Intel Technol. J. 6 (1) (2002). Last accessed: 5 April 2019.
[17] R. Longbottom, Roy Longbottom's Raspberry Pi, Pi 2 and Pi 3 Benchmarks, 2013, Last accessed: 8 April 2019., http://www.roylongbottom.org.uk/Raspberry%20Pi%20Benchmarks.htm.
[18] R. York, Benchmarking in Context: Dhrystone, 2002, ARM White paper. Last accessed: 29 April 2019., http://dell.docjava.com/courses/cr346/data/papers/DhrystoneMIPS-CriticismbyARM.pdf.
[19] R. Longbottom, MP Dhrystone Benchmarks - MP-DHRY, MP-DHRYPiA7, 2013, Last accessed: 8 April 2019., http://www.roylongbottom.org.uk/Raspberry%20Pi%20Multithreading%20Benchmarks.htm#anchor6.
[20] C. Bormann, CoAP - RFC 7252 Constrained Application Protocol, 2014, Last accessed: 24 April 2019., https://coap.technology/.
[21] M. Scarpiniti, E. Baccarelli, A. Momenzadeh, Virtfogsim: A Parallel toolbox for dynamic energy-Delay performance testing and optimization of 5G mobile-Fog-Cloud virtualized platforms, MDPI Appl. Sci. 9 (6) (2019), https://doi.org/10.3390/app9061160.
[22] I. Lera, C. Guerrero, C. Juiz, YAFS: A Simulator For IoT scenarios in fog computing, IEEE Access 7 (2019) 91745–91758, https://doi.org/10.1109/ACCESS.2019.2927895.
[23] T. Qayyum, A.W. Malik, M.A.K. Khattak, O. Khalid, S.U. Khan, FogNetSim++: A Toolkit for modelling and simulation of distributed fog environment, IEEE Access 6 (2018) 63570–63583, https://doi.org/10.1109/ACCESS.2018.2877696.
[24] A. Brogi, S. Forti, QoS-Aware Deployment of IoT applications through the fog, IEEE Int. Things J. 4 (5) (2017) 1185–1192, https://doi.org/10.1109/JIOT.2017.2701408.
[25] A. Brogi, S. Forti, A. Ibrahim, How to Best Deploy Your Fog Applications, Probably, Proceedings of the IEEE 1st International Conference on Fog and Edge Computing (ICFEC), (2017), pp. 105–114, https://doi.org/10.1109/ICFEC.2017.8.
[26] C. Sonmez, A. Ozgovde, C. Ersoy, EdgeCloudSim: An environment for performance evaluation of edge computing systems, Trans. Emerging Telecommun. Technol. 29 (11) (2018), https://doi.org/10.1002/ett.3493.