Preproceedings of the Federated Conference on
Computer Science and Information Systems pp. 37–56

A Distributed Application Placement and Migration Management Techniques for Edge and Fog Computing Environments

Mohammad Goudarzi*, Marimuthu Palaniswami†, Rajkumar Buyya*,
*The Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: mgoudarzi@student.unimelb.edu.au, rbuyya@unimelb.edu.au
† The Department of Electrical and Electronic Engineering,
The University of Melbourne, Australia
Email: palani@unimelb.edu.au

***Abstract*—Fog/Edge computing model allows harnessing of resources in the proximity of the Internet of Things (IoT) devices to support various types of latency-sensitive IoT applications. However, due to the mobility of users and a wide range of IoT applications with different resource requirements, it is a challenging issue to satisfy these applications' requirements. The execution of IoT applications exclusively on one fog/edge server may not be always feasible due to limited resources, while the execution of IoT applications on different servers requires further collaboration and management among servers. Moreover, considering user mobility, some modules of each IoT application may require migration to other servers for execution, leading to service interruption and extra execution costs. In this article, we propose a new weighted cost model for hierarchical fog computing environments, in terms of the response time of IoT applications and energy consumption of IoT devices, to minimize the cost of running IoT applications and potential migrations. Besides, a distributed clustering technique is proposed to enable the collaborative execution of tasks, emitted from application modules, among servers. Also, we propose an application placement technique to minimize the overall cost of executing IoT applications on multiple servers in a distributed manner. Furthermore, a distributed migration management technique is proposed for the potential migration of applications' modules to other remote servers as the users move along their path. Besides, failure recovery methods are embedded in the clustering, application placement, and migration management techniques to recover from unpredicted failures. The performance results demonstrate that our technique significantly improves its counterparts in terms of placement deployment time, average execution cost of tasks, the total number of migrations, the total number of interrupted tasks, and cumulative migration cost.**

I. Introduction

THE NUMBER of latency-sensitive applications of Internet of Things (IoT) devices has been increasing due to recent advances in technologies so that many applications rely on remote resources for their execution. Due to latency-sensitive nature of these applications and the huge amount of data that they generate, traditional cloud computing cannot efficiently satisfy the requirements of IoT applications, and they experience high latency and energy consumption while communicating to cloud servers (CSs) [1], [2]. The fog/edge computing paradigm addresses these issues by providing an intermediate layer of distributed resources between IoT devices and CSs that can be accessed with lower latency [3], [4], [5]. However, the provided resources of fog/edge servers for IoT applications are limited and with less variety in comparison to the resources of CSs [6]. In our view, fog computing has a hierarchical and distributed structure that harnesses the resources of both CSs and Fog Servers (FSs) at different hierarchical fog levels, while lower-level FSs have fewer computing resources compared to higher-level FSs, but they are accessible with lower latency [1], [7], [8], [9], [10]. However, edge computing does not have this hierarchical structure and does not use resources of CSs [11] (although some works use these terms interchangeably).

Real-time IoT applications can be modeled as a set of lightweight and interdependent application modules in fog computing environments so that such application modules alongside their allocated resources form the data processing elements of various IoT applications [7], [1]. Considering different requirements of applications' modules, they can be placed on one FS, different FSs in the same hierarchical level, FSs in different hierarchical levels, and/or CSs for the execution [1], [12]. Besides, as the number of IoT applications increases, more requests are forwarded to FSs that may overload them. Hence, a dynamic application placement technique is required to efficiently place interdependent modules of IoT applications on remote servers while meeting their requirements.

Alongside the importance of suitable application placement techniques, there are yet several issues to be addressed. The coverage ranges of lower-level FSs are limited, and IoT users have different mobility patterns. Besides, interdependent modules of each IoT application may be deployed on several FSs. Hence, as the IoT user moves towards its destination, the application response time and IoT device energy consumption can be negatively affected [13]. Therefore, the migration of interdependent modules of each application among FSs, which incurs service interruption and additional cost, is an important and yet a challenging issue. Several migration techniques

decide when, how, and where application modules can migrate when IoT users change their location in the fog/edge computing environments, such as [14], [15], [16], [17]. However, these techniques either focus on the migration of a single application module without considering other deployed modules [13] or consider an IoT application as a set of independent application modules. An IoT application may consist of several interdependent modules, and the migration technique should consider the configuration of all interdependent modules when an IoT user moves towards its destination. Hence, the migration of IoT applications, consisting of several interdependent modules, is an important challenge to be addressed, especially in hierarchical fog computing environments in which modules may be placed on different hierarchical levels.

Also, in fog computing, there are several studies that consider the application placement and migration management engines (i.e., decision engines) have a global view about topology and resources of all FSs and CSs [6], [18] while there are other studies that assume decision engines only have a local view about resources and topology of servers in their proximity [10], [9], [19]. In these latter techniques, the decision engines act in the distributed manner so that each FS that receives the application placement and/or migration request try to use the available resources in its proximity (which can be accessed with lower latency) to place/migrate the application modules as much as possible. However, if there are no available resources, the rest of the placement and migration will be handled by higher-level FSs in the hierarchy. Considering communication with higher-level FSs incurs higher latency compared to communication among FSs at the same hierarchical level, the clustering of FSs (if it is possible) at the same hierarchical level can provide sufficient resources (with less latency in comparison to higher-level FSs) to serve real-time IoT applications and reduce the amount of communication with higher-level FSs.

In this paper, we address these issues and propose efficient distributed application placement and migration management techniques to satisfy the requirements of real-time IoT applications while users move.

The main contributions of this paper are as follows.

- We propose a new weighted cost model based on IoT applications' response time and IoT devices' energy consumption for application placement and migration of IoT devices in hierarchical fog/edge computing environments to minimize cost of running real-time IoT applications.
- We put forward a dynamic and distributed clustering technique to form clusters of FSs at the same hierarchical levels so that such servers can collaboratively handle IoT application requirements with less execution cost.
- Considering the NP-Complete nature of application placement and migration problems in fog/edge computing environments, we propose a distributed application placement and migration management techniques to place/migrate modules of real-time applications on different levels of hierarchical architecture based on their requirements.
- We embed failure recovery methods in clustering, application placement, and migration management techniques to recover from unpredicted failures.

The rest of paper is organized as follows. Relevant works of application placement and migration management techniques in edge and fog computing environments are discussed in section II. The system model and problem formulations are presented in section III. Section IV presents our proposed distributed clustering, application placement, and migration management technique. We evaluate the performance of our technique and compare it with the state-of-the-art techniques in section V. Finally, section VI concludes the paper and draws future works.

II. Related Work

In this section, related works that address both application placement and mobility issues at the same time as their main challenges in the context of edge/fog computing are studied. These works are categorized into independent and dependent categories based on the dependency mode of their applications' granularity (e.g., modules). In the dependent category, constituent parts of IoT applications (i.e., modules) can be executed only when their predecessor modules complete their execution, while IoT applications that are modeled as a set of independent modules do not have this constraint.

A. Edge Computing

In the independent category, Wang et al. [20] formulated service migration as a distance-based Markov Decision Process (MDP), which considers the distance between an IoT user and service provider as its main parameter. Then, they proposed a numerical technique to minimize the migration cost of users. Wang et al. [21] and Yang et al [22] considered deterministic mobility conditions, in which the potential paths between source and destination are priori known, and proposed placement techniques, performed on the IoT device, to minimize the delay. Since paths and available edge devices are priori known, as the IoT user moves, the current in-contact edge device can send the required information to the next edge device. Ouyang et al. [17] proposed an edge-centric application placement and mobility management technique that are executed on the network operator and one-hop edge devices respectively. They proposed a distributed approximation scheme based on the best response update technique to optimize the mobile edge service performance. Liu et al. [23] proposed a mobility-aware offloading and migration technique to maximize the total revenue of IoT devices by reducing the probability of migration. Zhu et al. [24] proposed a mobility-aware application placement in vehicular scenarios with constraints on service latency and quality loss. In this technique, some of the vehicles generate tasks while other vehicles provide computing services as remote servers. Zhang et al. [25] proposed a deep reinforcement technique to minimize the delay of IoT tasks. Yu et al. [26] proposed a technique to minimize the delay of tasks while satisfying the energy consumption of a single IoT user moving among edge servers.

In the dependent category, Sun et al. [27] and Qi et al. [28] proposed a mobility-aware application placement technique in which placement decision engines run on IoT devices. The authors of [27] considered a single IoT device and proposed an IoT-centric energy-aware mobility management technique to minimize the application delay while authors of [28] proposed an edge-centric and knowledge-driven online learning method to adapt to the environment changes as vehicles move.

B. Fog Computing

In the independent category, Wang et al.[16] proposed a solution to place a single service instance of each IoT user on a remote server when multiple IoT users exist in the system. They proposed both offline and online approximation algorithms, performed on the cloud, to find the optimal and near-optimal solutions respectively. Wang et al. [29] and Wang et al. [13] proposed edge-centric application placement and mobility management technique when multiple IoT users with a single module exist in the system. The main goal of [29] is Maximizing IoT users' gain through offloading and reducing the number of migrations, while the main goal of authors of [13] is minimizing the service delay.

In the dependent category, Shekhar et al. [6] and Bittencourt et al. [19] proposed mobility-aware application placement techniques for IoT application, consisting of multiple inter-dependent modules while considering prior mobility information. The authors in [6] proposed a cloud-centric technique, called URMILA, in which the centralized controller makes the placement decision for all IoT applications to satisfy their latency requirements. Besides, whenever the decision is made, even in case the user leaves the range of its immediate server, there is no migration algorithm to migrate modules to new servers, which incurs a higher cost for the users. The authors in [19] proposed an edge-centric solution based on the edgeward-placement technique [10] for placement of IoT applications while considering their targeted destination. In this proposal, however, the potential of clustering is not considered. So, whenever the immediate server cannot serve the application modules, the modules are forwarded to the next hierarchical layer for possible placement and migration.

C. A Qualitative Comparison

Key elements of related works are identified and presented in Table I and compared with ours in terms of the main category, IoT application, architectural, and placement and mobility management engines' properties. The IoT application properties identify and compare dependency mode (either independent or dependent) of IoT applications, modules' number (either single or multiple modules per application), and heterogeneity (whether the specification of modules is same (i.e., homogeneous) or different (i.e., heterogeneous)). Architectural properties contain the number of IoT devices (either single or multiple), whether hierarchical fog architecture is considered or not, and clustering technique (whether a clustering technique is applied on edge/fog servers or not). Placement and mobility management engines contain positions of placement, mobility management engines, failure recovery capability, and the decision parameters used in each proposal.

Our work proposes an edge-centric application placement and mobility management technique for an environment consisting of multiple IoT devices with heterogeneous applications (consisting of several dependent modules with heterogeneous requirements) and multiple remote servers (either CSs or FSs) deployed in a hierarchical architecture. Considering the potential of the clustering of FSs in the hierarchical fog computing environment, we propose a weighted cost model of response time and energy consumption for the application placement and migration techniques. The proposed weighted cost model considers the dependency among modules of IoT applications which plays an important role in application placement and migration management. Second, we put forward a distributed and dynamic clustering technique by which FSs of the same hierarchical level can form a cluster and collaboratively provide faster and more efficient service for IoT applications. This latter is because the communication overhead between FSs of the same hierarchical level is usually less than communication with higher-level FSs [1]. Although resources of each lower-level FS is less than each higher-level FS, aggregated resources of lower-level FSs, obtained through clustering, can be used to manage IoT applications modules in lower-level FSs with less response time and energy consumption. Third, we propose a distributed application placement and migration techniques for hierarchical fog computing environments to minimize the weighted cost of running real-time IoT applications. Finally, due to the highly dynamic nature of such systems, there is a high chance of failures in the system, for which we propose light-weight failure recovery methods in the clustering, application placement, and migration management techniques.

III. System Model and Problem Formulation

We consider a system consisting of N mobile IoT users (so that each user has one IoT device), F heterogeneous FSs distributed in the proximity of IoT users, and a centralized cloud. FSs follow a hierarchical topology, in which lower-level FSs can be accessed with lower latency while providing fewer resources in comparison to higher-level FSs that provide more resources but can be accessed with higher latency [1], [9]. Besides, we assume that each IoT device is connected to one FS in the lowest hierarchical level, so that this FS is responsible for the application placement and mobility management of that IoT device. The set of all available servers is represented as $\mathcal{S}$ with $|\mathcal{S}| = M$ and $M > F$. The 2-tuple $(h, i) \in \mathcal{S}$ $(0 \leq h, 1 \leq i)$ represents one server, in which h represents the hierarchical level of the server and i denotes the server's index at that hierarchical level. If we assume there are L hierarchical fog layers, $(L+1, 1)$ demonstrates the centralized cloud data-center placed at the top-most level. Moreover, the $(0, n)$ denotes the nth IoT device. Fig. 1 represents a view of our system model and how IoT devices move among different FSs. Also, it shows the in-cluster communications (in case clustering is applied) and communications between FSs at different hierarchical levels in this environment.

Table I: A qualitative comparison of related works with ours

Techniques	Category	Application Properties			Architectural Properties			Placement and Mobility Management Engines					
											Decision Parameters		
		Dependency	Module Number	Heterogeneity	Number of IoT Devices	Hierarchical Fog Architecture	Clustering Technique	Placement Engine Position	Mobility Management Engine Position	Failure Recovery	Time	Energy	Weighted
[20]	Edge Computing	Independent	Single	Heterogeneous	Single	×	×	Edge Centric	Edge Centric	×	✓	×	×
[21]			Multiple	Heterogeneous	Multiple	×	×	IoT Device Centric	Edge Centric	×	✓	×	×
[17]			Single	Heterogeneous	Multiple	×	×	Edge Centric	Edge Centric	×	✓	×	×
[23]			Single	Heterogeneous	Multiple	×	×	Edge Centric	Edge Centric	×	✓	✓	✓
[22]			Multiple	Heterogeneous	Multiple	×	×	IoT Device Centric	Edge Centric	×	✓	×	×
[24]			Multiple	Heterogeneous	Single	×	×	Edge Centric	Edge Centric	×	✓	×	×
[25]			Single	Homogeneous	Single	×	×	Edge Centric	Edge Centric	×	✓	×	×
[26]			Multiple	Heterogeneous	Single	×	×	Edge Centric	Edge Centric	×	✓	✓	×
[27]		Dependent	Multiple	Heterogeneous	Single	×	×	IoT Device Centric	IoT Device Centric	×	✓	✓	×
[28]			Multiple	Heterogeneous	Multiple	×	×	IoT Device Centric	Edge Centric	×	✓	×	×
[16]	Fog Computing	Independent	Single	Heterogeneous	Multiple	×	×	Cloud Centric	Cloud/Edge Centric	×	✓	×	×
[29]			Single	Heterogeneous	Multiple	×	×	Edge Centric	Edge Centric	×	✓	✓	✓
[13]			Single	Heterogeneous	Multiple	×	×	Edge Centric	Edge Centric	×	✓	×	×
[6]		Dependent	Multiple	Homogeneous	Single	×	×	Cloud Centric	Cloud Centric	×	✓	×	×
[19]			Multiple	Heterogeneous	Multiple	✓	×	Edge Centric	Edge Centric	×	✓	×	×
Proposed Solution			Multiple	Heterogeneous	Multiple	✓	✓	Edge Centric	Edge Centric	✓	✓	✓	✓

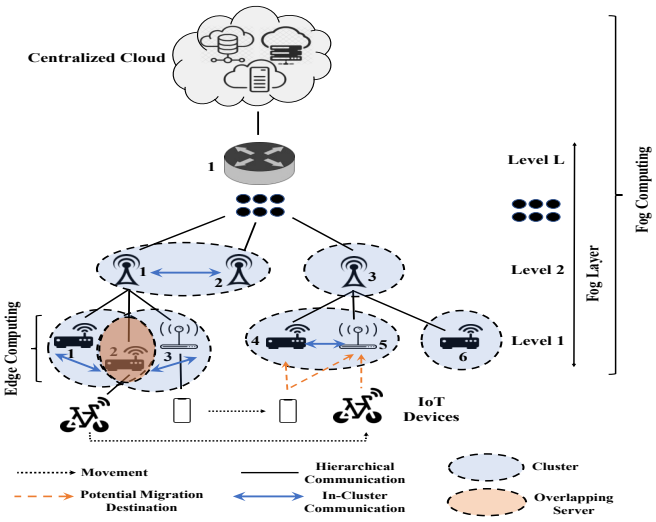


Figure 1: A view of our system model

Each FS can form a cluster either by other nearby FSs at the same hierarchical level or by itself. Moreover, each FS in lth hierarchical level may belong to different clusters in that hierarchical layer. The cluster member (CM) list of each FS is defined as $List_{cl}(h,i)$, which is empty if the FS (h,i) does not have any CMs. Besides, for each FS, we define a children list, $List_{ch}(h,i)$, containing server specification of immediate lower-level FSs, to which it has direct hierarchical communication links. The sole parent server of each FS is defined as $par(h,i) = (h',i')$ which refers to the immediate higher-level FS. We assume that in-cluster communications are faster than hierarchical communications [1]. Hence, clustering FSs, while incurs additional cost due to running clustering algorithm, can improve the quality of service for IoT users. Moreover, each FS has a list, called $\Omega(h,i)$, containing server specification of itself, its children, and all FSs belonging to the Ω of its children. To illustrate, considering Fig. 1, the $\Omega(2,1) = \{(2,1),(1,1),(1,2),(1,3)\}$ and $List_{ch}(2,1) = \{(1,1),(1,2),(1,3)\}$, and $\Omega(2,2) = \{(2,2)\}$ and $List_{ch}(2,2) = \{\}$. If we assume the maximum number of fog layers is three (i.e., $L = 3$) in this example, then $\Omega(3,1) = \{(3,1),(2,1),(2,2),(2,3),(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)\}$, and the $List_{ch}(3,1) = \{(2,1),(2,2),(2,3)\}$.

We consider that FSs and CSs use container technology to run IoT applications' modules [13], [30]. So, we assume that FSs have access to images of all containers ($Cnts$) while such $Cnts$ may be active if they are running on the server or inactive (i.e., the container images are accessible, but the containers are not running) otherwise [13]. Moreover, for each container, according to the application module that it serves, an amount of ram size at the runtime is assigned to keep the state $Cnt^{ram}_{v_{n,j}}$ [31]. Table II summarizes the parameters used in this paper and their respective definitions.

A. Application Model

We consider real-time IoT applications working based on the Sense-Process-Actuate model, in which sensors transmit tasks periodically according to their sample rate [1], [10]. The emitted sensors' tasks should be forwarded to different modules of the IoT applications for processing based on dependency model among constituent modules. When each module receives tasks from predecessor modules as input, it processes tasks and produces respective tasks as its output to be forwarded to next modules [1], [32]. Finally, results

Table II: Parameters and respective definitions

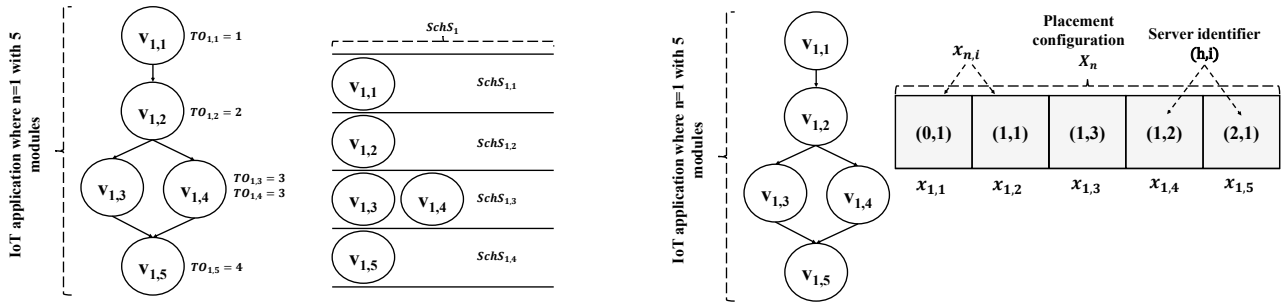
Parameter	Definition	Parameter	Definition
CSs	Cloud Servers	FSs, FS	Fog Servers, Fog Server
$CNTs$, CNT	Containers, Container	N	Number of mobile IoT devices
F	Number of heterogeneous fog servers (FSs)	$\mathcal{S}$	The set of all available servers
M	Number of available servers	(h,i)	The 2-tuple showing one server in which h represents the hierarchical level of the server and i denotes the server's index at that hierarchical level
$List_{ch}(h,i)$	The list containing server specification of children for the server (h,i)	$par(h,i)$	The sole parent of the server (h,i) in the hierarchical system
$\Omega(h,i)$	The set containing server specification of server (h,i), its children, and all FSs belonging to the Ω of its children	CM	Cluster Member
$List_{cl}(h,i)$	The list containing server specification of cluster members for the server (h,i)	G_n	Directed Acyclic Graph (DAG) of the nth IoT application
$\mathcal{V}_n$	The set of modules belonging to the nth IoT application	$\mathcal{E}_n$	The set of data flows between modules belonging to the nth IoT application
$v_{n,i}$, $v_{n,j}$	The ith and/or jth module belonging to the nth IoT application	$e_{n,i,j}$	The data flow from module $v_{n,i}$ to module $v_{n,j}$ of the nth IoT device
$\mathcal{P}(v_{n,j})$	The set of predecessor modules of the module $v_{n,j}$	$TO_{n,i}=t$	The topological order of ith module of the nth IoT application is equalt to t
$SchS_n$	The schedule set of the nth IoT application consisting of subsets of modules with the the same TO value t	$SchSn,t$	A subset of $SchS_n$ showing modules with the same TO value t (i.e., modules that can be executed in parallel)
$e^{ins}_{n,i,j}$	The amount of instructions in terms of Million Instruction that the module $v_{n,j}$ receives from $v_{n,i}$ for processing	$e^{dsize}_{n,i,j}$	The size of data that the module $v_{n,i}$ generates as an output to be sent to module $v_{n,j}$
$v^{mtd}_{n,i}$	The maximum tolerable delay for the module $v_{n,i}$	X_n	The placement configuration of the nth IoT application
$x_{n,i}$	The placement configuration for each module $v_{n,i}$ of the nth IoT application in the X_n	$\Psi(X_n,t)$	The weighted cost of modules in the tth schedule while considering the placement configuration X_n.
$\lvert SchS_n\rvert$	The number of schedules for the nth IoT application	$T_{x_{n,j}}$	The overall delay of each module (i.e., $v_{n,j}$) based on its assigned server
$Cnts_{(h,i)}$	The number of instantiated $Cnts$ on the server (h,i)	$Cap_{(h,i)}$	The maximum capacity of server (h,i) to instantiate $Cnts$.
$\Gamma(X_n,t)$	The weighted cost of modules in the tth schedule while considering the placement configuration X_n	$\Theta(X_n,t)$	The energy consumption of modules in the tth schedule while considering the placement configuration X_n
$T^{lat}_{x_{n,j}}$	The inter-nodal latency between the servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ are placed	$T^{exe}_{x_{n,j}}$	The computing execution time of tasks, emitted from the $v_{n,i}$ to be executed on the $v_{n,j}$
$T^{tra}_{x_{n,j}}$	The transmission time between between the module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$	$cpu(x_{n,j})$	The computing power of the assigned server (in terms of MIPS) for the module $v_{n,j}$
γ^{tra}	The transmission time between source and destination servers	B_{up}, B_{down}, $B_{cluster}$	The bandwidth of the one server to the parent server, to the child server, and to its CMs, respectively
$NST_i(H), NSE_i(H)$	They define the next intermediate server to reach the destination server	$chRule$	It identifies whether any children of the current server has a route to the destination server or not
$chRule$	It identifies whether any CMs of the current server has a route to the destination server or not	$\Upsilon((\Omega(h,i)),(h',i'))$	It shows whether $\Omega(h,i)$ contains (h',i') or not (i.e., meaning that there is one hierarchical path from (h,i) to the (h',i'))
γ^{lat}	The inter-nodal latency between source and destination servers	$lat(up)$, $lat(down)$, $lat(cluster)$	The inter-nodal latency of one server to the parent server, to the child server, and to its CMs, respectively
$\Psi^{mig}((X_n,X'_n,ts)$	The weighted migration cost of nth IoT application from the current configuration X_n to the new configuration X'_n	$\gamma^{mig}(x_{n,i},x'_{n,i})$	The migration cost of one module from current configuration $x_{n,i}$ to the new configuration $x'_{n,i}$
$\gamma^{lat}_{mig}((h,i),(h',i'))$	The migration latency between current and new servers	$dsize^{mig}$	The size of dump data and states that should be transferred between current and new servers
$e^{ins,r}_{n,i,j}$	The amount of remaining instructions of task $e^{ins,r}_{n,i,j}$ to be executed on the new server after migration	$E(x_{n,j})$	The overall energy consumption of each module (i.e., $v_{n,j}$) based on its assigned server
$E^{exe}_{x_{n,j}}$	The computing energy consumption of tasks, emitted from the $v_{n,i}$to be executed on the $v_{n,j}$	$E^{lat}_{x_{n,j}}$	The energy consumption incurred due to inter-nodal latency between the servers on which module $v_{n,j}$and its predecessors $\mathcal{P}(v_{n,j})$ are placed
$E^{tra}_{x_{n,j}}$	The transmission energy consumption between between the module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$	P_{cpu}, P_i, P_t	The CPU power of the IoT device, the idle power of IoT device, and transmission power of the IoT device
ϑ^{tra}	The transmission energy consumption between source and destination servers	ϑ^{lat}	The energy consumption incurred due to inter-nodal latency between servers
$\Gamma^{mig}((X_n,X'_n),t)$	The migration time of nth IoT application from the current configuration X_n to the new configuration X'_n considering schedule t	$\Theta^{mig}((X_n,X'_n),t)$	The migration energy consumption of nth IoT application from the current configuration X_n to the new configuration X'_n considering schedule t

will be forwarded to the actuator as the last module. In this work, we assume that both sensor and actuator modules of IoT applications reside in IoT devices [19].

Real-time IoT application belonging to the nth IoT device is represented as a Directed Acyclic Graph (DAG) of its modules $G_n=(\mathcal{V}_n,\mathcal{E}_n), \forall n\in\{1,2,\cdots,N\}$, where $\mathcal{V}_n=\{v_{n,i}|1\leq i\leq|\mathcal{V}_n|\}$ denotes the set of modules belonging to the nth IoT device, and $\mathcal{E}_n=\{e_{n,i,j}|v_{n,i},v_{n,j}\in\mathcal{V}_n,\ v_{n,i}\in\mathcal{P}(v_{n,j}),\ i\neq j\}$ shows the set of data flows between modules. Since IoT applications are modeled as DAGs, each module $v_{n,j}$ cannot be executed unless all its predecessor modules, denoted as $\mathcal{P}(v_{n,j})$, finish their execution. To illustrate, $e_{1,1,2}$ represents that execution of module $v_{1,2}$ depends on the execution of the module $v_{1,1}$. Moreover, we define a Topological Order value t for each module i of the nth IoT application as $TO_{n,i}=t$. We define a schedule set for the nth IoT application, called $SchS_n$, consisting of modules with the same TO value t as its subsets. The $SchS_{n,t}$ specify modules with the same TO value t (i.e., modules that can be executed in parallel). In addition, the set of successor modules of module $v_{n,j}$ is defined as $Succ(v_{n,j})$. Fig 2a shows an IoT application, the TO value for each module, and the schedule set $SchS_n$ based on the TO values of its modules. Besides, We define the output of each module $v_{n,i}$ is a task consisting of two values to be forwarded to next modules based on data flows of the IoT application. The first value is the amount of instructions in terms of Million Instruction (MI) that the module $v_{n,j}$ receives from $v_{n,i}$ for processing, shown as $e^{ins}_{n,i,j}$, and the second value is the size of data $e^{dsize}_{n,i,j}$ the module $v_{n,i}$ generates as its output to be forwarded to module $v_{n,j}$ [10].

B. Problem Formulation

The placement configuration of the application belonging to the nth IoT application is shown as X_n. Also, $x_{n,i}=(h,i)$ denotes the placement configuration for each module $v_{n,i}$ of



(a) An IoT application and its corresponding schedules

(b) A candidate server configuration for the IoT application

Figure 2: An example of IoT application, its schedules and a candidate server configuration

the nth IoT application in the X_n based on the specification of the server. To illustrate, $x_{n,i} = (1, 3)$ shows that the ith module of nth IoT device is assigned to a server in the first hierarchical level where the server index is 3. Moreover, if the ith module of the nth IoT device is assigned to run locally on itself, $x_{n,i} = (0, n)$. Fig 2b presents a sample DAG of an IoT application and a candidate placement configuration.

1) Placement weighted cost model: The goal of application placement is to find a suitable configuration for modules of each real-time IoT application to minimize the weighted cost $\Psi(X_n, t)$ of running applications in terms of the response time of tasks and energy consumption of IoT devices:

$$\min_{w_1,w_2\in[0,1]} \sum_{t=1}^{|SchS_n|} \Psi(X_n, t), \quad \forall n \in \{1, 2, \cdots, N\} \tag{1}$$

where

$$\Psi(X_n, t) = w_1 \times \Gamma(X_n, t) + w_2 \times \Theta(X_n, t) \tag{2}$$

$$s.t. \quad C1: \ Size(x_{n,j}) = 1, \ \forall x_{n,j} \in X_n \ , \quad n \in \{1, 2, \cdots, N\}, 1 \le i \le |\mathcal{V}_n| \tag{3}$$

$$C2: \ Cnts(h, i) \le Cap(h, i), \ \forall \ (h, i) \in \mathcal{S} \tag{4}$$

$$C3: \ \Psi(x_{n,i}, t) \le \Psi(x_{n,j}, t), \ \forall v_{n,i} \in \mathcal{P}(v_{n,j}) \tag{5}$$

where $|SchS_n|$ represents the number of schedules, and $\Gamma(X_n, t)$ and $\Theta(X_n, t)$ show the response time model and energy consumption model, respectively, of modules in the tth schedule while considering the placement configuration X_n. Moreover, w_1 and w_2 are control parameters to tune the weighted cost model according to user requirements. We assume the number of available servers M is more than or equal to the maximum number of modules in the tth schedule for parallel execution (i.e., $|SchS_{n,t}| \le M$). We suppose that each module of an IoT application can be exactly assigned to one Cnt of one remote server. $C1$ indicates that each module i of the nth IoT application can only be assigned to one server at a time, and hence the size of $x_{n,j}$ is equal to 1 [2], [33]. $C2$ denotes that the number of instantiated $Cnts$ on the server (h, i) is less or equal to the maximum capacity of that server $Cap(h, i)$. Besides, $C3$ guarantees that the predecessor modules of $v_{n,j}$ (i.e., $\mathcal{P}(v_{n,j})$) are executed before the execution of module $v_{n,j}$ [33].

*a) **Response time model:*** The goal of this model is to find the best possible configuration of servers for each IoT application so that the overall response time for each IoT application becomes minimized. In order to only consider response time model as the main objective, the control parameters of weighted cost model (Eq. 2) can be set to $w_1 = 1$ and $w_2 = 0$.

$$\Gamma(X_n, t) = \begin{cases} T(x_{n,j}), & \text{if } |SchS_{n,t}| = 1 \quad \text{(a)} \\ max(T(x_{n,j})), & \text{otherwise} \\ \forall x_{n,j} \in X_n | v_{n,j} \in SchS_{n,t} & \text{(b)} \end{cases} \tag{6}$$

The Eq. 6.a represents the condition in which the number of modules in the tth schedule is one (i.e, $|SchS_{n,t}| = 1$), and hence, the time of that schedule is equal to the time of that module based on its assigned server $T(x_{n,j})$. Besides, the Eq. 6.b refers to the condition in which the number of modules in the tth schedule is more than one (i.e., several modules can be executed in parallel). In this latter case, the time of the tth schedule is equal to the maximum time of all modules that can be executed in parallel.

The overall delay of each module (i.e., $v_{n,j}$) based on its candidate configuration (i.e., $x_{n,j}$) is defined as the sum of inter-nodal latency between servers ($T^{lat}_{x_{n,j}}$), the computing time per module ($T^{exe}_{x_{n,j}}$), and the data transmission time between $v_{n,j}$ and all of its predecessor modules ($T^{tra}_{x_{n,j}}$). It is formulated as:

$$T(x_{n,j}) = T^{exe}_{x_{n,j}} + T^{lat}_{x_{n,j}} + T^{tra}_{x_{n,j}} \tag{7}$$

The computing execution time of module $v_{n,j}$ depends on tasks emitted from its predecessors (i.e., $\mathcal{P}(v_{n,j})$) for processing by $v_{n,j}$. The computing time of $v_{n,j}$ is estimated as:

$$T^{exe}_{x_{n,j}} = \sum \frac{e^{ins}_{n,i,j}}{cpu(x_{n,j})}, \quad \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \tag{8}$$

where $cpu(x_{n,j})$ demonstrates the computing power of the assigned server (in terms of Million Instruction per Second (MIPS)) for the module $v_{n,j}$. Moreover, the $e^{ins}_{n,i,j}$ shows the amount of instructions in terms of MI that the module $v_{n,j}$ receives from $v_{n,i}$ for the processing.

The transmission time between module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ of the application belonging to the nth IoT device is calculated as:

$$T^{tra}_{x_{n,j}} = \max(\gamma^{tra}(e^{dsize}_{n,i,j},(h,i),(h',i'))), \quad \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \quad x_{n,i}=(h,i), x_{n,j}=(h',i') \tag{9}$$

Due to the hierarchical nature of fog computing, the transmission time of one task (γ^{tra}) between each pair of dependent modules $v_{n,i}$ and $v_{n,j}$ is recursively obtained based on visited servers between source and destination. The (h,i) and (h',i') show server specifications of source and destination servers on which modules $v_{n,i}$ and $v_{n,j}$ are assigned, respectively. By visiting each intermediate server between source and destination servers, the value of source server (h,i) is updated while the value of destination server remains unchanged. To reduce the length of equations, we consider $(e^{dsize}_{n,i,j},(h,i),(h',i')) = H$.

$$\gamma^{tra}(H) = \begin{cases} \frac{e^{dsize}_{n,i,j}}{B_{up}} + \gamma^{tra}(H'), & NST_i(H) = NST_1 | NST_4 | NST_6 \\ \frac{e^{dsize}_{n,i,j}}{B_{down}} + \gamma^{tra}(H'), & NST_i(H) = NST_2 \\ \frac{e^{dsize}_{n,i,j}}{B_{cluster}} + \gamma^{tra}(H'), & NST_i(H) = NST_3 | NST_5 \\ 0, & NST_i(H) = NST_7 \end{cases} \tag{10}$$

where B_{up}, B_{down}, and $B_{cluster}$ refer to the bandwidth of current server to parent server, to child server, and to cluster server, respectively. Besides, H' is defined as what follows:

$$H' = (e^{dsize}_{n,i,j},(h^{''},i^{''}),(h',i')) \tag{11}$$

$$(h^{''},i^{''}) = NST_i(H) \tag{12}$$

The Eq. 11 shows the data size and destination server of H' is exactly the same as H, and the only difference is the specification of the source server $(h^{''},i^{''})$ which is obtained from the output of $NST_i(H)$ (i.e., $(h^{''},i^{''}) = NST_i(H)$. The $NST_i(H)$ defines the next intermediate server to reach the destination server for each edge $e_{n,i,j}$.

$$NST_i(H) = \begin{cases} Par(h,i), & \text{if } h < h' & i=1 \\ chRule, & \text{if } h > h' \ \& \ chRule \neq \varnothing & i=2 \\ clRule, & \text{if } h \oplus h' = 0 \ \& \ i \oplus i' \neq 0 \ \& \ clRule \neq \varnothing & i=3 \\ Par(h,i), & \text{if } h \oplus h' = 0 \ \& \ i \oplus i' \neq 0 \ \& \ clRule = \varnothing & i=4 \\ clRule, & \text{if } h > h', \ \& \ clRule \neq \varnothing & i=5 \\ Par(h,i), & \text{if } h > h' \ \& \ chRule = \varnothing \ \& \ clRule = \varnothing & i=6 \\ (0,0), & \text{if } h \oplus h' = 0 \ \& \ i \oplus i' = 0 & i=7 \end{cases} \tag{13}$$

$$chRule = \text{if } \exists (h^{''},i^{''}) \in List_{ch}(h,i) | \Upsilon((\Omega(h^{''},i^{''}),(h',i')) = 1, \text{return } (h^{''},i^{''}), \text{ else return } \varnothing \tag{14}$$

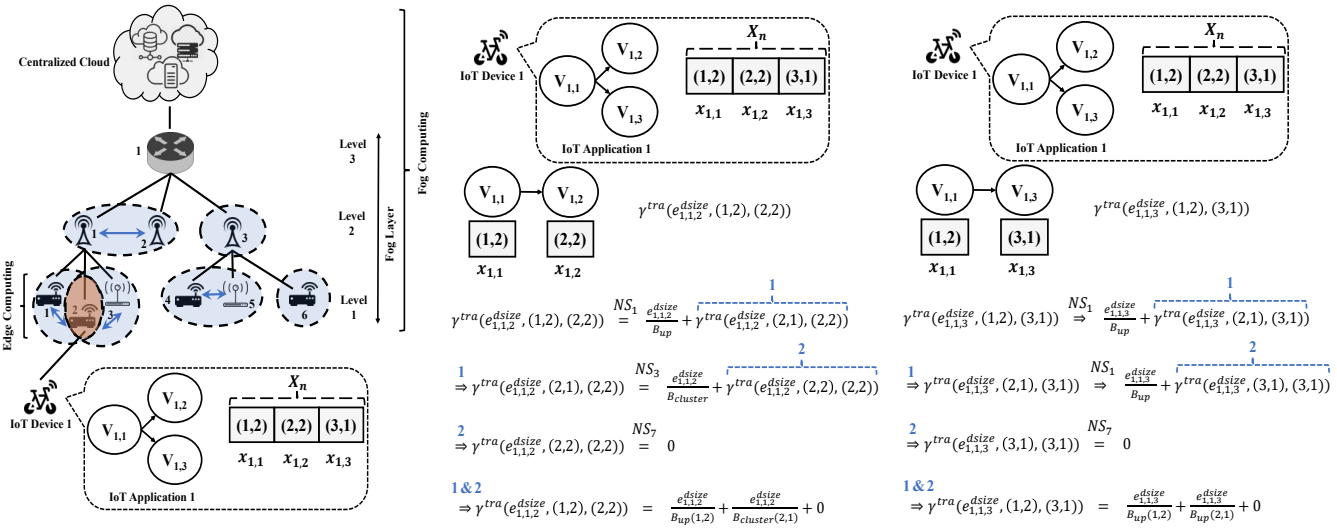
$$clRule = \text{if } \exists (h^{''},i^{''}) \in List_{cl}(h,i) | \Upsilon((\Omega(h^{''},i^{''}),(h',i')) = 1, \text{return } (h^{''},i^{''}), \text{ else return } \varnothing \tag{15}$$

The $\Upsilon((\Omega(h^{''},i^{''})),(h',i'))$ is equal to 1 if $\Omega(h^{''},i^{''})$ contains (h',i') (i.e., meaning that there is one hierarchical path from $(h^{''},i^{''})$ to the (h',i')) and is equal to 0 if (h',i') does not exist. Moreover, the $\oplus$ is XOR binary operation. The chRule (Eq. 14) says that if the server (h,i) has a children $(h^{''},i^{''})$ in its $List_{ch}$ which has a hierarchical path to the destination server $(h^{'},i^{'})$, the specification of this server $(h^{''},i^{''})$ should be returned. The clRule (Eq. 15) presents that if the server (h,i) has a CM $(h^{''},i^{''})$ in its $List_{cl}(h,i)$ which has a hierarchical path to the destination server $(h^{'},i^{'})$, the specification of this server $(h^{''},i^{''})$ should be returned. Based on the aforementioned rules, $NST(H)$ finds the next server to which the data should be sent and calculates the transmission cost. The NST_1 of Eq. 13 states that if the hierarchical level of the current server is less than destination server, the $Par(h,i)$ should be checked in the next step. The NST_2 represents the case that the hierarchical level of the current server is higher than the destination server and the current server has a child through which the destination server can be reached. The NST_3 states the condition that the current and destination servers are in the same hierarchical level, and one of the CMs has a route to the destination server. The NST_4 indicates that if the current and the destination servers are in the same level, and there is no route to destination using CMs, the parent should be checked in the next step. The NST_5 states that if the level of the current server is higher than the destination server, and a CM has a path to the destination server, the cluster server should be selected in the next step. The NST_6 states that if the level of current server is higher than the destination server, and there exists no route from children nor from CMs, the parent server should be traversed. Finally, the NST_7 is the ending condition for this recursive process and states that if the current and destination server is same, the cost is zero. Fig 3 represents an example of obtaining transmission time between source and destination servers.

Inter-nodal latency $T^{lat}_{x_{n,j}}$ between servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ are placed is calculated as:

$$\Gamma^{lat}_{X_{n,j}} = \max(\gamma^{lat}((h,i),(h',i'))), \quad \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \quad x_{n,i}=(h,i), x_{n,j}=(h',i') \tag{16}$$

where γ^{lat} shows the inter-nodal latency between source and destination servers (i.e., (h,i) and (h',i') respectively) on



(a) An IoT application and its candidate configuration

(b) Transmission time for the data flow $e_{1,1,2}$

(c) Transmission time for the data flow $e_{1,1,3}$

Figure 3: A example of calculating transmission time based on a candidate configuration

which $v_{n,i}$ and $v_{n,j}$ are placed. It is calculated similar to the transmission time. To reduce the equation size, we consider $((h,i),(h',i')) = A$.

$$\gamma^{lat}(A) = \begin{cases} lat_{up} + \gamma^{lat}(A'), & NST_i(A) = NST_1 | NST_4 | NST_6 \\ lat_{down} + \gamma^{lat}(A'), & NST_i(A) = NST_2 \\ lat_{cluster} + \gamma^{lat}(A'), & NST_i(A) = NST_3 | NST_5 \\ 0, & NST_i(A) = NST_7 \end{cases} \tag{17}$$

where lat_{up}, lat_{down}, and $lat_{cluster}$ correspond to up-link, down-link, and cluster-link inter-nodal latency respectively, and depends on the hierarchical level of servers. Besides, A' is defined as what follows:

$$A' = ((h^{''}, i^{''}), (h', i')) \tag{18}$$

The Eq. 18 shows the destination server (i.e., (h', i')) of A' is exactly the same as A, and the only difference is the specification of the source server $(h^{''}, i^{''})$ which is obtained from the output of $NST(A)$. The $NST(A)$ performs exactly the same as $NST(H)$ (i.e., Eq. 13) to find the next intermediate server, and all equation from Eq. 13 to Eq. 15 are valid here.

*b) **Energy consumption model**:* The goal of this model is to find a suitable placement configuration of application modules to minimize the energy consumption of the nth IoT device. To only consider energy consumption model as the main objective, the control parameters of weighted cost model (Eq. 2) can be set to $w_1 = 0$ and $w_2 = 1$.

$$\Theta(X_n, t) = \begin{cases} E(x_{n,j}), & \text{if } |SchS_{n,t}| = 1 \quad \text{(a)} \\ max(E(x_{n,j})), & \text{otherwise} \\ \forall x_{n,j} \in X_n | v_{n,j} \in SchS_{n,t} & \text{(b)} \end{cases} \tag{19}$$

where $|SchS_n|$ shows the number of schedules, and $\Theta(X_n, t)$ represents the energy consumption of modules in the tth schedule while considering the placement configuration X_n.

The overall energy consumption of each module (i.e., $v_{n,j}$) based on its candidate configuration (i.e., $x_{n,j}$) is defined as the sum of energy consumed for inter-nodal latency between servers ($E^{lat}_{x_{n,j}}$), the computing of each module ($E^{exe}_{x_{n,j}}$), and the data transmission between $v_{n,j}$ and all of its predecessor modules ($E^{tra}_{x_{n,j}}$). It is formulated as:

$$E(x_{n,j}) = E^{exe}_{x_{n,j}} + E^{lat}_{x_{n,j}} + E^{tra}_{x_{n,j}} \tag{20}$$

The computing energy consumption for module $v_{n,j}$ depends on its assigned server and can be derived from:

$$E^{exe}_{x_{n,j}} = \begin{cases} T^{exe}_{x_{n,j}} \times P_{cpu}, & \text{if } x_{n,j} = (h,i) \;\&\; h = 0 \\ T^{idle}_{x_{n,j}} \times P_i, & \text{if } x_{n,j} = (h,i) \;\&\; h \neq 0 \end{cases} \tag{21}$$

Because only the energy consumption of IoT devices is considered in this work, whenever application modules run on remote servers, the energy consumption of IoT device is equal to the idle time $T^{idle}_{x_{n,j}}$ multiplied to the power consumption of IoT device in its idle mode P_i. Besides, P_{cpu} is the CPU power of the IoT device on which the module $v_{n,j}$ runs.

The energy consumption for data transmission between the module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ of the application belonging to the nth IoT device is calculated as follows:

$$\begin{aligned} E^{tra}_{x_{n,j}} = \max(\vartheta^{tra}(e^{dsize}_{n,i,j}, (h,i), (h',i'))), \\ \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \\ x_{n,i} = (h,i), x_{n,j} = (h',i') \end{aligned} \tag{22}$$

where, to reduce the length of equations, we consider $H = (e^{dsize}_{n,i,j}, (h,i), (h',i'))$. Similar to response time model, (h,i) and (h',i') show the specifications of source and destination servers on which modules $v_{n,i}$ and $v_{n,j}$ runs, respectively. The transmission energy consumption between each pair of

dependent modules ($\vartheta^{tra}(H)$) is calculated as follows:

$$\vartheta^{tra}(H)=\begin{cases}(\frac{e^{dsize}_{n,i,j}}{B_{up}}\times P_t)+(\gamma^{tra}(H')\times P_i), & NSE_i(H)=NSE_1\\ (\frac{e^{dsize}_{n,i,j}}{B_{down}}\times P_t)+(\gamma^{tra}(H')\times P_i), & NSE_i(H)=NSE_2\\ \gamma^{tra}(H')\times P_i, & NSE_i(H)=NSE_3\end{cases} \tag{23}$$

where P_t presents the transmission power of the IoT device, and the NSE_i shows transmission configuration based on H.

$$NSE_i(H)=\begin{cases}H'=(e^{dsize}_{n,i,j},Par(h,i),(h',i')), & \text{if } h<h' \;\&\; i=1\\ & h=0,\\ H'=(e^{dsize}_{n,i,j},(h,i),Par(h',i')), & \text{if } h>h' \;\&\; i=2\\ & h'=0,\\ H'=H, & \text{otherwise, } i=3\end{cases} \tag{24}$$

NSE_1 states the data flow is starting from an IoT device as the source server to remote servers as destination. Hence, the respective transmission energy consumption is equal to the required time to send the data to the parent server of IoT device multiplied by P_t, plus the IoT device's idle time (in which the data is transmitted from parent server to the destination) multiplied by P_i. Moreover, NSE_2 represents the invocation starting from remote servers as the source to the IoT device as the destination. It is important to note that the transmission power of IoT device P_t is active only if one of the modules is assigned to the IoT device and another module run on the remote servers, because we only consider the energy consumption from the IoT device's perspective. In other conditions, the transmission energy consumption is equal to the transmission time γ^{tra} (obtained from Eq. 10), in which the IoT device is in idle mode, multiplied by P_i (NSE_3).

The inter-nodal energy consumption $E^{lat}_{x_{n,j}}$ between servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ are placed is calculated as:

$$\begin{aligned}E^{lat}_{X_{n,j}}&=\max(\vartheta^{lat}((h,i),(h',i'))),\\ &\forall e_{n,i,j}\in\mathcal{E}_n|v_{n,i}\in\mathcal{P}(v_{n,j}),\\ &x_{n,i}=(h,i),x_{n,j}=(h',i')\end{aligned} \tag{25}$$

where ϑ^{lat} shows the energy consumption incurred due to inter-nodal delay between source and destination servers on which $v_{n,i}$ and $v_{n,j}$ are placed. This latter is calculated similar to transmission energy consumption based on the $NSE_i(A)$ [33], [2]. To reduce the equation size, $((h,i),(h',i'))=A$.

$$\vartheta^{lat}(A)=\gamma^{lat}(A)\times P_i \tag{26}$$

where the $\gamma^{lat}(A)$ is obtained from Eq. 17.

2) Migration weighted cost model: We assume that the migration of modules belonging to the nth IoT device from current servers to new servers only happens due to the mobility of IoT devices. We consider pre-copy memory migration in which the current servers still running while transferring pre-dump to the new servers [13], [31]. The goal of migration cost model is to minimize the the downtime plus required cost of executing remaining instructions on the new servers. The migration weighted cost model is defined as:

$$\min_{w_1,w_2\in[0,1]}\Psi^{mig}((X_n,X'_n),t),\quad \forall t\in|SchS_n|,\quad \forall n\in\{1,2,\cdots,N\} \tag{27}$$

where

$$\Psi^{mig}((X_n,X'_n),t)=w_1\times\Gamma^{mig}((X_n,X'_n),t)+w_2\times\Theta^{mig}((X_n,X'_n),t) \tag{28}$$

$$s.t.\qquad C1:\sum_{t=1}^{|SchS_n|}\Psi(X'_n,t)\leq\sum_{t=1}^{|SchS_n|}\Psi(X_n,t)+\epsilon \tag{29}$$

where $\Gamma^{mig}((X_n,X'_n),t)$ and $\Theta^{mig}((X_n,X'_n),t)$ represent the additional time and energy consumption incurred by the migration of modules of tth schedule in the downtime (when the service is interrupted). The C1 states the service cost for tasks emitted from modules of nth IoT device in the new configuration X'_n should be less or roughly the same while considering the previous configuration X_n. The ϵ shows an acceptable additional service cost in the migration. Moreover, constraints C1, C2, and C3 from Eq. 1 are valid here as well.

*a) **Migration time model**:* The migration time is considered as the execution time required to finish remaining instructions on the new servers plus the downtime. This latter includes the time for suspending the $Cnts$ in current servers, transmission of the dump and states, and $Cnts$' resuming time on the new servers. Since, in the downtime, a specific amount of dump data and states should also be transferred between servers ($dsize^{mig}$), the migration latency $\gamma^{lat}_{mig}((h,i),(h',i'))$ and migration transmission time between current and new servers $\gamma^{tra}_{mig}(dsize^{mig},(h,i),(h',i'))$ to transfer this data are also important [31]. Besides, the $Cnts$' stopping time plus its resuming time are considered as a constant I^{mig}. The migration time is defined as:

$$\begin{aligned}\Gamma^{mig}((X_n,X'_n),t)&=Max(\gamma^{mig}(x_{n,i},x'_{n,i})),\\ &\forall x_{n,i}\in X_n,\forall x'_{n,i}\in X'_n|v_{n,i}\in SchS_{n,t},\\ &x_{n,i}=(h,i),x'_{n,i}=(h',i')\end{aligned} \tag{30}$$

where

$$\begin{aligned}\gamma^{mig}(x_{n,i},x'_{n,i})&=\gamma^{lat}_{mig}((h,i),(h',i'))+I^{mig}\\ &+\gamma^{tra}_{mig}(dsize^{mig},(h,i),(h',i'))+\frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})}\end{aligned} \tag{31}$$

where $\gamma^{mig}(x_{n,i},x'_{n,i})$ represents the migration cost of module $v_{n,i}$ from its current server $x_{n,i}$ to its new server $x'_{n,i}$. The γ^{tra}_{mig} and γ^{lat}_{mig} are calculated based on 10 and 17, respectively. Also, $\frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})}$ shows the execution time of remaining instructions of task $e^{ins,r}_{n,i,j}$ on the new server (h',i').

*b) **Migration energy consumption model**:* The additional energy consumption of IoT device, incurred by the migration, depends on the execution of remaining instructions and the downtime.

$$\begin{aligned}\Theta^{mig}((X_n,X'_n),t)&=Max(\vartheta^{mig}(x_{n,i},x'_{n,i})),\\ &\forall x_{n,i}\in X_n,\forall x'_{n,i}\in X'_n|v_{n,i}\in SchS_{n,t},\end{aligned} \tag{32}$$

$$x_{n,i} = (h,i), x^{'}_{n,i} = (h',i')$$

where

$$\vartheta^{mig}(x_{n,i}, x'_{n,i}) = \vartheta^{lat}_{mig}((h,i),(h',i')) + I^{mig} + \vartheta^{tra}_{mig}(dsize^{mig},(h,i),(h',i')) + \vartheta^{exe}_{mig}(x'_{n,i}) \quad (33)$$

where $\vartheta^{mig}(x_{n,i}, x'_{n,i})$ represents the amount of energy consumed by the IoT device in the migration of each module of application from its current server $x_{n,i}$ to its new server $x^{'}_{n,i}$. The ϑ^{tra}_{mig} and ϑ^{lat}_{mig} represent the energy consumption incurred due to the transmission and migration latency between current and new servers. They are calculated based on 23 and 26, respectively. Also, the $\vartheta^{exe}_{mig}(x'_{n,j})$ shows the energy consumption required for the execution of remaining instructions of task $e^{ins,r}_{n,i,j}$ on the new server (h',i').

$$\vartheta^{exe}_{mig}(x'_{n,i}) = \begin{cases} \frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})} \times P_{cpu}, & \text{if } x'_{n,i} = (h',i') \;\&\; h' = 0 \\ \frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})} \times P_i, & \text{if } x'_{n,i} = (h',i') \;\&\; h' \neq 0 \end{cases} \quad (34)$$

C. Optimal Decision Time Complexity

We assume M servers exist in the hierarchical fog/edge computing environment and the maximum number of modules in each IoT application is K. Each module of an IoT application can be assigned to one of the M candidate servers at a time. Hence, for an IoT application with K modules, the Time Complexity (TC) of finding the global optimal solution for the application placement and the migration is $O(M^K)$. This cost is prohibitively high and prevents us from obtaining the global optimal solution in real-time [34]. Hence, we propose distributed algorithms to find an acceptable solution in a polynomial time for application placement and migration techniques in hierarchical fog computing environments.

IV. Proposed Technique

In this section, we present a fog server architecture to support distributed application placement, migration management, and clustering (as depicted in Fig. 4) by extending the fog server architecture proposed in [1]. Each FS in [1] is composed of three main components: controller, computational, and communication. We extend this architecture to support clustering and mobility management of IoT users in a distributed manner.

In our FS architecture, the Controller Component monitors and manages the Communication and Computational Components. It consists of three decision engine blocks and several meta-data blocks to store important information. The *Clustering Engine* is responsible for forming a distributed cluster with its in-range FSs and updating CMs' information in the *Cluster Info* and *Routing Info* meta-data. The *Application Placement Engine* is responsible for placement of IoT applications' modules to minimize the overall cost of running real-time IoT applications. It checks *Cluster Info*, *Resource Info*, and *Routing Info* meta-data for making placement decision, and updates the *Placement Info* and *Resource Info* meta-data blocks to store the configuration of application modules and available resources in this FS, respectively. The *Migration Management Engine* of each FS controls migration process of applications' modules when IoT users move. This module considers all meta-data blocks including the current mobility information of the users (i.,e *Mobility Info*), and decides the migration destination of application modules. Based on its decision, *Placement Info* and *Resource Info* will be updated to store last changes in the configuration of application modules.

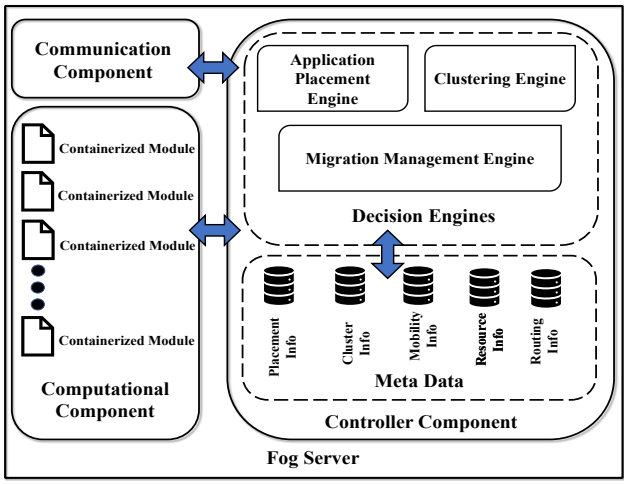


Figure 4: A view of fog server architecture

The Computational Component provides resources for the execution of application modules that are assigned to this FS based on the container technology. Besides, the Communication Component is responsible for network functionalities such as routing and packet forwarding, just to mention a few [1].

A. Dynamic Distributed Clustering

Since FSs usually have fewer resources in comparison to CSs, one FS may not be able to provide service for all modules of one application. Moreover, in some scenarios, several IoT devices are connected to the same FS, and hence, the FS may not be able to serve all application modules of different IoT devices due to its limited resources. Thus, other modules of one application should be placed on either CSs or higher-level FSs for the execution. However, in a hierarchical fog computing environment, in which the potential clustering of FSs is considered, application modules can be placed or migrated to other FSs in the same cluster. It can reduce the placement and migration cost of application modules.

We consider that FSs belonging to the same hierarchical layer can form a cluster by any in-range FSs at the same hierarchical level and swiftly communicate together using the Constrained Application Protocol (CoAP), Simple Network Management Protocol (SNMP), and so forth. Therefore, the communication delay within a cluster is lower than communication using up-link and down-link [1]. Besides, in a reliable IoT-enabled system, it is expected that the fog infrastructure providers have applied efficient networking techniques to ensure steady communication among the FSs through less variable inter-nodal latency [1]. Algorithm 1 provides an overview of the dynamic distributed clustering technique.

When an FS joins the network, it receives and stores *CandidParent* control messages from FSs residing in the immediate upper layer. The new FS finds coordinates of its position and estimates the average latency to all candidate parents. It selects the FS with the minimum distance as its parent and sends an acknowledgment to it using *ParentSelection* method. Moreover, the new FS broadcasts a *FogJoining* control message, containing its position and coverage range, to its one-hop neighbors (lines 2-7). FSs receiving this message send back a *replyNewFog* control message with their list of active and inactive $Cnts$, positions' specifications, and their coverage range to the new FS. Besides, they update their CM list $List_{cl}$ with specifications of this new FS (lines 8-14). As the new FS receives *replyNewFog* message, it builds its CM list $List_{cl}$ with specifications of FSs residing in the same hierarchical layer. Alongside storing lists of active and inactive $Cnts$ of its CMs, positions, and their coverage range (lines 15-21). This distributed mechanism helps FSs to dynamically update their CM lists when a new FS joins the network.

We consider that each FS can leave the network in normal conditions (e.g., when the low-level FS is switched off by its user) or due to a failure (such as hardware or software failures). Before an FS leaves the network in normal conditions either permanently or temporarily, we assume that all of its assigned tasks should be finished. Hence, it only needs to send *StartFogLeaving* control message to its CMs to update the $List_{cl}$ of themselves, to its parent server, and to its children to find a new parent (lines 22-25). All FSs that receive *FogLeaving* control message remove all information related to this FS from their entries. Also, the children of the leaving FS that receive this control message call the *ParentSelection* method to update their parent (lines 26-32). In case of a fatal error, in which the leaving FS cannot send a control message to the parent, CMs, and children, its immediate parent runs the *StartFogFailureRecovery* and sends *FogFailureRecovery* control message to its children list $List_{ch}$ so that they can remove entries related to the failed FS (lines 33-39). It is important to note that this latter process takes more time in comparison to the *FogLeaving* process in normal conditions due to the higher latency of uplink and downlink communications. Besides, if any FS children loose their connection to their parent, they can run the *ParentSelection* method to choose a new parent.

In addition, each FS sends the latest information about its $List_{ch}$ to its parent FS if any changes happen. This helps higher-level FSs update their Ω.

Algorithm 1: Dynamic distributed clustering

```
   Input  : RCM: Received Control Message
1  switch RCM do
2      case CandidParent do
3          ParentSelection()
4          message.add(getPosition(),coverRange)
5          message.type(FogJoining)
6          Broadcast(message)
7      end
8      case FogJoining do
9          message.add(getPosition(),coverRange)
10         message.add(getActiveCnts(),getInactiveCnts())
11         message.type(ReplyNewFog)
12         send(RCM.getSourceAddr(), message)
13         List_cl.update(RCM.getData())
14     end
15     case ReplyNewFog do
16         List_cl.update(RCM.getData())
17         MapActiveCnt_cl.put(RCM.getSourceAddr(),
18         message.getListActiveCnts())
19         MapInActiveCnt_cl.put(RCM.getSourceAddr(),
20         message.getListInActiveCnts())
21     end
22     case StartFogLeaving do
23         message.type(FogLeaving)
24         Broadcast(message)
25     end
26     case FogLeaving do
27         List_cl.remove(RCM.getSourceAddr())
28         List_ch.remove(RCM.getSourceAddr())
29         if RCM.getSourceAddr() == this.Parent) then
30             ParentSelection()
31         end
32     end
33     case StartFogFailureRecovery do
34         for i = 1 to List_ch.size() do
35             message.type(FogFailureRecovery)
36             message.setFailedFog(failedFog.getAddr())
37             send(List_ch.get(i).getSourceAddr(),message)
38         end
39     end
40     case FogFailureRecovery do
41         List_cl.remove(RCM.getFailedFogAddr())
42     end
43 end
```

B. Application Placement

Due to the time consuming nature of finding the optimal solution (Section III-C) for the application placement problem, a Distributed application placement technique (DAPT) is proposed to find a well-suited solution in a distributed manner (Algorithm 2). The DAPT starts whenever an application placement request arrives, and the serving FS tries to place application modules on appropriate servers so that real-time tasks, emitted from modules, can be processed with the minimum cost. Considering the weighted cost (Eq. 1), DAPT attempts to place modules of IoT applications in one/several FSs on the lowest-possible layer while considering the potential of clustering. However, if available resources in that/those FSs are not sufficient, it considers upper layer FSs or/and CSs to place the rest of modules. In this way, DAPT reduces the search space of Eq. 1 for each FS by only considering itself, its parent FS, and its CMs, and aims at reducing the overall weighted cost. Moreover, a distributed failure recovery method is embedded in DAPT to recover from possible failures.

The immediate FS that receives the placement request from an IoT device is considered as the application placement controller ($controller$) for that IoT device. If the controller is performing the placement of a set of modules or a parent FS receives placement request from its children and the failure recovery mode is not active (lines 3-28), the *ClusterCheck* method returns the list of CMs and their available resources (line 4). Then, the list of ready servers S_R containing parent FS, current FS, and available CMs is created (line 5). This list contains all servers that current FS considers for the placement of modules in that hierarchical layer. Next, the *FindOrder* method checks either topological order of modules (TO_n) are available or not. If it is not available, it considers the DAG $\mathcal{G}_n$ of nth IoT application, and using the Breadth-First-Search (BFS) Algorithm finds topological order of all modules, and

Algorithm 2: An overview of DAPT

Input : $\mathcal{G}_n$: The DAG of nth IoT device, $U_{\mathcal{G}_n}$: A subset of unassigned modules from $\mathcal{G}_n$, X_n: The configuration of assigned modules, $controller_{ID}$: ID of the placement controller
Output : X_n

1 s_{ID}: this.ID
2 $List_{cl}$: this.getClusterMembers()
3 **if** *(controller(n) || ReqFromChild) & !DAPTFailureRecovery(n)* **then**
4 $List^{A}_{cl}$=ClusterCheck($List_{cl}$)
5 S_R=ReadyServers($List^{A}_{cl}$,this.parent,s_{ID})
6 $SchS_n$=FindOrder($\mathcal{G}_n$)
7 $U_{(\mathcal{G}_n)}$=Sort($U_{(\mathcal{G}_n)}$, $SchS_n$)
8 **if** $S_R - Par(s_ID) \neq \varnothing$ **then**
9 **for** $i = 1$ *to* $U_{\mathcal{G}_n}$*.size()* **do**
10 v=$U_{(\mathcal{G}_n),i}$
11 ID_{min}=FindMinCost(S_R,$\mathcal{G}_n$,X_n,v)
12 **if** ID_{min} == s_{ID} **then**
13 res_v=CalService(v)
14 **if** *this.*$Cnts$*.contains(v)* & **then**
15 ScaleCnts(v,res_v)
16 **else**
17 StartCnt(v)
18 **end**
19 UpdateConfig(X_n,v,s_{ID})
20 **end**
21 **else**
22 $ReqList$.update(v,ID_{min})
23 **end**
24 **end**
25 PlaceReqToServers($ReqList$,$\mathcal{G}_n$,X_n,S_R,TO_n,$SchS_n$)
26 **else**
27 PlacePar($\mathcal{G}_n$,$U_{\mathcal{G}_n}$,X_n,TO_n,$SchS_n$)
28 **end**
29 **else if** *!controller(n) & !DAPTFailureRecovery(n)* **then**
30 **for** $i = 1$ *to* $U_{\mathcal{G}_n}$*.size()* **do**
31 v=$U_{(\mathcal{G}_n),i}$
32 res_v=CalService(v)
33 **if** *this.*$Cnts$*.contains(v)* & $res_v \leq$ *this.Resource* **then**
34 ScaleCnts(v,res_v)
35 UpdateConfig(X_n,v,s_{ID})
36 NotifyController(v, s_{ID},$controller_{ID}$)
37 **else**
38 **if** $res_v \leq$ *this.Resource* **then**
39 StartCnt(v)
40 UpdateConfig(X_n,v,s_{ID})
41 NotifyController(v, s_{ID},$controller_{ID}$)
42 **else**
43 SendDAPTFailureRecovery(n,v,$controller_{ID}$,s_{ID})
44 **end**
45 **end**
46 **end**
47 **else**
48 DAPTFailureRecovery(n,v,S_R,X_n)
49 **end**

creates $SchS_n$ (line 6). This latter helps to identify modules that do not have any dependency and can be executed in parallel. Then, $Sort$ method defines priority value for modules that can be executed in parallel (i.e., modules with the same topological order) based on non-increasing order of their rank value (line 7). The rank of each module is defined as:

$$Rank(v_{n,j}) = \begin{cases} C^{exe}_{n,j} + \max(C^{tra}_{n,j,z} + Rank(v_{n,z})) & \text{if } v_{n,j} \neq exit \\ \forall v_{n,z} \in Succ(v_{n,j}), & \\ C^{exe}_{n,j}, & \text{if } v_{n,j} = exit \end{cases} \tag{35}$$

where $C^{exe}_{n,j}$ shows the average weighted execution cost of module $v_{n,j}$, and $C^{tra}_{n,j,z}$ depicts the transmission cost of module $v_{n,j}$ and $v_{n,z}$, which are calculated as:

$$C^{exe}_{n,j} = w_1 \times \widetilde{T^{exe}_{x_{n,j}}(S_R)} + w_2 \times \widetilde{E^{exe}_{x_{n,j}}(S_R)} \tag{36}$$

$$C^{tra}_{n,j,z} = w_1 \times \widetilde{\gamma^{tra}_{n,j,z}(S_R)} + w_2 \times \widetilde{\vartheta^{tra}_{n,j,z}(S_R)} \tag{37}$$

where $\widetilde{T^{exe}_{x_{n,j}}(S_R)}$ and $\widetilde{E^{exe}_{x_{n,j}}(S_R)}$ show the average execution time and energy consumption of each module considering available servers in the S_R. The execution time $T^{exe}_{x_{n,j}}$ and energy consumption $E^{exe}_{x_{n,j}}$ of each module per server are obtained from Eq. 8 and Eq. 21 respectively. Besides, $\widetilde{\gamma^{tra}_{n,j,z}(S_R)}$ and $\widetilde{\vartheta^{tra}_{n,j,z}(S_R)}$ shows the average transmission time and energy consumption between modules $v_{n,j}$ and $v_{n,z}$ considering available servers in the S_R. The transmission time $\gamma^{tra}_{n,j,z}$ and transmission energy consumption $\vartheta^{tra}_{n,j,z}$ between each pair of servers in the S_R can be obtained from Eq. 10 and Eq. 23, respectively. Moreover, w_1 and w_2 are control parameters to tune the weighted cost. The rank is calculated recursively by traversing the DAG of application, starting from the exit module. The $Sort$ method can find the critical path of the DAG and gives higher priority to the modules that incur higher execution cost among modules that can be executed in parallel. Hence, the probability of placement of these modules on lower-level FSs increases. This latter is important since the resources of lower-level FSs are limited compared to higher-level FSs, but they can be accessed with less communication cost. Hence, if modules are more communication and latency-sensitive, they can be placed on lower-level FSs with higher priority while if they are computation-intensive modules, that cannot be efficiently executed on the lower-level FSs, they can be forwarded to higher-level FSs with higher priority. If S_R contains any candidate server except its parent, for each module v of $U_{\mathcal{G}_n}$, the *FindMinCost* receives the S_R, $\mathcal{G}_n$, and configuration X_n, as its input and finds the minimum cost for the execution of the module v based on current solution configuration X_n (i.e., based on the assigned servers' configuration to the predecessors of this module). Although in fog computing environments, a large number of FSs are deployed as candidate servers, the DAPT only considers FSs in the S_R, to which the serving FS can communicate with the lowest possible transmission and inter-nodal cost. Moreover, we assume that FSs do not have a global view of all FSs in the environment. Therefore, the search space in each hierarchical layer is reduced while the suitable candidate servers for real-time and latency-sensitive IoT applications are kept. After prioritizing modules, the execution cost of each module based on the available servers in S_R is calculated using *FindMinCost* method. This method checks the available resources required to run or scale the $Cnts$ to run these modules on available servers. Then, among the servers that meet these requirements, it returns the ID of the selected FS, ID_{min}, that can execute module v while minimizing the overall application cost using Eq.1 (line 11). If the current FS is selected, and it has active Cnt, the *ScaleCnt* method scales the resources so that it can serve this module (line 15). If there is no active Cnt in this FS, it should run a new Cnt, which incurs a Cnt startup cost (line 17). The candidate solution configuration X_n is updated accordingly so that the new configuration can be considered

for the placement of the rest of the modules (line 19). If the selected FS is among the CMs or parent FS, the module v and its corresponding assigned server are stored in the request list $ReqList$ (line 22) so that it can be forwarded to their destination using the *PlaceReqToServers* (line 25). This method sends modules to assigned serves along with the topological order of this IoT application TO_n, schedules $SchS_n$, and current solution configuration X_n. Finally, in a case that the S_R is empty, meaning that the current controller does not have any resources and also it does not have any candidate servers with sufficient resources, it sends all modules to the parent FS so that the placement can be started in the higher hierarchical levels by means of the *PlacePar* method (line 27). If the parent FS receives the placement request from its children, it checks the possibility of placement of received modules on its S_R. The background reason is if one FS receives some modules for placement from its children FSs, it means that those modules are either more computation-intensive rather than latency/communication-intensive, or the children FSs did not have sufficient resources for these modules. However, if one FS receives a placement request from its CMs, it starts the deployment of modules on the condition that the available resources meet the modules' requirements.

If serving FS is not the controller FS and the failure recovery mode is not active (i.e, the placement request is forwarded to CMs), it iterates over the received modules (i.e., $U_{\mathcal{G}_n}$) and calculates the required amount of resources for each module *CalService(v)*. If it has enough resources, it starts the module, and using *NotifyController* method sends an acknowledgment for the controller FS. However, if due to any problem this FS cannot place this module, it runs *SendDAPTFailureRecovery* method, which sends a failure message to the controller FS so that the controller can make a new decision (lines 29-47).

If failure recovery mode is active, it means that one or several servers cannot properly execute assigned modules. Hence, the DAPT algorithm calls *DAPTFailureRecovery* method. This method receives failed modules of nth IoT application and finds corresponding FSs from the solution configuration X_n. If it has several candidate servers in S_R, it removes specification of the failed FS from S_R. Then, it iterates over the rest of available servers to finds FSs for these modules that minimize the execution cost. However, if the current FS only has its parent sever in the S_R, *DAPTFailureRecovery* sends a control message to activate *DAPTFailureRecovery* method of the parent FS. (line 48). It helps to check the possibility of placement of these modules in higher hierarchical layers.

C. Migration Management Technique (MMT)

As the user of nth IoT device is moving away from its current low-level FS (i.e., its controller FS) to a new low-level FS, the current controller FS should initiate the migration process to find a new controller FS, and migrate the current data and states of running $Cnts$ to new FSs. We suppose IoT devices can detect distributed low-level FSs (eg., using beacons, GPS, etc) and update their list of sensed FSs $List^n_{SFog}$ periodically. Whenever the controller FS realizes that the IoT device n is about to leave (e.g., through the received signal to noise ratio), it receives $List^n_{SFog}$ from the IoT device and initiates the migration process. The goals of the migration management technique (MMT) is to 1) find a new controller FS with the maximum sojourn time for the IoT device and 2) find a set of substitute servers for processing of IoT application's modules while minimizing the migration cost (Eq. 27). The Algorithm 3 shows an overview of the distributed migration process.

Whenever a controller FS realizes the nth IoT device is about to leave its coverage range, it initiates *MigrationInitiate* to find a new controller FS for the IoT device. The current controller FS receives the list of sensed low-level FSs $List^n_{SFog}$ from nth IoT device and removes its s_{ID} from this list so that it cannot be selected as a new controller FS (line 4). The mobility information of each user *mobInfo(n)* contains its average speed and its direction. Moreover, in the clustering technique, each FS learns the position and coverage ranges of its CMs. Considering the aforementioned values, the controller FS can estimate the sojourn time of this IoT device for each CM. The *MobilityAnalyzer* method (line 5) receives *mobInfo(n)* and $List^n_{SFog}$ and checks whether the $List^n_{SFog}$ contains any CMs of the current FS controller. Moreover, it finds specifications of other FSs belonging to $List^n_{SFog}$ through its CMs, if possible. The *MobilityAnalyzer* then creates two separate lists for reachable FSs ($List_{reach}$) and unreachable FSs ($List_{unreach}$) from $List^n_{SFog}$. The former one contains any FSs of $List^n_{SFog}$ which are among CMs of the current controller FS or those that can be accessed through its CMs, while the latter one refers to FSs to which the controller FS does not have access either directly or through its CMs. The *MobilityAnalyzer* method gives higher priority to FSs of $List_{reach}$ because the required information for the new controller to start its procedures can be more efficiently transferred to these FSs compared to those FSs to which it does not have direct access. The MMT considers $resources$ of FSs belonging to $List_{reach}$, and if they have enough resources to serve modules that are currently assigned to the current controller FS, it estimates the sojourn time of nth IoT device for those candidate FSs. Then, it returns the ID of the FS with sufficient resources and the maximum estimated sojourn time. It is important to note that assigning the controller role to a new FS with maximum sojourn time can reduce the number of possible future migrations, which leads to fewer service interruptions due to migration downtime. On the condition that no FSs of $List_{reach}$ contains enough resources, it returns the ID of FS with the maximum sojourn time. However, if $List_{reach}$ is empty, this method returns the ID of one of the FSs from $List_{unreach}$ randomly. Then, current controller FS sends a *NewControllerReq* message to $dest_{ID}$, containing the DAG of nth IoT device application $\mathcal{G}_n$, *mobilityInfo(n)*, and the current configuration of assigned servers X_n (lines 7-9).

When an FS receives *NewControllerReq* message, it adds the IoT device n to its *controllerList* to serve this IoT device as its new controller FS (lines 11-12). This new controller FS is responsible for the rest of migration management. It retrieves the current configuration X_n and the previous controller ID,

Algorithm 3: Migration Management Technique

```
   Input  : RCM: Received Control Message, G_n: The DAG of nth IoT
            device, mobInfo(n): The mobility data of the IoT device n,
            X_n: The configuration of assigned modules, controller_ID: ID
            of the controller, List^n_SFog: Sensed fog devices' List of IoT
            device n
1  switch RCM do
2      case MigrationInitiate do
3          List^n_SFog=List^n_SFog.remove(s_ID)
4          dest_Id=MobilityAnalyzer(n,mobInfo,List_cl,List^n_SFog)
5          message.add(G_n,mobInfo(n),X_n,TO_n,SchS_n)
6          message.type(NewControllerReq)
7          send(dest_ID,message)
8          controller_pre(n)=true
9      end
10     case NewControllerReq do
11         n=RCM.getIoTDevice
12         getcontrollerList().add(n)
13         X_n=RCM.getConfig(n)
14         ID_PreCon=RCM.getSourceAddr()
15         List^sorted_Cnts =SortCntsSize(G_n, Cnts^ram)
16         MapServer_pre=FindPreServersConfig(X_n)
17         for t = 1 to |SchS_n| do
18             sendMigReqToServers(MapServer_pre,List^sorted_Cnts,SchS_n.t)
19             WaitForServersNotifications()
20         end
21     end
22     case MigrationReq do
23         ReqInfo=RCM.getInfo()
24         Modules= ReqInfo.getModules()
25         S_R=ReadyServers(this.getCMs(),this.getID(),this.getChildren())
26         if !S_R.isEmpty() then
27             for i = 1 to Modules.size() do
28                 SortedCostList=∅
29                 for j = 1 to S_R.size() do
30                     MigCostTemp=CalMigCost(Modules_i,S_R,j)
31                     CostList.update(S_R,j,MigCostTemp)
32                 end
33                 SortedCostList=Sort(CostList)
34                 Server_ID=FindMigrationDestination(SortedCostList)
35                 sendMigrationDestination(Modules_i,X_n,Server_ID)
36             end
37         end
38         else
39             SendMigReqToServers(this.Parent(),ReqInfo)
40         end
41     end
42     case MigrationDestination do
43         v=RCM.getModule()
44         res_v=calService(v)
45         if res_v ≤ this.resources then
46             sendMigrationStart(v,FS^v_pre,FS^v_new)
47             UpdateConfig(X_n,v,s_ID)
48             NotifyController(v,s_ID,controller_ID)
49         else
50             SendMMTFailureRecovery(n,v,controller_ID,s_ID)
51         end
52     end
53     case StartMigration do
54         Migrate(v,RCM.FS^v_new)
55         UpdateResoure(v)
56         if controller_pre(n) & MigrationFinish(n) then
57             controller_pre(n)=false
58             getControllerList().remove(n)
59         end
60     end
61     case MMTFailureRecovey do
62         MMTFailureRecovery(n,v,controller_ID,s_ID)
63     end
64 end
```

ID_{PreCon}, from the received message RCM (lines 13-14). The *SortCntsSize* method descendingly sorts $Cnts$ based on their allocated runtime Ram $Cnts^{ram}$ (line 15). The background reason is the amount of dump and state to be transferred in the downtime is directly related to $Cnts^{ram}$ [31]. The migration of $Cnts$ with larger $Cnts^{ram}$ incurs higher cost in terms of migration time and energy (Eq. 27). Hence, to reduce the total migration cost, MMT gives higher priority to modules with heavier $Cnts^{ram}$ so that the migration decision can be made sooner, and they can be migrated before other modules. Next, *FindPreServersConfig* method retrieves assigned servers' specifications for all application modules and stores them in $MapServer_{pre}$ (line 16). The migration cost (Eq. 27) is defined as the maximum migration cost for each application module while considering X_n and its new configuration X'_n. The goal is to minimize this migration cost while it is subject to the condition that the new configuration X'_n provides better application execution cost or roughly the same with previous configuration X_n (Eq.29). So, the MMT retrieves modules of each schedule based on $SchS_n$ and send their corresponding information alongside $MapServer_{pre}$ and $List^{sorted}_{Cnts}$ to *sendMigReqToServers* method. It creates a list of modules based on the hierarchical layer on which modules are previously assigned. Modules of each hierarchical layer are also sorted based on allocated Ram size, obtained from $List^{sorted}_{Cnts}$. This method sends $MigrationReq$ messages alongside respective modules' information to FSs that are responsible for making the migration decision. As MMT acts in a distributed manner and FSs at each layer only has information about their parent, children, and CMs, migration decisions for modules of each layer are made by the new controller, its parent, or ancestors in the hierarchy. To illustrate, considering Fig. 1, we assume an IoT application has three modules in one of its schedules and two of them were previously assigned on FS (1,3) (prior controller), and one on FS (2,1). If we assume that the new controller is FS (1,4), it makes migration decision for modules that previously assigned on FS (1,3) while $par(1,4)$ (i.e., FS (2,3)) makes migration decision for the module that previously assigned on FS (2,1). After sending migration requests $migrationReq$, FS (1,4) waits to receive notifications and new configuration of modules for that schedule and then iterates over next schedules (lines 17-20).

When an FS receives *MigrationReq* message, the FS retrieves the information and forwarded modules from the received message (lines 23-24). Then, the list of ready servers S_R is created based on CMs, and children. If the S_R does not contain any available servers, all the modules are forwarded to the parent FS for making migration decision (line 39), while if it contains servers, it tries to minimize the migration cost based on the specification of available servers (line 26-37). This FS considers a list of $modules$, sorted descendingly based on $Cnts^{ram}$, for making migration decision. Hence, the migration of modules that incur higher migration costs in each schedule is performed with higher priority, leading to less overall migration costs in that schedule. Then, for each selected module, the migration cost is estimated and stored in the $CostList$ (line 29-32). The *Sort* method sorts the migration costs ascendingly so that servers with lower migration cost receives higher priority (line 33). Then, the $FindMigrationDestination$ method selects a new server for the module, considering $SortedCostList$, which minimizes the migration cost while it does not negatively affect the application's running cost. Hence, this method iterates over

$SortedCostList$, sorted ascendingly based on the migration costs, and selects the server that satisfies the Eq. 29 (line 34). Finally, the *sendMigrationDestination* method sends a $MigrationDestination$ message to the selected FS to check its resources and start the migration of the respective module.

The FS receiving *MigrationDestination* checks whether it has enough resources to serve the module v or not (lines 42-44). If this FS can serve the module v, it sends a *StartMigration* message to the FS^{v}_{pre} so that it can start the migration. Then, it updates the X_n with its s_{ID} and notifies the controller (lines 45-468). If it cannot serve this module due to any reason, it runs the *SendMMTFailureRecovery* method to send a failure message to the controller FS (lines 49-51).

The *MMTFailureRecovey* is working as the same as *DAPTFailureRecovey*. The only difference is that the migration cost in the MMT is obtained from Eq. 27 (lines 59-61).

Whenever an FS receives a *StartMigration* message, it starts the migration and then frees the previously assigned resources (lines 53-55). Moreover, if the FS was previously the controller for the nth IoT device, and it finishes the migration of all assigned modules belonging to that IoT device, the FS removes the nth IoT device from its *controllerList* (lines 56-59).

D. Complexity Analysis

The Time Complexity (TC) of the clustering phase (Algorithm 1) depends on the size of $List_{cl}$ and $List_{ch}$, and candidate parents in the immediate upper level for each FS. In the worst-case scenario, if we assume all FSs reside in one cluster and/or they have only one parent. Hence, the TC of *remove* method belonging to the *FogLeaving* and *FogFailureRecovery* is $O(F)$, and the TC of the *StartFogFailureRecovery* is $O(F)$. Moreover, the TC of *ParentSelection* method of *CandidParent* is $O(F)$ in the worst-case scenario if we assume one FS has $F-1$ candidate parent. Hence, the TC of the clustering step in the worst-case scenario is $O(F)$. Moreover, in the best-case scenario, the number of FSs in $List_{cl}$ and/or the size of the $List_{ch}$ is one, and the TC of the best-case is $O(1)$.

To find the TC of DAPT (Algorithm 2), we suppose that the size of the largest IoT application is K. So, in the worst-case scenario, the size of $U_{\mathcal{G}_n}$ is K. The *FindOrder* method finds the topological order of the DAG using BFS algorithm with the TC of $O(K+|\mathcal{E}|)$, in which $|\mathcal{E}|$ represents the number of data flows. In the dense DAG, the $|\mathcal{E}|$ is of $O(K^2)$. Moreover, the TC of *Sort* Algorithm is $O(FK^2)$ in the worst-case scenario. In the worst-case scenario, all FSs reside in one cluster and have enough resources for any requests. Hence, the worst-case TCs of *ClusterCheck*, *ReadyServers*, *FindMinCost*, and *DAPTFailureRecovery* are of $O(F)$, $O(F)$, $O(FK)$, and $O(FK)$, respectively. Hence, the worst-case TC of DAPT Algorithm is $O(FK^2+FK)$. In the best-case scenario, the DAG of the application can be sparse so that the TC of *FindOrder* and *Sort* algorithms become $O(K)$ and $O(1)$, respectively. Moreover, in the best-case scenario, the number of available servers in one cluster is one, and hence, TCs of *ClusterCheck*, *ReadyServers*, *FindMinCost*, and *DAPTFailureRecovery* are of $O(1)$, $O(1)$, $O(K)$, and $O(K)$, respectively. So, TC of DAPT in the best-case scenario is $O(K)$.

The TC of the *MigrationInitiate* from Algorithm 3 depends on the TC of *MobilityAnalyzer*. In the worst-case scenario, all the FSs reside in one cluster and the IoT device can sense all of them. So, the size of the list of sensed FSs $List^{n}_{SFog}$ is equal to F. Hence, in the worst-case, the TC of creating $List_{reach}$ and $List_{unreach}$ is of $O(F^2)$ while in the best-case scenario, it is of $O(F)$ when there is only one FS in the cluster. Moreover, the worst-case TC of finding maximum sojourn time is $O(F)$. So, the TC of *MigrationInitiate* in the worst-case is $O(F^2)$ while in the best-case, it is of $O(F)$. The TC of the *NewControllerReq* in the worst-case is $O(KLogK+FK)$ while TC of *NewControllerReq* in the best-case scenario is $O(KLogK)$ when there is only one FS in each cluster. The TC of *MigrationReq* in the worst-case scenario depends on the TCs of $CalMigCost$ and $Sort$ which are $O(FK)$ and $O(FKLogF)$ while in the best-case scenario they are O(K). The TC of *MigrationDestination* depends on the TC of $MMTFailureRecovery^{mig}$ and is of $O(F)$ at the worst-case and $O(1)$ in the best-case scenario. Therefore, the TC of the MMT in the worst-case scenario is $O(F^2+FK+KLogK+FKLogF)$ while in the best-case scenario is $O(KLogK)$.

Considering TCs of all methods, the TC of our technique in the worst-case scenario is $O(F^2+FK^2+FKLogF)$ while in the best-case scenario, it is $O(F+KLogK)$.

V. Performance Evaluation

In this section, the system setup and parameters, and detailed performance analysis of our technique, in comparison to its counterparts, are provided.

A. System Setup and Parameters

We extended the iFogSim simulator [10] for the implementation and evaluation of distributed mobility management, clustering, and failure recovery techniques. We used DAGs of two real-time applications, namely the Electroencephalography tractor beam game (EEGTBG) [10], [19] and ECG Monitoring for Health-care applications (ECGMH) [9] to create our DAGs. Both applications consist of a sensor and display modules that are placed in the IoT device (e.g., smartphone, wearable devices, etc). Other modules can be placed either on distributed FSs or CSs based on the distributed application placement decisions and/or the migration technique. Data transmission intervals for ECG and EEG sensors are 10ms and 15ms, respectively [1], [10]. Besides, we assume the amount of RAM allocated to each container at the runtime for state size is randomly selected from 50-75 MBytes [31]. The total amount of data to be transferred in the downtime (i.e., $dsize^{mig}$) is just a few MBytes [31], which is randomly selected from 5-10% of each container's allocated RAM in the runtime.

We simulate a 2km $\times$ 1km area, in which the coverage range of FSs situated in the first and second layers is assumed to be 200m and 400m, respectively. The system consists of one layer of IoT devices, three layers of heterogeneous FSs, and a layer [1], [7], [9]. The IoT device layer consists of 80 IoT devices,

while the number of FSs in level 1, level 2, and level 3 are 30, 5, and 1, respectively. The computing power (CPU) of IoT devices is considered as 500 MIPS [33], while the computing power of level 1 FSs is randomly selected from [3000-4000] MIPS [33], [19]. Besides, the total computing power of level 2 FSs, level 3 FSs, and CS are considered as 8000 MIPS, 10000 MIPS, and 80000 MIPS, respectively [7], [9]. Besides, the latencies between IoT devices to level 1 FSs, level 1 FSs to level 2 FSs, level 2 FSs to level 3 FSs, and level 3 FSs to cloud servers are 5ms, 25ms, 50ms, and 150ms, respectively [1], [9], [7]. The upstream and downstream network capacity of IoT devices are 100 Mbps and 200 Mbps, respectively. The upstream, downstream, and clusterlink network capacity for FSs and the CSs are also considered to be 10 Gbps [7], [9]. Moreover, clusters can be formed among the level 1 and level 2 FSs with their in-range FSs of the same hierarchical layer. The communication latency among the FSs residing in level 1 clusters and FSs residing in level 2 clusters are [3-5] ms and [20-25] ms, respectively [1], [9]. The processing power consumption, idle power consumption, and transmission power consumption of IoT devices are 0.9W, 0.3W, and 1.3W, respectively [35], [2]. User trajectories are generated by a variation of the random walk mobility model [27], [20], in which each user selects a direction, chooses a destination anywhere toward that direction, and moves towards it with a uniformly random speed. The user arriving at the destination can choose a new random direction.

Table III: Evaluation Parameters

Parameter	**Value**
Simulation Time	100,200,300,400 (S)
Area	2km × 1km
Users' Speed	[0.5-4] m/s
Latency (ms)	
ECG Sensor Data Transmission Interval	10
EEG Sensor Data Transmission Interval	15
ECG and EEG Sensor ↔ IoT Device	2
IoT Device ↔ Level 1 FS	5
Level 1 FS ↔ Level 2 FS	25
Level 2 FS ↔ Level 3 FS	50
Level 3 FS ↔ Cloud	150
L1 Clusters	[3-5]
L2 Clusters	[20-25]

B. Performance Study

We conducted seven experiments evaluating system size analysis, average execution cost of tasks, cumulative migration cost, the total number of migrations, Total number of Interrupted Tasks (TIT) due to the migration, Failure recovery analysis, and optimality analysis. In the experiments, to obtain the weighted cost of placement and migration, the w_1 and w_2 are set to 0.5. To analyze the efficiency of our technique, we extended two other counterparts in the dependent category of fog computing proposals as follows:

- *MAAS*: This is the extended version of the technique called Mobility-Aware Application Scheduling (MAAS) [19] working based on edgeward-placement technique. The main concern of this edge-centric technique is to place dependent modules of IoT applications on remote servers based on their pre-known mobility pattern (i.e., source, destination, and the potential paths between them are known in advance) of users. In MAAS, if an FS cannot place modules on itself, the modules should be forwarded to the parent server for placement. We extended this technique to support the migration as the users move among remote servers in the runtime while considering the destination and potential paths are not priori-known.
- *Urmila*: This is the extended version of Ubiquitous Resource Management for Interference and Latency-Aware services (Urmila) [6] which proposes a mobility-aware technique for placement of dependent modules of IoT applications while mobility pattern of users are priori-known. In this technique, the central controller is placed in the highest level FS, and makes placement decisions for IoT applications consisting of dependent modules. We extended this technique so that the central controller helps remote servers to migrate dependent modules of applications as the IoT users move.

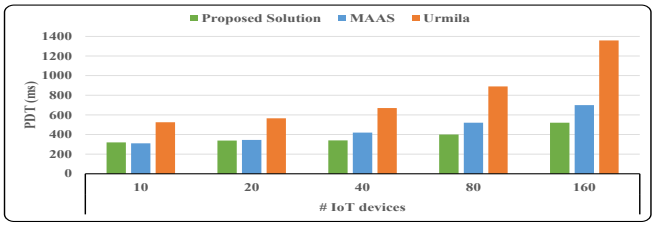
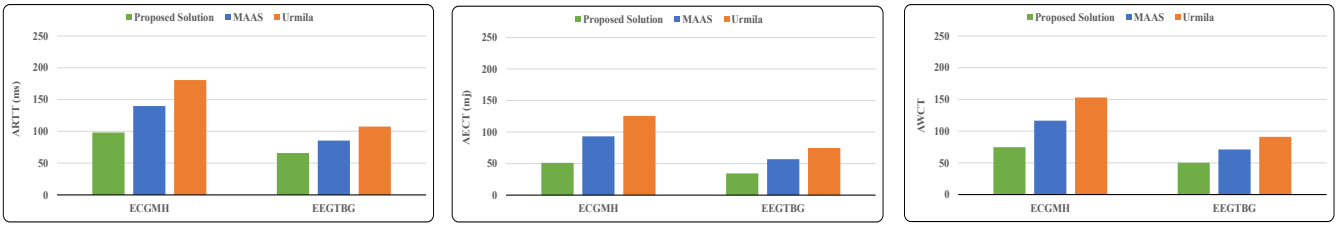


Figure 5: Placement Deployment Time (PDT)

1) System size analysis: In this experiment, we study the effect of number of IoT devices on the Placement Deployment Time (PDT). The PDT shows the period between the start of sending placement requests from IoT devices up to the time the deployment of application modules of IoT devices on FSs are finished. Obviously, the PDT includes the decision time in which FSs make placement decisions and the container startup cost on the servers. Regardless of the quality of solutions that each technique provides, the PDT helps to understand how long the IoT devices should wait until the service can start. In this experiment, the number of IoT devices is increased from 10 to 160 by multiplication of two. Although the number of IoT devices increases in this experiment, we fixed the number of FSs so that we can analyze how different techniques work when the number of placement requests increases significantly. Besides, it is clear that our technique, due to its distributed manner, can easily manage the increased number of placement requests when the number of FSs increases.

In Fig. 5, the PDTs of our proposed solution and MAAS are significantly lower than Urmila, specifically in a larger number of IoT devices. This latter is mainly because our solution and MAAS use a distributed placement engine while Urmila uses a centralized approach. When the placement decision engine receives incoming placement requests, it should make placement decisions and then manage the deployments of



(a) Average Response Time of Tasks (ARTT) (b) Average Energy Consumption of Tasks (AECT) (c) Average Weighted Cost of Tasks (AWCT)

Figure 6: Average execution cost of tasks

application modules in different servers according to solutions' configuration. In Urmila, all of the placement requests should be forwarded to the centralized entity, meaning that the number of arriving placement requests in the decision engine is larger than the distributed placement techniques. Hence, the processing of these requests on the centralized controller takes more time compared to the distributed placement engines, especially when the number of IoT devices increases. Moreover, our solution outperforms the MAAS since it tries to place more application modules in the lowest hierarchical layer, compared to MAAS, which incurs less deployment time.

2) Average execution cost of tasks: This experiment shows the average execution cost of tasks emitted from a sensor module until they arrive at actuator in 400 seconds of simulation.

As it can be seen from Fig 6, our proposed solution outperforms the MAAS and Urmila in terms of Average Response Time of Tasks (ARTT), Average Energy Consumption of Tasks (AECT), and Average Weighted Cost of Tasks (AWCT). In the MAAS, each FS, from the lowest to the highest hierarchical level, attempts to place modules on itself or forwards them to its parent server for the placement or handling of the migration process. Therefore, it does not consider other potential servers at the same hierarchical level, which incurs higher transmission and inter-nodal costs. The pure Urmila, on the other hand, does not migrate the application modules to servers that are closer to the moving IoT devices, and hence, the average execution cost of tasks, emitted from IoT devices, increases significantly. In our distributed technique, however, each FS considers potential servers at the same hierarchical level (for placement and migration) if those servers are among its CMs. In this way, we decrease the large search space of centralized techniques, while we use the benefits that servers at the same hierarchical level can provide. Also, since modules with higher costs have higher placement priority, the possibility of their placement on more suitable servers are higher compared to other modules. This latter leads to better placement decisions that minimize the cost of executing tasks. It is important to note that the average execution cost of the EEGTBG is lower than the ECGMH. It is because tasks' instruction number in the EEGTBG is lower than of ECGMH ones.

3) Total number of migrations: This experiment studies the total number of migrations that occurred during 400 seconds due to the IoT users' movement.

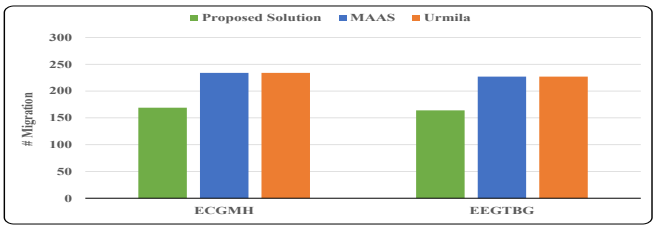
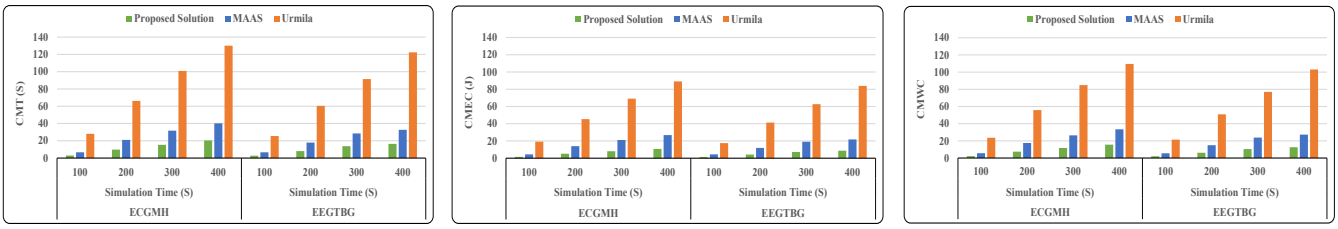


Figure 7: Total number of migrations

It can be seen from Fig. 7 that our technique leads to a smaller number of migrations in comparison to its counterparts. This is because our solution considers the current mobility information of IoT devices such as current speed and direction. Since the controller FS has coordinates of its CMs and current mobility information of leaving IoT devices (e.g., their average speed and their direction while in the range of the current controller FS), the serving FS can estimate a sojourn time for all candidate remote servers for the migration. Hence, by the migration of modules to the remote server with the highest sojourn time (in case sufficient resources are available), the number of possible migrations decreases. The extended MAAS and Urmila only try to reduce the migration cost by migrating modules to new remote servers, while they do not consider current mobility information of IoT devices and their sojourn time in remote servers. Hence, they may select remote servers in which the IoT devices stay only for a short period.

4) Cumulative migration cost: This experiment analyzes the Cumulative Migration Cost (CMC) of IoT devices for ECGMH and EEGTBG in different simulation times. The term cumulative refers to the aggregate migration cost of all IoT devices.

As Fig 8 shows, our solution outperforms its counterparts in terms of Cumulative Migration Time (CMT), Cumulative Migration Energy Consumption (CMEC), and Cumulative Migration Weighted Cost (CMWC) for both ECGMH and EEGTBG applications. As the simulation time increases, the cost of all techniques grows, however, Urmila experiences a faster increase in comparison to our solution and MAAS. This latter is because the Urmila's controller is placed at the highest hierarchical layer, which incurs significant inter-nodal and transmission cost when the controller manages migrations between the old and new remote servers in the downtime.



(a) Cumulative Migration Time (CMT)

(b) Cumulative Migration Energy Consumption (CMEC)

(c) Cumulative Migration Weighted Cost (CMWC)

Figure 8: Cumulative Migration Cost

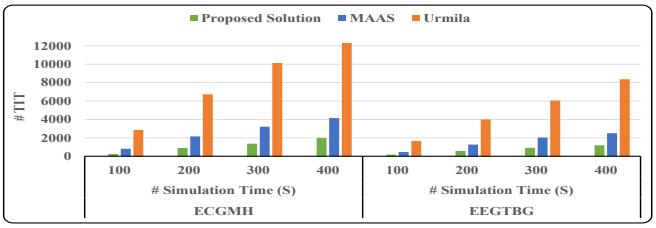


Figure 9: Total number of interrupted tasks

Besides, the migration cost of MAAS is more than our solution, since whenever the resources of controller finishes, the MAAS migrates the application modules to higher layers, and hence, the emitted tasks to/from those modules experience higher cost. Also, the total number of migrations in Urmila and MAAS are higher than ours, which apparently increases their cumulative migration costs. The slight difference between cost of ECGMH and EEGTBG is because the tasks generated from the ECGMH's modules are heavier than EEGTBG's ones in terms of their MI. So, the processing time of remaining instructions of tasks (i.e., $e^{ins,r}_{n,i,j}$) that migrated from old server to new server is higher for the ECGMH compared to the EEGTBG (in case the computing powers of old and new servers are roughly the same).

5) Total number of interrupted tasks (TIT): This experiment analyzes the Total number of Interrupted Tasks (TIT) in the downtime. During migration downtime, there is no active service provider for incoming tasks from the modules deployed on the IoT device for a while. Hence, service interruptions happen in the downtime, in which the generated tasks experience higher delays or even they can be discarded, compared to the tasks that are generated when there is no migration. The IoT users receive smoother results with lower TIT.

Fig. 9 presents the TIT of techniques for ECGMH and EEGTBG in different simulation times. It can be seen that our solution outperforms its counterparts in different simulation times for the ECGMH and EEGTBG. The migration time has a direct impact on the TIT, and the techniques with higher migration time lead to larger TIT. This latter is because as the migration time increases, the number of delayed (or even dropped) tasks grows faster. It can be seen from Fig. 9 that the Urmila results in larger TIT than two other techniques because of its higher migration time. Moreover, due to our smaller migration time, the TIT of our solution is smaller than other techniques for both ECGMH and EEGTBG applications. It is worth mentioning that the TIT of techniques for EEGTBG applications is smaller than of ECGMH ones. This latter is due to a higher data transmission interval for the EEG sensor in EEGTBG compared to the ECG sensor of ECGMH, which means that the number of emitted tasks per second for the EEGTBG application is smaller than the ECGMH application. Hence, applications with shorter task emission interval (here, the ECGMH application) suffer more from higher migration time.

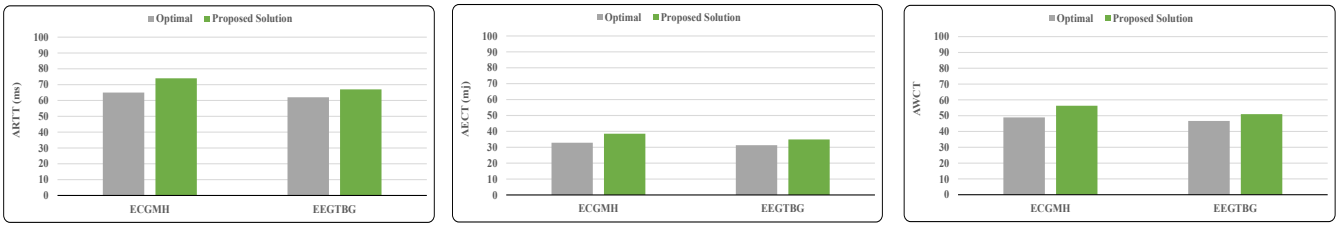
Table IV: Failure Recovery Analysis

Applications	Experiment	Techniques		
		Proposed Solution (FR Mode)	MAAS (No FR)	Urmila (No FR)
ECGMH	Total Number of Migrations	177	234	234
	Total Number of Interrupted Tasks	2095	4152	12302
EEGTBG	Total Number of Migrations	169	227	227
	Total Number of Interrupted Tasks	1228	2504	8361

6) Failure recovery analysis: In this experiment, we study the effect of the failure recovery method in the migration process. The MAAS and Urmila do not have any failure recovery methods and their results are just presented here for comparison purposes. The results of our technique with a failure recovery method (FR Mode) are presented in Table IV when there is a 5% probability of failure in the migration process.

Table IV illustrates that our technique with the failure recovery method (FR Mode) can recover from failures while it still outperforms its counterparts in terms of the total number of migrations and TIT. The obtained results of the average execution cost of tasks and cumulative migration cost in the FR Mode are roughly the same with the Non-FR Mode and they are not provided here. Since the Urmila and MAAS do not have any failure recovery methods, in case of any failures, their placement and/or migration process remains incomplete. However, in our technique, we embedded the failure recovery method for which it accepts a small overhead while it does not stop working if any failures occur.

7) Optimality analysis: In this experiment, we compare the performance of our proposed solution with the optimal values. To obtain the optimal results, we used an optimized version of the Branch-and-Bound algorithm to search all possible candidate configurations for application placement, in which the bounding function helps to faster prune the search space [36]. Since finding the optimal solution is very time consum-



(a) Average Response Time of Tasks (ARTT) (b) Average Energy Consumption of Tasks (AECT) (c) Average Weighted Cost of Tasks (AWCT)

Figure 10: Optimality analysis results

ing, in this experiment, we only consider 20 IoT devices in a hierarchical fog computing environment consisting of 15 candidate servers.

Fig.10 shows the results of optimality analysis in terms of Average Response Time of Tasks (ARTT), Average Energy Consumption of Tasks (AECT), and Average Weighted Cost of Tasks (AWCT). The results show that our solution has an average of 12% difference with the optimal results. However, considering the large number of FSs distributed in the proximity of IoT users, obtaining the optimal solutions, due to their large search spaces, is not practically possible, especially for real-time IoT applications.

VI. Conclusions and Future Work

We proposed a new weighted cost model for minimizing the overall response time and energy consumption of IoT devices in a hierarchical fog computing environment, in which heterogeneous FSs and CSs provide services for IoT devices. In order to enable collaboration among remote servers and provide better services for IoT applications, we proposed a dynamic and distributed clustering technique among FSs of the same hierarchical level. Considering the heterogeneous resources of remote servers and the dynamic nature of such computing environments, we also proposed a distributed application placement technique to place interdependent modules of IoT applications on appropriate remote servers while satisfying their resource requirements. Also, to manage potential migrations of IoT applications' modules among remote servers, due to IoT users' mobility, a distributed migration management technique is proposed. The main goal of this latter is to reduce the migration cost of IoT applications. Finally, we embedded light-weight failure recovery methods to handle possible unpredicted failures that may happen in such dynamic computing environments. The effectiveness of our technique is analyzed through extensive experiments and comparisons by the state-of-the-art techniques in the literature. The obtained results demonstrate that our technique improves its counterparts in terms of placement deployment time, average execution cost of tasks, the total number of migrations, cumulative migration cost of all IoT devices, and the total number of interrupted tasks due to migration.

As part of future work, we will extend our cost model to consider the energy consumption of servers and monetary cost. Moreover, we plan to consider different migration models such as pre-copy, post-copy, and hybrid, and analyze how they affect IoT applications with different resource requirements. Finally, we plan to integrate these techniques in real container-based distributed frameworks such as FogBus2 [14] framework to better analyze proposed techniques in real-world scenarios.

References

[1] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Latency-aware application module management for fog computing environments," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 1, p. 9, 2018.

[2] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, "An application placement technique for concurrent iot applications in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1298–1311, 2020.

[3] M. Goudarzi, M. Palaniswami, and R. Buyya, "A fog-driven dynamic resource allocation technique in ultra dense femtocell networks," *Journal of Network and Computer Applications*, vol. 145, p. 102407, 2019.

[4] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. Leung, "An efficient computation offloading management scheme in the densely deployed small cell networks with mobile edge computing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2651–2664, 2018.

[5] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[6] S. Shckhar, A. Chhokra, H. Sun, A. Gokhale, A. Dubey, and X. Koutsoukos, "Urmila: A performance and mobility-aware fog/edge resource management middleware," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 118–125.

[7] M. Taneja and A. Davy, "Resource aware placement of iot application modules in fog-cloud computing paradigm," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 1222–1228.

[8] A. Kiani, N. Ansari, and A. Khreishah, "Hierarchical capacity provisioning for fog computing," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 962–971, 2019.

[9] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 71–81.

[10] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[11] L. Yang, H. Zhang, X. Li, H. Ji, and V. C. Leung, "A distributed computation offloading strategy in small-cell networks integrated with mobile edge computing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2762–2773, 2018.

[12] S. Jošilo and G. Dán, "Computation offloading scheduling for periodic tasks in mobile edge computing," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 667–680, 2020.
[13] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2019, (in press).
[14] Q. Deng, M. Goudarzi, and R. Buyya, "Fogbus2: a lightweight and distributed container-based framework for integration of iot-enabled systems with edge and cloud computing," in *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, 2021, pp. 1–8.
[15] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 140–147, 2017.
[16] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1002–1016, 2016.
[17] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.
[18] M. Adhikari, S. N. Srirama, and T. Amgoth, "Application offloading strategy for hierarchical fog environment through swarm optimization," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4317–4328, 2019.
[19] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 26–35, 2017.
[20] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, 2019.
[21] Z. Wang, Z. Zhao, G. Min, X. Huang, Q. Ni, and R. Wang, "User mobility aware task assignment for mobile edge computing," *Future Generation Computer Systems*, vol. 85, pp. 1–8, 2018.
[22] C. Yang, Y. Liu, X. Chen, W. Zhong, and S. Xie, "Efficient mobility-aware task offloading for vehicular edge computing networks," *IEEE Access*, vol. 7, pp. 26 652–26 664, 2019.
[23] Z. Liu, X. Wang, D. Wang, Y. Lan, and J. Hou, "Mobility-aware task offloading and migration schemes in scns with mobile edge computing," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–6.
[24] C. Zhu, G. Pastor, Y. Xiao, Y. Li, and A. Ylae-Jaeaeski, "Fog following me: Latency and quality balanced task allocation in vehicular fog computing," in *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2018, pp. 1–9.
[25] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *Future Generation Computer Systems*, vol. 96, pp. 111–118, 2019.
[26] F. Yu, H. Chen, and J. Xu, "Dmpo: Dynamic mobility-aware partial offloading in mobile edge computing," *Future Generation Computer Systems*, vol. 89, pp. 722–735, 2018.
[27] Y. Sun, S. Zhou, and J. Xu, "Emm: Energy-aware mobility management for mobile edge computing in ultra dense networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2637–2646, 2017.
[28] Q. Qi, J. Wang, Z. Ma, H. Sun, Y. Cao, L. Zhang, and J. Liao, "Knowledge-driven service offloading decision for vehicular edge computing: A deep reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 5, pp. 4192–4203, 2019.
[29] D. Wang, Z. Liu, X. Wang, and Y. Lan, "Mobility-aware task offloading and migration schemes in fog computing networks," *IEEE Access*, vol. 7, pp. 43 356–43 368, 2019.
[30] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 778–790, 2020.
[31] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: a performance evaluation," *Sensors*, vol. 19, no. 7, p. 1488, 2019.
[32] M. Goudarzi, Z. Movahedi, and M. Nazari, "Mobile cloud computing: a multisite computation offloading," in *2016 8th International Symposium on Telecommunications (IST)*. IEEE, 2016, pp. 660–665.
[33] X. Xu, Q. Liu, Y. Luo, K. Peng, X. Zhang, S. Meng, and L. Qi, "A computation offloading method over big data for iot-enabled cloud-edge computing," *Future Generation Computer Systems*, vol. 95, pp. 522–533, 2019.
[34] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality mmogs," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–14.
[35] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, no. 4, pp. 51–56, 2010.
[36] M. Goudarzi, M. Zamani, and A. T. Haghighat, "A fast hybrid multi-site computation offloading for mobile cloud computing," *Journal of Network and Computer Applications*, vol. 80, pp. 219–231, 2017.