PROACTIVE AND DYNAMIC TASK SCHEDULING IN FOG-CLOUD ENVIRONMENT

Hoang Phan

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 2019

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Hoang Phan**

Entitled: **Proactive and Dynamic Task Scheduling in Fog-cloud Environment**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. H. Harutyunyan _____ Chair

Dr. Y.-G. Gueheneuc _____ Examiner

Dr. D. Goswami _____ Examiner

Dr. B. Jaumard _____ Supervisor

Approved _____
Dr. Volker Haarslev, Graduate Program Director

_____ 20 _____ _____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Proactive and Dynamic Task Scheduling in Fog-cloud Environment

Hoang Phan

Fog computing was introduced for the first time by Cisco in 2012. Since then, there has been a great number of studies on fog computing, in which vacant and free-of-charge computing resources in local networks provide low-latency services to end devices. Even though traditional architecture with scalable and powerful central servers in cloud can accommodate those tasks, it is costly to allocate resources in cloud to execute all those tasks. In addition, it falls short of satisfying Quality of Service (QoS) requirements in terms of waiting time because of long distance communication between servers and user end devices.

In this thesis, we discuss dynamic scheduling problem in fog-cloud collaboration environment for real-time applications when QoS is strict and when an answer is useless if the corresponding application finishes its execution after a pre-defined deadline. By taking into account an admission control procedure to grant only requests whose deadline requirements are feasible with respect to the available resources in the network, we study a proactive scenario using different strategies to calculate schedules and to assign resources, within the admission control procedure to accommodate an incoming request. Then, we propose our heuristic with four variants corresponding to four different strategies, with the adjustment of a trade-off cost-makespan factor in an utility function. When evaluating performance with some baseline methods in such proactive scenario, the numerical results show that our variants can meet deadline requirements for more applications while exploiting more efficiently the resources in the fog layer and being charged less for using cloud.

Keywords: fog computing, cloud computing, dynamic scheduling, real-time scheduling, task scheduling, workflow applications, DAG, QoS requirements, heterogeneous systems.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Brigitte Jaumard for all of her support, guidance, patience and kindness during my time at Concordia University work on my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

As Internet-of-Things (IoT) has gradually become one of the most important technological evolutions in the last decade, the number of connected devices has increased tremendously and it is expected to reach 20 billion in 2020 (Gartner, Inc. 2017). This trend generates not only a huge volume of data that must be transferred over the network but also makes the quality of service (QoS) become much more demanding. For example, augmented reality (AR) and virtual reality (VR) applications require a great amount of computation work and also highly responsive communication to provide acceptable experience to users. Even though common devices such as smart phones, laptops, tablets developed remarkably in the last few decades, they are still limited in battery life, computation and storage capacities to offer good user-experience when executing computation intensive or resource-consuming tasks. Thus, there is a need of offloading those high-demanding tasks to another workstation whose computation and storage capacity are more powerful and available. Although cloud computing, with its flexibility to scale on-demand (on both computation and storage), can accommodate fairly the execution overhead and storage shortage issues, it would be expensive to serve a huge number of requests using cloud resources. It would also fall short of offering adequate experience to users due to round-trip communication between end-devices and remote servers in the cloud. Fog computing (FC) can come into play to overcome those limitations of traditional central cloud computing architecture as it brings computation and data storage closer to end devices.

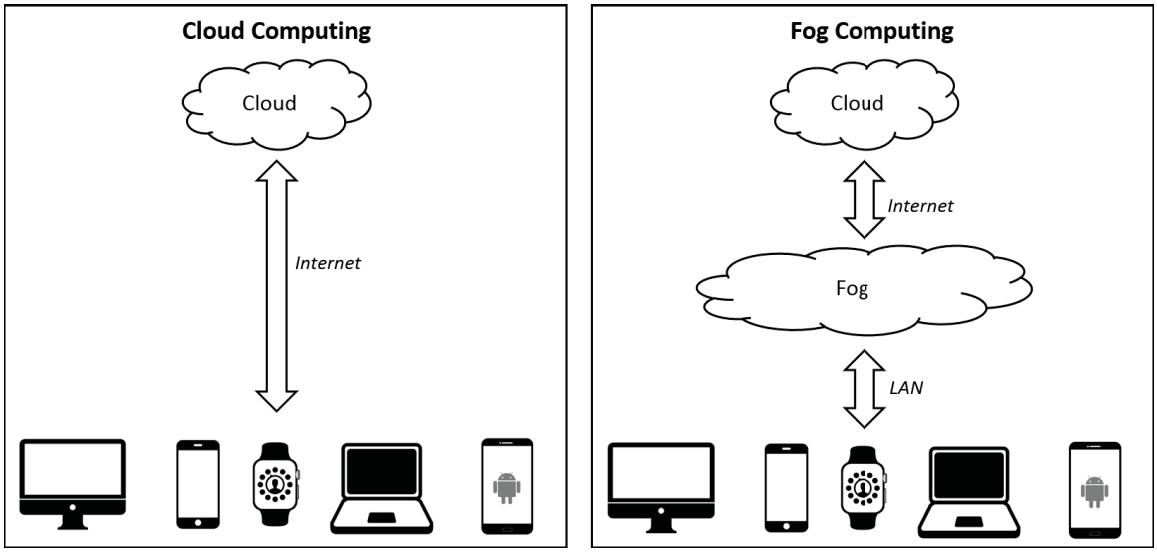

Figure 1: Traditional Cloud Computing vs. Fog Computing

Fog computing, first introduced by Cisco, is an additional infrastructure layer, which consists of network devices at the edge, that resides between IoT devices and remote servers on cloud. As a complementary layer to cloud computing, resources in the fog are usually not as powerful as those in the cloud but they are more capable than end-user devices, thus, can be responsible for serving part of the requests that arrive at the network. This paradigm, hence, eliminates the needs of sending requests and transferring data all the way long to the core servers on cloud to be served, which helps improve the responsiveness and ease the burden to handle an enormous number of requests for the cloud.

In general, each request, representing the application it is constructed from, that is sent from an end-user device usually consists of a set of tasks that must be executed. Those tasks can either be independent (bag of tasks) or dependent (workflow) on each other (Mach et al. 2017). In the case where tasks are independent, they can be offloaded separately to resources in the network to be executed simultaneously and in parallel. However, in the case where requests are workflows, most of the time tasks depend on input from other tasks and they cannot start until all of predecessors finish execution. In such scenario, we need to consider inter-task data dependencies when scheduling tasks onto resources in the system and executing tasks in parallel is not always applicable in this case.

Requirements/Features	Cloud Computing	Fog Computing
Latency	High	Low
Network Access Type	Mostly WAN	LAN (WAN)
Server Location	Anywhere within the network	At the edge
Mobility Support	Low	High
Distribution	Centralized	Distributed
Task/Application Needs	Higher computation power	Lower latency
User Device	Computers, mobile devices (limited)	Mobile-smart-wearable devices
Management	Service Providers	Local Business
Number of Servers	High	Low

Table 1: Characteristics of Cloud Computing and Fog Computing Comparison (extracted from Baktir et al. 2017)

1.2 Motivation Scenario

With all the advantages that fog computing can offer over traditional system infrastructures, there are a good number of use cases that can benefit from deploying fog computing into the network such as the following scenario. A private organization offers a wide range of internal services that are accessible through mobile devices such as smart phones, tablets, wearable devices, etc. In particular, the services that the organization offers are time-critical, in other words, they are sensitive to timing constraints such that if an application fails to finish the execution within the required time frame, the response will become useless. Because the services are for internal use, users will not need to pay anything to use the services and it is the organization who will be charged for anything occurring while providing the services. Because of such characteristics, the organization realized that having traditional centralized servers on cloud would not be a good solution since it would degrade considerably the quality in the service that it offers to users while having to pay a great sum that is charged for using resources in the cloud. As a result, the organization decided to exploit computation resource of devices in proximity with end users such as routers, switches, sensors, etc. available in the local area network (LAN). Thus, with this intermediate layer of powerful processors, instead of having all requests from users being sent to cloud servers to be processed, part of them will be served within the local network which helps to offer the services more responsively to users and remarkably reduce the cost of using external resources in the cloud. However,

because those local devices are not specialized to mainly provide high level services, they are usually not as powerful as servers in cloud. With an ultimate goal to meet deadline constraints of as many applications as possible for less money, the organization expects to compromise cloud cost and execution time rather than only aims to minimize application's makespan all the time.

1.3 Thesis Contributions

To solve the problem in the example scenario above, in this research, we address the problem of dynamic task scheduling in a real-time scenario where we will schedule multiple applications, in which each application is in form of dependent tasks and represented as a graph, onto a heterogeneous fog-cloud collaboration environment without prior knowledge of upcoming IoT requests from users. Specifically, we propose a proactive scenario in which there is a request-admission control procedure to anticipate whether an application is feasible for the system to meet its pre-defined time constraints or not, using multiple strategies, before really allocating tasks to computing resources in the network. Then, we introduce a heuristic that computes application schedules using an utility function, in which, just by adjusting value of a cost-makespan factor β, we can obtain a schedule that either favors monetary cost rather than execution time or vice versa. The main objective of our heuristic is to maximize the guarantee ratio of accepted applications whose deadline requirements are satisfied while making good use of resources in the fog layer and minimizing cloud cost. To evaluate how our approach performs compared to other methods using simulation, we introduce 4 variants whose values of β are different to represent 4 scheduling strategies. Last but not least, we also implemented a generator to create data sets of simulated network traffic for evaluation purpose.

1.4 Plan of the Thesis

The remainder of this paper is organized as follows: Chapter 2 summarizes some related articles in the literature regarding task scheduling in heterogeneous computing environments as well as in the domain of fog computing. Later, chapter 3 provides more details regarding the system infrastructure with the problem models that we take into account.

Also, in chapter 3, we explain our proposed method as well as our proactive scenario. Then, we introduce our data generator in chapter 4. Finally, we give description of how we evaluate the performance of our method compared with others in Chapter 5, followed by our conclusion in the last chapter.

Chapter 2

Literature Review

This section discusses various task scheduling methods in heterogeneous systems in general and then introduces some popular papers on fog computing.

Overall, task scheduling in heterogeneous systems, which consist of a combination of machines that have different characteristics or capacities, is a popular research topic that has been tackled by many researchers. As Ullman 1975 proved that task scheduling in heterogeneous system is an NP-complete problem, researchers are expected to find heuristic methods that produce task assignments to processors that are close to exact solutions. In the context of static scheduling in which information regarding computation work and data dependencies of tasks are given beforehand, one of the most popular heuristics is HEFT (Heterogeneous Earliest Finish Time) proposed by Topcuoglu et al. 2002, which consists of 2 phases namely *task prioritizing* and *processor selection*. The general idea of HEFT is to minimize execution time of a single application by trying to assign each task, starting from the highest prioritized one, to a processor that has the minimum combined cost of communication time and computation time. HEFT does not take into account other constraints such as monetary cost of using cloud resources, deadline requirements, etc. Bittencourt et al. 2008 addressed the task scheduling problem in a heterogeneous system using a method called PCH (Path Clustering Heuristic). The authors proposed a clustering strategy in which a path of the workflow, consisting of a cluster of dependent tasks, will be scheduled for execution on a same processor to minimize communication cost and improve the performance of executing the whole workflow.

Various studies address the task scheduling problem in a dynamic context with multiple applications. For example, Qin et al. 2005 presented two methods, namely DALAP and DASAP, that schedule application tasks with earliest deadline first, which leads to local optimal solutions that likely increase the chance of shortening execution time of the whole application. The authors applied an admission-control mechanism to see whether an request arriving at the network is feasible for the system to finish its execution before the deadline or not before dispatching it, which avoid using resources for executing infeasible requests. Stavrinides et al. 2011 proposed a list-scheduling approach in which different bin-packing techniques are used during the processor selection phase to select potential idle time slots (schedule holes) to execute tasks to achieve the highest ratio of applications that meet their latency requirements. These methods can be considered classic task scheduling heuristics in heterogeneous systems, the authors did not take into account any collaboration of fog or cloud computing, which usually have constraints in monetary cost.

Even though task scheduling in fog-cloud environment is relatively new compared to other computing systems, it has been widely investigated in recent years considering multiple requirement factors (i.e., energy consumption, user-defined budget, etc.) and in various scenarios of fog computing. One of the first attempts that addresses workflow-based task scheduling in fog-cloud environment is CMaS (Cost-Makespan aware Scheduling) proposed by Pham et al. 2017. Instead of focusing only on application makespan, CMaS also takes into consideration the cost that would be charged for using resources in the cloud for computation and communication. By considering the two constraints at the same time, the authors of CMaS assemble a balance schedule between monetary cost and makespan, which is under a user-defined deadline constraint. However, this approach only intends to schedule tasks in the context of static scheduling with single application, that is not compatible with the dynamic nature of IoT traffic that fog-cloud systems usually need to deal with.

In contrast to static scheduling, there are not so many studies considering the dynamic context with multiple real-time applications in fog-cloud-like environments. In Stavrinides et al. 2018, a heuristic called Hybrid-EDF is proposed to address the scenario of dynamic scheduling in fog-cloud system for multiple real-time applications, where there is no knowledge about future tasks, taking into consideration cloud cost that is charged for computation and communication and prescribed deadline. In this

method, the authors allocate computation-intensive tasks with low communication demands to servers in the cloud while scheduling communication-intensive tasks with low computation requirements to resources in the fog. Hybrid-EDF can utilize possible gaps in the schedule and exploit idle resources in the network to accommodate multiple applications. However, even though the paper proposed a scenario whose models are close to real life fog-cloud environments, Hybrid-EDF does not make use of resources in the system really efficiently. For all applications that arrive at the network, the heuristic calculates optimal execution location for each task and then allocates them to resources for execution immediately without considering whether it is feasible for the system to satisfy QoS requirements of the applications or not, which could lead to a waste of resources, and also money if those resources are in the cloud. Table 2 summarizes how task scheduling references in the literature differ from our heuristic.

Reference	Environment	Application type	No. of applications	Static/ dynamic	Min. makespan	Min. monetary cost	Tradeoff makespan-cost	Deadline constraint	Admission control	Multiple attempts
Topcuoglu et al. 2002	Heterogeneous	Workflow	Single	Static	Yes	No	No	No	No	No
Qin et al. 2005	Heterogeneous	Workflow	Multiple	Dynamic	Yes	No	No	Yes	Yes	No
Stavrinides et al. 2011	Heterogeneous	Workflow	Multiple	Dynamic	Yes	No	No	Yes	No	No
Pham et al. 2017	Fog and cloud	Workflow	Single	Static	Yes	Yes	Yes	Yes	No	Yes
Stavrinides et al. 2018	Fog and cloud	Workflow	Multiple	Dynamic	Yes	Yes	No	Yes	No	No
Our heuristic	Fog and cloud	Workflow	Multiple	Dynamic	Yes	Yes	Yes	Yes	Yes	Yes

Table 2: References of Task scheduling in the literature and our heuristic

Chapter 3

Our Heuristic

3.1 Problem Formulation

3.1.1 System infrastructure

Following the scenario mentioned above, the system infrastructure can be divided into three main layers:

Cloud layer: this is the highest layer in which there is a set of servers with different specifications that are hosted remotely by third-party cloud service providers such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud. Within the system infrastructure, the servers in the cloud layer usually have the most powerful processors with the largest storage and memory capacity so they can execute all kinds of requests from users. Furthermore, each server in the cloud layer has multiple processing cores and we assume that there is one VM running on each core. It is possible to have multiple tasks running on a same server in the cloud in that a task will be executed on a separate VM.

Fog layer: this layer resides in the middle, between the other two layer, consists of a set of heterogeneous processing nodes that have specialized capability of serving most of requests in the network. Processing nodes in this layer can be devices within the local network such as routers, switches, sensors, etc. As they are not made to primarily execute application tasks from users, these complementary resources have limited computational capacity as well as shortage of storage and memory. To facilitate collaboration between resources within the network, we assume that these resources are connected with servers in the cloud over Internet. In addition, processing nodes in the

fog layer are single core which means there will be at most one task to be executed on a processing node at a time.

End-user devices layer: this is the bottom-most layer representing end-user devices such as smartphones, wearable devices or tablets. For simplification, each device runs an application with a set of tasks that need to be offloaded to higher levels to be executed to meet QoS requirements.

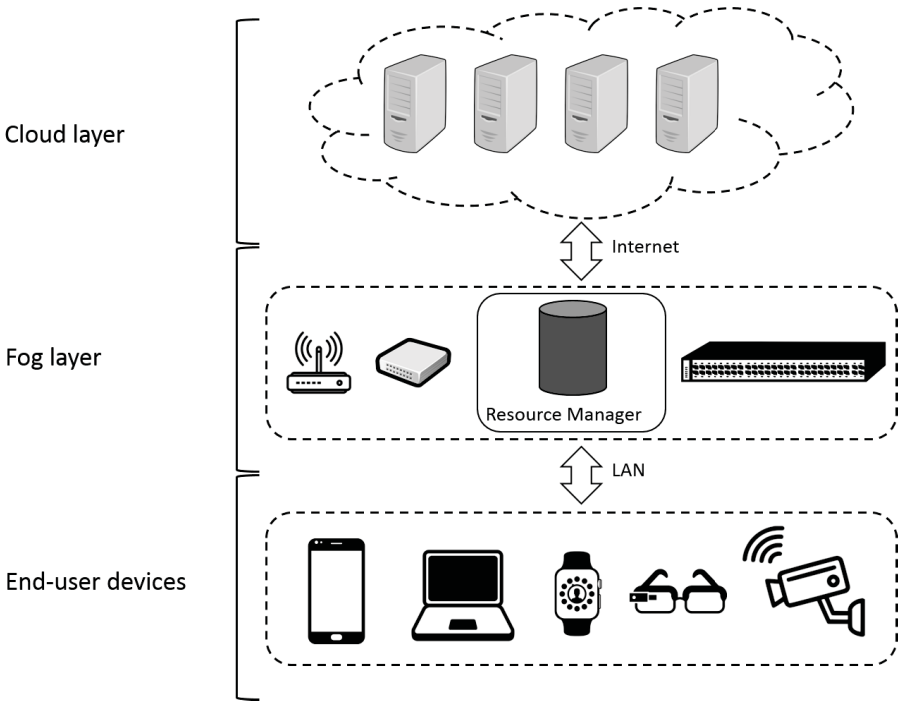

Figure 2: System architecture

As discussed in the literature (Pham et al. 2017, Qin et al. 2005), our infrastructure has a component that we call *resource manager* (RM) responsible for managing resources and scheduling requests arriving at the network. Before being offloaded to resources in the network, all requests will be submitted to this component with all information about computation work and storage needed for execution. Moreover, with the ability of collecting frequently status of machines in the network, we assume that RM, in which there is an admission control that will be explained further in this section, will have enough information to calculate schedules and anticipate whether deadline requirements of applications can be met. Only feasible applications, whose deadline can be satisfied, will be dispatched to designated resources for execution and will be rejected without being sent to resources. For simplification, we assume that scheduling time and dispatching

time are insignificant and these factors will not be accumulated toward application's makespan.

Since the fog layer is actually part of the local network (LAN), it is theoretically more stable and robust compare to the Internet, we assume connections between devices in the middle layer are much more responsive and stable than those between devices belong to different layers where communication is done via Internet.

3.1.2 Problem model

In general, the problem of scheduling a task graph that consists of precedence constrained tasks on a set of given resources in a network is a problem of mapping each task on a resource such that we can obtain the optimal value of an utility function.

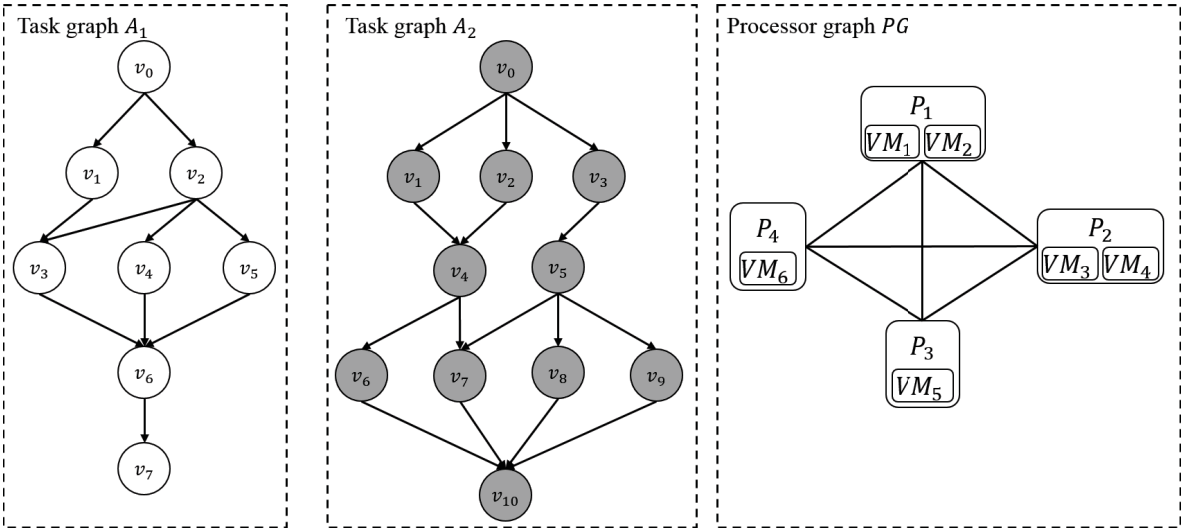

Figure 3: An example of task graphs and processor graph

Processor graph

A processor graph PG, consists of a set of processors in the cloud layer N_{CLOUD} and the fog layer N_{FOG}, it is a complete graph in which there is always a connection between any two processors. Each processing node P_i is characterized by:

- Number of cores (or number of VMs) NCORE_i
- Processing rate PROC_i
- Upload bandwidth BWU_i

- Download bandwidth BWD_i
- Cost per time unit COST_i for using a VM on P_i

In our model, the processing nodes in the cloud layer have at least 2 cores whereas those in the fog layer have only 1 core. By offloading tasks to the cloud layer, we execute different tasks simultaneously on different VM, that are running on different cores of a same processor, which help increasing the overall throughput of the scheduling without affecting execution of other concurrent tasks. Furthermore, all VMs running on all processing cores on processor P_i will have the same processing rate PROC_i and communication between different tasks on a same processor, either on a same or different VMs, is negligible, thus will be counted as zero. We also take into account the considerable difference in communication time between processors belonging to the same or different layers by having local bandwidth BW_{FOG}=1Gbps and BW_{CLOUD}=250Mbps which will be applied for data transfer between any two nodes within the fog layer and the cloud layer, respectively.

Task graph

Each input application $a_i \in A$ in our model is represented by a directed acyclic graph (DAG) $G = (V, E)$ in which each node $v_i \in V$ represents a task and each edge $e_{ij} \in E$ between two nodes v_i and v_j denotes the dependency constraints between them. Each node v_i in the task graph has a non-negative weight c_i represents for number of instructions as computation works of the task. The time that a VM VM_j would need to execute task v_i is then given by:

$$w(v_i, \text{VM}_j) = c_i / \text{PROC}_j \tag{1}$$

Each edge e_{ij} has a non-negative weight d_{ij} denotes the amount of data transferred from v_i to v_j before task v_j getting executed. Communication cost for transferring data of d_{ij} between VM_s (where task v_i is scheduled on) and VM_d (where task v_j is scheduled on) is then defined as:

$$c(d_{ij}, \text{VM}_s, \text{VM}_d) = \begin{cases} 0, & \text{if } s = d \\ d_{ij} / \text{BW}(s, d), & \text{if } s \neq d \end{cases} \tag{2}$$

where $\text{BW}(s, d)$ is the bandwidth for the communication from VM_s to VM_d. It is determined as:

$$\text{BW}(s,d) = \begin{cases} \text{BW}_{\text{FOG}} = 1\text{Gbps}, & \text{if } \{s,d\} \in N_{FOG} \\ \text{BW}_{\text{CLOUD}} = 250\text{Mbps}, & \text{if } \{s,d\} \in N_{CLOUD} \\ \min(\text{BWU}_s, \text{BWD}_d), & \text{otherwise} \end{cases} \tag{3}$$

A task that has no predecessors is called *entry task*, whereas a task without any successors is called *exit task*, denoted as v_{entry} and v_{exit}, respectively. A task can only be executed once all the precedent tasks finish and all data dependency has arrived at the scheduled processing node for that task.

Regarding QoS requirements, each task itself does not have any particular deadline but each application a_i has a constraint DELAY_i which denotes the maximum delay that the user who sends a_i can wait for the network to process it after arriving at the network at time ARRIVAL_i. An application needs to finish execution within a required timeframe and any response outside the timeframe will be considered useless. We assume that communication time to send results back to end devices is negligible thus counted as zero since response data is relatively small. As a result, to have application a_i meet its deadline requirement, finish time of all exit tasks of a_i is also finish time of a_i and will be defined as:

$$\text{DEADLINE}_i = \text{ARRIVAL}_i + \text{DELAY}_i \tag{4}$$

In this thesis, the terms of *application*, *request*, *task graph* and *DAG* are used interchangeably.

Monetary cost

We only consider monetary cost that is charged whenever we assign tasks to VMs in the cloud layer as they are hosted by third-party service providers. For simplification, the cost of using cloud resources will be calculated based on the time that the VMs on those resources are occupied for computation. We assume that the monetary cost for using resources in the fog layer is zero.

3.2 Proactive Scenario and Compromised Makespan-Cost Ratio Heuristic

3.2.1 Proactive Scenario

Unlike the strategy Hybrid-EDF proposed in Stavrinides et al. 2018 and the majority of heuristics in task scheduling that consider multiple applications scenario, we will not allocate tasks immediately to selected execution locations before knowing that the network can finish executing the whole application before its deadline. In our proposed methods as well as other heuristics that we use to evaluate performance, before allocating any application that arrives at the network to the resources, we will let them go through an admission control procedure. This admission control procedure will anticipate if an application can finish before deadline by calculating makespan of the application when allocating each task to a VM such that we would obtain the optimal value of an utility function for each task. As we anticipate how allocate tasks to resources in the network, we also take into account the state of the network with other tasks that are currently being executed as well as how those resources would be occupied with tasks of the current application that are being anticipated to be scheduled. After having calculated all the tasks of current application on how they would be allocated onto the resources of the network, if all exit tasks of the application in the anticipated schedule finish before its pre-defined deadline then the application will be marked as "Accepted". The application will be allocated into the network for real execution. Otherwise, the application will be considered as "Rejected". Our algorithm will avoid wasting resources of the network spending on executing requests whose QoS requirements cannot be satisfied.

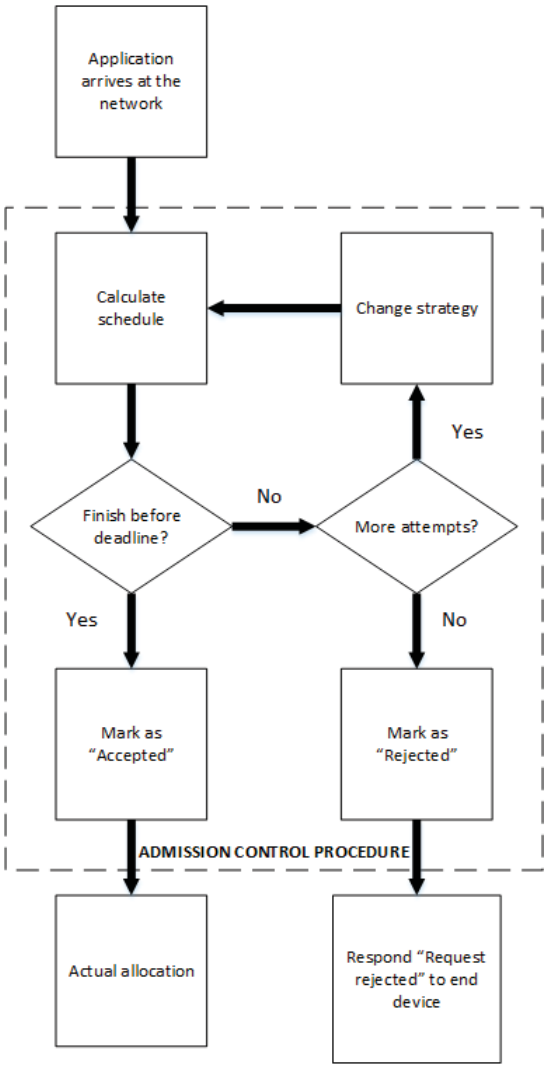

Figure 4: Flowchart of our proactive scenario

As we know beforehand whether QoS requirements of application can be met or not without really executing any application tasks on resources, users will not wait until part of their applications (or even the whole applications) get executed to get a response from the system if their requests are feasible or not. For simplification, we assume that the machine of RM is well equipped with good capacity such that calculation time that it will need to compute schedules and forward information of applications from users to resources is negligible which can be counted as zero. With this assumption, we consider a proactive scenario in which it is possible to use different strategies in multiple attempts to accommodate an application when it fails (being marked as "Rejected") to finish the execution within the required timeframe in earlier attempts. However, we are also aware

that the number of attempts must not be impractical, thus, should be limited, as it may cause severe delay and will degrade overall performance of examined applications. Hence, we assume that the delay caused by calculation of schedule remain negligible when the number of maximum attempts allowed in the admission control procedure is not greater that 3.

After having reached the maximum number of allowed attempts, if none of proposed schedules that are computed by the scheduler component of RM can make an application finish execution before its deadline, the application still remains "Rejected". The process of how the admission control grants an application is illustrated in the flowchart in Figure 4.

To compromise the outcome schedule with makespan and cloud cost, we proposed a heuristic with several variants that basically apply a same formula of utility function but with different values of a makespan-cost factor. By using this compromising factor, we specify the inclination level on makespan and monetary cost when selecting execution location of tasks. As we attempt to change strategies in turn in order to fit requests into the network, we solely adjust the value of makespan-cost factor in the utility function to favor more execution time.

3.2.2 Admission Control Procedure (Computing Schedule)

In the admission control procedure, different task scheduling strategies can be examined in turn to anticipate a suitable schedule for a given application into the network at a specific time using simulation. In general, there are two primary phases namely *task selection* and *VM/processor selection* in our heuristic. In the task selection phase, scheduling priority of every task will be computed and each task will then be selected by the scheduler starting from the one with the highest priority. While in the VM/processor selection phase, the scheduler will determine execution location of a selected task to achieve the optimal value of our objective function.

Task selection phase

In this phase, each task v_i will be given a scheduling priority recursively defined as:

$$pri(v_i) = \begin{cases} \overline{w(v_i)} + \max\limits_{v_j \in succ(v_i)} (\overline{c(e_{ij})} + pri(v_j)), & \text{if } v_i \neq v_{exit} \\ \overline{w(v_i)}, & \text{if } v_i \equiv v_{exit} \end{cases} \quad (5)$$

in which $\overline{w(v_i)}$ is the average time that a processor/VM in the network take to execute task v_i and $\overline{c(e_{ij})}$ is the average time to transfer e_{ij} between two processors/VMs in the network. These average factors are calculated as:

$$\overline{w(v_i)} = \frac{c_i}{\overline{\text{PROC}_i}} \tag{6}$$

$$\overline{c(e_{ij})} = \frac{d_{ij}}{\overline{\text{BW}}} \tag{7}$$

where the mean processing rate $\overline{\text{PROC}_i}$ and the mean bandwidth $\overline{\text{BW}}$ of the fog-cloud environment are defined as follow:

$$\overline{\text{PROC}_i} = \frac{(\sum_{P_i \in PG} \text{PROC}_i \times \text{NCORE}_i)}{\sum_{P_i \in PG} \text{NCORE}_i} \tag{8}$$

$$\overline{\text{BW}} = \frac{\sum_{P_i \in PG} \text{BWU}_i + \sum_{P_i \in PG} \text{BWD}_i}{2 \times |PG|} \tag{9}$$

with $|PG|$ is the number of processors in the network.

As the priority, also called upward-rank, of each node in the task graph depends on its child nodes, the precedence constraints of task graphs will then be preserved when we sort all the tasks in a non-increasing order of node priority.

VM selection phase

Similarly to the scheduling mechanism of HEFT, we use insertion-based policy that takes into account available time slots at which resources are idle to allocate tasks to for execution. A task v_i can only start its execution if all of its predecessor tasks finish their execution and transferring data as input to the selected VM of v_i. Let $t_f(v_i, \text{VM}_n)$ be the finish time of task v_i on VM_n. Ready time $t_r(v_i, \text{VM}_n)$ of task v_i on VM_n, at which all the required data as input for v_i from its predecessors has arrived at VM_n, is defined as:

$$t_r(v_i, \text{VM}_n) = \max_{v_j \in pred(v_i)} [t_f(v_j, \text{VM}_m) + c(d_{ji}, \text{VM}_m, \text{VM}_n)] \tag{10}$$

For entry task v_{entry}, there is no need for it to have any input from other tasks to start the execution, $t_r(v_{entry}, \text{VM}_n)$ will either be the arrival time of the application ARRIVAL or the earliest idle time slot on VM_n that can host v_{entry}.

Then, the main job in VM selection phase for each task v_i is to look for a VM whose earliest idle time slot can host the execution of v_i (an example of an idle time slot that can host execution of a task is shown in Figure 5) such that our utility function can achieve the optimal value with that selection. Let $EST(v_i, \text{VM}_n)$ and $EFT(v_i, \text{VM}_n)$ be the earliest start time and earliest finish time of task v_i on VM_n, respectively. For an idle time slot $[t_A, t_B]$ on VM_n to host task v_i, there must be no task being executed within this time slot on VM_n and

$$\max\{t_A, t_r(v_i, \text{VM}_n)\} + w(v_i, \text{VM}_n) \leq t_B \tag{11}$$

$EST(v_i, \text{VM}_n)$ and $EFT(v_i, \text{VM}_n)$ are then defined by:

$$EST(v_i, \text{VM}_n) = \max\{t_A, t_r(v_i, \text{VM}_n)\} \tag{12}$$

$$EFT(v_i, \text{VM}_n) = EST(v_i, \text{VM}_n) + w(v_i, \text{VM}_n) \tag{13}$$

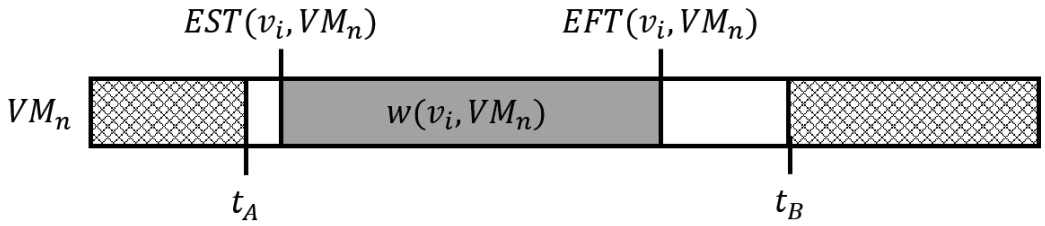

Figure 5: An example of an idle time slot $[t_A, t_B]$ on VM_n that can host task v_i

Our proposed method also takes into account monetary cost of using cloud resources when a task is allocated to a VM in the cloud layer. As we assume that the monetary cost charged for communication to and from cloud resources is much less than that of the computation, we only consider the cost for the time a selected VM in the cloud layer will be occupied for execution. The cloud cost of executing task v_i allocated to VM_n is:

$$cost(v_i, \text{VM}_n) = \text{COST}_n \times w(v_i, \text{VM}_n) \tag{14}$$

After having computed all earliest finish time of a task on all VMs in the network as well as monetary cost that would be charged for using those computing resources, we will choose a VM from which we can obtain the optimal trade-off value for our utility

function for a specific task v_i. The utility function to determine the trade-off value between monetary cloud cost and execution time of task v_i on VM_n is defined by:

$$\begin{aligned} U(v_i, \text{VM}_n) = \beta \times \frac{cost(v_i, \text{VM}_n)}{\max\limits_{\text{VM}_m \in PG} cost(v_i, \text{VM}_m)} \\ + (1-\beta) \times \frac{EFT(v_i, \text{VM}_n) - \min\limits_{\text{VM}_m \in PG} EFT(v_i, \text{VM}_m)}{\max\limits_{\text{VM}_m \in PG} EFT(v_i, \text{VM}_m) - \min\limits_{\text{VM}_m \in PG} EFT(v_i, \text{VM}_m)} \end{aligned} \tag{15}$$

where β $(0 \leq \beta \leq 1)$ is the makespan-cost factor reflecting the inclination between monetary cost of using the cloud and execution time when selecting execution location of a task. Execution location of task v_i will then be the VM that has the minimum value of $U(v_i, \text{VM}_n)$. When $\beta = 0$, the heuristic become a dynamic version, in the context of real-time applications, of HEFT in such all the VMs that will be selected will only favor execution time to minimize overall application's makespan. At the other extreme, only resources in the fog layer will be selected to execute tasks when $\beta = 1$ they always have $U(v_i, \text{VM}_n) = 0$. The utility function can achieve a balance trade-off value when monetary cost and execution time have equal weights in the formula, thus the neutral value of β is 0.5.

After having task v_i to be scheduled on VM_n, the earliest start time $EST(v_i, \text{VM}_n)$ and the earliest finish time $EFT(v_i, \text{VM}_n)$ will equal to the actual start time and the actual finish time of task v_i, which are denoted as $AST(v_i)$ and $AFT(v_i)$, respectively. Then, after having scheduled all the tasks of a task graph, the makespan of that task graph is $AFT(v_{exit})$.

An application will only be marked "Accepted" if its exit task finishes before DEADLINE_i and then, all the tasks of the application will be scheduled accordingly as computed. Otherwise, the application will be marked "Rejected" without wasting any resources on executing the whole task graph. An illustration of output representing how tasks are assigned to processors is shown in Figure 6.

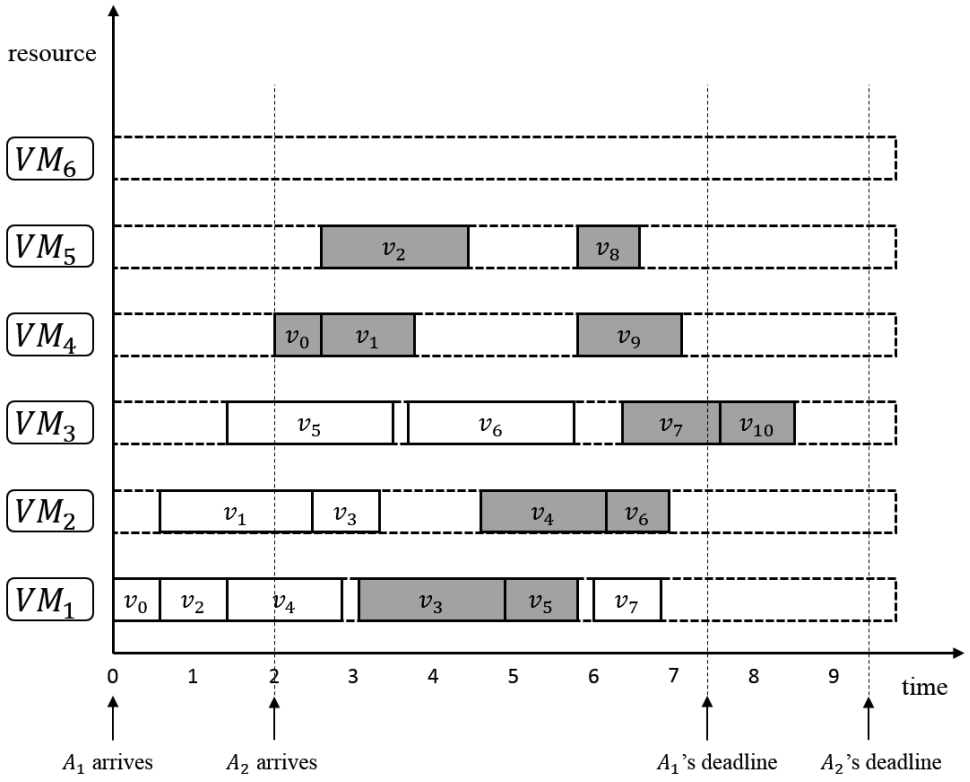

Figure 6: An example of task scheduling for applications given in Figure 3

3.2.3 Adjusting the Value of Makespan-Cost Factor β

In this paper, we examine 4 variants of our heuristic and they differ from one and another on which value of β to start with, how to adjust β in latter attempt(s) and the number of maximum attempts allowed in the admission control procedure.

Firstly, from the definition of the utility function, we can see that the higher the value of β is, the more the function will become dependent on monetary cost, which means if we would want to have requests to be executed within a shorter period of time, we should decrease the value of β to make selections of VM less sensitive with monetary cost and vice versa. Generally, the strategy of our heuristic is to always start first with a higher value of β to aim for less expensive execution locations. If then the application fails to finish its execution before deadline, a smaller value of β will be chosen which should help produce a new schedule whose makespan is expected to be smaller in an effort to meet time constraints. By starting to accommodate each application with a high value of β first, we expect to obtain more cost-effective schedules before adjusting

β to a lower value to favor more execution time. In the last attempt, for all the variants of our heuristic, we use $\beta = 0$ as the last resort, which will likely help produce a schedule whose makespan we consider as the shortest that our heuristic can possibly construct to accommodate an application.

Regarding how to adjust the value of β if deadline requirements cannot be met in the first attempt, we examine the possibilities of adjusting either manually or automatically. In the case where β will be adjusted manually, other values of β in latter attempts are specifically defined in advance whereas on the other hand, new value of β will be determined using a formula based on the difference between makespan produced by previous strategy and application's deadline. As an attempt to adjust automatically, we propose a formula to estimate the right value of β as follows:

$$\beta = \begin{cases} 0 & \text{, if } \textsc{lateness} > 1 \\ (1 - \textsc{lateness})/2 & \text{, otherwise} \end{cases} \tag{16}$$

where LATENESS is defined as:

$$\textsc{lateness} = \frac{(AFT(v_{exit}) - \textsc{arrival}) - \textsc{delay}}{\textsc{delay}} \tag{17}$$

Finally, by investigating different numbers of maximum attempts to accommodate applications, we will evaluate whether the heuristic could grant notably more requests if more strategies are allowed in the admission control procedure. All the parameters of the 4 variants, namely ProactiveCUHEFT, ProactiveCMKCRHEFT, ProactiveCMKCR-manual, ProactiveCMKCR-auto, as well as other approaches for evaluation purpose are specified in Table 5 and will be discussed further in Chapter 5.

Chapter 4

Data Instances

We now describe a generic generator of data instances for FC. Each data instance has two key parts: the FC infrastructure and FC traffic.

4.1 Network Infrastructure

The network infrastructure that we use for experimenting and evaluating will be constructed by different types of processors with quantity for each type can be found in Table 3. As being mentioned in problem model section, each core of a processing node can run a separate VM thus number of cores of a processing node is equal the number of running VMs on that node. Hence, there are in total 12 VMs in the fog layer while there are 18 VMs belong to the cloud layer. In addition to different bandwidth constants (BW_{FOG}=1Gbps and BW_{CLOUD}=250Mbps) in the two layer of processors, upload bandwidth and download bandwidth of each processing node will be generated randomly in the range of [50-100 Mbps]. Those bandwidths will be used to calculate communication time that processing nodes will need to transfer data between those that are in different layers as it will be transmitted over the Internet.

	Processing rate	Number of cores	Cost per time unit	Quantity
Fog type 1	1.6 GHz	1	0	3
Fog type 2	1.8 GHz	1	0	3
Fog type 3	2.0 GHz	1	0	3
Fog type 4	2.2 GHz	1	0	3
Cloud type 1	2.6 GHz	2	1.0	1
Cloud type 2	2.6 GHz	4	1.3	1
Cloud type 3	3.0 GHz	2	1.6	1
Cloud type 4	3.0 GHz	4	1.9	1
Cloud type 5	3.4 GHz	2	2.3	1
Cloud type 6	3.4 GHz	4	2.6	1

Table 3: Characteristics of processing nodes in the fog layer and the cloud layer

4.2 FC Traffic

We will define the FC traffic by a set of applications in which each application will be represented by a task graph, with precedence constraints on some tasks, while other tasks can be run in parallel.

4.2.1 Characteristics of the FC Traffic

The traffic to evaluate performance of the proposed method will be characterized by its overall number of applications (n^{APPS}) to be scheduled as well as the application arrival gap (γ) which denotes the gap of time between applications that will arrive at the network, i.e., one application arrives after another after γ seconds. Indeed, we will evaluate the performance of our algorithm as we change the value of the arrival gap. The higher the value of the arrival gap is, the fewer applications will reach the system in a certain period of time, the less busy the network will be and vice versa.

Computation works	[0.1-1.5] 10^9 CPU-cycles
Data dependency	[0.5-20] MB
Application deadline	[CPL, 2CPL]

Table 4: Characteristics of tasks in the data set

4.2.2 Generation of Task Graphs

Several authors have proposed algorithms for generating randomly DAG such as Topcuoglu et al. 2002, Shivle et al. 2004 and Park 2004 with common parameters namely number of tasks, average degree of a node, communication and computation ratio (CCR). Inspired by those algorithms, we developed a random graph generator which has the following parameters:

- Number of tasks in the graph n^{TASKS}.
- Shape parameter of the graph α. From this parameter, number of layers (or height of task graph) is then randomly chosen from a uniform distribution whose mean equal to $\alpha.\sqrt{n}$. Whereas, number of tasks of each layer (width of each layer) is generated randomly from a uniform distribution whose mean value equal to $\frac{\sqrt{n}}{\alpha}$. Using α, we can choose to generate thin and long graphs by choosing $\alpha \ll 1.0$ or dense and shorter graphs when $\alpha \gg 1.0$.
- Communication to computation ratio CCR. CCR value of an application represents the relation between the average communication cost between tasks in the graph over the average computation cost of all tasks. An application can be considered as computation-intensive if the value of CCRs is low, whereas a high value of CCR indicates that the application is communication-intensive.
- Maximum delay DELAY. This parameter represents the maximum delay that the application can wait, after arriving at the network, for being executed. A schedule that takes more than DELAY to finish the execution will be considered as unacceptable, thus get rejected.

After having determined randomly the number of tasks in each layer using shape parameter α as explained above, we will iterate through each task, from the lowest layer to the highest layer, to establish random out-link dependencies from current task to a random subset of tasks which belong to higher layers. Size of the subset will also be selected randomly in the range of [1, $|T_i^{\text{HIGHER}}|$] in which T_i^{HIGHER} denotes the set of tasks that belong to higher layers than that of task v_i. Once we finish generating links between tasks only from those in the lower layer to those in the higher layer, we run an additional process, using deep-first search (DFS) algorithm, to remove transitivity in the graph to avoid redundant dependencies.

The computation works c_i of task v_i is selected randomly within a range as denoted in Table 4. Using parameter CCR, average weight $\overline{d_{ij}}$ of edge e_{ij}, which denotes data dependency between tasks v_i and v_j, is then defined by:

$$\overline{d_{ij}} = CCR \times \overline{C_{comp}(v_i)} \times \overline{\mathrm{BW}} \tag{18}$$

where $\overline{\mathrm{BW}}$ is the average bandwidth of the network and $\overline{C_{comp}(v_i)}$ is the average cost for a processor to execute task v_i defined by:

$$\overline{C_{comp}(v_i)} = \frac{c_i}{(\sum_{P_i \in PG} \mathrm{PROC}_i \times \mathrm{NCORE}_i) / \sum_{P_i \in PG} \mathrm{NCORE}_i} \tag{19}$$

From the average weight $\overline{d_{ij}}$, amount of data dependency of edge e_{ij} will then be set randomly with a value from the following range:

$$0.9 \times \overline{d_{ij}} \leq d_{ij} \leq 1.1 \times \overline{d_{ij}} \tag{20}$$

Finally, we add a dummy entry task and a dummy exit task to tasks that have no predecessors and successors, respectively. Those dummy tasks cause no effects to the overall schedule length of the whole graph as their constraint values will be 0.

To evaluate further the performance of our approach, for each experiment, we generate a data set of task graphs where values of the parameters are given below.

- $n^{\mathrm{APPS}} = 10000$
- $SET_{n^{\mathrm{TASKS}}} = \{20, 40, 60, 80, 100\}$
- $SET_{CCR} = \{0.2, 0.5, 1, 2, 5\}$
- $SET_{\alpha} = \{0.5, 1, 1.5, 2\}$
- $SET_{\gamma} = \{0.4, 0.6, 0.8, 1.0, 1.3, 1.5, 1.8\}$

4.2.3 Generation of Maximum Delay

Length of a path of a task graph is the sum of communication cost (time that is used for transferring data dependency between nodes) and computation cost (time that is used for execution of a node) of all the nodes and edges on the path, from start to finish. Critical path length (CPL) is the length of the longest path in the graph. Maximum

delay, DELAY, of an application will then be defined by selecting randomly from a range that depends on the CPL of the task graph.

$$CPL \leq \text{DELAY} \leq 2CPL \tag{21}$$

Chapter 5

Performance Evaluation

5.1 Performance metrics

The goal of our scheduling heuristic is to produce task schedules with good trade-off between cloud cost and overall completion time of tasks to maximize the number of accepted requests. In other words, our heuristic aims to allocate tasks to resources in the network such that to have requests executed within pre-defined deadline but using as less cloud computing power as possible. We are also interested in investigating to see if resources in the fog would be exploited efficiently.

- *Guarantee ratio (GR)*, specifies the percentage of requests (or task graphs) that finish execution before deadline among the set of requests in the data set that are fed to the network.
- *Cloud cost*, which is the total cost of using cloud resources for executing tasks that are offloaded to the cloud layer.
- *Fog resource occupancy*, signifies the percentage of resources in the fog layer that are utilized to execute incoming requests.

5.2 Comparison Strategies

Details of all the strategies to be evaluated are as follow:

- *Dynamic Heterogeneous Earliest Finish Time (DynamicHEFT)*: just like the static

version introduced in Topcuoglu et al. 2002, DynamicHEFT assigns tasks to minimize only the combined cost of communication time and computation time without taking into account other constraints (i.e., monetary cost of using cloud resources), thus $\beta = 0$. In addition, DynamicHEFT has only one attempt to compute schedule within the admission control procedure.

- *Cloud Unaware (CU)*: in contrast to DynamicHEFT, CU takes into account only resources in the fog layer which will certainly have zero cost for using the cloud ($\beta = 1$). CU also has at most one attempt in admission control procedure.

- *Compromised Makespan-Cost Ratio (CompromisedMKCR)*: similar to DynamicHEFT and CU, instead of having multiple attempts to accommodate a task graph to the network when it fails to meet the deadline requirements, CompromizedMKCR has only one chance to compute schedule with the neutral value of cost-makespan factor $\beta = 0.5$.

- *Proactive Cloud Unaware - HEFT (ProactiveCUHEFT)*: the strategy of this approach is actually straight forward by using two baseline heuristics CU ($\beta = 1$) and HEFT ($\beta = 0$). The idea of ProactiveCUHEFT is trying to schedule every request arrives at the network using CU - which has zero cloud cost. In case the schedule produced by CU would not finish before the deadline, DynamicHEFT will be applied in the latter attempt.

- *Proactive CompromisedMKCR - HEFT (ProactiveCMKCRHEFT)*: with also two attempts allowed in the admission control procedure just as same as in ProactiveCUHEFT, this variant, however, uses CompromisedMKCR in the first attempt instead of CU before employing DynamicHEFT in the second attempt.

- *Proactive CompromisedMKCR - manual (ProactiveCMKCR-manual)*: this variant is just as same as ProactiveCMKCRHEFT with the values of β are fixed at $\beta = 0.5$ and $\beta = 0$ in the first and last attempt, respectively. However, ProactiveCMKCR-manual has 3 attempts to compute schedules and uses $\beta = 0.2$, to increase the chance of assigning tasks to resource in the fog, in the second attempt before going to the extreme where $\beta = 0$. This strategy is expected to use less resource in the cloud than ProactiveCUHEFT and ProactiveCMKCRHEFT.

- *Proactive CompromisedMKCR - auto (ProactiveCMKCR-auto)*: with the same idea of having at most 3 attempts in admission control procedure, ProactiveCMKCR-auto applies formula (16) to determine the value of β in the second attempt without using a pre-defined value as inProactiveCMKCR-manual.

- *HybridEDF* (Stavrinides et al. 2018): unlike the other heuristics, there is no admission control in HybridEDF thus, tasks of any applications arrive at the network will be scheduled until either all the tasks are allocated or the deadline is reached, whatever comes first.

	Number of attempts	**First** β	**Second** β	**Third** β
DynamicHEFT	1	0	N/A	N/A
CU	1	1	N/A	N/A
CompromisedMKCR	1	0.5	N/A	N/A
ProactiveCUHEFT	2	1	0	N/A
ProactiveCMKCRHEFT	2	0.5	0	N/A
ProactiveCMKCR-manual	3	0.5	0.2	0
ProactiveCMKCR-auto	3	0.5	auto (≥ 0.2)	0
HybridEDF	1	N/A	N/A	N/A

Table 5: Difference between heuristics to be evaluated

5.3 Results

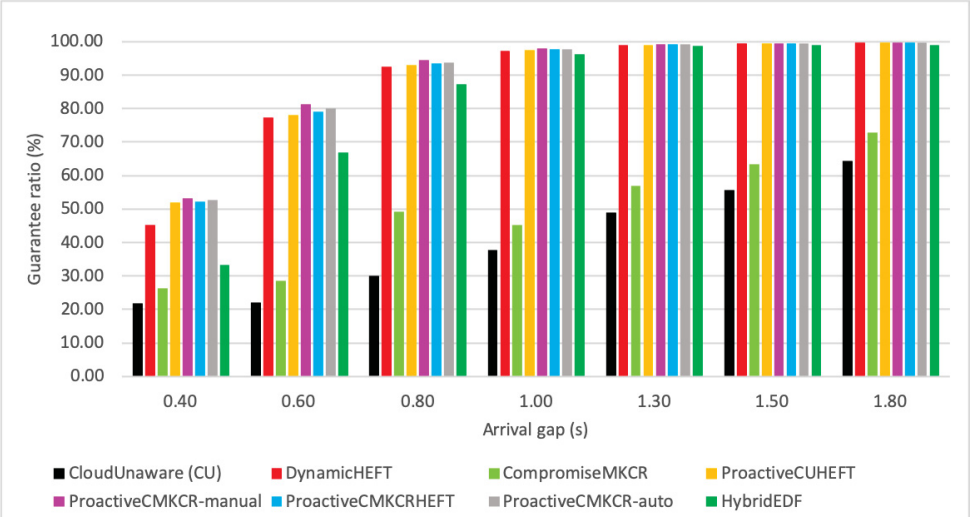

Figure 7: Guarantee ratio comparison

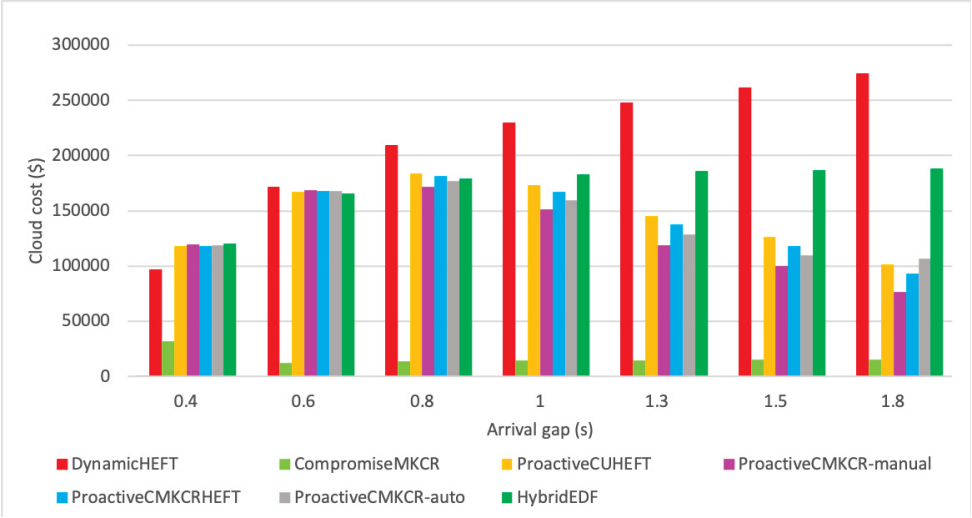

Figure 8: Cloud cost comparison

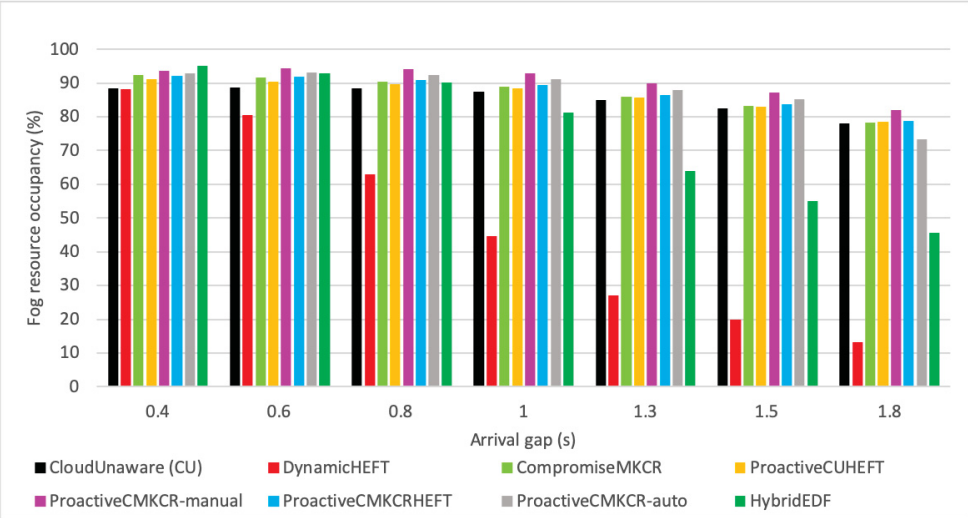

Figure 9: Fog resource occupancy comparison

In general, that all the heuristics share the same behavior to accommodate more applications as we increase the arrival gap, which makes the network become less busy and more resources available for service. However, we can see from the results that when the network becomes really busy with too many requests arrive at the network in a short period of time ($\gamma = \{0.4, 0.6, 0.8\}$), even though many applications will be rejected, our variants stand out with the highest ratio of applications finish before deadline while making a good use of resources in the fog layer.

Among the strategies that performs well in terms of the guarantee ratio metric, all the proactive variants offer timely service for very competitive cloud costs compared with DynamicHEFT and HybridEDF in all examined workload scenarios. Our proactive heuristics always allocate tasks to processors that may produce a more economical outcome first before going with an extreme approach if applications cannot be finished before the deadline in the first attempt. This factor also affects how the heuristics take advantage of resources in the network. For example, we can see that all the heuristics make really good use of this free-of-charge layer when the network is overloaded as the resources are occupied more than 85% of the time. However, as we increase the value of arrival gap of applications that arrive at the network to make the network less busy, both DynamicHEFT and HybridEDF start to fall short of using resources in the fog layer well, especially DynamicHEFT. All the proactive approaches still offer competitive service while exploiting efficiently fog resources.

We can also see that even though HybridEDF can satisfy deadline constraints of a

decent number of applications that arrive at the network, it is relatively more costly in terms of monetary cost for the cloud than the proactive strategies. This occurs when applying HybridEDF because, without an admission control, there are tasks that are offloaded to cloud resources but later, one or some antecedent tasks end up unsuccessfully finishing the execution before the application's deadline. In HybridEDF, execution location of a task is determined based on the ratio of its computation work over communication constraints with its successors, this method becomes strict and less flexible in terms of exploiting resources in the fog. As a consequence, a relatively low occupancy of resources in the fog layer when using HybridEDF compared to the other heuristics.

In addition, as expected, with the value of β is set at the two extremes, the two baseline methods CU ($\beta = 1$) and DynamicHEFT ($\beta = 0$) behave opposite in the examined cases. It can be seen from Figure 7 that CU always accommodate the least number of requests since tasks are allocated to only processors in the fog thus less resource are available in this approach compared to the other methods. However, as tasks are always offloaded to machines in the fog, CU has zero cost for the cloud as well as a high occupancy of resources in the fog layer. At the other extreme, DynamicHEFT performs contrarily with comparatively high ratio of accepted applications among the investigated methods. Because DynamicHEFT only minimizes the overall makespan of applications, in most cases, tasks will be scheduled to be executed in powerful processors in the cloud layer which makes DynamicHEFT, unsurprisingly, become the most expensive approach with the lowest occupancy of resources in the fog among the examined methods.

Also using the same scheme to grant only feasible applications to the network like DynamicHEFT and CU by using only one attempt in admission control, CompromisedMKCR performs relatively poor as the corresponding result of this method is only slightly better than the baseline method CU even though it is not limited to only resources in the fog. This shows that even in the case where we apply a neutral value of cost-makespan factor ($\beta = 0.5$) in our utility function, we will not likely obtain a fairly balance outcome if we use only one attempt.

Furthermore, the way β is adjusted, either manually or automatically, does not seem to impact remarkably on the performance as there is no notable difference between the variants in all the evaluated metrics. With 3 attempts allowed in the admission control procedure, ProactiveCMKCR-manual and ProactiveCMKCR-auto do not perform much better than ProactiveCUHEFT and ProactiveCMKCRHEFT, those that have at

most only 2 attempts. This indicates that more flexibility in terms of number of strategies allowed in the admission control would not likely guarantee much more profitable outcomes.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We proposed several variants of a proactive strategy that schedule tasks to resources in a fog-cloud environment in which we assume that an admission control is deployed to only allocate tasks whose application's deadline constraint can be met by the resources in the network. Moreover, unlike most of studies in the literature, we apply multiple attempts to schedule tasks using different strategies that favor makespan over monetary cost when choosing execution location on processing nodes to allocate tasks to if the execution cannot finish before the deadline. Results show that by accepting only feasible applications and dynamically adjusting makespan-cost factor β, our proposed heuristics can use resources in the network more efficiently with a noticeably higher rate of accepted applications with less money charged for using resources in the cloud than the other evaluated heuristics. In addition, our experiments show that all of our proactive variants exploit resources in the fog with the highest rate of occupancy in this intermediate layer, not only when the traffic is light but also when the network is overloaded with many applications arrive in a short period of time. Lastly, from the results, we see that it is beneficial to use multiple attempts to assign tasks to processing nodes which helps not only minimizing cloud cost but also maximizing the number of applications whose deadline are met. However, there should be a thorough observation to consider and decide how much flexibility we should have in the admission control procedure as there are not much difference between those that have a maximum of 2 and 3 attempts to accommodate requests.

6.2 Future Work

In the future, we aim to take into account other constraints such as RAM and energy consumption because these are factors that impact how the resources would be allocated or where tasks would be offloaded. Likewise, regarding elements that may affect response time of a request, we would also like to examine further more accurately how much time the task scheduler needs to calculate schedules and dispatch tasks to resources. Last but not least, we are interested in investigating how we can choose the right value of cost-makespan factor β automatically in the second attempt instead of adjusting it manually or with arbitrary formulas. Ideally, the difference of the values of β between attempts should be determined based on how far the makespan, produced by the schedule using the last value of β, is from the deadline.

Bibliography

Gartner, Inc. (2017). *Leading the IoT - Gartner Insights on How to Lead in a Connected World*. `https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf`. [Online; accessed 11-Oct-2019].

Mach, P. and Z. Becvar (Aug. 2017). "Mobile edge computing: a survey on architecture and computation offloading". In: *Ieee communications surveys tutorials* 19.3, pp. 1628–1656.

Baktir, A. C., A. Ozgovde, and C. Ersoy (June 2017). "How can edge computing benefit from software-defined networking: a survey, use cases, and future directions". In: *Ieee communications surveys tutorials* 19.4, pp. 2359–2391.

Ullman, J.D. (1975). "Np-complete scheduling problems". In: *Journal of computer and system sciences* 10.3, pp. 384–393.

Topcuoglu, H., S. Hariri, and M.-Y. Wu (Mar. 2002). "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *Ieee transactions on parallel and distributed systems* 13.3, pp. 260–274.

Bittencourt, Luiz F. and Edmundo R. M. Madeira (June 2008). "A performance-oriented adaptive scheduler for dependent tasks on grids". In: *Concurr. comput. : pract. exper.* 20.9, pp. 1029–1049.

Qin, Xiao and Hong Jiang (2005). "A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters". In: *Journal of parallel and distributed computing* 65.8, pp. 885–900.

Stavrinides and Helen D. Karatza (2011). "Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques". In: *Simulation modelling practice and theory* 19.1. Modeling and Performance Analysis of Networking and Collaborative Systems, pp. 540–552.

Pham, X.-Q. et al. (2017). “A cost- and performance-effective approach for task scheduling based on collaboration between cloud and fog computing”. In: *International journal of distributed sensor networks* 13 (11), pp. 1–16.

Stavrinides and H.D. Karatza (2018). “A hybrid approach to scheduling real-time IoT workflows in fog and cloud environments”. In: *Multimedia tools and applications*, pp. 1–17.

Shivle, S. et al. (July 2004). “Mapping of subtasks with multiple versions in a heterogeneous ad hoc grid environment”. In: *Third international symposium on parallel and distributed computing/third international workshop on algorithms, models and tools for parallel computing on heterogeneous networks*, pp. 380–387.

Park, Gyung-Leen (2004). “Performance evaluation of a list scheduling algorithm in distributed memory multiprocessor systems”. In: *Future generation computer systems* 20.2. Modeling and simulation in supercomputing and telecommunications, pp. 249–256.