UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

IN INTERNATIONAL JOINT Ph.D. WITH

LA ROCHELLE UNIVERSITÉ

ÉCOLE DOCTORALE EUCLIDE

ADYSON MAGALHÃES MAIA

IOT SERVICE PLACEMENT WITH LOAD DISTRIBUTION AND SERVICE MIGRATION IN EDGE COMPUTING FOR 5G NETWORKS

FORTALEZA

2021

ADYSON MAGALHÃES MAIA

IOT SERVICE PLACEMENT WITH LOAD DISTRIBUTION AND SERVICE MIGRATION IN EDGE COMPUTING FOR 5G NETWORKS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Doutor em Ciência da Computação. Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Miguel Franklin de Castro.

Orientador (Cotutela): Prof. Dr. Yacine Ghamri-Doudane.

Coorientador: Prof. Dr. Dario Vieira Conceição.

FORTALEZA

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M184i Maia, Adyson Magalhães.
IoT service placement with load distribution and service migration in edge computing for 5g networks / Adyson Magalhães Maia. – 2021.
175 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2021.
Orientação: Prof. Dr. Miguel Franklin de Castro.
Coorientação: Prof. Dr. Yacine Ghamri-Doudane, Prof. Dr. Dario Vieira Conceição.

1. Service placement. 2. Load distribution. 3. Service migration. 4. Internet of things. 5. Edge computing. I. Título.

CDD 005

ADYSON MAGALHÃES MAIA

IOT SERVICE PLACEMENT WITH LOAD DISTRIBUTION AND SERVICE MIGRATION IN EDGE COMPUTING FOR 5G NETWORKS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Doutor em Ciência da Computação. Área de concentração: Ciência da Computação.

Aprovada em: 31/03/2021.

This thesis is dedicated to my parents and friends.

ACKNOWLEDGEMENTS

My Ph.D. journey was a challenging and enjoyable experience that could not be achieved without the support of many, to whom I would like to express my gratitude.

First and foremost, I would like to thank my advisors and co-advisor, Prof. Miguel Franklin, Prof. Yacine Ghamri-Doudane, and Prof. Dario Vieira, for all their encouragement, guidance, contributions, and opportunities during this long journey.

Thank you to all my fellow colleagues from the GREat and L3i laboratories for their fellowship and helpful discussions. Special thanks to Beatriz, Belmondo, Christiano, Deborah, Elvys, Hamza, Ítalo, Jefferson, Joseane, Juliana, Lana, Lionel, Mariana, Marwa, Muzzamil, Nour, Paulo Arthur, Rainara, Rayner, Rim, Sabrine, Salah, Tales, Thomas, Viviana. I also would like to thank Jaminny for helping me to review some papers during the Ph.D. journey.

I must mention my friends from Fortaleza, Bruno, Caio, Cecilia, Fabio, Hendrio, Kleber, Mailson, Monyke, Regina, and Schwarzenegger, with whom I have been sharing more than 15-years of friendship.

Almost last but not least, I am grateful to my family, including my parents, Adahilson and Francisca, that always supported and encouraged me.

Finally, I would like to thank UFC, EFREI, ULR, and CAPES for the opportunity and financial support of this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

"History has its eyes on you. Who lives, who dies, who tells your story?"

(Lin-Manuel Miranda)

ABSTRACT

Edge Computing (EC) is a promising concept to alleviate some of the cloud computing limitations in supporting Internet of Things (IoT) applications, especially time-sensitive applications, by bringing computing resources closer to end users, at the network edges. As promising as EC is, it also faces many challenges. These are mainly related to the resource management in the vast, distributed, dynamic, and heterogeneous setting brought by EC. A relevant issue for resource management is the service placement problem, which is the decision-making process of determining where to place different applications or services over the EC infrastructure according to some constraints, requirements, and performance goals. This decision-making process can thus be extended to include other related issues, such as load distribution and service migration. In this thesis, we investigate the IoT services placement with load distribution and service migration in the context of next generation networks with EC capabilities, such as the fifth-generation (5G) mobile system. First, we address service placement with load distribution as single and multi-objective problems and we the proposal to solve these using a well-chosen genetic algorithm. Analytical results show that through our proposed formulation and the associated proposed algorithms, we are able to outperform other benchmark algorithms in terms of multiple conflicting objectives, such as response deadline violation, operational cost, and service availability. Then, in order to handle load fluctuations, we propose a centralized limited look-ahead prediction control that periodically readjusts service placement and load distribution decisions by taking into account the performance-cost trade-off of service migrations. Evaluation results show that our predictive control has even better system performance regarding response deadline violations with a small additional migration cost compared to benchmark algorithms. Finally, we address the scalability issue faced by centralized decision-making process by designing a hierarchical distributed service placement solution. The evaluation of our distributed control indicates that the trade-off between the system performance and the scalability of decision-making depends on how the control decision problem is decomposed.

Keywords: service placement; load distribution; service migration; internet of things; edge computing; 5G network.

RESUMO

Edge Computing (EC) é um conceito promissor para mitigar algumas das limitações da computação em nuvem no suporte às aplicações da Internet das Coisas (Internet of Things - IoT), especialmente às aplicações sensíveis ao tempo, ao trazer recursos computacionais para a proximidade dos usuários finais nas bordas da rede. Por mais promissor que a EC seja, este conceito também enfrenta muitos desafios. Estes desafios estão principalmente relacionados à gestão de recursos em ambiente vasto, distribuído, dinâmico e heterogêneo trazido pela EC. Uma questão relevante para o gerenciamento de recursos é o problema de colocação de serviço, que é o processo de tomada de decisão para determinar onde colocar diferentes aplicações ou serviços na infraestrutura da EC de acordo com algumas restrições, requisitos e metas de desempenho. Este processo de tomada de decisão pode ser estendido para incluir outras questões relacionadas, como a distribuição de carga e a migração de serviço. Esta tese investiga a colocação de serviços IoT com distribuição de cargas e migração de serviços no contexto das redes de próxima geração com suporte à EC, tal como a rede móvel de quinta geração (5G). Primeiramente, esta tese aborda a colocação de serviços com distribuição de carga como problemas mono-objetivo e multiobjetivo e propõe resolvê-los usando algoritmo genético. Os resultados analíticos mostram que, por meio de nossa formulação e dos algoritmos propostos associados, é possível superar outros algoritmos de benchmark em termos de múltiplos objetivos conflitantes, tais como a violação de prazo de resposta, o custo operacional e a disponibilidade de serviço. Em seguida, para lidar com as flutuações de carga, é proposto um controle centralizado e preditivo que reajusta periodicamente as decisões de colocação de serviço e distribuição de carga de acordo com a relação custo-benefício das migrações de serviço. Os resultados da avaliação mostram que o controle preditivo proposto tem um desempenho de sistema ainda melhor em relação às violações do prazo de resposta e um pequeno custo de migração adicional em comparação com os algoritmos de benchmark. Finalmente, é tratado o problema de escalabilidade enfrentado por um processo de tomada de decisão centralizado ao projetar uma solução hierárquica e distribuída da colocação de serviços. A avaliação do controle distribuído proposto indica que a compensação entre o desempenho do sistema e a escalabilidade da tomada de decisões depende de como o problema de decisão é decomposto.

Palavras-chave: colocação de serviço; distribuição de carga; migração de serviço; internet das coisas; edge computing; rede 5G.

RÉSUMÉ

L'Edge Computing (EC) est un concept prometteur pour atténuer certaines des limitations du cloud computing dans la prise en charge des applications Internet des Objets (Internet of Things - IoT), en particulier les applications sensibles au délai, en rapprochant les ressources informatiques des utilisateurs à la périphérie du réseau. Aussi prometteur que soit l'EC, ce concept est également confronté à de nombreux défis. Ceux-ci sont principalement liés à la gestion des ressources dans ce cadre étendu, distribué, dynamique et hétérogène qu'apporte l'EC. Un des problèmes majeurs pour cette gestion des ressources est le problème du placement des services ou applications. Ce problème peut ainsi être étendu pour inclure d'autres aspects connexes, tels que la répartition de la charge et la migration de service. Dans cette thèse, nous étudions le placement des services IoT avec distribution de charge et migration de services dans le contexte de réseaux de nouvelle génération dotés de capacités EC, tels que le système mobile de cinquième génération (5G). Premièrement, nous abordons le placement de services avec la distribution de charge comme un problème mono-objectif, puis un problème multi-objectifs. Nous proposons alors de les résoudre en utilisant un algorithme génétique spécifique. Les résultats analytiques montrent que grâce à notre formulation et aux algorithmes proposés, nous sommes en mesure de suppléer les autres algorithmes de référence en termes lorsque nous considérons des objectifs multiples et contradictoires, comme la violation du délai de réponse, le coût opérationnel et la disponibilité du service. Afin de gérer les fluctuations de charge, nous proposons ensuite un contrôle centralisé et prédictif qui réajuste périodiquement les décisions de placement de services et de distribution de charge en tenant compte du compromis performance-coût lié aux migrations de services. Les résultats de l'évaluation montrent que notre contrôle prédictif offre des performances du système encore meilleures en ce qui concerne les violations de délai mais au prix d'une légère augmentation du coût liée à la migration. Enfin, nous abordons le problème d'extensibilité auquel est confrontée toute prise de décision centralisée en concevant une solution hiérarchique et distribuée pour le placement de services. L'évaluation de notre contrôle distribué indique que le compromis entre les performances du système et l'extensibilité de la prise de décision dépend de la façon dont le problème de décision de contrôle est décomposé.

Mots clés: placement de services; distribution de charge; migration de services; internet des objets; informatique en périphérie; réseau 5G.

LIST OF FIGURES

Figure 1 – The three-tier architecture of Edge Computing 22
Figure 2 – Research methodology 28
Figure 3 – Basic IoT architecture 31
Figure 4 – Comparison between VM and container 33
Figure 5 – MEC system architecture 38
Figure 6 – Generation transition in BRKGA 43
Figure 7 – BRKGA flowchart 44
Figure 8 – NSGA-II classification and selection procedures 45
Figure 9 – Example of population classified by NSGA-II 46
Figure 10 – LLC controller framework 49
Figure 11 – A basic working principle of LLC 50
Figure 12 – Taxonomy for service placement problem in Edge Computing 53
Figure 13 – Proposed Edge Computing system for 5G networks 70
Figure 14 – Example of an EC infrastructure represented as a graph 70
Figure 15 – Different application designs 73
Figure 16 – Example of the variables of the static service placement with load distribution problem. 78
Figure 17 – BRKGA+NSGA-II flowchart 85
Figure 18 – Decoding example for Algorithm 2 88
Figure 19 – Performance of different elite and mutant group sizes 97
Figure 20 – Performance of different elite probability P_{elite} values 98
Figure 21 – Performance of different stopping criteria threshold $\hat{I}_{mdr}^{min}$ values 99
Figure 22 – Maximum deadline violation 100
Figure 23 – Overall operational cost 101
Figure 24 – Average service availability 102
Figure 25 – Proposed Edge Computing system with mobile and inactive devices 106
Figure 26 – Response time estimation for a dynamic request flow 112
Figure 27 – Impact of a migration delay in different requests 115
Figure 28 – Decoding example for Algorithm 3 120
Figure 29 – Chromosome sequence generation examples 122
Figure 30 – Experiment network scenario for the dynamic service placement problem . . 126

Figure 31 – On-and-Off workload pattern as a two-state Markov chain 128
Figure 32 – Synthetic workload patterns 129
Figure 33 – Experiment flowchart for the dynamic service placement problem 129
Figure 34 – Performance of different load chunk size $\lambda_{\%}$ values 131
Figure 35 – Performance of different prediction horizon H values 132
Figure 36 – Average deadline violation per time step 133
Figure 37 – Average operational cost per time step 134
Figure 38 – Average migration cost per time step 134
Figure 39 – Control architectures 137
Figure 40 – Example of time scale difference between global and local controllers . . . 141
Figure 41 – Subsystems scenario cases 160
Figure 42 – Performance for different numbers of subsystems with 10 applications and 10,000 users 161
Figure 43 – Execution time of the hierarchical distributed controllers 163
Figure 44 – Average number of application replicas placed in different network parts . . 163

LIST OF TABLES

Table 1 – Comparison of Cloud Computing and Edge Computing 34
Table 2 – Comparison of Edge Computing concepts 36
Table 3 – Performance comparison between 4G and 5G 36
Table 4 – Key Performance Indicators (KPIs) to assess the performance of 5G use case categories 37
Table 5 – Classification of static service placement approaches in Edge Computing . . 65
Table 6 – Classification of dynamic service placement approaches in Edge Computing . 66
Table 7 – Main notations of the static service placement problem 71
Table 8 – Performance evaluation parameters for the static service placement problem . 95
Table 9 – Main notations of the dynamic service placement problem 107
Table 10 – Proposed control algorithms 123
Table 11 – Performance evaluation parameters for the dynamic and centralized service placement problem 127
Table 12 – Main notations of the distributed service placement problem 139
Table 13 – Primary and secondary publications of this thesis 168

LIST OF ALGORITHMS

Algorithm 1 – Crowding distance assignment 46
Algorithm 2 – Chromosome decoder for the static service placement problem 87
Algorithm 3 – Chromosome decoder for the dynamic service placement problem with a single time step 119
Algorithm 4 – The procedure of obtaining a control input sequence from a chromosome sequence for the dynamic service placement problem 121
Algorithm 5 – Chromosome decoder for the dynamic service placement problem with multiple look-ahead time steps and simple sequence generation 122
Algorithm 6 – Chromosome decoder for the dynamic service placement problem with multiple look-ahead time steps and general sequence generation 123
Algorithm 7 – Chromosome decoder for the global one-time step representation 148
Algorithm 8 – Chromosome decoder for the local one-time step representation 154

LIST OF ABBREVIATIONS AND ACRONYMS

3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
ARIMA	AutoRegressive Integrated Moving Average
ASP	Application Service Provider
BRKGA	Biased Random-Key Genetic Algorithm
BS	Base Station
CC	Cloud Computing
EC	Edge Computing
eMBB	enhanced Mobile Broadband
ETSI	European Telecommunications Standards Institute
GA	Genetic Algorithm
IaaS	Infrastructure-as-a-Service
IERC	IoT European Research Cluster
ILP	Integer Linear Programming
INLP	Integer Nonlinear Programming
InP	Infrastructure Provider
IoT	Internet of Things
IPS	Instructions Per Second
IT	Information Technology
KPI	Key Performance Indicator
LLC	Limited Look-ahead Control
LP	Linear Programming
MDP	Markov Decision Process
MDR	Mutual Domination Rate
MEC	Multi-access Edge Computing
MEC	Mobile Edge Computing
MILP	Mixed-Integer Linear Programming
MINLP	Mixed-Integer Nonlinear Programming
mMTC	massive Machine Type Communications
MPC	Model Predictive Control

NIST	National Institute of Standards and Technology
NLP	Nonlinear Programming
NSGA-II	Non-dominated Sorting Genetic Algorithm II
OS	Operating System
PaaS	Platform-as-a-Service
QoS	Quality of Service
RAM	Random-Access Memory
RAN	Radio Access Network
RFID	Radio Frequency Identification
SaaS	Software-as-a-Service
SLA	Service Level Agreement
URLLC	Ultra Reliable Low Latency Communications
VM	Virtual Machine

CONTENTS

1 INTRODUCTION . . . 21
1.1 Contextualization . . . 21
1.2 Motivation . . . 24
1.3 Research Questions and Contributions . . . 26
1.4 Research Methodology . . . 27
1.5 Organization . . . 28
2 BACKGROUND . . . 30
2.1 Internet of Things . . . 30
2.2 Infrastructure Technologies . . . 32
2.2.1 Cloud Computing . . . 32
2.2.2 Edge Computing . . . 33
2.2.3 5G Network . . . 35
2.3 Mathematical Programming . . . 39
2.3.1 Multi-Objective Optimization . . . 40
2.4 Genetic Algorithms . . . 42
2.4.1 Biased Random-Key Genetic Algorithm . . . 42
2.4.2 Non-dominated Sorting Genetic Algorithm . . . 45
2.4.3 Stopping Criteria . . . 47
2.5 Limited Look-ahead Control . . . 48
2.6 Summary . . . 51
3 RELATED WORK . . . 52
3.1 Classification Taxonomy . . . 52
3.1.1 Problem Formulation . . . 53
3.1.2 Controller Design . . . 54
3.1.2.1 Dynamic Assignment . . . 55
3.1.3 System Modeling . . . 55
3.1.3.1 Infrastructure Model . . . 55
3.1.3.2 Application Model . . . 56
3.1.3.3 Dynamic Model . . . 57
3.1.4 Solution Technique . . . 57
3.2 Classification of Service Placement Approaches . . . 58

3.2.1 ***Static Placement without Load Distribution*** 58
3.2.2 ***Static Placement with Load Distribution*** 59
3.2.3 ***Dynamic Placement without Load Distribution*** 60
3.2.4 ***Dynamic Placement with Load Distribution*** 62
3.2.5 ***Discussion*** 63
3.3 **Summary** 67
4 **A STATIC APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION** 68
4.1 **System Model** 68
4.1.1 ***Infrastructure Model*** 69
4.1.2 ***Application Model*** 72
4.1.3 ***User Model*** 73
4.2 **Problem Statement and Formulation** 74
4.2.1 ***Response Time Estimation*** 74
4.2.1.1 *Network Delay* *75*
4.2.1.2 *Processing Delay* *75*
4.2.2 ***Problem Variables and Constraints*** 76
4.2.3 ***Single-Objective Formulation*** 78
4.2.3.1 *Linearization and Relaxation* *79*
4.2.3.1.1 *Node Capacity Constraint* 79
4.2.3.1.2 *Queue Stability Constraint* 80
4.2.3.1.3 *Objective Function* 80
4.2.3.1.4 *Linear Formulation* 81
4.2.4 ***Multi-Objective Formulation*** 81
4.3 **A Genetic-Based Proposal** 83
4.3.1 ***Chromosome Representation and Decoder*** 85
4.3.1.1 *Decoder Algorithm* *86*
4.3.1.2 *Complexity Analysis* *88*
4.3.2 ***Initial Population*** 88
4.3.2.1 *Complexity Analysis* *91*
4.3.3 ***Next Population and Stopping Criteria*** 91
4.3.4 ***Complexity Analysis of the Genetic Algorithm*** 92

4.4 **Performance Analysis** . . . 93
4.4.1 ***Performance Metrics*** . . . 93
4.4.2 ***Evaluated Algorithms*** . . . 93
4.4.3 ***Analysis Setup*** . . . 94
4.4.4 ***Different Parameters Settings*** . . . 96
4.4.4.1 *Elite and Mutant Group Size* . . . *96*
4.4.4.2 *Elite Probability* . . . *96*
4.4.4.3 *Stopping Criteria Threshold* . . . *97*
4.4.5 ***Results and Discussion*** . . . 97
4.4.5.1 *Maximum Deadline Violation* . . . *98*
4.4.5.2 *Overall Operational Cost* . . . *100*
4.4.5.3 *Average Service Availability* . . . *102*
4.5 **Summary** . . . 103
5 **A DYNAMIC AND CENTRALIZED APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION** . . . 105
5.1 **System Model** . . . 106
5.1.1 ***Infrastructure Model*** . . . 107
5.1.2 ***Application Model*** . . . 108
5.1.3 ***User Model*** . . . 108
5.2 **Problem Statement and Formulation** . . . 109
5.2.1 ***Modeling the System Dynamics*** . . . 110
5.2.1.1 *System State Estimation* . . . *112*
5.2.2 ***Optimization Formulation*** . . . 115
5.3 **Control Algorithms** . . . 116
5.3.1 ***One-Time Step Algorithm*** . . . 116
5.3.1.1 *Chromosome Representation* . . . *117*
5.3.1.2 *Chromosome Decoder* . . . *117*
5.3.1.3 *Genetic Operations* . . . *120*
5.3.2 ***H-Steps Look-Ahead Algorithm*** . . . 121
5.3.2.1 *Simple Sequence* . . . *121*
5.3.2.2 *General Sequence* . . . *122*
5.4 **Performance Analysis** . . . 123

5.4.1 ***Evaluated Algorithms*** 124
5.4.2 ***Performance Metrics*** 124
5.4.3 ***Analysis Setup*** 125
5.4.4 ***Different Parameters Settings*** 130
5.4.4.1 *Load Chuck Size* *130*
5.4.4.2 *Prediction Horizon Length* *130*
5.4.5 ***Results and Discussion*** 131
5.4.5.1 *Deadline Violation* *132*
5.4.5.2 *Operational Cost* *133*
5.4.5.3 *Migration Cost* *133*
5.5 **Summary** 135
6 **A DYNAMIC AND DISTRIBUTED APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION** 136
6.1 **Hierarchical Distributed Control Design** 136
6.2 **Global Controller** 138
6.2.1 ***System Dynamics*** 140
6.2.1.1 *System State Estimation* *142*
6.2.2 ***Optimization Formulation*** 144
6.2.3 ***Control Algorithm*** 146
6.3 **Local Controller** 149
6.3.1 ***System Dynamics*** 150
6.3.1.1 *System State Estimation* *151*
6.3.2 ***Optimization Formulation*** 152
6.3.3 ***Control Algorithm*** 153
6.3.4 ***Cooperative Control Design*** 153
6.4 **Performance Analysis** 157
6.4.1 ***Evaluated Algorithmic Solutions*** 157
6.4.2 ***Performance Metrics*** 158
6.4.3 ***Analysis Setup*** 159
6.4.4 ***Results and Discussion*** 160
6.5 **Summary** 164
7 **CONCLUSION** 165

7.1 **Contributions** 165
7.2 **Perspectives** 167
BIBLIOGRAPHY 170

1 INTRODUCTION

1.1 Contextualization

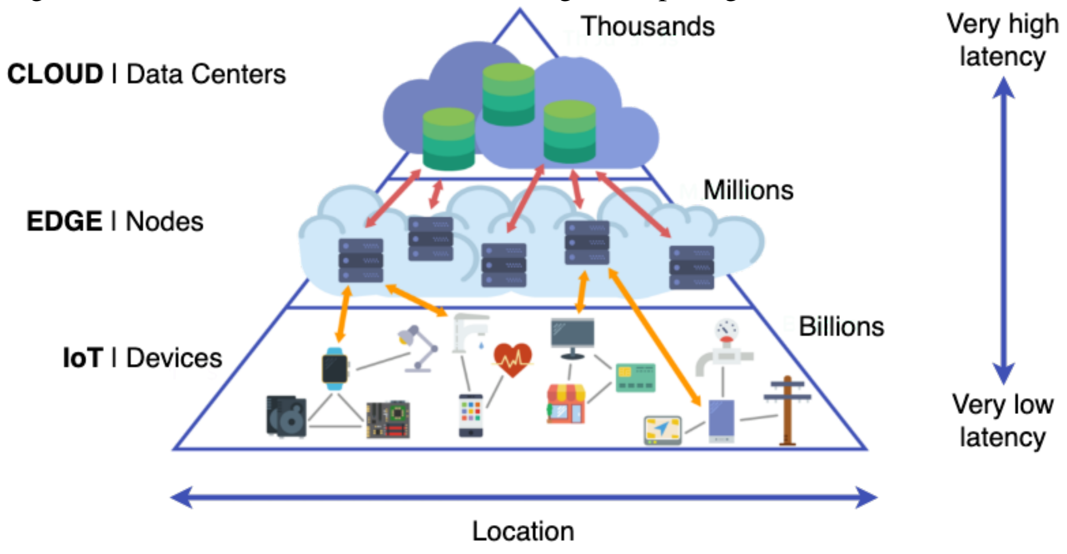
With the progressing development of computing and wireless technologies, the Internet of Things (IoT) is a paradigm driving a digital transformation in our daily lives (GUBBI *et al.*, 2013; BALAJI *et al.*, 2019). In the IoT paradigm, smart objects or things are essential building blocks, which include mobile phones, vehicles, home appliances, sensors, actuators, and any other embedded devices. These devices will be interconnected in order to exchange data related to real and virtual worlds among themselves through modern communication network infrastructures. Therefore, IoT expands existing human-to-human communication to human-to-machine and machine-to-machine communications. Furthermore, IoT envisions the creation of new and innovative applications in diverse areas, such as supply chain management, transportation and logistics, connected vehicles, healthcare, smart home, smart buildings, and smart cities (PERERA *et al.*, 2014).

The continued growth of IoT brings more connected devices to collect and consume data across the network. Due to the resource constraints of those devices, a common approach is to use Cloud Computing (CC) to execute data analysis remotely. Some of the benefits of using CC are (*i*) the flexible pricing model (pay-as-you-go), (*ii*) the on-demand and elastic delivery of virtualized resources (e.g., computing, storage, and network resources), and (*iii*) the scalable computing model (PAN; MCELHANNON, 2018).

The backbone of CC is based on building a few large data centers in various parts of the world to serve a huge number of users. However, this means that all user-generated data or requests need to be transmitted to a remote centralized data center, which may result in long (network) latencies. For some time-sensitive IoT applications (e.g., factory automation, intelligent transport systems, emergency response, interactive mobile gaming, augmented reality, and mission-critical applications) requiring real-time responses at 10ms or even 1ms (SCHULZ *et al.*, 2017), the delay caused by transferring data to the cloud is unacceptable. Moreover, some data processing can be made locally without having to be transmitted to the cloud. Even when some decisions are made in the cloud, it may be unnecessary and inefficient to send the large volume of data to the cloud for processing and storing because not all data is useful for decision making and analysis (HU *et al.*, 2017). Another drawback of CC is the lack of fast and direct access to local contextual information (e.g., precise user location, local network conditions, and user mobility behavior) while provisioning resources to an application (LIU *et al.*, 2018). Therefore, these challenges caused by the explosive growth of IoT cannot be addressed only by the Cloud Computing model.

Several similar concepts have emerged in academia and industry to overcome some limitations of only using CC. Among those concepts are Cloudlet (SATYANARAYANAN *et al.*, 2009), Fog Computing (BONOMI *et al.*, 2012), and Mobile Edge Computing/Multi-access Edge Computing (ETSI, 2016). The common denominator of those concepts is the extension of Cloud Computing by bringing cloud services and resources (e.g., computing, storage, and networking resources) closer to end users on geo-distributed nodes at the network edges (ROMAN *et al.*, 2018; BILAL *et al.*, 2018). We use the term Edge Computing (EC) to encompass these partially overlapping and complementary concepts. In general, EC adds a new layer of distributed general-purpose computing nodes between the end-user devices and cloud data centers, as shown in Figure 1 (BAKTIR *et al.*, 2017). A variety of edge nodes forms the edge layer, where a node may have diverse resource capabilities and supports the execution of applications or services through virtualization technologies. Some examples of such edge nodes are cellular base stations, routers, switches, and wireless access points.

Figure 1 – The three-tier architecture of Edge Computing



Source: Adapted from <https://bit.ly/37VL0Mu>.

By offering cloud resources at the edge layer, applications running on edge nodes in the vicinity of their end-user devices can filter, aggregate, or analyze data close to its source. Consequently, Edge Computing can (*i*) minimize latency and response time; (*ii*) reduce core and cloud networks traffic; (*iii*) decrease power consumption of end-user devices by running computeintensive applications in the edge layer; as well as (*iv*) make better location- and context-aware decisions (LIU *et al.*, 2018). The characteristics of EC ensure a wide range of applications and use cases that can benefit from being deployed at the edge, such as healthcare, augmented and virtual reality, multi-player gaming, interactive multimedia, video analytics, smart environments, industrial control systems, vehicular communications, road traffic monitoring (BILAL *et al.*, 2018).

EC can also be considered a key technology to overcome some challenges of the next fifth-generation (5G) cellular network, such as extremely low latency, reduction on core network traffic, and delivery of real-time contextual information (TALEB *et al.*, 2017). The 5G network is driven by the current trends toward IoT and the growth of mobile Internet traffic to surpass the limitations of 4G networks by guaranteeing a thousand-fold communication capacity increase, extremely low latency, and a massive number of connections in a cost and energy-efficient manner (SHAFI *et al.*, 2017; MARABISSI *et al.*, 2018). Moreover, the ubiquitous connectivity of 5G provides the 4A (Anytime, Anyplace, Anyone, and Anything) connectivity for an IoT environment (KITANOV *et al.*, 2016). That is, people and things can connect at any time to any place with anything and anyone.

Compared to the traditional cloud infrastructure, Edge Computing has distinguishing characteristics (HU *et al.*, 2017; KHAN *et al.*, 2019). First, Cloud Computing usually locates its resources in a few centralized data centers, but there will be a dense geo-distribution of edge nodes in EC. Furthermore, the availability of the cloud services depends on the distance of multi-hop between an end-user device and a cloud data center, while edge nodes are one or few hops away from end users. Edge nodes are more heterogeneous than cloud servers, i.e., edge nodes come with different form factors and resources capabilities. However, edge nodes are generally more resource-limited than cloud data centers. Lastly, EC supports mobility and has, consequently, a more dynamic environment than CC.

Despite the benefits of Edge Computing, it is still a recent research topic and faces several challenges. A major challenge is related to resource management due to the vast, distributed, dynamic, and heterogeneous EC environment (LIU *et al.*, 2018; BILAL *et al.*, 2018). We are particularly interested in the service placement problem, which is a resource management issue related to the decision of ideal places (i.e., whether on a node in the edge or within the cloud) to deploy multiple applications or services according to some demands and constraints in order to optimize desired objectives (TÄRNEBERG *et al.*, 2017; FILHO *et al.*, 2018). Moreover, in the service placement context, we use the notion of application and service interchangeably to designate a program or software that performs a function or suite of related functions of benefit to their end users.

In the next section, we further discuss service placement as a non-trivial problem that can include some related sub-problem, such as service migration and load distribution.

1.2 Motivation

In the Edge Computing context, a major concern is related to allocating shared resources for running applications through virtualization technologies (e.g., Virtual Machine (VM) or container) (BILAL *et al.*, 2018; TOCZÉ; NADJM-TEHRANI, 2018; NAHA *et al.*, 2018). Accordingly, service placement, service migration, and load distribution are crucial issues that should be addressed to achieve efficient resource management on Edge Computing.

Application or service placement is the decision-making process of selecting a destination computer-based node or server to host an application. This decision is made autonomously by some Infrastructure Provider (InP) throughout a resource management tool, called controller, on behalf of Application Service Providers (ASPs), which do not directly determine where to place their applications (FILHO *et al.*, 2018). Usually, an ASP only signs a Service Level Agreement (SLA) defining the Quality of Service (QoS) requirements to be fulfilled by the InP. In this way, a service placement scheme maps each application onto some hosting node to optimize a set of performance-related objectives while meeting all requirements defined by ASPs and InP.

Over time, the service placement decision needs to be further reassessed as application workload conditions change, and the current solution mapping cannot meet some requirements (FILHO *et al.*, 2018). Such changes may be due to user mobility or different workload patterns. Application or service migration can be understood as the movement of applications from one server to another. Moreover, service migration provides the ability to adjust the placement of applications to satisfy the QoS requirements continually, such as low response time. Although the movement of applications may improve system performance, it may lead to some problems (PIETRI; SAKELLARIOU, 2016). For instance, it may cause performance degradation due to excessive reallocation, system overhead, resources wastage by some applications, and lack of them to other ones. As the migration overhead may not be negligible, frequent migrations need to be avoided, especially when the migration cost exceeds its benefits. On the other hand, a delayed reallocation may lead to SLA violations or an increased cost from resource over-provisioning. Therefore, the reassessment of a placement decision should consider the benefits and costs of service migrations.

Another challenge is provisioning adequate resources to handle the fluctuating workloads generated by application users. That is, how many resources should be allocated to each application to handle a dynamic incoming workload. A simple solution to this problem is using resource over-provisioning to handle high peak loads. However, over-provisioning is unsuitable for EC due to high costs and limited resource capacity at edge nodes (SKARLAT *et al.*, 2017b; BILAL *et al.*, 2018). Meanwhile, a service placement solution mappings can be many-to-many, i.e., an application can be placed onto one or more nodes, and a node can host more than one application. In this way, service placement may consider the load balancing to distribute workloads of an application across multiple nodes. Hence, load distribution can be applied to improve resource usage, increase availability, reduce response time, and avoid over-provisioning (XU *et al.*, 2017).

Although the service placement, service migration, and load distribution problems can be separately optimized in independent procedures, the placement decision may affect the two other problems, and vice versa (URGAONKAR *et al.*, 2015). For instance, an application workload can only be distributed to nodes selected by a service placement decision to host this application. On the other hand, a service placement procedure may avoid deploying an application in an overloaded server whose incoming workload depends on a load distribution decision. Hence, an optimal decision-making process may require a complex joint optimization of these problems.

We can find several research works in the literature concerning service placement in Cloud Computing. Pires and Barán (2015), Pietri and Sakellariou (2016), Carvalho *et al.* (2018), and Filho *et al.* (2018) published surveys on this topic over the past few years. However, research works for CC cannot be directly applied to EC because they neither consider the distinct characteristics of EC (e.g., a large, distributed, heterogeneous, and dynamic environment) nor the time-sensitive application requirements.

There are some research works in the literature related to service placement in Edge Computing, such as Tärneberg *et al.* (2017), Gu *et al.* (2017), Skarlat *et al.* (2017b), Zhao and Liu (2018). However, the majority of these works do not handle the scalability and dynamism of EC because the decision-making process is centralized and static. Furthermore, some works do not consider the limited resource capabilities of edge nodes, requirements of time-sensitive applications, or conflicted optimization objectives.

1.3 Research Questions and Contributions

In this thesis, we aim to address the aforementioned concerns to provide decision-making approaches for the service placement problem and its related sub-problems, specifically load distribution and service migration, in the context of cellular networks with Edge Computing capabilities (e.g., 5G networks). In order to achieve this goal, we investigate the following research questions through the three main contributions of this thesis:

- **Research Question 1 (RQ1)**. *How to make service placement and load distribution decisions to deploy multiple IoT applications or services in an Edge Computing infrastructure according to certain infrastructure constraints, application requirements, and performance criteria?*
- **Research Question 2 (RQ2)**. *How to reassess the service placement and load distribution decisions due to dynamic application loads by taking into account the benefits and costs of service migrations?*
- **Research Question 3 (RQ3)**. *How to make scalable and optimized (service placement, load distribution, and service migration) decisions in a large Edge Computing environment?*

The above-mentioned research questions follow an incremental methodology where each subsequent question adds new aspects to be considered for the service placement problem. Consequently, we also have incremental contributions as a contribution extends the previous ones. Our three main contributions are described below.

In our first contribution, we jointly address the service placement and load distribution problems in a static Edge Computing scenario where all information required in the decision-making process is provided in advance and does not change over time. We formalize the decision-making process as an optimization problem that considers EC infrastructure resource capacity constraints and different application characteristics (e.g., response deadline, resource demand, scalability, and availability). The formalized problem aims to minimize the potential occurrence of SLA violations in terms of application response deadlines. Then, we extended this single-objective problem to include multiple conflicting performance-related objectives (e.g., operational cost and service availability) and still prioritizing metrics related to time-sensitive applications, such as minimizing deadline violations. In order to solve the single and multi-objective formulated problems, we propose a genetic-based meta-heuristic that combines Biased Random-Key Genetic Algorithm (BRKGA) (GONÇALVES; RESENDE, 2011) and Non-dominated Sorting Genetic Algorithm II (NSGA-II) (DEB *et al.*, 2002) with specific evolutionary operations to generate feasible solutions to these problems.

The second contribution considers that application loads might vary in spatial and temporal domains in a dynamic EC scenario due to user mobility or workload patterns. In order to handle this dynamic load, we design a centralized controller that readjusts the application placement and load distribution decisions over time. Moreover, the designed controller adopts a proactive approach, called Limited Look-ahead Control (LLC) (ABDELWAHED *et al.*, 2004), that prepares the EC system in advance for predicted load fluctuations. The proactive preparation can include the pre-deployment of an application through service migration to a region that will soon request this application. As a result of this pre-deployment, the negative impact of service migration delays on future response time can be reduced. According to the LLC concept, we formulate the dynamic service placement with load distribution problem as a multi-objective optimization control problem over a look-ahead prediction horizon. We then extend the genetic algorithm proposed in our first contribution to solve this dynamic control problem.

Our third and last contribution tackles the scalability issue of a centralized decision-making process in a large EC environment. Here, we propose a hierarchical distributed limited look-ahead control approach to reduce the dimensionality of the overall control problem. In this hierarchical distributed control, the entire EC system is partitioned into subsystems. Each subsystem contains a disjoint subset of EC nodes and has its own local controller responsible for control decisions regarding service placement, load distribution, and service migration within this subsystem. At the upper control layer, the global controller receives system-wide information and provides local control goals for the lower control layer, which is composed of local controllers that may exchange information to coordinate their control decisions. Moreover, the global controller considers the entire EC system in a more abstract way to avoid the same scalability issue of a centralized controller.

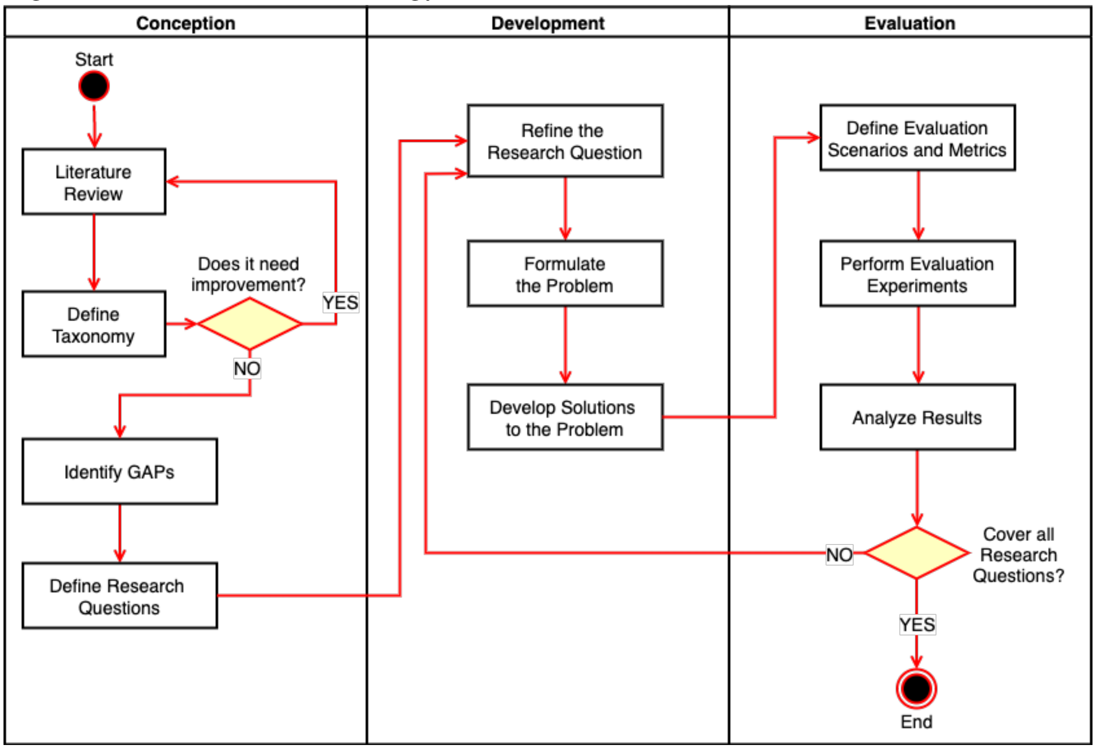
1.4 Research Methodology

An overview of the research methodology used in this thesis is presented in Figure 2, which is organized into three main phases: (*i*) conception, (*ii*) development, and (*iii*) evaluation.

In the conception phase, we review the literature and define a classification taxonomy to identify related work issues. Based on the identified gaps, we define the research problems and questions. After the first phase, the development and evaluation phases are executed using an incremental approach, as briefly discussed in Section 1.3.

In the development phase, we refine the literature review focusing on each research question to discover challenges and solution techniques. For a specific research question, we formally model the related problem and develop solutions (algorithms) to solve it. Next, we define use-case scenarios and metrics to validate and evaluate the proposed solutions through numerical experiments in the evaluation phase. Based on the experiment analysis, we identify strengths, weaknesses, and major contributions of the proposal. Then, we address some of the found limitations in the development and evaluation phases of the subsequent research question. Therefore, these two phases are only completed when all research questions have been covered.

Figure 2 – Research methodology


Conception
Development
Evaluation
Start
Literature Review
Define Taxonomy
Does it need improvement?
YES
NO
Identify GAPs
Define Research Questions
Refine the Research Question
Formulate the Problem
Develop Solutions to the Problem
Define Evaluation Scenarios and Metrics
Perform Evaluation Experiments
Analyze Results
Cover all Research Questions?
NO
YES
End

Source: Author.

1.5 Organization

This thesis consists of seven chapters, including the introduction presented in this chapter. The remaining chapters are organized as follows:

- **Chapter 2** outlines the main concepts related to this thesis: Internet of Things, Cloud Computing, Edge Computing, and 5G Networks. It also introduces mathematical programming as a formalism to model optimization problems throughout this thesis. This chapter then overviews genetic algorithms, including BRKGA and NSGA-II, as a method to solve optimization problems. Furthermore, the LLC concept is presented in this chapter as a technique to control dynamic systems proactively.
- **Chapter 3** presents the related work on service placement in Edge Computing. It also identifies and discusses the limitations of existing works found in the literature according to a classification taxonomy that considers different aspects of a service placement approach.
- **Chapter 4** details our first main contribution, a static approach to service placement with load distribution. Specifically, we model an EC system where replicas of an application can be placed over the EC infrastructure to distribute user-generated load among these replicas. Then, we jointly formulate the service placement and load distribution problems as a single or multi-objective optimization problem. Moreover, we develop a genetic-based algorithm to solve the formulated problem, and we analytically compared its performance against some benchmarking algorithms.
- **Chapter 5** describes our second main contribution, a dynamic and centralized approach to service placement with load distribution and service migration. By following the LLC concept, we model how an EC system evolves under controllable (service placement and load distribution) actions and uncontrollable (user-generated load) events. Moreover, we extend our genetic algorithm to select control actions that optimize system performance over a limited look-ahead prediction horizon. We also provide a performance evaluation where our proposal is compared with different benchmarking algorithms in a scenario with dynamic load synthetically generated.
- **Chapter 6** presents our last main contribution, a dynamic and distributed approach to service placement with load distribution and service migration. In this distributed approach, we decompose that overall control problem into a set of local control problems that are solved in a hierarchical cooperative fashion. Preliminary performance evaluation results are then discussed to compare our centralized and distributed approaches.
- **Chapter 7** concludes this thesis by summarizing the achieved contributions and discussing future research directions.

2 BACKGROUND

This chapter presents the main concepts of areas referenced in this thesis, which are useful for understanding the challenges and solutions mentioned in this research work. Section 2.1 presents the Internet of Things concept, whereas Section 2.2 discusses Cloud Computing, Edge Computing, and 5G network as enabling infrastructure technologies for IoT applications. Section 2.3 introduces the mathematical programming notion to formally model optimization problems. Next, Section 2.4 presents genetic algorithms as a method to solve optimization problems. In Section 2.5, Limited Look-ahead Control is presented as a technique to proactively control dynamic systems. Finally, Section 2.6 summarizes this chapter.

2.1 Internet of Things

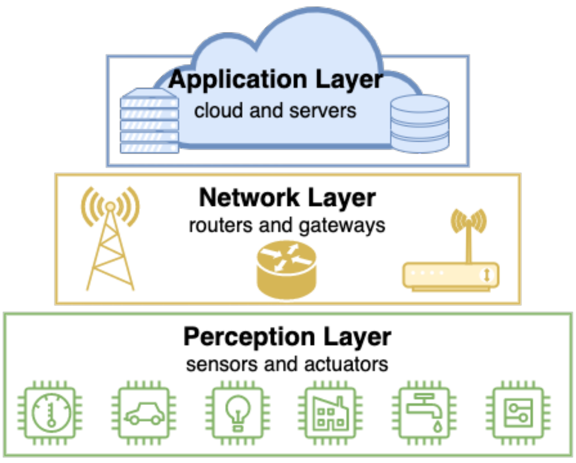
In 1999, Kevin Ashton first coined the term Internet of Things (IoT) in the context of supply chain management by using Radio Frequency Identification (RFID) technology (ASHTON, 2009). After that, the term has evolved to encompass a broader range of application domains (e.g., healthcare, utilities, and transport) and technologies (e.g., wireless sensor networks, mobile networks, big data, and cloud/edge computing). However, there is no unified definition for the IoT term, and there are several definitions proposed by academic and industry organizations (SINGH *et al.*, 2019). For instance, the IoT European Research Cluster (IERC) defined IoT as a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network (VERMESAN *et al.*, 2011).

The fundamental characteristics of IoT can be represented in a basic architecture shown in Figure 3, which is comprised of three distinct layers (AL-FUQAHA *et al.*, 2015): (*i*) perception, (*ii*) network, and (*iii*) application layers. These layers can be briefly described as follows:

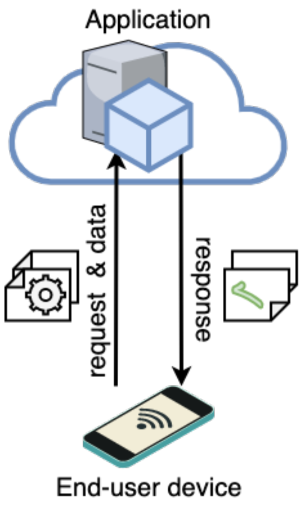
- **Perception layer**. This layer aims to collect useful information from the monitored environment, which are then transformed into digital data. For this purpose, this layer is composed of various types of physical devices, sensors, and actuators. Moreover, these devices, also known as smart objects, are able to exchange data via the network layer with applications and other devices.

- **Network layer**. This layer provides the facility to interconnect the smart objects. Furthermore, it is also in charge of data transmission between the perception and the application layers. Depending on the device features, diverse network technologies (e.g., ZigBee, Bluetooth, WiFi, 4G, and 5G.) can be used to guarantee communication in IoT.
- **Application layer**. It is responsible for delivering services to the end users. For this purpose, applications can store, aggregate, process, and manage the received data from the perception layer. This application layer covers numerous vertical markets, such as smart home, smart building, transportation, industrial automation, and smart healthcare.

Figure 3 – Basic IoT architecture



(a) Three-layer architecture



(b) Application operational pattern

Source: Author.

Following this architecture, IoT applications generally have a common operational pattern (YU *et al.*, 2018). Due to limitations of energy, storage, and computational capability, end-user devices send through the network their collected data in a request message to an application deployed in a remote server. After processing the received request, the application will send the results back to the end users in a response message. This request processing can be data filtering, aggregation, storage, or analysis. Moreover, the nature of a request processing response can be diverse. For example, it can be to intervene in the physical environment through actuators.

IoT relies on an enormous amount of technologies to support its applications (AL-FUQAHA *et al.*, 2015). The next section presents Cloud Computing, Edge Computing, and 5G networks as infrastructure technologies related to this thesis that facilitate the operation of IoT applications.

2.2 Infrastructure Technologies

2.2.1 Cloud Computing

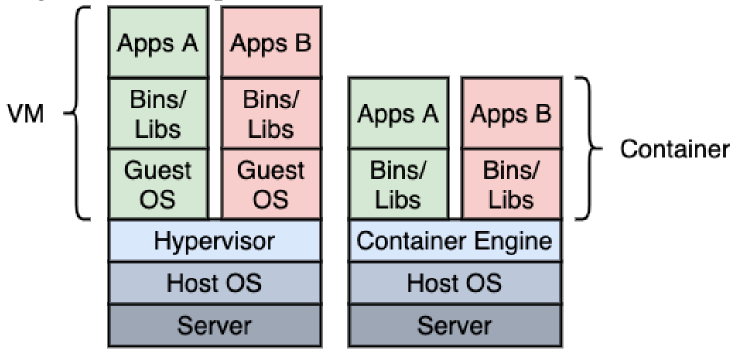
According to the National Institute of Standards and Technology (NIST), Cloud Computing (CC) can be defined as a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (MELL; GRANCE, 2011).

Cloud Computing provides several services that make it attractive to business owners. These CC services can be classified into three types based on the business model (VAQUERO *et al.*, 2009): (*i*) Infrastructure-as-a-Service (IaaS), (*ii*) Platform-as-a-Service (PaaS), and (*iii*) Software-as-a-Service (SaaS). IaaS refers to the on-demand provisioning of virtualized resources (e.g., CPU, memory, and storage) while PaaS provides software environments for developing, deploying, and managing applications. Finally, SaaS provides software applications and composite services to end users and other applications. However, in practice, IaaS and PaaS providers are often parts of the same organization (e.g., Google and Amazon). Therefore, PaaS and IaaS providers are often called InPs or cloud providers, and SaaS providers are named ASPs (ZHANG *et al.*, 2010).

A technical foundation of Cloud Computing lies in the virtualization of resources. For instance, to deploy and run applications in CC, two virtualization technologies are commonly used (TALEB *et al.*, 2017): (*i*) Virtual Machine (VM) and (*ii*) container. The VM technology enables flexibility, isolation, and fine-grained control of resources for applications hosted on virtual machines through a hypervisor software. Moreover, a VM is a physical hardware abstraction and requires a complete Operating System (OS), binaries, and libraries to run applications. As a result, a VM may unnecessarily use a large number of physical machine resources and have slow OS boot processes. These two characteristics can make it difficult to initialize and migrate applications quickly.

In contrast, the container technology occurs at the OS level, where the OS kernel allows the creation of multiple instances (containers) isolated from user-space to run different applications. A container engine (e.g., Docker[1]) utilizes the OS kernel to manage the lifecycles of containers. Compared to a VM image, a container image has a smaller size because it contains only the software packages required to run a specific application. Hence, a container enables easy instantiation and rapid migration of an application due to its lightweight nature. Despite these benefits, containers are less secure than VMs because the kernel is shared among different containers. Specifically, threats and failures at the kernel level, as well as in a container, can affect all containers (SULTAN *et al.*, 2019). Figure 4 illustrates the discussed difference between the architectural representation of VM and container.

Figure 4 – Comparison between VM and container



Source: Author.

2.2.2 *Edge Computing*

Edge Computing is in its early years, still lacking standardized definitions, architectures, and protocols (BILAL *et al.*, 2018). This lack of a standard definition may lead to misunderstandings in the relationship between Cloud Computing and Edge Computing concepts. For instance, one misconception is that EC will move or replace CC, when, instead, it should be considered as a complement to CC. Indeed, EC has some similar characteristics to CC by extending cloud services to the network edges, such as on-demand provisioning of virtualized resources.

Despite the similarities, Edge Computing possesses distinguishing characteristics from Cloud Computing, as shown in Table 1. The main difference between EC and CC lies in the location of their nodes (KHAN *et al.*, 2019). Edge nodes are widely geographically distributed, whereas CC places nodes in a few centralized locations. On the other hand, CC utilizes large data centers with high resource capabilities as nodes, whereas edge nodes are more heterogeneous

[1] https://www.docker.com

and limited in terms of resource capacities. Due to this limited capacity, edge nodes occupy less space than cloud ones, and they can be located a single or few network hops away from end-user devices (YOUSEFPOUR *et al.*, 2019). This proximity to users allows EC to offer significantly lower latencies when compared to CC. Moreover, edge nodes can provide real-time local contextual information to applications (e.g., user mobility and network status) that are not available or limited in CC (GEDEON *et al.*, 2019).

Table 1 – Comparison of Cloud Computing and Edge Computing

Characteristic	**Cloud Computing**	**Edge Computing**
Infrastructure	Centralized	Distributed
Geo-distribution	Locally clustered	Widespread
Resource capacity	High	Low
Heterogeneity	Low	High
Latency	High	Low
Distance to end users	Far (multiple hops)	Near (one or few hops)
Local context awareness	Limited	Supported
Mobility	Limited	Supported

Source: Author.

In recent years, several similar concepts have emerged regarding EC implementation schemes. In the following paragraphs, three main EC implementation concepts are presented:

– **Cloudlet Computing**. Cloudlet is a term coined by Satyanarayanan *et al.* (2009), and it acts as a data center in a box deployed at facilities close to mobile users, such as coffee shops, shopping malls, company buildings, train stations, and hospitals. A cloudlet consists of resource-rich servers or a cluster of servers running VMs to provide the resources demanded by mobile applications. Moreover, a cloudlet is connected to the Internet and offers one-hop wireless access, mainly WiFi, to nearby end users. Hence, cloudlets represent the middle tier of a three-tier architecture, i.e., mobile device, cloudlet, and cloud tiers (TALEB *et al.*, 2017).

– **Mobile Edge Computing**. It is an initiative of the European Telecommunications Standards Institute (ETSI) to bring the capabilities of Cloud Computing and Information Technology (IT) service environment to the Radio Access Network (RAN) of cellular networks (ETSI, 2016). In 2017, ETSI expanded the Mobile Edge Computing (MEC) scope not only to encompass cellular networks but also to include multiple types of network accesses (e.g., 3G, 4G, 5G, WiFi, and fixed access technologies). Thus, it changed the name from Mobile Edge Computing to Multi-access Edge Computing (MEC), yet the acronym MEC remained.

– **Fog Computing**. Cisco initially introduced the concept of Fog Computing as an analogy to the natural phenomenon of fog (DOLUI; DATTA, 2017). Just as the clouds are far above the sky, the fog is closer to the people. Therefore, a fog infrastructure is decentralized and based on nodes located at any point between end-user devices and the cloud. Moreover, fog nodes are heterogeneous and can range from being resource-poor to powerful servers (e.g., end devices, routers, switches, access points, set-top boxes, and 5G base stations) (ROMAN *et al.*, 2018).

Although these concepts (cloudlet, fog, and MEC) aim to bring cloud services closer to end-user devices, there are few subtle differences among them (DOLUI; DATTA, 2017; MOURADIAN *et al.*, 2018). The first difference regards the edge node location as follows: cloudlet is a data center in a box with WiFi access in which users connect directly; nodes in MEC are deployed to cellular base stations, where users connect directly; and fog nodes can be anywhere on the infrastructure, so there is no guarantee of access at a single hop to a fog node. The second difference is that cloudlet relies exclusively on VM technology, while MEC and fog can use other virtualization technologies besides VMs, such as containers. The third difference is that cloudlet and MEC may work in stand-alone mode, i.e., there is no iteration with the cloud. On the other hand, a fog infrastructure always includes the cloud. The final difference is related to the targeted application types, as cloudlet focuses only on computing offloading, while MEC and fog target any application type (e.g., computing offloading, data storage, caching, and processing). Table 2 summarizes these differences.

In this thesis, we use the Edge Computing term to encompass the different emerging concepts that extend the Cloud Computing capabilities to the edge. Furthermore, an edge or EC node refers to any computer-based node with resource capabilities along the path between end-user devices and cloud data centers.

2.2.3 5G Network

Cellular network technologies are continuously evolving in a manner where each generation provides performance enhancements, for example, in terms of data rate, latency, and connection density. The fifth-generation cellular (5G) network continues this trend by providing a significant performance improvement of its predecessors (SHAFI *et al.*, 2017). Compared to

Table 2 – Comparison of Edge Computing concepts

Characteristic	Fog Computing	Mobile-Edge Computing	Cloudlet Computing
Node devices	Routers, switches, access points, gateways	Servers running in base stations	Data Center in a box
Node location	Varying between end-users devices and cloud	RAN	Local/Outdoor installation
Software architecture	Based on fog abstraction layer	Based on mobile orchestrator	Based on cloudlet agent
Proximity	One or multiple hops	One hop	One hop
Access mechanisms	Bluetooth, WiFi, cellular networks	cellular networks	WiFi
Virtualization technology	VM, container	VM, container	VM

Source: Adapted from (DOLUI; DATTA, 2017).

the fourth-generation (4G), 5G intends to achieve a 1000-fold system capacity growth, a 5-fold reduction in end-to-end latency, an energy efficiency of at least ten times, and a 20 times increase of transfer rate (BARB; OTESTEANU, 2020). Table 3 presents the performance comparison between 4G and 5G.

Table 3 – Performance comparison between 4G and 5G

Key Requirements	4G	5G
Peak data rate	1 Gbit/s	20 Gbit/s
User experienced data rate	10 Mbit/s	100 Mbit/s
Mobility	350 km/h	500 km/h
Latency	10 ms	< 1 ms
Connection density	10^5 devices/km^2	10^6 devices/km^2
Area traffic capacity	0.1 Mbit/s/m^2	10 Mbit/s/m^2

Source: Barb and Otesteanu (2020)

5G networks envisage not only performance improvement but also support a wide variety of usage scenarios and applications, which are broadly categorized as follows (ITU-R, 2015):

– **enhanced Mobile Broadband (eMBB).** It is an evolution of existing human-centric mobile broadband applications by improving performance and seamlessly increasing the user experience. This usage scenario covers a range of cases, such as wide-area coverage and hotspot. The wide-area coverage case expects seamless coverage and medium to high mobility, with improved user data rates compared to that offered by 4G. The hotspot case supports high user density and needs very high traffic capacity, but only requires mobility at pedestrian speeds. Moreover, the hotspot case requires higher data rates than the wide-area coverage.

- **Ultra Reliable Low Latency Communications (URLLC).** It has stringent requirements for capabilities such as throughput, latency, and availability. Some examples include tactile Internet applications, factory automation, remote medical surgery, and intelligent transport systems.
- **massive Machine Type Communications (mMTC).** It typically consists of a very large number of connected devices transmitting a relatively low volume of delay-tolerant data. These devices are required to be low cost and have a long battery life.

Table 4 presents the minimum performance requirements for the three 5G use case categories. These requirements are extremely challenging and diverse. Hence, the 5G networks must be flexible and adaptable to a variety of application scenarios. According to the European Telecommunications Standards Institute (ETSI), Edge Computing plays an essential role in achieving some of these requirements, such as low end-to-end latency and bandwidth efficiency (KEKKI *et al.*, 2018).

Table 4 – Key Performance Indicators (KPIs) to assess the performance of 5G use case categories

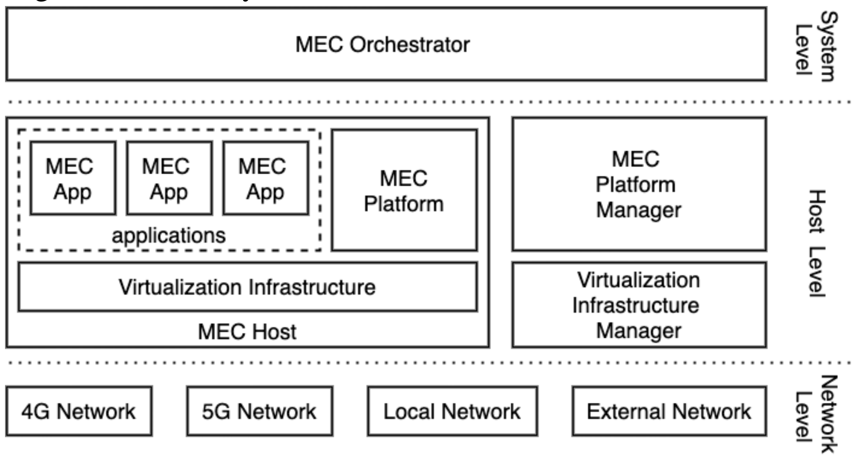
KPI	**Key Use Case**	**Values**
Density	mMTC	$\geq 10,000$ devices/km^2
Mobility	eMBB	Up to 500 km/h
Peak data rate	eMBB	Downlink: 20 Gbps, Uplink: 10 Gbps
User data rate	eMBB	Downlink: 100 Mbps, Uplink: 50 Mbps
User plane latency	eMBB, URLLC	4 ms (eMBB), 1 ms (URLLC)
Control plane latency	eMBB, URLLC	20 ms
Reliability	URLLC	Frame error rate $< 10^{-5}$
Availability	URLLC	> 99 %

Source: Marabissi *et al.* (2018)

ETSI also defines a reference architecture for Multi-access Edge Computing (MEC) that can be integrated into 5G networks (ETSI, 2016). As shown in Figure 5, the MEC architecture is composed of entities grouped into trees levels: (*i*) system, (*ii*) host, and (*iii*) network levels. The MEC orchestrator plays a central role in the system level as it maintains an overall view of the entire MEC system based on deployed MEC hosts, available resources, available MEC services, and topology. It is also responsible for selecting MEC hosts for application instantiation, on-boarding of application packages, triggering application relocation, and triggering application instantiation and termination.

The host level consists of the MEC platform manager, the virtualization infrastructure manager, and the MEC host. The MEC platform manager is responsible for managing the life cycle of applications, providing element management functions, and controlling the application rules and requirements. Meanwhile, the virtualization infrastructure manager is responsible for allocating virtualized resources, preparing the virtualization infrastructure to run software images, provisioning MEC applications, and monitoring application faults and performance. The MEC host facilitates applications, offering a virtualized resource infrastructure and a set of fundamental functionalities (MEC services) required to execute applications, known as the mobile edge platform. Finally, the underlying network level offers integration and connectivity to a variety of accesses, including 4G and 5G networks.

Figure 5 – MEC system architecture



Source: Adapted from (ETSI, 2016) and (KEKKI *et al.*, 2018).

Although the MEC orchestrator can consider diverse application requirements and information on the resources currently available in the MEC system to select one or more MEC hosts to instantiate applications, the ETSI does not intend to specify the actual algorithm that makes this selection. Therefore, it creates new research opportunities to address the application or service placement selection. The rest of this chapter presents some techniques used in this thesis to formalize and solve the service placement problem in Edge Computing.

2.3 Mathematical Programming

A decision-making problem, such as the service placement problem, involves finding the best decision under given circumstances. In general, this decision process aims to minimize or maximize an objective function that measures the goodness of a decision. Hence, optimization is central to a decision-making problem (CHONG; ZAK, 2004). Mathematical optimization, also known as mathematical programming, consists of modeling the optimization in terms of objective, variables, and constraints related to the problem. Formally, a general (single-objective) optimization problem can be expressed as:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & h_i(x) = 0, \qquad i = 1, \ldots, P \\ & g_j(x) \leq 0, \qquad j = 1, \ldots, Q \\ & x \in \mathscr{X} \end{aligned} \tag{2.1}$$

where $f : \mathscr{X} \to \mathbb{R}$ is the (single-)objective function, $h_i : \mathscr{X} \to \mathbb{R}$ are equality constraints, $g_j : \mathscr{X} \to \mathbb{R}$ are inequality constraints, and x is the n-dimensional decision variable. Here, the optimization is written as a minimization problem, but a maximization problem can be obtained by replacing $f(x)$ by $-f(x)$ in problem (2.1).

A point or solution $x \in \mathscr{X}$ is feasible if it satisfies all constraints $h_i(x)$ and $g_j(x)$. Then, problem (2.1) is feasible if there is at least one feasible solution, and infeasible otherwise. Moreover, a solution x^* is said to be an optimal solution if it is feasible and $f(x^*) \leq f(x)$ for all feasible points.

Optimization problems can be classified based on the nature of their variables and functions, as highlighted below:

- **Linear Programming (LP)**. In this case, both objective and constraints are linear functions. Moreover, the decision variables are continuous, i.e., $x \in \mathscr{X} \subset \mathbb{R}^n$. In practice, some algorithms (e.g., simplex method) solve this problem quickly and efficiently, achieving the optimal decision (CHONG; ZAK, 2004).
- **Nonlinear Programming (NLP)**. It includes problems where some of the constraints or objective functions are nonlinear, and the variables are still continuous. Under certain conditions (e.g., convexity), some NLP problems can also be solved quickly and efficiently (BERTSEKAS, 1997).

- **Integer Linear Programming (ILP)**. Similar to LP, ILP problems have linear objectives and constraints; however, the variables are integers, i.e., $x \in \mathscr{X} \subset \mathbb{Z}^n$. As a result of discrete variables, ILP is NP-hard (SCHRIJVER, 1998). It is possible to apply algorithms (e.g., branch-and-bound techniques) to get the exact optimal result in some cases. However, many problems are intractable and, thus, heuristic algorithms are used instead to obtain near-optimal results.
- **Integer Nonlinear Programming (INLP)**. It includes problems where some of the constraints or objective functions are nonlinear, and the variables are discrete. As NLP, INLP is also NP-hard due to integer variables.
- **Mixed-Integer Linear Programming (MILP)**. In this case, the problem variables are composed of discrete and continuous parts, i.e., $x \in \mathscr{X} \subset \mathbb{R}^{n_1} \times \mathbb{Z}^{n_2}$. Moreover, the objective and constraints are linear functions.
- **Mixed-Integer Nonlinear Programming (MINLP)**. It is also a type of integer programming with discrete and continuous variables, but some (objective and constraint) functions are nonlinear. MINLP is still NP-hard, and, in practice, it can be harder to solve than its linear counterpart (BURER; LETCHFORD, 2012).

Typical methods to solve nonlinear problems (e.g., NLP, INLP and MINLP) are associated with relaxation and constraint enforcement concepts (BELOTTI *et al.*, 2013). A relaxation technique transforms a nonlinear problem into an easier one to be solved; for instance, a linearization relaxation that transforms the nonlinear problem into an LP, ILP or MILP problem. Then, a constraint enforcement procedure excludes solutions that are feasible to the relaxation, but not to the original problem.

2.3.1 Multi-Objective Optimization

Although problem (2.1) aims to optimize a single objective function $f(x)$, many real-world decision-making processes try to simultaneously optimize multiple objective functions while satisfying a list of constraints (BRANKE *et al.*, 2008). Let $F = (f_1, f_2, \ldots, f_M)$ be a list of objective functions and, then, a general multi-objective optimization problem can be formulated as:

$$\begin{aligned} \min \quad & F(x) = (f_1(x), f_2(x), \dots, f_M(x)) & \\ \text{s.t.} \quad & h_i(x) = 0, & i = 1, \dots, P \\ & g_j(x) \leq 0, & j = 1, \dots, Q \\ & x \in \mathscr{X} & \end{aligned} \tag{2.2}$$

Unlike single-objective optimization problems that may have a unique optimal solution, in a multi-objective case, conflicts among objectives usually prevent the problem from having a single optimal solution that can simultaneously optimize all objectives. In this way, the improvement of an objective may lead to the deterioration of another. Therefore, it is necessary to search for compromise solutions by considering trade-offs among the conflicting objectives.

The concept of Pareto dominance plays a vital role in finding a set of best trade-off solutions that cannot be improved in any of the objectives without degrading at least one of the other objectives (BRANKE *et al.*, 2008). This concept is formally defined as follows:

Definition 2.1 (Pareto dominance) *Given the multi-objective problem* (2.2)*, a feasible solution* x_1 *Pareto-dominates another solution* x_2*, expressed as* $x_1 \prec x_2$*, when*

$$\begin{aligned} x_1 \prec x_2 \textit{ iff } f_i(x_1) &\leq f_i(x_2) \qquad \forall i \in \{1, 2, \dots, M\} \\ \textit{and } f_j(x_1) &< f_j(x_2) \qquad \exists j \in \{1, 2, \dots, M\} \end{aligned}$$

The set of all solutions that are not dominated by any other solution is called the Pareto optimal set or the Pareto optimal front. These non-dominated solutions are considered equally good if there is no additional preference information.

When a decision maker is able to provide preference information related to the objectives, classical a priori methods can be used to solve the multi-objective problem (BRANKE *et al.*, 2008). For instance, a scalarization technique (e.g., weighted sum and ε-constrained methods) transforms a multi-objective problem into a single-objective optimization based on some preference information. Another example of a priori method is the lexicographic method that consists of solving a sequence of single-objective optimizations based on a preference order among the objective functions.

On the other hand, when preference information is not available or is hard to obtain in practice, methods based on the Pareto dominance concept can be used to solve the multi-objective problem. In particular, evolutionary algorithms, such as genetic algorithms, are broadly adopted in the literature for multi-objective optimization (ZHOU *et al.*, 2011). Hence, the next section presents genetic algorithms to handle both single and multiple objectives cases.

2.4 Genetic Algorithms

Genetic Algorithms (GAs) are meta-heuristic methods for solving optimization problems inspired by the process of natural selection. In a GA, a population of candidate solutions, called individuals, to an optimization problem is evolved toward an optimal solution (GOLDBERG; HOLLAND, 1988; HOLLAND *et al.*, 1992). Each individual has a corresponding chromosome that encodes the solution. For instance, a traditional chromosome is represented by a string or vector of binary values, but other encodings are also possible. Moreover, a chromosome is associated with a fitness level, which corresponds to the objective function value of the solution it encodes. The GA is an iterative process where at each step, called generation, it creates a new population by recombining or randomly mutating chromosome elements of selected individuals of the current population to produce offspring individuals that make up the next generation. Individuals are selected stochastically, but those with better fitness are preferred over those that are less fit. Usually, GA terminates when either produces a maximum number of generations or reaches a satisfactory fitness level.

A genetic approach has the advantages of not being limited to linear or single-objective problems and being implemented in a parallel computing environment (CUI *et al.*, 2017). In the two next subsections, we exemplify the wide-spread applicability of GAs by describing two algorithms employed in this thesis. Subsection 2.4.1 presents a GA that handles constrained optimizations, whereas Subsection 2.4.2 introduces a GA for multi-objective problems. After presenting these two algorithms, Subsection 2.4.3 discuss stopping criteria for GAs.

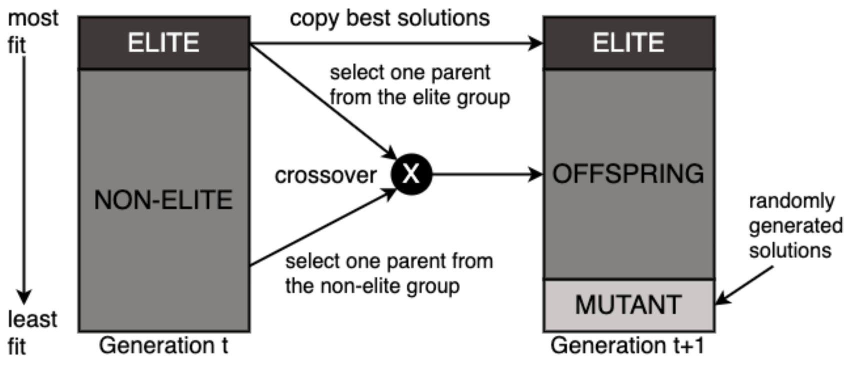
2.4.1 Biased Random-Key Genetic Algorithm

GAs are typically applied for unconstrained optimization problems. A common way of incorporating constraints into a GA is through penalty functions that add a certain value to the objective function based on the amount of constraint violation present in a specific solution. Nonetheless, it may be extremely difficult to estimate good penalty factors or even generate a single feasible solution for some complex optimization problems (COELLO, 2002).

An alternative method to handle constrained-optimization problems in GAs is to develop (*i*) special solution representations to simplify the shape of the search space, and (*ii*) special operators to preserve the feasibility of solutions at all times. Gonçalves and Resende (2011) adopt this method by proposing a Biased Random-Key Genetic Algorithm (BRKGA) where chromosomes are represented as a vector of randomly generated real numbers. Moreover, a deterministic algorithm, named decoder, takes any chromosome as input and associates it with a feasible solution of an optimization problem, for which an objective value or fitness can be computed. In other words, a chromosome gives instructions on how to build a feasible solution.

Figure 6 illustrates how BRKGA evolves the current population for the next generation. The initial population is made up of vectors of random values, or keys, in the real interval $[0,1]$. After the fitness of each individual is computed by the decoder, the algorithm partitions the population into two groups of individuals: a small group of elite individuals, i.e., those with the best fitness values, and the remaining set of non-elite individuals. BRKGA uses an elitist strategy to keep all of the current elite individuals, without modification, in the next generation. This strategy keeps track of good solutions found during the algorithm iterations, resulting in a monotonically improving heuristic. It also adds a small number of mutated individuals in the next generation to escape from entrapment in local minima. A mutant individual is simply a vector of random keys generated in the same way that the initial population is created. The next population is completed by offspring individuals produced through the mating process called crossover.

Figure 6 – Generation transition in BRKGA

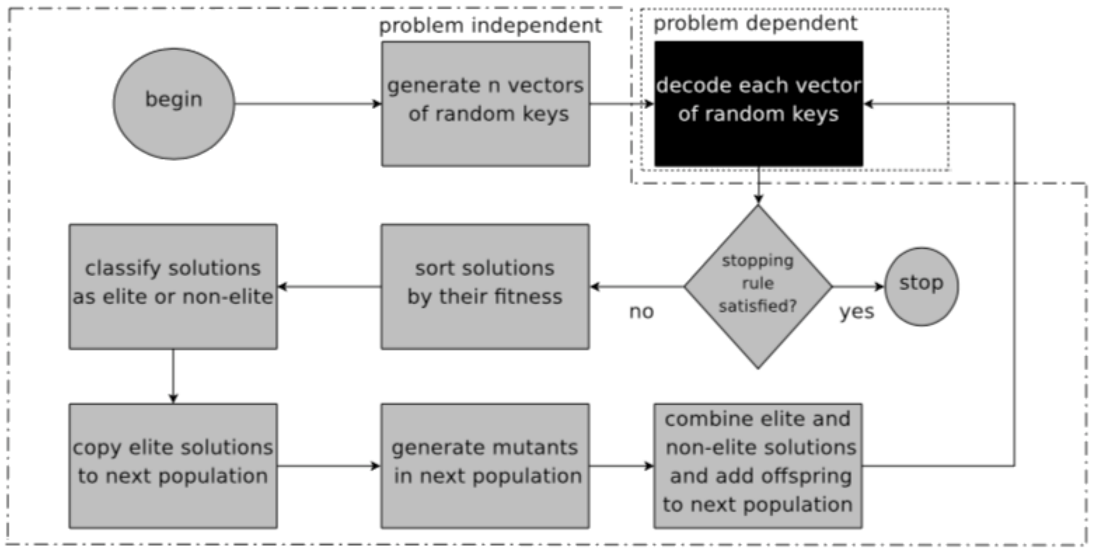

most fit
ELITE
copy best solutions
ELITE
select one parent from the elite group
crossover
X
OFFSPRING
NON-ELITE
randomly generated solutions
select one parent from the non-elite group
MUTANT
least fit
Generation t
Generation t+1

Source: Adapted from (GONÇALVES; RESENDE, 2011)

In BRKGA, each new offspring individual produced by crossover is a combination of one solution randomly selected from the group of elite individuals and one from the set of non-elite individuals in the current population. More specifically, BRKGA uses parameterized uniform crossover (SPEARS; JONG, 1995) to combine two parent solutions and obtain a new offspring. Let N be the length of the chromosome modeled as a vector and P_{elite} as the probability that an offspring inherits a genetic characteristic of its elite parent. Then, in the parameterized uniform crossover, each element at the position $i \in \{1,\ldots,N\}$ of an offspring vector takes on the value of the i-th element of the elite parent with probability P_{elite} and the value of the i-th element of the non-elite parent with probability $1-P_{\text{elite}}$. By setting $P_{\text{elite}} > 0.5$, the offspring is more likely to inherit characteristics of the elite parent than those of the non-elite parent. Another observation is that offspring solutions resulting from mating are always feasible because of the assumption that any random key vector can be decoded into a feasible solution.

Figure 7 shows an overview of the BRKGA flowchart, which is divided into two parts: (*i*) the problem-independent and (*ii*) the problem-dependent parts. The problem-independent part has no knowledge of the problem being solved, and it is responsible for the solution searching process based on the genetic operations (initialization, crossover, classification, selection, mutation, and stop operations). The only connection to the optimization problem being solved is the problem-dependent part of the GA, where the decoder produces solutions from random-key vectors and computes the fitness of these solutions. Therefore, when designing a new meta-heuristic for a specific optimization problem, we only need to specify the chromosome representation and the decoder algorithm.

Figure 7 – BRKGA flowchart


problem independent
problem dependent
begin
generate n vectors of random keys
decode each vector of random keys
stopping rule satisfied?
yes
stop
no
sort solutions by their fitness
classify solutions as elite or non-elite
copy elite solutions to next population
generate mutants in next population
combine elite and non-elite solutions and add offspring to next population

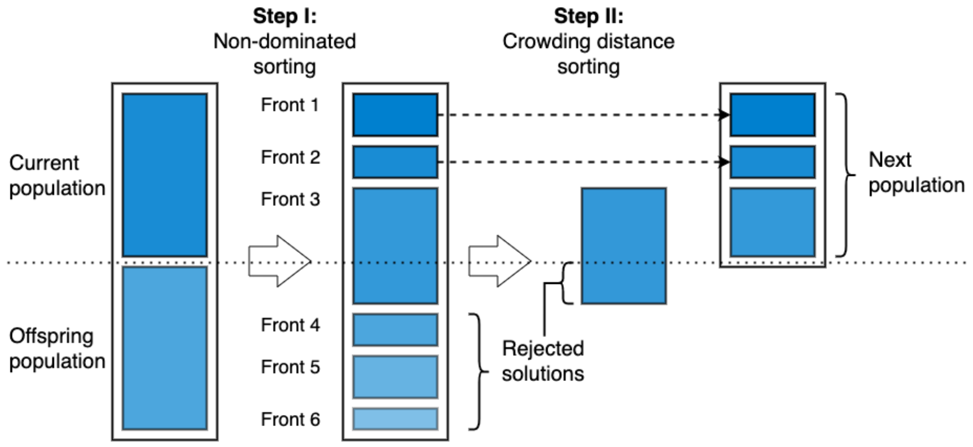
Source: Gonçalves and Resende (2011)

2.4.2 Non-dominated Sorting Genetic Algorithm

In each iteration, a genetic approach evolves a population of individuals toward better solutions. This population-based approach is a perfect match for multi-objective optimization problems, where an iteration can simultaneously find multiple compromised solutions according to the Pareto dominance concept (BRANKE *et al.*, 2008). The Non-dominated Sorting Genetic Algorithm II (NSGA-II) proposed by Deb *et al.* (2002) is one of the most popular multi-objective genetic algorithms, which uses elitist principle, non-dominance sorting, and crowding distance sorting to find Pareto-optimal solutions.

As shown in Figure 8, a new population is comprised of the current population and its offspring individuals. NSGA-II then classifies this new population by a fast sorting procedure with two steps: (*i*) non-dominated sorting and (*ii*) crowding distance sorting. In the first step, NSGA-II calculates the non-domination level of each solution. In other words, it counts the number of other solutions that dominate a specific solution as the non-domination level of this specific solution. Then, this first step sorts solutions according to the ascending level of non-domination, and it groups solutions with the same level in a front.

Figure 8 – NSGA-II classification and selection procedures



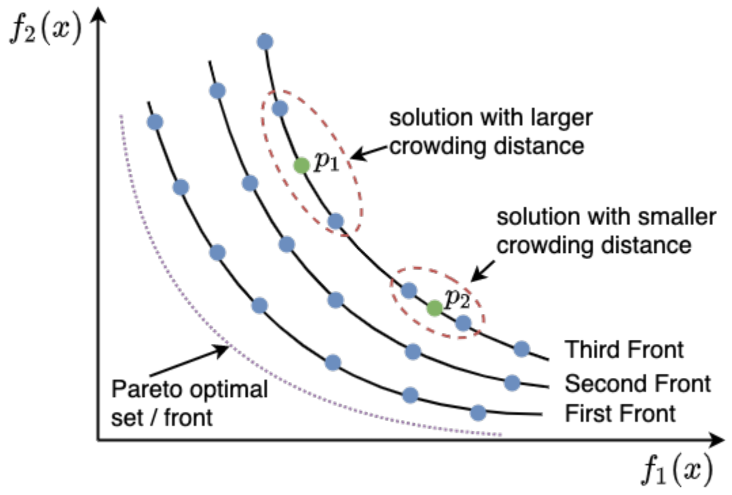
Source: Adapted from (DEB *et al.*, 2002).

The classification second step is a diversity preservation mechanism. This mechanism orders solutions on the same front using crowding distance, which informs the density of solutions surrounding a particular solution. Algorithm 1 proposed by Deb *et al.* (2002) calculates the crowding distance of a solution or point as the average distance of two points on either side of this point along each of the objectives. In order to preserve solution diversity, less dense solutions are preferred and, thus, the second step sorts a population in descending order of crowding distance in each front.

```
Algorithm 1: Crowding distance assignment
   Data: List of solutions in front $\mathscr{F}$, list of objective functions $F = (f_1, f_2, \ldots, f_M)$
   Result: Crowding distance $D$
 1 $L \leftarrow |\mathscr{F}|$;                          // number of solutions in the front
 2 forall $i \in \mathscr{F}$ do
 3     $D_i \leftarrow 0$;                    // initial crowding distance of solution $i$
   /* For each objective function                                          */
 4 forall $f_k \in F = (f_1, f_2, \ldots, f_M)$ do
 5     $f_k^{max}, f_k^{min} \leftarrow$ maximum and minimum values of function $f_k$;
 6     $S \leftarrow$ sort solutions in $\mathscr{F}$ according to the objective function $f_k$;
 7     $D_{S_1} \leftarrow \infty$;                      // distance of the first element in $S$
 8     $D_{S_L} \leftarrow \infty$;                       // distance of the last element in $S$
       /* For each solution in the front                                   */
 9     for $i \leftarrow 2$ to $L - 1$ do
10         $D_{S_i} \leftarrow D_{S_i} + \frac{f_k(S_{i+1}) - f_k(S_{i-1})}{f_k^{max} - f_k^{min}}$; // distance of the $i$-th element in $S$
```

Figure 9 depicts the result of a population classification by NSGA-II for an optimization problem with two objective functions, f_1 and f_2. In this figure, solutions as points in a curve belong to the same front. Moreover, NSGA-II groups solution in fronts so that the i-th front is more close to the Pareto optimal front than the $(i+1)$-th front. We can also observe that solution p_1 has a crowding distance larger than solution p_2 because p_1 is more distant from its neighbors in the third front than solution p_2.

Figure 9 – Example of population classified by NSGA-II


$f_2(x)$
solution with larger crowding distance
p_1
solution with smaller crowding distance
p_2
Third Front
Second Front
First Front
Pareto optimal set / front
$f_1(x)$

Source: Author.

After sorting the new population, the resulted position of an individual is assigned as its rank. Then, NSGA-II only selects the best-ranked individuals/solutions for the next population. Finally, the overall complexity of the classification and selection procedures is $O\left(MN_{\text{pop}}^2\right)$, where M is the number of objective functions and N_{pop} is the population size (DEB *et al.*, 2002).

2.4.3 Stopping Criteria

In a GA, a stopping criterion is invoked at the end of its current iteration. The most popular criterion is just to terminate the algorithm after a given number of iterations/generations $t_{\max}$. However, an inadequate $t_{\max}$ may result in unsatisfactory solutions in terms of optimality if the $t_{\max}$ value is too low or a waste of computational resources if $t_{\max}$ is too high. Hence, another stopping criterion that detects scenarios where there is no sense in proceeding with the algorithm execution may be necessary. For instance, a GA terminates when the solution obtained so far is satisfactory or a better one is unlikely to be produced.

Martí *et al.* (2016) propose a stopping criterion for multi-objective problems called MGBM after the authors surnames. MGBM combines a local improvement indicator, named Mutual Domination Rate (MDR), and a global evidence-gathering process to detect situations where no further progress will be made. Moreover, MGBM can be integrated with NSGA-II by using the concept of non-dominated fronts.

The MDR indicator I_{mdr} contrasts the non-dominated individuals of the current and preceding iterations in order to compute a measure of the improvement produced by the current iteration. Equation (2.3) specifies this indicator where $\mathscr{F}_{t-1}^1$ and $\mathscr{F}_t^1$ are the first non-dominated front produced by NSGA-II at iteration/generation $t-1$ and t, respectively. The function $\Delta(A,B)$ returns the set of elements of A that are dominated by at least one element of B. According to Martí *et al.* (2016), the order of complexity of calculating I_{mdr} is $O\left(M|\mathscr{F}_{t-1}^1||\mathscr{F}_t^1|\right)$ when there are M objective functions.

$$I_{\text{mdr}}(t) = \frac{|\Delta\left(\mathscr{F}_{t-1}^1, \mathscr{F}_t^1\right)|}{|\mathscr{F}_{t-1}^1|} - \frac{|\Delta\left(\mathscr{F}_t^1, \mathscr{F}_{t-1}^1\right)|}{|\mathscr{F}_t^1|} \tag{2.3a}$$

$$\Delta(A,B) = \{x \mid x \in A \text{ and } \exists y \in B, y \prec x\} \tag{2.3b}$$

The $I_{\text{mdr}}(t) \in [-1,1]$ indicator can be interpreted as follows. If $I_{\text{mdr}}(t) = 1$, the entire first-front population of iteration t is better than its predecessor. If $I_{\text{mdr}}(t) = 0$, there has not been any substantial progress. In the worst case, $I_{\text{mdr}}(t) = -1$ indicates that iteration t has not improved any of its predecessor solutions.

In order to detect if a GA has made progress, MGBM uses a recursive estimation based on a simplified Kalman filter to predict when the MDR indicator has stabilized around zero. By using one-dimensional estimation, and assuming full progress estimation from the start and that the observation noise is equal to the initial error covariance, the progress estimation $\hat{I}_{\mathrm{mdr}}$ at generation $t \geq 1$ can be defined as (MARTÍ *et al.*, 2016):

$$\hat{I}_{\mathrm{mdr}}(t) = \begin{cases} 1 & t = 1 \\ \hat{I}_{\mathrm{mdr}}(t-1) + \frac{1}{t+1}\left(I_{\mathrm{mdr}}(t) - \hat{I}_{\mathrm{mdr}}(t-1)\right) & t > 1 \end{cases} \tag{2.4}$$

Therefore, a GA terminates when the progress estimation falls below a defined threshold, i.e., $\hat{I}_{\mathrm{mdr}}(t) < \hat{I}_{\mathrm{mdr}}^{\mathrm{min}}$. As a safety measure for a possible non/slow convergence of $\hat{I}_{\mathrm{mdr}}(t)$, the maximum number of iterations can also be used in conjunction with MGBM.

2.5 Limited Look-ahead Control

In a dynamic computer system, various performance-related parameters must be continuously tuned to achieve the desired performance under dynamic operating conditions. For instance, an Edge Computing system reassesses its service placement decisions over time to meet a certain response time for a time-varying service demand while minimizing operational costs. Feedback methods in control theory, such as Model Predictive Control (MPC), are promising approaches to automate these dynamic systems. A feedback control first observes the current system state, and then, it takes corrective action, if any, to achieve the desired performance. MPC is a proactive feedback control that takes control actions accordingly to current measured information and predicted future events. However, traditional feedback controls, including MPC, assume a linearized model for system dynamics with continuous variables and unconstrained state space (KANDASAMY *et al.*, 2006).

The Limited Look-ahead Control (LLC) is a proactive method similar to MPC but supporting nonlinear systems with mixed continuous and discrete variables under explicit and dynamic operating constraints (ABDELWAHED *et al.*, 2004). Hence, the basic LLC concept is to operate a system by continuously monitoring its current state and selecting control actions that best satisfy the given specifications when applied in the system. Moreover, control actions are obtained by optimizing system behavior, as forecasted by a mathematical model, for the specified performance criteria over a limited look-ahead prediction horizon.

In order to apply a mathematical technique, an abstract model of the underlying system is required. In an LLC approach, the following discrete-time equation describes the system dynamics:

$$s(t+1) = \Phi\left(s(t), c(t), e(t)\right) \tag{2.5}$$

where t is the discrete-time index, $s(t)$ is the system state or output, $c(t)$ denotes the control input or decision variable, and $e(t)$ represents the environment input or disturbance at time step t. In general, environmental inputs are uncontrollable (e.g., system incoming workload), but they can be estimated using well-known forecasting techniques, such as the AutoRegressive Integrated Moving Average (ARIMA) described in (HYNDMAN; ATHANASOPOULOS, 2018). Furthermore, $\Phi(\cdot)$, called the system dynamics or behavioral model, captures the relationship between a system state and its (control and environment) inputs.

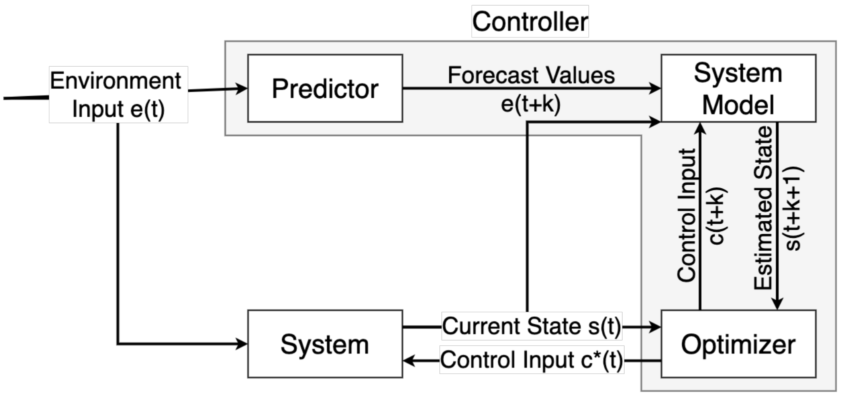
Figure 10 shows the overall framework of a decision-maker controller using LLC. The functional components within the framework are described as follows:

– **Predictor.** It forecasts relevant environment parameters over a limited look-ahead prediction horizon of length H to be used by the system model.

– **System Model.** Given the current system state $s(t)$ and the predicted environment inputs, this component estimates a system state $s(t+k+1), k \in [t, t+H-1]$ within the prediction horizon H by using the dynamic model $\Phi(\cdot)$ when it receives a control input $c(t+k)$.

– **Optimizer.** Given a sequence of control inputs $\pi_c = \{c(k) \mid k \in [t, t+H-1]\}$, it uses the system model to constructs a set of future states from the observed state $s(t)$ up to a prediction horizon H. This component objective is to select the optimal sequence $\pi_{c^*} = \{c^*(k) \mid k \in [t, t+H-1]\}$ of control decisions that optimize the system performance while satisfying both state and input constraints.

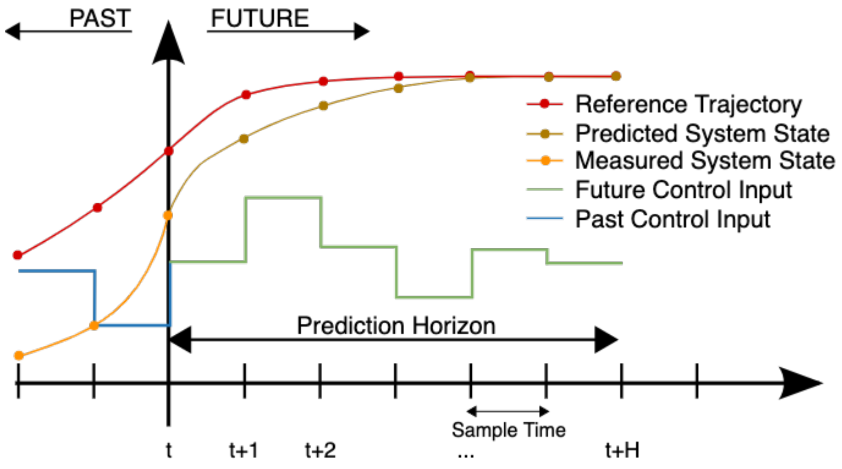
Figure 10 – LLC controller framework



Source: Adapted from (ABDELWAHED *et al.*, 2004).

Figure 11 illustrates the LLC basic working principle for a system with a one-dimensional system state and control input. At the current time t, a sequence of future control inputs or actions is selected based on how close the predicted system states are to the desired performance reference trajectory over a discrete and limited look-ahead prediction horizon of length H. For each time step within the prediction horizon, the predicted system state depends on the selected control input applied to the system at this time step, the past system states, and the system dynamics model.

Figure 11 – A basic working principle of LLC



Source: Adapted from <https://bit.ly/2GXOCni>.

At the beginning of a time step t, the controller performs the optimization of problem (2.6) to select a sequence of control inputs π_{c^*}. In this problem, $f(\cdot)$ is the performance function to be optimized, and $\mathscr{C}$ is the finite set of all possible control inputs. In addition, $h(\cdot)$ and $g(\cdot)$ are lists of equality and inequality constraints upon the system state and inputs, respectively. After solving the problem, the controller applies the first control input $c^*(t)$ of the selected sequence into the system. This process is repeated at time step $t+1$ when the new measured system state $s(t+1)$ is available.

$$\begin{aligned}
\min \quad & \sum_{k=t}^{t+H-1} f\left(s(k+1), c(k), e(k)\right) & & \\
\text{s.t.} \quad & s(k+1) = \Phi\left(s(k), c(k), e(k)\right), & & \forall k \in [t, t+H-1] \\
& h\left(s(k), c(k), e(k)\right) = 0, & & \forall k \in [t, t+H-1] \\
& g\left(s(k), c(k), e(k)\right) \leq 0, & & \forall k \in [t, t+H-1] \\
& c(k) \in \mathscr{C}, & & \forall k \in [t, t+H-1]
\end{aligned} \tag{2.6}$$

In order to solve problem (2.6), it is sufficient to use an algorithm that exhaustively evaluates all possible operating states within the prediction horizon to determine the best control input for a system with few control options and a small prediction horizon. However, more advanced techniques, such as GAs, in a system with a large control input set are necessary to solve problem (2.6) due to the exponential growth of possible input combinations for a control sequence (ABDELWAHED *et al.*, 2004; BAI; ABDELWAHED, 2009). Therefore, this thesis intends to develop LLC algorithms in the context of the service placement problem, which can have diverse control options.

2.6 Summary

This chapter described the main concepts involved in this thesis work: Internet of Things (IoT), Cloud Computing (CC), Edge Computing (EC), 5G Network, Mathematical Programming, Genetic Algorithm (GA), and Limited Look-ahead Control (LLC).

First, this chapter presented a brief overview of IoT concepts, including an operational pattern for IoT applications based on a three-layer architecture. After that, CC, EC, and 5G concepts were explained as enabling technologies for IoT. Furthermore, similarities and differences among CC, EC, and other related concepts (Fog Computing, Cloudlet Computing, and MEC) were discussed.

Second, this chapter introduces mathematical programming as a formal manner to model optimization problems, such as the service placement problem. Furthermore, this chapter presents a classification of optimization problems based on the nature of their variables, constraints, and objective functions. Then, genetic algorithms were depicted as a meta-heuristic method for solving optimization problems. After that, this chapter detailed two genetic algorithms (BRKGA and NSGA-II) employed in this thesis to handle constrained and multi-objective problems. Moreover, MGBM was presented as a stopping criterion for multi-objective genetic algorithms.

Finally, the LLC was presented as a proactive method to control systems operating under dynamic environmental conditions, such as the service placement control for an Edge Computing system with dynamic load addressed in this research work.

3 RELATED WORK

This chapter presents the related works on service placement problem in Edge Computing, which is this thesis topic. Given the complexity of this problem, the existing literature considers different assumptions, characteristics, and strategies to propose efficient service placement. Consequently, we present a taxonomy to classify the surveyed works in order to identify their strengths and limitations and, thus, point out research opportunities.

The remainder of the chapter is organized as follows. First, we describe a taxonomy to compare service placement approaches in Section 3.1. Then, Section 3.2 discusses and classifies the related work to identify research gaps. Finally, this chapter is summarized in Section 3.3.

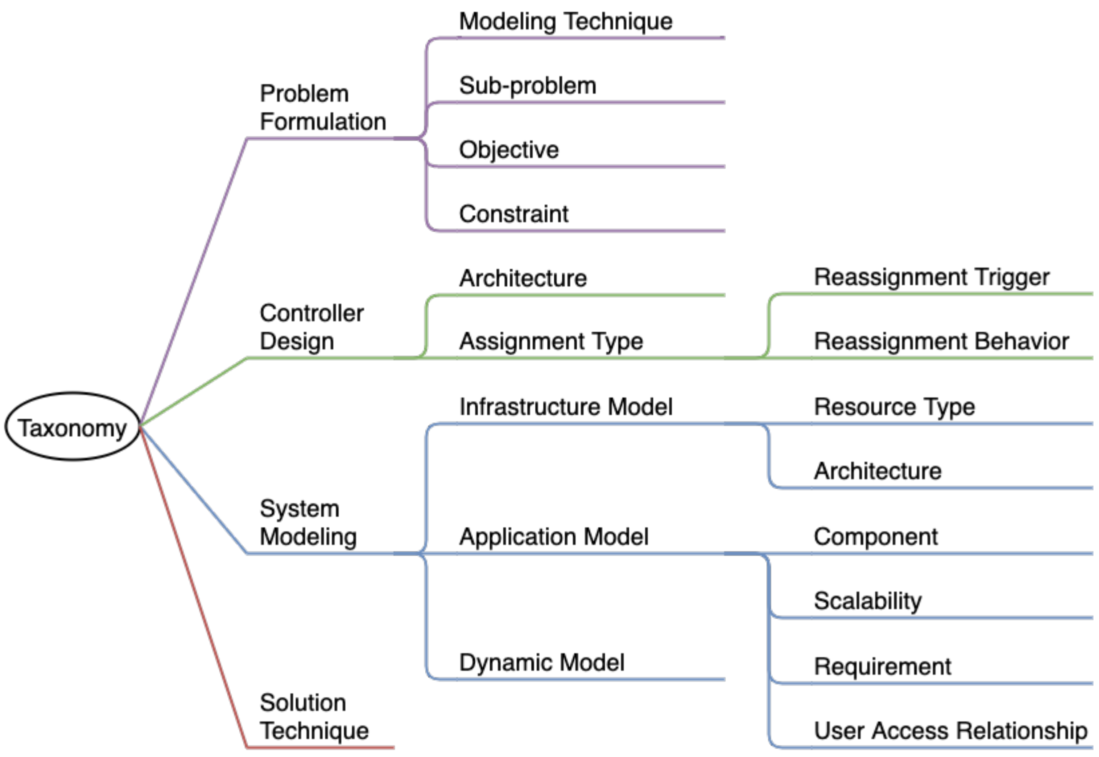
3.1 Classification Taxonomy

There are some taxonomies to classify service placement approaches in the literature. In the Cloud Computing (CC) context, Pires and Barán (2015) propose a taxonomy for Virtual Machine (VM) placement based on three main aspects: optimization approach (mono-objective, multi-objective solved as mono-objective, and pure multi-objective), objective function, and algorithm technique. In (PIETRI; SAKELLARIOU, 2016), the authors categorize VM placement in CC using the following criteria: sub-problem (initial placement and reallocation problem), optimization objective, scheduling type (event-driven, periodic, and threshold-based), optimization technique, and evaluation platform. Masdari *et al.* (2016) classify VM placement schemes as static and dynamic, and a dynamic placement can be further characterized as reactive or proactive.

Regarding service placement in Edge Computing (EC), Brogi *et al.* (2020) survey existing proposals based on their considered constraints, optimized metrics, and the algorithmic solution used. In (SALAHT *et al.*, 2020), the authors propose a service placement taxonomy by taking into consideration the following elements: control plan design (centralized vs. distributed), placement characteristic (online vs. offline), system dynamicity, and mobility support.

Based on the aforementioned taxonomies, we present a taxonomy to provide an in-depth classification of works related to the service placement problem in EC. As shown in Figure 12, this taxonomy has four main aspects: (*i*) problem formulation, (*ii*) controller design, (*iii*) system modeling, and (*iv*) solution technique. We describe these aspects in the next subsections.

Figure 12 – Taxonomy for service placement problem in Edge Computing



Source: Author.

3.1.1 Problem Formulation

Typically, the service placement problem is seen as an optimization problem that can be formally modeled by different techniques to deal with various sub-problems, objectives, and constraints. Thus, the following four elements can be used to classify how the service placement problem is formulated in the literature:

Modeling technique. There are different techniques to model an optimization problem. Mathematical programming is often found in the literature to model an optimization problem with goals and constraints. It includes ILP, MILP, MINLP, and other variations. Other techniques, such as game theory and Markov Decision Process (MDP), can also be used to capture different aspects of the problem.

Sub-problem. Besides the service placement decision, the problem statement can also address other related sub-problems, such as service migration and load distribution. Service placement refers to the mapping of applications or services onto hosting nodes. Service migration or reallocation involves the readjustment overtime of placement decisions due to system changes. Load distribution or balancing refers to the distribution of workloads, tasks, or requests among multiple nodes hosting the associated application. Moreover, these sub-problems can be formulated and optimized either independently or jointly.

Objective. It is a function that measures the suitability of a solution within the optimization process. There are several objective functions in the literature to be minimized or maximized; for instance, minimize cost, response time, and migration cost. Thus, we classify placement proposals in the following categories: (*i*) single-objective, (*ii*) multi-objective as single-objective, (*iii*) pure multi-objective. A single-objective approach considers the optimization of only one objective function. In the second category, multiple objective functions are combined into one objective function. Meanwhile, a pure multi-objective approach directly optimizes a collection of objective functions by using multi-objective operations, such as the Pareto dominance.

Constraint. An optimization process can include a set of constraints specifying conditions regarding decision variables that must be satisfied by a feasible solution. Particularly for the placement problem, we classify constraints as those related to infrastructure or applications. In the infrastructure case, a common constraint concerns the availability of computing, storage, and network resources. Application constraints may include the satisfaction of QoS requirements and the limitation of costs within a budget.

3.1.2 Controller Design

An essential part of the development of a placement strategy is the design of a controller responsible for the decision-making process. We depict hereafter two aspects of this design, the architecture and assignment type, as follows:

Architecture. A controller architecture informs where the decision process is performed. A centralized approach consists of a single central control unit operating all tasks related to the decision-making process. A distributed architecture maintains a central unit, but some of its tasks are distributed among multiple sub-units to increase scalability. Unlike these two architectures, a decentralized controller does not have a central unit. Instead, several control units compute their own decisions based on local resources and information. For instance, each hosting node can decide for itself what services to deploy to it. Furthermore, a control unit can make decisions independently or through coordination with other units.

Assignment type. A service placement assignment can be done either statically or dynamically. The static assignment, also known as an offline or initial assignment, performs a placement decision at the pre-execution time and does not change it for a long time. It requires all information related to the optimization process to be available in advance. On the other hand, a dynamic or online assignment occurs during the controller run-time. Moreover, in this dynamic case, placement decisions take into account the current state of the system and reallocation actions.

3.1.2.1 Dynamic Assignment

We can further classify a dynamic controller according to when a placement reassignment will be triggered and if it has a reactive or proactive behavior.

Reassignment trigger. A triggering mechanism is required to reassess the placement decision, where periodic, event-driven, and threshold-based are common triggers. A periodic trigger invokes the optimization process at predefined intervals. In an event-driven mechanism, some decisions are taken when certain events (e.g., the arrival of a new application) are detected. A threshold-based method monitors performance-related metrics to trigger the placement reassessment when a threshold is exceeded.

Reassignment behavior. A controller has a reactive behavior when it changes the current placement mapping only after the system has reached a particular undesired state. In contrast, a proactive controller involves prediction-based approaches that help prepare in advance for system state changes.

3.1.3 System Modeling

The statement of service placement problem goes through the description of a system model, which we can decompose in the following parts: (*i*) infrastructure, (*ii*) application, and (*iii*) dynamic models.

3.1.3.1 Infrastructure Model

The EC infrastructure comprises the network of hosting nodes and the resources required to deploy applications. We can categorize this infrastructure based on the resource types and architecture considered in its model, which are described as follows:

Resource type. In order to deploy an application in a node, it is necessary to allocate resources to this application. A placement strategy can focus on allocating a single resource type (e.g., CPU, memory, storage, and bandwidth) or multiple resource types.

Architecture. Generally, EC has an architecture composed of three-tiers: end-user devices, edge, and cloud layers. Thus, the placement of services can occur in a single (i.e., edge layer) or multiple tiers (e.g., edge and cloud layers).

3.1.3.2 Application Model

Diverse characteristics can be used to model applications deployed in EC. We highlight three main characteristics: components, scalability, requirements, and user access relationship.

Component. An application is generally composed of several functional parts called components. According to the composition abstraction level, a placement approach can focus on deploying these components as separate or unified pieces. At the high-grained abstraction level, applications are designed to be self-contained, i.e., containing all their functional components, and then placed as a single piece based on the container or VM technologies. Meanwhile, at the fine-grained level, an application contains a set of components (e.g., micro-services) that can be placed separately in different locations. Furthermore, each component has its own characteristics, and there may be a dependency graph between components. Hence, at the fine-grained level, application placement refers to the placement of its components.

Scalability. Some applications support scalability or elasticity to handle workload variation without degrading the performance. An application can scale on the vertical and horizontal axes. Vertical scaling means adding or removing resources of a single hosting node to an application. Likewise, horizontal scaling refers to placing an application in more nodes or removing the application from some nodes hosting it.

Requirement. Applications define requirements that should be satisfied by placement approaches. We summarize some of those requirements based on QoS (e.g., maximum response time and minimum availability), resource, and cost.

User access relationship. Concerning the access relationship between end users and applications, we can category the access into two cases: single-user and multiple-users. The single case means that each end user has a dedicated application. In this case, a user sends request to a single application, and an application is only accessed by its single user. In contrast, several users may send requests to the same application in the multiple-users case.

3.1.3.3 Dynamic Model

Another classification criterion of placement approaches is whether they address the system dynamics, which can be related to infrastructure, application, and user. In a dynamic infrastructure, the network topology can evolve due to, for instance, the join and leave of nodes. Other infrastructure characteristics, such as resource capabilities and allocation cost, can also vary over time. Similarly, applications may enter and leave the system, and their characteristics can change over time. For the user point-of-view, the mobility and requests pattern are usually modeled as dynamic aspects.

3.1.4 Solution Technique

In the literature, we can identify four main algorithm techniques used to solve the service placement problem (PIRES; BARÁN, 2015; SALAHT *et al.*, 2020): deterministic, approximation, heuristic, and meta-heuristic algorithms. In a deterministic or exact algorithm, the optimal solution is computed by using a mathematical programming solver or performing an exhaustive search (i.e., by enumerating all solutions). An approximation algorithm ensures that the obtained solution is not more or less distant than a predetermined factor of the optimum solution.

As performing the optimal solution may take a long processing time, heuristic algorithms can obtain reasonably good results in less computational time by taking advantage of any problem specificity. However, unlike deterministic and approximation algorithms, heuristic-based approaches do not provide any optimality guarantees.

Meta-heuristics are high-level heuristics that also try to obtain good solutions in a reasonable time. For instance, Genetic Algorithm, Ant Colony Optimization, Particle Swarm Optimization are meta-heuristics inspired by nature. Unlike a heuristic designed for a specific problem, meta-heuristics are problem-independent techniques that explore the solution space more thoroughly to hopefully escape from a local optimum (ABDEL-BASSET *et al.*, 2018). A local optimum is the best solution to a problem within a small neighborhood of possible solutions, but worse than the (global) optimal solution.

3.2 Classification of Service Placement Approaches

VM placement is a well-studied topic in Cloud Computing. Pires and Barán (2015), Pietri and Sakellariou (2016), Masdari *et al.* (2016), Carvalho *et al.* (2018), and Filho *et al.* (2018) published surveys on this topic over the past few years. However, service placement approaches to conventional CC do not consider that an Edge Computing environment is more distributed, heterogeneous, latency-sensitive, and resource-limited than a cloud environment.. Therefore, we are only interested in the placement approaches of multiple applications in EC. Moreover, other problems such as computation offloading (MACH; BECVAR, 2017), content caching (WANG *et al.*, 2017b), and live VM migration (ZHANG *et al.*, 2018) are beyond this thesis scope. Still, these problems can be seen as complementary to the service placement one.

Based on the assignment types presented in the previous section, we identify two main scenarios for service placement in EC: static placement and dynamic placement scenarios. These two scenarios can be divided into sub-scenarios specifying whether or not they address load distribution. Thus, in the next subsections, we discuss and compare service placement approaches according to the following cases: (*i*) static placement without load distribution, (*ii*) static placement with load distribution, (*iii*) dynamic placement without load distribution, and (*iv*) dynamic placement with load distribution.

3.2.1 Static Placement without Load Distribution

In this scenario, placement approaches map each application onto some hosting node while satisfying a set of constraints and optimizing objective metrics. However, the provided mapping solution is fixed or does not change for a long time. Furthermore, placement approaches in this scenario do not consider load distribution and, thus, each service is usually mapped to a single node.

In (TÄRNEBERG *et al.*, 2017), the authors discuss the placement of applications in edge nodes to minimize the overall running cost. That is, the work aims to reduce the edge resource consumption due to the assumption that the running cost is proportional to resource usage. The authors present an iterative local search heuristic among neighboring solutions in a search tree to find a near-optimal solution. A limitation of this work is the assumption of having sufficient resources for all applications at the network edge. Moreover, it only considers the network latency as an impact factor of application response time, neglecting other factors such as processing time.

In (SKARLAT *et al.*, 2017b) and (SKARLAT *et al.*, 2017a), the authors examine the service placement in a hierarchical and distributed edge architecture to maximize the number of applications placed on edge nodes rather than the cloud. These works also prioritize time-sensitive applications to satisfy their response time deadline requirements. Moreover, the authors design a decentralized control where each control node performs the application placement decision among its child nodes in the hierarchical network infrastructure. The work (SKARLAT *et al.*, 2017a) extends (SKARLAT *et al.*, 2017b) by including a GA to solve the optimization problem. However, the proposed GA can generate infeasible solutions, which may degrade the algorithm performance.

According to (SPINNEWYN *et al.*, 2017), an EC environment is more susceptible to unpredictable failures than the cloud. As a result, this characteristic significantly affects the reliability of applications deployed in an EC environment. Therefore, the authors investigate the placement of multi-component applications to optimize multi-objectives while satisfying the application minimum availability requirement. This work applies a scalarization to transform a multi-objective optimization into a single-objective problem. The formulated problem is then solved using a GA or a heuristic based on subgraph isomorphism detection. Although this work allows the deployment of multiple replicas for an application component, they are only seen as backup entities in case of failure. Hence, there is no load distribution among these replicas.

3.2.2 Static Placement with Load Distribution

This scenario includes placement approaches where an application can be replicated and deployed in different locations simultaneously. Moreover, it is possible to distribute the load among the various application replicas. Therefore, a placement approach establishes where to place each application replica and how to distribute the incoming load among these replicas. However, established decisions are fixed in this scenario.

Zhao and Liu (2018) address service placement and load distribution problems in MEC while targeting to minimize the average response time. The authors propose a heuristic strategy that tries to select, for each application, hosting nodes with low average response time among all users requesting this application. Nevertheless, the proposed solution does not take into consideration the response time deadline requirement that is particularly important for latency-sensitive applications.

In (GU *et al.*, 2017), the authors also investigate both service placement and load distribution problems in MEC. Gu *et al.* (2017) formulate the optimization problem to minimize an overall cost while satisfying the maximum tolerable delay of time-sensitive medical applications. However, the authors only examine the application deployment in base stations of a cellular network, ignoring other possible locations such as the core network and cloud data centers. In addition, the work assumes that there are sufficient resources in the base stations to deploy all applications while satisfying the delay requirement.

Yang *et al.* (2016) present a study for joint optimization of service placement and load distribution problems in a static scenario. The authors design a two-step heuristic to minimize the average response time of all application requests while satisfying the nodes capacity constraints. First, the heuristic relaxes the problem by disregarding the node capacity constraint and solves the relaxed problem using a linear programming solver. Then, a greedy strategy tries to obtain a feasible solution for the original problem from the optimal solution of the relaxed problem. A drawback of this work is the assumption that applications are homogeneous, as all requests consume the same amount of resources to be processed.

In (KATSALIS *et al.*, 2016), the authors investigate VM scheduling and placement decision in MEC to (*i*) maximize infrastructure provider revenue, (*ii*) minimize SLA violations, and (*iii*) ensure fairness in resource allocation among service providers. Even though the work investigates SLA violation in terms of response time, it considers the processing time responsible only for the response delay, neglecting the network delay. Furthermore, the work applies a weighted sum scalarization to transform the multi-objective problem into a single-objective one. Unfortunately, setting weights for each objective is not an easy task for a decision-maker.

3.2.3 Dynamic Placement without Load Distribution

A simple way to dynamically place services is to periodically reassess the placement decision through a static approach. However, this strategy does not consider the cost of a service migration operation. Hence, in this scenario, we only discuss dynamic assignment works that address the trade-off between costs and benefits of service migrations. Nevertheless, this scenario does not take into account load distribution.

Tärneberg *et al.* (2017) extend its static service placement by considering that applications demand may vary over time due to mobile users, leading to application migrations. In order to avoid network overload due to frequent application migrations, the authors add a penality/barrier to changing the current placement decision in the optimization formulation. However, it is not easy to define the best penalty parameter values in a heterogeneous EC environment.

In (WANG *et al.*, 2017a), the authors aim to minimize the costs of place services that arrive and leave the system over time. Two cost types are examined: local cost related to the system performance at a specific time and service migration cost. Furthermore, the work proposes two solutions to the formulated problem. The first solution assumes that the arrival and departure time for all services are known, and it uses a look-ahead procedure to optimized predicted costs periodically. In contrast, the second solution only defines a service place when it arrives, but without changing the location of services already deployed. However, this work does not consider that a system may have constraints (e.g., resource capacity) that invalidate some placement options.

Most works in this scenario adopt the Follow-me Cloud/Edge approach to handle dynamic loads caused by user mobility. In this approach, each user is associated with a dedicated application to execute its offloaded tasks. Moreover, an application may be migrated to another node to follow user mobility and maintain satisfactory service performance. For instance, Sun and Ansari (2020) use this approach to minimize the non-green energy consumption while ensuring end-to-end latency below a predefined requirement. Nevertheless, frequent migrations may increase service interruption and resource consumption. To address this issue, the work (OUYANG *et al.*, 2018) applies a Lyapunov optimization to minimize time-average service latency under a long-term migration cost budget. Ouyang *et al.* (2018) also propose a decentralized scheme based on a non-cooperative congestion game to increase solution scalability. In (GAO *et al.*, 2019), the authors aim to minimize user delay comprising access queueing, communication, and migration delays. They then design a heuristic algorithm with approximation guarantees that only redistributes applications if the total non-migration delay of all users significantly exceeds the total migration delay. Authors in (YU *et al.*, 2019) adopt mobility prediction to pre-migrate applications, and thus, avoiding the impact of migration delay in future user-perceived service latency. However, a Follow-me Cloud/Edge approach usually ignores that several users may request the same application, and multiple replicas of this application can be in the system.

3.2.4 Dynamic Placement with Load Distribution

This scenario deals with service placement, service migration, and load distribution in a dynamic EC system. The diverse decision factors make this a very challenging scenario to be addressed.

Besides the static service placement, Yang *et al.* (2016) also examine dynamic service placement. In this dynamic case, the authors take user mobility and access patterns to predict future load. Then, a greedy algorithm jointly solves the service placement and load distribution problems to minimize request latency, resource consumption, and migration cost within a look-ahead prediction window. However, the proposed prediction model has low accuracy with a non-short time window, affecting requests latency.

In (URGAONKAR *et al.*, 2015), the authors jointly model service placement, service migration, and load distribution problems as a Markov Decision Process to optimize transmission and migration costs while providing maximum delay guarantees. Due to the computational complexity, the work relaxed and decoupled the problem into independent sub-problems that are solved periodically using heuristics based on the Lyapunov optimization technique (NEELY, 2010). However, this relaxation replaces the maximum delay constraints by queue stability constraints, which only provides worst-case delay guarantees.

Authors in (YU *et al.*, 2017) study dynamic Virtual Machine placement and request distribution to minimize network traffic from requests data and VMs migration. They propose a heuristic algorithm that prioritizes the placement of VMs for serving the most critical request flows, which are flows with higher bandwidth requirements or with a larger number of requests. Following the critical order, it searches available nearby nodes to place the VMs of each flow. A shortcoming of this work is that the prioritization based only on bandwidth may affect the performance of applications that have other requirements as critical, such as time-sensitive applications.

The work in (FARHADI *et al.*, 2019) addresses service placement and request scheduling for data-intensive applications to maximize the expected number of served requests, but decisions for these problems are separated in different time scales. Service placement happens on a larger scale to prevent system instability, while requests are scheduled on a smaller scale to support real-time services. It also imposes a budget constraint to control service migration costs. Furthermore, the authors present an extended formulation that optimizes the service placement problem across a predicted time window. However, they do not take into account that requests may have QoS requirements (e.g., maximum response time) to be served.

3.2.5 Discussion

According to the presented taxonomy in Section 3.1, Tables 5 and 6 categorize the surveyed works on static and dynamic service placement in Edge Computing, respectively. Based on the provided tables, this subsection discusses and identifies some research gaps that we intend to address in this thesis.

A differential aspect of Edge Computing over Cloud Computing is the inclusion of time-sensitive applications. In general, those applications have a QoS requirement related to the maximum delay (deadline) of accessing an application. However, some works, such as (TÄRNEBERG *et al.*, 2017), (ZHAO; LIU, 2018), (URGAONKAR *et al.*, 2015) and (FARHADI *et al.*, 2019), disregard this requirement entirely, or they do not tackle the impact of the network, processing, and migration delays on its assurance. Another identified limitation is the formulation of hard constraints for this delay requirement by assuming that it is always possible to satisfy it for all applications. In a practical scenario, some applications are deployed in a remote cloud data center due to resource constraints on the edge layer, which may lead in some cases to violations of the deadline requirement.

Regarding the optimization objective, most surveyed studies only have a single objective. However, a single objective does not capture the diversity of goals and performance metrics of different entities involved in the service placement problem. Moreover, many multi-objective works transform the problem into a single-objective problem through some scalarization method. Nevertheless, these methods require a global order of preference among the objectives, which can be quite hard or even impossible to exist. Thus, the challenge is to develop methods that directly address the optimization of multiple and conflicting objectives.

An application characteristic that lacks more exploration in the service placement literature is the scalability support. This characteristic can play a crucial role in adapting applications to a distributed, heterogeneous, and dynamic EC environment. In addition, different applications may have distinct degrees of scalability, including not supporting it. However, few works deal with different scalability degrees for heterogeneous applications.

A consequence of scalable applications is the possibility of load distribution to improve performance. As service placement and load distribution decisions can affect each other, a robust decision process may require a joint optimization of these two decisions. Although static assignment approaches, such as those in Table 5, can give us insights about some peculiarities of service placement problem in EC, disregarding dynamic aspects of the system (e.g., service migration and user mobility) may lead to undesired poor performance. Hence, a dynamic approach is more suited for a real system than a static one, but it is the most challenging to design. Consequently, only a few studies address service placement, service migration, and load distribution in an EC environment at the same time.

In a dynamic placement, proactive behavior may be helpful or even required to prevent the system from reaching undesired states, especially for applications with critical and strict requirements. For instance, Yu *et al.* (2019) and Yang *et al.* (2016) try to predict user mobility in their proactive and dynamic placement. However, in Table 6, none of the related work on dynamic placement with load distribution addresses the impact of its decisions on the performance (e.g., response time) of time-sensitive applications. Thus, the challenge is to design dynamic and proactive approaches to the service placement problem.

A common point in most related works is the decision process being performed in a centralized way. However, a centralized decision process may suffer scalability issues, which can be a bottleneck in a vastly distributed environment. On the other hand, a decentralized decision is highly scalable, but it may produce more inferior solutions than those obtained by a centralized process due to a lack of a global system view. Hence, another challenge is to design a non-centralized control with results close to those achieved by a central controller.

Table 5 – Classification of static service placement approaches in Edge Computing

Reference	Problem Formulation				Controller Design				System Model							Solution Technique
									Infrastructure		Application					
	Modeling Technique	Sub-Problem	Objective	Constraint	Architecture	Assignment Type	Reassignment Trigger	Reassignment Behavior	Resource Type	Architecture Tier	Component	Scalability	Requirement	User Access Relationship	Dynamic	
(TÄRNEBERG *et al.*, 2017)	MP	SP	S	Res QoS	C	Static	-	-	M	S	S	V	Res QoS	M	-	Heu
(SKARLAT *et al.*, 2017b)	MP	SP	S	Res QoS	Dec	Static	-	-	M	M	M	None	Res QoS	M	-	Det
(SKARLAT *et al.*, 2017a)	MP	SP	S	Res QoS	Dec	Static	-	-	M	M	M	None	Res QoS	M	-	MH
(SPINNEWYN *et al.*, 2017)	MP	SP	MaS	Res QoS	C	Static	-	-	M	S	M	H	Res QoS	M	-	Heu MH
(ZHAO; LIU, 2018)	MP	SP LD	S	Res	C	Static	-	-	S	M	S	V, H	Res	M	-	Heu
(GU *et al.*, 2017)	MP	SP LD	S	Res QoS	C	Static	-	-	M	S	S	V, H	Res QoS	M	-	Heu
(YANG *et al.*, 2016)	MP	SP LD	S	Res	C	Static	-	-	M	M	S	H	Res	M	-	Heu
(KATSALIS *et al.*, 2016)	MP	SP LD	MaS	Res	C	Static	-	-	M	M	S	H	Res QoS	M	-	Det
Proposal in Chapter 4	MP	SP LD	S, M	Res	C	Static	-	-	M	M	S	V, H	Res QoS	M	-	MH

Source: Author.
Note: MP = Math. Programming, SP = Service Placement, LD = Load Distribution, MaS = Multi-objective as Single-objective, Res = Resource, C = Centralized, Dec = Decentralized, S = Single, M = Multiple, V = Vertical, H = Horizontal, Det = Deterministic, Heu = Heuristic, MH = Meta-Heuristic, - = Not Applicable

Table 6 – Classification of dynamic service placement approaches in Edge Computing

Reference	Problem Formulation				Controller Design				System Model							Solution Technique
									Infrastructure		Application					
	Modeling Technique	Sub-Problem	Objective	Constraint	Architecture	Assignment Type	Reacssignment Trigger	Reacssignment Behavior	Resource Type	Architecture Tier	Component	Scalability	Requirement	User Access Relationship	Dynamic	
(TÄRNEBERG *et al.*, 2017)	MP	SP SM	MaS	Res QoS	C	Dyn	Periodic	Reac	M	S	S	V	Res QoS	M	User	Heu
(WANG *et al.*, 2017a)	MP	SP SM	S	None	C	Dyn	Periodic Event	Proa	-	M	S	None	-	M	App	Det Heu
(SUN; ANSARI, 2020)	MP	SP SM	S	Res QoS	C	Dyn	Periodic	Reac	S	S	S	None	QoS	S	User	Heu
(OUYANG *et al.*, 2018)	MP GT	SP SM	S	Budget	C Dec	Dyn	Periodic	Reac	S	S	S	None	Res	S	User	Heu
(GAO *et al.*, 2019)	MP	SP SM	S	Res	C	Dyn	Threshold	Reac	S	S	S	None	Res	S	User	Appr
(YU *et al.*, 2019)	MP	SP SM	S	-	C	Dyn	Periodic	Proa	M	S	S	None	Res QoS	S	User	Heu
(YANG *et al.*, 2016)	MP	SP, SM LD	MaS	Res	C	Dyn	Periodic	Proa	M	M	S	H	Res	M	User	Heu
(URGAONKAR *et al.*, 2015)	MDP	SP, SM LD	S	-	C	Dyn	Periodic	Reac	-	M	S	H	QoS	M	User	Heu
(YU *et al.*, 2017)	MP	SP, SM LD	S	Res	C	Dyn	Periodic	Reac	M	S	S	H	Res QoS	M	User	Heu
(FARHADI *et al.*, 2019)	MP	SP, SM LD	S	Res Budget	C	Dyn	Periodic	Reac Proa	M	M	S	V, H	Res	M	User	Appr Heu
Proposal in Chapter 5	MP	SP, SM LD	M	Res	C	Dyn	Periodic	Proa	M	M	S	V, H	Res QoS	M	User	MH
Proposal in Chapter 6	MP	SP, SM LD	M	Res	Dis	Dyn	Periodic	Proa	M	M	S	V, H	Res QoS	M	User	MH

Source: Author.
Note: MP = Math. Programming, GT = Game Theory, SP = Service Placement, SM = Service Migration, LD = Load Distribution, SM = Service Migration, MaS = Multi-objective as Single-objective, Res = Resource, C = Centralized, Dec = Decentralized, Dis = Distributed, Dyn = Dynamic, Reac = Reactive, Proa = Proactive, S = Single, M = Multiple, V = Vertical, H = Horizontal, Det = Deterministic, Appr = Approximation, Heu = Heuristic, MH = Meta-Heuristic, - = Not Applicable

3.3 Summary

This chapter investigated and classified existing works related to service placement problem in Edge Computing based on a presented taxonomy. The investigation identified some gaps found in these works, which are summarized as follows:

– Current works neglect the maximum delay requirement of time-sensitive applications, do not consider different aspects that impact this requirement satisfaction (e.g., communication and processing delays), or assume there is enough resource in the edge layer to deploy all applications;
– Most service placement approaches optimize a mono objective or transform multiple objectives into a single one, and thus, a multi-objective optimization lacks further investigation;
– The application scalability property needs to be further examined when jointly optimizing service placement and load distribution;
– Few works handle service placement, service migration, and load distribution proactively in a dynamic system. Moreover, none of them address the impact of their decisions on the performance of time-sensitive applications; and
– The vast majority of the service placement approaches have centralized control decisions that may experience scalability issues in a large EC environment.

The next chapter presents a static service placement with load distribution addressing the first three identified gaps. The next-to-last and last gaps are covered in Chapters 5 and 6, respectively.

4 A STATIC APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION

In this chapter, we are interested in the service placement and load distribution decisions in a static Edge Computing (EC) scenario. In this static scenario, all information required in the decision-making process is provided in advance and does not change over time. We address these decisions as an optimization problem that considers the EC infrastructure constraints and different application characteristics (response deadline, resource demand, scalability, and availability). Moreover, the formulated problem aims to reduce Service Level Agreement (SLA) violations in terms of application response deadlines and, at the same time, optimize other conflicting performance-related objectives (e.g., operational cost and service availability). The main contributions of this chapter are the following:

– We present a system model where multiple replicas of an application can be placed in different parts of the EC infrastructure to distribute requests (load) among these replicas.
– We jointly formulate the service placement and load distribution as a single-objective optimization problem to minimize the potential occurrence of SLA violations. We also apply linear and relaxation transformations into the formulated problem to solve it using a linear solver.
– The single-objective problem is extended to include multiple conflicting objectives. In addition, the formulated multi-objective problem allows the prioritizing of metrics related to time-sensitive applications.
– We propose a Genetic Algorithm (GA) to solve the formulated single and multi-objective problems. The proposed algorithm combines BRKGA and NSGA-II with specific evolutionary operations for the addressed problem in this chapter.

The remainder of the chapter is organized as follows. Section 4.1 presents the EC system model. We formally formulate the service placement and load distribution problem in Section 4.2. In Section 4.3, we describe our proposed GA to solve the formulated optimization problem. Then, Section 4.4 analyses the performance of our proposal. Finally, Section 4.5 concludes this chapter.

4.1 System Model

Our Edge Computing system model consists of an Infrastructure Provider (InP), various Application Service Providers (ASPs), and end-user devices. The InP owns and maintains the EC infrastructure containing EC nodes geographically distributed between end-user devices and a remote cloud data center. These EC nodes can provide diverse resources (e.g., processing, memory, storage, and networking resources) to host and operate applications through virtualization technologies, such as VM or container. Moreover, some nodes also act as network routers and (wireless) access points.

An ASP offers applications to end-user devices by renting on-demand resources from the InP to deploy and operate its application on EC nodes. However, ASPs do not directly determine where to place their applications. Usually, an ASP only signs a SLA defining the Quality of Service (QoS) requirements to be fulfilled by the InP. In this way, the InP is responsible for selecting the places to deploy applications based on the requirements specified by each ASP.

End-user devices are connected directly with access point nodes over wired or wireless links. Over time, devices send requests or tasks to be processed by an application. A device request is routed among the nodes up to a target node hosting the required application. As an application can be deployed in multiple nodes in our system model, the target node is selected based on a load distribution decision. In the target node, the application then puts the arrived request in a waiting queue for processing. Finally, after completing the request processing, the resulted response is sent back to the device.

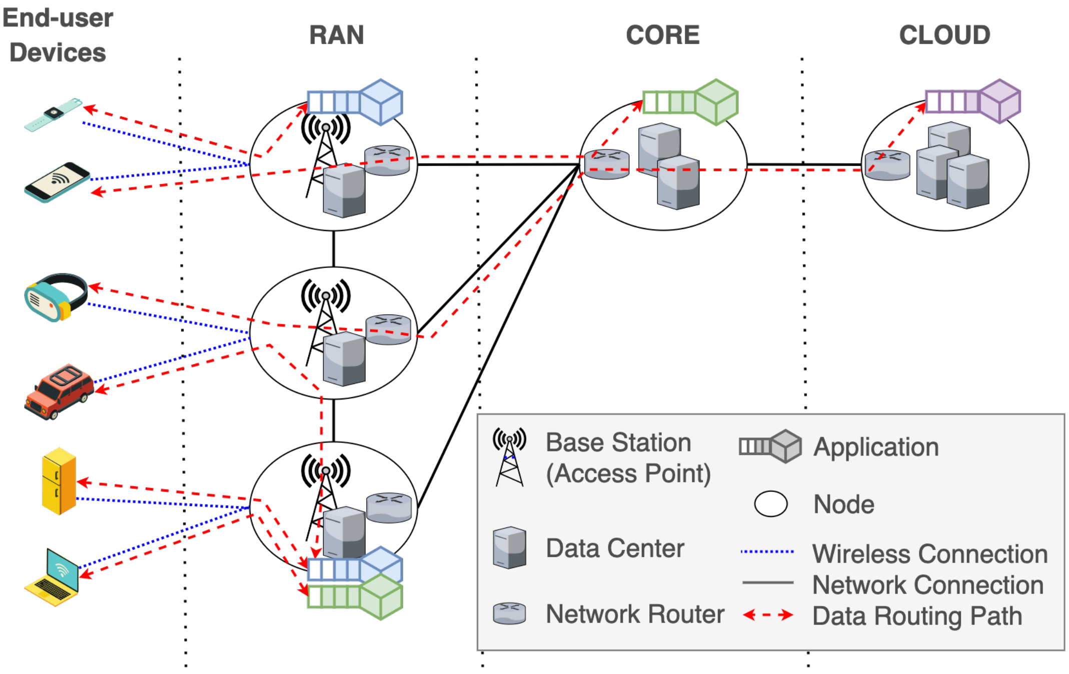
Figure 13 illustrates our system model in a cellular network scenario with Edge Computing capabilities (e.g., a 5G network). In this scenario, applications can be hosted on nodes located on the Radio Access Network (RAN), Core Network, and Cloud Computing regions. If an application runs on a Base Station (BS) in the RAN region, then a request may be routed among neighboring Base Stations (BSs) to reduce traffic at the core and decrease transmission delays. However, not all applications can be deployed to BSs because of the limited computing resources in this region. Therefore, some applications are hosted in the core or the cloud while carrying about not violating some placement constraints.

In the remainder of this section, we further detail the main features of the infrastructure, application, and user models. Moreover, Table 7 summarizes the main notations used in this chapter.

4.1.1 Infrastructure Model

The EC infrastructure is modeled as an undirected and connected graph $G = (\mathscr{V}, \mathscr{E})$, where the vertices $\mathscr{V}$ are EC nodes and the edges $\mathscr{E}$ are network links between the nodes. We

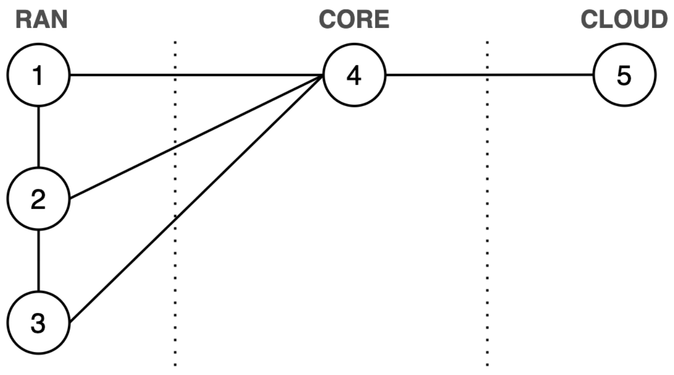
Figure 13 – Proposed Edge Computing system for 5G networks



Source: Author.

assume all vertices are accessible by any other vertex in the graph through multiple hops. In addition, end-user devices and their connections are not represented in G. Figure 14 illustrates the EC infrastructure presented in Figure 13 as an undirected and connected graph.

Figure 14 – Example of an EC infrastructure represented as a graph



Source: Author.

A link $l = (m, n) \in \mathscr{E}$ corresponds to a (physical or virtual) network connection between nodes m and n, and it has the following attribute:

– **Transmission Delay** $D^{\text{net}}_{a,l}$ is the average amount of time it takes for a request for an application a to be transmitted in the network link l.

In the proposed system model, we can specify different types of resources, where $\mathscr{R}$ is the set of considered resources. For instance, the set $\mathscr{R} = \{\text{CPU}, \text{RAM}, \text{DISK}\}$ is made up of

Table 7 – Main notations of the static service placement problem

Symbol	**Description**
System Model	
$\mathscr{V},\mathscr{E},\mathscr{R},\mathscr{A},\mathscr{U}$	Set of nodes, links, resource types, applications, and users, respectively
$D^{\text{net}}_{a,l}$	Network delay for application (app) a in a link l
$N^{\text{cap}}_{n,r}$	Total capacity of resource r on node n
$N^{\text{cost}}_{n}(\lambda)$	Resource allocation cost on node n for an app with workload λ
N^{avail}_{n}	Availability probability of node n
A^{rd}_{a}	Response deadline of app a
A^{max}_{a}	Maximum number of replicas for app a
$A^{r}_{a}(\lambda)$	Demand of resource r for a replica of app a with workload λ
A^{work}_{a}	CPU work size of a request for app a
A^{req}_{a}	Request generation rate for app a
A^{avail}_{a}	Availability probability of app a
$\mathscr{U}_{a,n}$	Set of users connected to node n requesting application a
$\mathscr{U}_{a}$	Set of all users requesting application a in the entire system
Problem Formulation	
$\mathscr{F}_{a,m,n}$	Set of requests from users attached to node m to an instance of app a hosted on node n
$Q_{a,m}$	Number of requests for app a generated from users attached to node m
Q_{a}	Total number of requests for app a in the system
$\lambda_{a,n}$	Request arrival rate of app a on node n
$\mu_{a,n}$	Service rate of app a on node n
$x=(\rho,\gamma,\delta)$	Decision variables
$\rho_{a,n}$	Whether node n hosts an instance of app a or not
$\gamma_{a,m,n}$	Whether request flow $\mathscr{F}_{a,m,n}$ exists or not
$\delta_{a,m,n}$	Number of requests in the flow $\mathscr{F}_{a,m,n}$
$A^{r}_{a,1},A^{r}_{a,2}$	Constants of a linear resource demand function $A^{r}_{a}(\cdot)$

Source: Author.

processing (CPU), Random-Access Memory (RAM) and disk storage resources (DISK). RAM and disk storage are measured in bytes, while CPU can be measured in Instructions Per Second (IPS).

In graph G, a node or vertex can represent a (mini) data center, a (wireless) access point, a network router, or all of them at the same time if they are co-located in a physical site. We also include cloud data centers as a single cloud node in the graph, i.e., $\{cloud\} \subset \mathscr{V}$. Furthermore, each node $n \in \mathscr{V}$ has the following parameters:

– **Resource Capacity** $N^{\text{cap}}_{n,r}$ is a number describing the total capacity of resource $r \in \mathscr{R}$ on node n.

- **Usage Cost** $N_n^{\text{cost}}(\lambda)$ is a function specifying the (monetary) cost of allocating resources for an application with a workload $\lambda \geq 0$ on the node. We define the workload $\lambda_{a,n}$ as the (average) arrival rate of requests for application a in node n.
- **Availability** N_n^{avail} is the probability that the node n will not fail.

We assume that the cloud node has an unlimited capacity for all resources (i.e., $N_{cloud,r}^{\text{cap}} = \infty, \forall r \in \mathscr{R}$) due to the capacity difference between the cloud and a mini data center close to the users.

4.1.2 Application Model

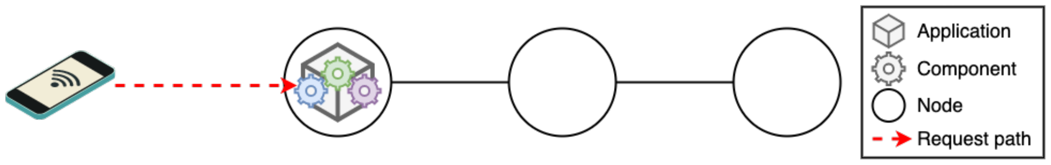
Let $\mathscr{A}$ be the set of all different applications to be placed over the EC infrastructure. We consider that one or more instances of these applications can be deployed within the system, but these instances or replicas are independent of each other. In this way, an application instance is designed to be self-contained and deployed as a single piece, such as a VM or container. This application design was assumed to avoid additional delays in handling requests when placing the necessary components for the request processing in different locations. Moreover, the self-contained design follows the IoT architecture and operational pattern shown in Section 2.1.

Figures 15a and 15b exemplify an application composed of three functional components designed as single or multiple pieces, respectively. In Figure 15a, user requests only need to be dispatched to a node hosting the requested application to be processed. Meanwhile, requests pass throughout three different nodes hosting the necessary application components to be processed entirely in Figure 15b, resulting in additional network delays for response time compared to the single-piece design in Figure 15a.

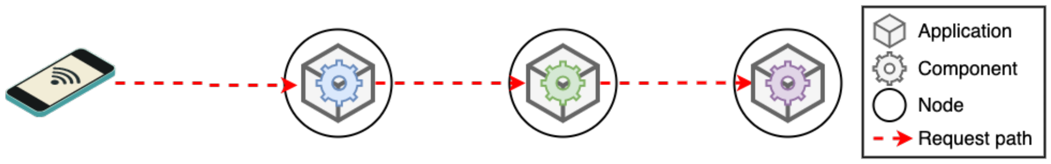
An application $a \in \mathscr{A}$ has the following attributes:

- **Response Deadline** A_a^{rd} is a number specifying the maximum time (i.e., deadline) allowed for responding a request for application a.
- **Maximum Number of Replicas** A_a^{max} describes how application a scales horizontally. For example, it can be set to $A_a^{\text{max}} = 1$, if the service does not allow replicas or $A_a^{\text{max}} = \infty$ if the maximum number of instances is undefined. Moreover, a node can only host one instance of each application.
- **Resource Demand** $A_a^r(\lambda)$ denotes how an application scales vertically. In other words, it is a non-decreasing function specifying the (average) amount of resources $r \in \mathscr{R}$ required

Figure 15 – Different application designs



(a) Application designed as a single piece



(b) Application designed as multiple pieces

Source: Author.

by a replica of application a with a workload λ. That is, if $\lambda > \lambda'$, then $A_a^r(\lambda) \geq A_a^r(\lambda')$. For instance, we can define a constant function if the vertical scaling is not supported or an increasing linear function if the amount of resources required is proportional to the workload.

- **CPU Work Size** A_a^{work} is a value indicating the (average) amount of processing required to get a response to a request for application a. It is measured by the number of instructions or clock cycles required to process a request.
- **Request Rate** A_a^{req} is the average request generation rate of an end-user device requesting application a. As common of IoT applications (METZGER *et al.*, 2019), it follows a Poisson distribution.
- **Availability** A_a^{avail} denotes the probability that an application replica is working without internal failure.

4.1.3 User Model

Let $\mathscr{U}$ bet the set of all end-user devices (or simply called users) in the system and, then, a user $u \in \mathscr{U}$ has the following properties:

- **Requested Application** $U_u^{\text{app}} \in \mathscr{A}$ specifies the application requested by the user. For the sake of simplicity, we assume that each user sends requests for only one application.
- **Attached Node** $U_u^{\text{node}} \in \mathscr{V}$ denotes the node acting as an access point where user u is connected to send its requests. In the static approach, the user is fixed and always attached to this node.

Then, we define $\mathcal{U}_{a,n}$ as the set of all users connected to node n requesting application a, and $\mathcal{U}_a$ is the set of all users requesting application a in the system.

4.2 Problem Statement and Formulation

In a practical scenario, it is not possible to place all applications on the edge of the network given the resource limitations of EC nodes in this region. Consequently, some applications are deployed further (i.e., in the core network or the cloud) from their users. This considerable distance between node and user may result in the response time of a request to exceed the deadline specified by some applications. Moreover, an overloaded node also increases response time, thus distributing the load among application replicas may mitigate this issue. Hence, both service placement and load distribution decisions may result in violations of the response deadline requirement, which is an important metric to be minimized for time-sensitive applications in an Edge Computings environment.

Infrastructure Provider and Application Service Providers often have many other performance metrics to optimize instead of just a single one. However, those multiple metrics are, in general, contradicting each other. For instance, an ASP wants to decrease response time while reducing the monetary cost of allocating resources to its application. On the other hand, decreasing costs implies using cheaper cloud resources and, thus, increasing response time due to the distance between end-user devices and a remote cloud data center.

Given the above context, we formulate the joint problem of service placement and load distribution to minimize deadline violation as a single objective, and multiple objectives in this section. In this chapter, we are only interested in the static, or offline, approach of the problem where applications and users do not move for a long time.

The remaining of this section is organized as follows. First, Subsection 4.2.1 presents an estimation of the response time of a request. Then, Subsection 4.2.2 specifies the variables and constraints of the problem. Subsection 4.2.3 presents a nonlinear and linear formulation of the single-objective case. Finally, Subsection 4.2.4 formulates the multi-objective case.

4.2.1 Response Time Estimation

As requests can be distributed among multiple replicas of an application, we define a request flow of an application as follows:

Definition 4.1 (Static Request Flow) *A request flow $\mathscr{F}_{a,m,n}$ is the set of requests for application $a \in \mathscr{A}$ generated by users attached to node $m \in \mathscr{V}$ (source node) and handled by a replica of a placed on node $n \in \mathscr{V}$ (target node).*

In this way, we are interested in estimating the response time of requests in each existing request flow, i.e., $|\mathscr{F}_{a,m,n}| > 0$. Thus, Equation (4.1) specifies the average response time $d_{a,m,n}$ of a request flow $\mathscr{F}_{a,m,n}$, where $d^{\text{net}}_{a,m,n}$ is the average time to send requests to a from users in m to node n and $d^{\text{proc}}_{a,n}$ is the average processing time of requests on n. We estimate both network and processing delays in the remainder of this subsection.

$$d_{a,m,n} = d^{\text{net}}_{a,m,n} + d^{\text{proc}}_{a,n} \tag{4.1}$$

4.2.1.1 Network Delay

The network delay of a request includes: (*i*) the communication delay between the requesting end-user device and the node to which it is attached, and (*ii*) the transmission delay from this latter node to a node hosting the application following a multi-hop routing path. It is important to note that a node where a user is attached to it can also host the application processing the user requests and, thus, the transmission delay of the second part is zero. Moreover, as different locations to place an application in the infrastructure do not affect the communication delay between devices and their attached nodes (ZHAO; LIU, 2018), we do not consider this delay in the network delay estimation and, thus, in the placement decision process. Then, we estimate the average network delay of a request flow $\mathscr{F}_{a,m,n}$ as:

$$d^{\text{net}}_{a,m,n} = \begin{cases} 0 & \text{if } m = n \\ \sum_{l \in \mathscr{P}_{m,n}} D^{\text{net}}_{a,l} & \text{otherwise} \end{cases} \tag{4.2}$$

where $\mathscr{P}_{m,n}$ is the set of links in a routing path from m to n. This set can be predetermined by some shortest routing path algorithm, such as the Floyd–Warshall algorithm (FLOYD, 1962; WARSHALL, 1962). Therefore, we assume that $d^{\text{net}}_{a,m,n}$ is a constant value calculated before starting the service placement decision process.

4.2.1.2 Processing Delay

For an application replica placed on a node, we model its request processing as an M/M/1 queueing model (SHORTLE *et al.*, 2018). Hence, this queuing model helps us to determine the processing delay of each application replica when distributing user-generated requests among these replicas.

As specified in our system model in Section 4.1, users continuously generate requests for an application a according to a homogeneous Poisson process with ratio A_a^{req}. Then, we define the request arrival rate $\lambda_{a,n}$ for the application a running on node n as the sum of all requests arriving at this node. Equation (4.3) expresses this request arrival rate, where $\delta_{a,m,n} = |\mathscr{F}_{a,m,n}| \in [0, Q_{a,m}]$ is an integer variable indicating the size of request flow $\mathscr{F}_{a,m,n}$ (i.e., number of requests in the flow), $Q_{a,m} = \lceil |\mathscr{U}_{a,m}| A_a^{\text{req}} \rceil$ is the total number of requests for application a generated by users attached to node m, and $|\mathscr{U}_{a,m}|$ is the cardinality of set $\mathscr{U}_{a,m}$.

$$\lambda_{a,n} = \sum_{m \in \mathscr{V}} \delta_{a,m,n} \tag{4.3}$$

Service times have an exponential distribution with rate parameter μ, where $1/\mu$ is the average service time in an M/M/1 queue. Thus, we express $1/\mu_{a,n}$ as the time to perform a request with CPU work A_a^{work} in a replica of application a in node n as:

$$\frac{1}{\mu_{a,n}} = \frac{A_a^{\text{work}}}{A_a^{\text{CPU}}(\lambda_{a,n})} \tag{4.4}$$

where $A_a^{\text{CPU}}(\lambda_{a,n})$ is the amount CPU resources allocated for the application replica having request arrival rate $\lambda_{a,n}$.

Finally, Equation (4.5) gives the average processing time of requests for application a running on node n according to the M/M/1 queueing model.

$$d_{a,n}^{\text{proc}} = \frac{1}{\mu_{a,n} - \lambda_{a,n}} \tag{4.5}$$

4.2.2 Problem Variables and Constraints

In order to jointly formulate the service placement and load distribution problems, we define $x = (\rho, \delta, \gamma)$ as a triple of problem variables. We describe these variables as follows:

1. **Application Placement** $\rho = \{\rho_{a,n} \mid a \in \mathscr{A} \text{ and } n \in \mathscr{V}\}$ is a set of binary variables, where $\rho_{a,n} \in \{0,1\}$ indicates whether a replica of an application a is placed on a node n or not.
2. **Load Distribution** $\delta = \{\delta_{a,m,n} \mid a \in \mathscr{A} \text{ and } m,n \in \mathscr{V}\}$ is a set of variables related to how requests are distributed, where $\delta_{a,m,n} \in \mathbb{Z}^+$ is the size of request flow $\mathscr{F}_{a,m,n}$.

3. **Request Flow Existence** $\gamma = \{\gamma_{a,m,n} \mid a \in \mathscr{A}$ and $m,n \in \mathscr{V}\}$ is an auxiliary set of binary variables, where $\gamma_{a,m,n} \in \{0,1\}$ specifies whether or not a request flow $\mathscr{F}_{a,m,n}$ exists between nodes m and n for an application a. This auxiliary set of variables will help us to linearize the formulated problem in Section 4.2.3.1.

A solution to the studied problem sets values for the above variables. Furthermore, a solution is feasible only if all the following constraints are met:

1. **Number of Replicas.** A node can only host a single replica of a given application. Moreover, the number of instances deployed in the system must respect the limits defined by the applications (i.e., $A_a^{\max}$), and all of them need to be placed.

$$1 \leq \sum_{n \in \mathscr{V}} \rho_{a,n} \leq A_a^{\max} \quad \forall a \in \mathscr{A} \tag{4.6}$$

2. **Request Flow Existence.** A request flow $\mathscr{F}_{a,m,n}$ only exists if a replica of application a is placed on node n and there are users attached to m requesting a.

$$\gamma_{a,m,n} \leq \rho_{a,n} Q_{a,m} \quad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.7}$$

3. **Request Flow Size.** If a flow $\mathscr{F}_{a,m,n}$ exists, its size must be at least one and at most equal to $Q_{a,m}$, the total number of requests generated by all users of application a connected to node m.

$$\gamma_{a,m,n} \leq \delta_{a,m,n} \leq \gamma_{a,m,n} Q_{a,m} \quad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.8}$$

4. **Load Conservation.** The aggregate size of all request flows for application a from the same source node m is equal to the total number of requests generated by users of application a connected to this node m. In other words, all requests must be distributed to some nodes.

$$\sum_{n \in \mathscr{V}} \delta_{a,m,n} = Q_{a,m} \quad \forall a \in \mathscr{A}, \forall m \in \mathscr{V} \tag{4.9}$$

5. **Node Capacity.** The total amount of resources demanded by applications placed on a node should not exceed its capacity.

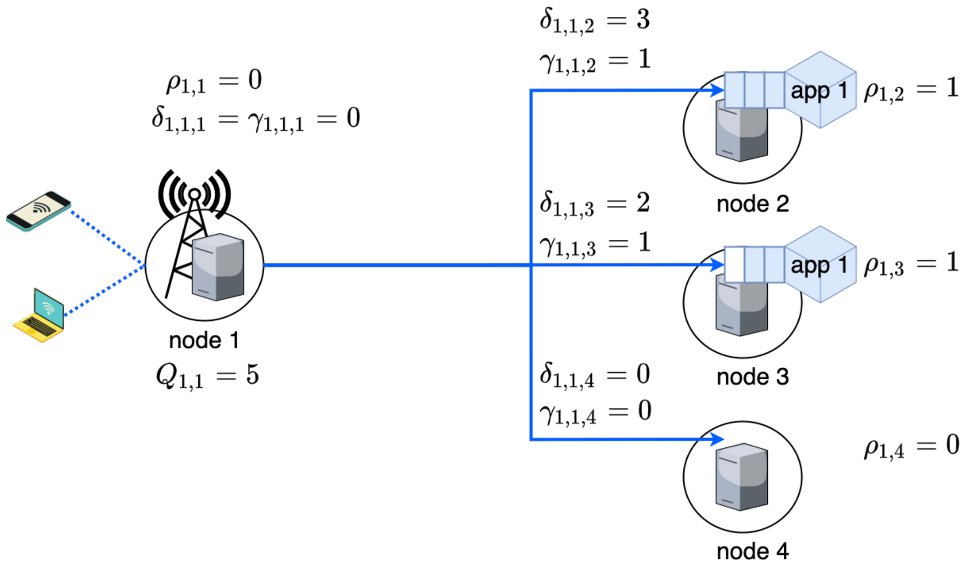
$$\sum_{a \in \mathscr{A}} \rho_{a,n} A_a^r(\lambda_{a,n}) \leq N_{n,r}^{\text{cap}} \quad \forall r \in \mathscr{R}, \forall n \in \mathscr{V} \tag{4.10}$$

6. **Queue Stability.** An M/M/1 queue is stable only if the average service rate is larger than its average arrival rate. This stability needs to be guaranteed for each application placed on a node.

$$\lambda_{a,n} < \mu_{a,n} \quad \forall a,n \,(\rho_{a,n} = 1), a \in \mathscr{A}, n \in \mathscr{V} \tag{4.11}$$

Figure 16 illustrates the defined variables $x = (\rho, \delta, \gamma)$ in a scenario with four nodes and one application. In this figure, application 1 is placed on nodes 2 and 3. Then, $\rho_{1,2} = \rho_{1,3} = 1$ and $\rho_{1,1} = \rho_{1,4} = 0$. The five requests for application 1 generated by users attached to node 1, i.e., $Q_{1,1} = 5$, are only distributed to nodes hosting this application. Specifically, nodes 2 and 3 receive 3 and 2 requests, respectively. Thus, $\delta_{1,1,2} = 3$ and $\delta_{1,1,3} = 2$, whereas $\delta_{1,1,1} = \delta_{1,1,4} = 0$. As a request flow only exists if the target node receives at least one request from the source node, then $\gamma_{1,1,2} = \gamma_{1,1,3} = 1$ and $\gamma_{1,1,1} = \gamma_{1,1,4} = 0$.

Figure 16 – Example of the variables of the static service placement with load distribution problem.



Source: Author.

4.2.3 *Single-Objective Formulation*

For the service placement problem with a single objective, our goal is to minimize the deadline violation of the system, which we define for the static service placement approach as the highest violation among all request flows in the system. We use this violation definition because we can linearize it late in this subsection. Then, we specify the deadline violation of an existing request flow $\mathscr{F}_{a,m,n}$ as the positive part of the difference between its average response time $d_{a,m,n}$ and the application response deadline A_a^{rd}, i.e., $\max\left(0,\, \gamma_{a,m,n} d_{a,m,n} - A_a^{\text{rd}}\right)$. Equation (4.12) expresses this objective function, where $[z]^+ = \max(0, z)$ is the positive part of a real number z.

$$\begin{aligned} f_{\mathrm{dv}}(x) &= \max_{x} \left\{ \left[\gamma_{a,m,n} d_{a,m,n} - A_a^{\mathrm{rd}} \right]^+ \right\} \\ x &= (\rho, \delta, \gamma) \end{aligned} \tag{4.12}$$

Then, the static service placement problem as a single-objective optimization problem is formulated as:

$$\begin{aligned} \min \quad & f_{\mathrm{dv}}(x) \\ \text{s.t.} \quad & x = (\rho, \delta, \gamma) \\ & \text{Equations (4.6) to (4.11)} \end{aligned} \tag{4.13}$$

4.2.3.1 Linearization and Relaxation

The optimization problem (4.13) is a Integer Nonlinear Programming (INLP) problem because constraints (4.10) and (4.11) and the objective function (4.12) are nonlinear. INLP is usually difficult to solve due to its high computational complexity (BURER; LETCHFORD, 2012). One way to reduce this complexity is to apply linearization and relaxation techniques. Therefore, we transform the problem (4.13) into a Mixed-Integer Linear Programming (MILP) problem as follows.

4.2.3.1.1 Node Capacity Constraint

For an application a, its resource demand function $A_a^r(\lambda)$ may be nonlinear for a specific resource type $r \in \mathscr{R}$. In this case, the function $A_a^r(\lambda)$ can be replaced by an over linear estimator $\bar{A}_a^r(\lambda)$ in the domain interval $[0, Q_a]$, as shown in Equation (4.14), where $A_{a,1}^r$, $A_{a,2}^r$ are constants, and Q_a is equal to $\sum_{n \in \mathscr{V}} Q_{a,n}$.

$$\bar{A}_a^r(\lambda) = A_{a,1}^r \lambda + A_{a,2}^r \tag{4.14}$$

Given that requests only arrive at nodes running the requested application according to Equations (4.3), (4.7) and (4.8), we have:

$$\rho_{a,n} \lambda_{a,n} = \lambda_{a,n} \tag{4.15}$$

By applying Equations (4.14) and (4.15) to Equation (4.10), the node capacity constraint can be rewritten as:

$$\sum_{a \in \mathscr{A}} \left(\lambda_{a,n} A_{a,1}^r + \rho_{a,n} A_{a,2}^r \right) \le N_{n,r}^{\mathrm{cap}} \quad \forall r \in \mathscr{R}, \forall n \in \mathscr{V} \tag{4.16}$$

4.2.3.1.2 Queue Stability Constraint

In order to have a linear queue stability constraint, we apply Equations (4.4) and (4.14) to constraint (4.11) and obtain:

$$\lambda_{a,n}\left(A_{a,1}^{\text{CPU}} - A_a^{\text{work}}\right) + A_{a,2}^{\text{CPU}} > 0 \quad \forall a,n\left(\rho_{a,n}=1\right), a \in \mathscr{A}, n \in \mathscr{V} \tag{4.17}$$

However, it must remove the strictness of the above inequality to obtain a standard form of a MILP problem. For this, it is added a small constant $\Theta \in (0,1]$. Furthermore, both sides of the inequality are multiplied by $\rho_{a,n}$ to ensure the queue existence constraint. Then, we further have:

$$\rho_{a,n}\lambda_{a,n}\left(A_{a,1}^{\text{CPU}} - A_a^{\text{work}}\right) + \rho_{a,n}A_{a,2}^{\text{CPU}} \geq \rho_{a,n}\Theta \quad \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \tag{4.18}$$

Finally, applying Equation (4.15) to the above result, we obtain the following linear queue stability constraint:

$$\lambda_{a,n}\left(A_{a,1}^{\text{CPU}} - A_a^{\text{work}}\right) + \rho_{a,n}A_{a,2}^{\text{CPU}} \geq \rho_{a,n}\Theta \quad \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \tag{4.19}$$

4.2.3.1.3 Objective Function

Problem (4.13) has a $\min\max f(x)$ objective, which is nonlinear because max function is nonlinear. This type of problem can be transformed into one without max function by replacing $\min\max f(x)$ for $\min z$, where z is a new variable, and adding a new constraint relating this variable to $f(x)$ (i.e., $f(x) \leq z$). In this way, the objective is linear, but it is necessary to check the linearity of the new constraint. Based on this transformation, we can add the following constraint in the problem:

$$\gamma_{a,m,n}d_{a,m,n} - A_a^{\text{rd}} \leq \varepsilon \quad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.20}$$

where $\varepsilon \geq 0$ is a new variable indicating the system deadline violation that we want to minimize. Moreover, given Equations (4.1), (4.2), (4.4), (4.5) and (4.14), we rewrite constraint (4.20) as:

$$\begin{aligned}&\left(\gamma_{a,m,n}\lambda_{a,n}d_{a,m,n}^{\text{net}} - \varepsilon\lambda_{a,n} - \lambda_{a,n}A_a^{\text{rd}}\right)\left(A_{a,1}^{\text{CPU}} - A_a^{\text{work}}\right)\\ &+ \gamma_{a,m,n}\left(A_{a,2}^{\text{CPU}}d_{a,m,n}^{\text{net}} + A_a^{\text{work}}\right) - A_{a,2}^{\text{CPU}}\left(A_a^{\text{rd}} + \varepsilon\right) \leq 0\\ &\qquad\qquad\qquad\qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V}\end{aligned} \tag{4.21}$$

However, in constraint (4.21), both $\gamma_{a,m,n}\lambda_{a,n}$ and $\varepsilon\lambda_{a,n}$ are bilinear terms (i.e., multiplication of two variables). We can relax these terms to obtain linear ones using McCormick envelopes (MCCORMICK, 1976). That is, we replace these bilinear terms with new variables ($\varphi_{a,m,n} = \gamma_{a,m,n}\lambda_{a,n}$ and $\psi_{a,n} = \varepsilon\lambda_{a,n}$) and add the following new linear constraints in the problem:

$$0 \le \gamma_{a,m,n} \le 1 \text{ and } 0 \le \lambda_{a,n} \le Q_a \text{ and } 0 \le \varepsilon \le E \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.22a}$$

$$0 \le \varphi_{a,m,n} \le \lambda_{a,n} \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.22b}$$

$$Q_a(\gamma_{a,m,n} - 1) + \lambda_{a,n} \le \varphi_{a,m,n} \le Q_a\gamma_{a,m,n} \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.22c}$$

$$0 \le \psi_{a,n} \le \lambda_{a,n}E \text{ and } \varepsilon Q_a + \lambda_{a,n}E - EQ_a \le \psi_{a,n} \le \varepsilon Q_a \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V} \tag{4.22d}$$

where E is a constant specifying the maximum deadline violation allowed. Then, we can rewrite Constraint (4.21) with the two new variables to have a linear constraint:

$$\begin{aligned}&\left(\varphi_{a,m,n}d^{\text{net}}_{a,m,n} - \psi_{a,n} - \lambda_{a,n}A^{\text{rd}}_a\right)\left(A^{\text{CPU}}_{a,1} - A^{\text{work}}_a\right)\\ &+ \gamma_{a,m,n}\left(A^{\text{CPU}}_{a,2}d^{\text{net}}_{a,m,n} + A^{\text{work}}_a\right) - A^{\text{CPU}}_{a,2}\left(A^{\text{rd}}_a + \varepsilon\right) \le 0\\ &\qquad\qquad\qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V}\end{aligned} \tag{4.23}$$

4.2.3.1.4 Linear Formulation

Let $\varphi = \{\varphi_{a,m,n} \mid a \in \mathscr{A} \text{ and } m,n \in \mathscr{V}\}$, $\psi = \{\psi_{a,n} \mid a \in \mathscr{A} \text{ and } n \in \mathscr{V}\}$, and $x = (\rho, \gamma, \delta, \varepsilon, \varphi, \psi)$. Then, we use the above linearizations and relaxations to formulate the MILP problem of the static service placement problem with a single objective as follows:

$$\begin{aligned}\min_x \quad & \varepsilon\\ \text{s.t.} \quad & x = (\rho, \delta, \gamma, \varepsilon, \varphi, \psi)\\ & \text{Equations (4.6) to (4.9), (4.16), (4.19), (4.22) and (4.23)}\end{aligned} \tag{4.24}$$

It is important to note that a solution to Problem (4.24) is also feasible for Problem (4.13), but it may present a higher objective value ε when applied to the original problem due to the bilinear relaxation.

4.2.4 Multi-Objective Formulation

Based on the single objective optimization problem (4.13) and given $F = (f_1, f_2, \dots, f_M)$ as a list of M performance-related functions, the multi-objective optimization problem for the static approach is formulated as:

$$\begin{aligned} \min \quad & F(x) = (f_1(x), f_2(x), \ldots, f_M(x)) \\ \text{s.t.} \quad & x = (\rho, \delta, \gamma) \\ & \text{Equations (4.6) to (4.11)} \end{aligned} \tag{4.25}$$

According to the InP or ASPs wishes, different objectives can be optimized. Some non-exhaustive performance-related functions are listed below:

- **Deadline Violation** f_{dv}. The violation level of the system defined in Equation (4.12) is a relevant metric to minimize for latency-sensitive applications.
- **Operational Cost** f_{cost}. Deploying applications on the system is not a free operation. Indeed, there is a cost charged, possibly monetary, to ASPs for the resources used by their applications according to the pay-as-you-go pricing model. For this reason, a provider aims to reduce the cost of running a product. Given $N_n^{\text{cost}}(\cdot)$ as the allocation cost function on a node n described in Section 4.1, then the total operational cost for an application is simply the sum of costs on each node hosting an application replica, i.e., $\sum_{n \in \mathscr{V}} \rho_{a,n} N_n^{\text{cost}}(\lambda_{a,n})$. Considering all applications, a performance metric to minimize is the overall operational cost, which is defined as:

$$\begin{aligned} f_{\text{cost}}(x) = & \sum_{a \in \mathscr{A}} \sum_{n \in \mathscr{V}} \rho_{a,n} N_n^{\text{cost}}(\lambda_{a,n}) \\ & x = (\rho, \delta, \gamma) \end{aligned} \tag{4.26}$$

- **Unavailability** f_{fail}. ASPs also want high availability for their applications. An application becomes unavailable when all of its replicas become unavailable. A replica is unavailable when there is a failure in the node hosting it, or an internal failure occurs in its software. In other words, a replica is available if there is no hardware and software failure. We formulate this unavailability of an application as $\prod_{n \in \mathscr{V}} \left(1 - \rho_{a,n} N_n^{\text{avail}} A_a^{\text{avail}}\right)$, where N_n^{avail} and A_a^{avail} are the availability probability of node n and application a respectively. In this way, maximizing the average availability or minimizing the average unavailability across all applications is a metric to be optimized, which we formally specify as follows:

$$\begin{aligned} f_{\text{fail}}(x) = & \frac{1}{|\mathscr{A}|} \sum_{a \in \mathscr{A}} \prod_{n \in \mathscr{V}} \left(1 - \rho_{a,n} N_n^{\text{avail}} A_a^{\text{avail}}\right) \\ & x = (\rho, \delta, \gamma) \end{aligned} \tag{4.27}$$

Unlike single-objective optimization problems that may have a unique optimal solution, in multi-objective optimization problems, conflicts among objectives usually prevent from having a single optimal solution that can optimize all objectives simultaneously. In this way, improvement of one objective may lead to deterioration of another. For instance, reducing unavailability or increasing availability of applications means raising costs by placing more replicas. On the other hand, decreasing costs implies using more cloud resources as they are generally cheaper than edge resources and, thus, increasing the deadline violation. Therefore, it is necessary to search for a set of best optimal compromise solutions by considering trade-offs among the conflicting objectives.

In the case where there is no preference among the objective functions, we can use the Pareto dominance operator, defined in Section 2.3.1, to obtain the set of optimal solutions for a multi-objective problem. However, for the optimization problem (4.25), it is important to improve the performance of time-sensitive applications, which can be considered a preference information. Moreover, we assume that a decision maker either does not have a preference order for other objectives or has difficulties obtaining it. Motivated by this fact, we propose the following modification of Pareto dominance:

Definition 4.2 (Preferred Dominance) *Let f_1 be the highest priority function to be optimized among the list of objective functions $F = (f_1, f_2, \dots, f_M)$. A feasible solution x_1 dominates another solution x_2, expressed as $x_1 \prec^1 x_2$, when*

$$\begin{aligned} &x_1 \prec^1 x_2 \textit{ iff } f_1(x_1) < f_1(x_2) \\ &\qquad\qquad \textit{or } (f_1(x_1) = f_1(x_2) \textit{ and } x_1 \prec x_2) \end{aligned}$$

In other words, the dominance operator $\prec^1$ prioritizes a selected function f_1 and, then, it is sufficient that a solution x_1 has a smaller value than another solution x_2 in f_1 in order for x_1 dominates x_2. Otherwise, if they have equal values for f_1, then the traditional Pareto dominance operator $\prec$ is used instead. Therefore, we can select a performance-related function to time-sensitive applications as a priority goal. For instance, we can set the deadline violation in Equation (4.12) as the primary objective to be optimized, i.e., $f_1 = f_{dv}$.

4.3 A Genetic-Based Proposal

Although well-known linear solvers, such as IBM ILOG CPLEX[1], can solve MILP problems, these problems are generally NP-Hard (ZHAO; LIU, 2018). Moreover, problem (4.24) is highly time-consuming due to a large number of variables. Another limitation of these solvers

[1] https://www.ibm.com/products/ilog-cplex-optimization-studio

is that they only address single-objective optimization problems. A way to overcome this limitation is transforming a multi-objective optimization into a single-objective problem through scalarizing methods, such as weighted sum. However, these methods assume that there is a global preference order among all objectives to be optimized, which may be hard to define in practical cases. Furthermore, in many multi-objective optimizations, it is difficult to obtain the exact Pareto optimal set. Therefore, we propose a meta-heuristic based on Genetic Algorithms to obtain near-optimal solutions. Some advantages of a genetic approach are that it is not limited to linear or single-objective problems and it could be implemented in a parallel environment (CUI *et al.*, 2017).

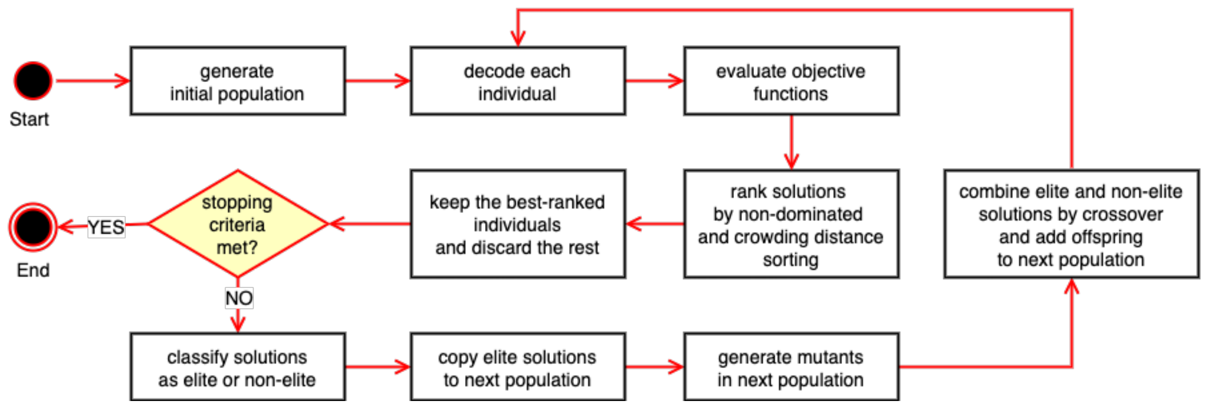
Despite the advantages of GAs, they are usually employed for unconstrained optimization problems. A simple method for incorporating constraints into a GA is by adding a penalty factor into the objective function based on how severe is the constraints violation in a specific solution. However, it may be hard to estimate good penalty factors or even generate feasible solutions for complex optimization problems (COELLO, 2002). Therefore, we can apply BRKGA, described in Section 2.4.1, to handle constrained optimization problems by establishing special solution representation and genetic operators to preserve solutions feasibility.

As BRKGA was initially designed for single-objective optimizations, it lacks support for pure multi-objective problems. On the other hand, GAs are suited for multi-objective problems due to the simultaneous evaluation of many candidate solutions. Hence, we extend BRKGA by incorporating the idea of Pareto optimality during the better fit selection process. More specifically, we include the non-dominance sorting and diversity preservation mechanism of the NSGA-II, presented in Section 2.4.2, in the population classification strategy of BRKGA.

Figure 17 shows the flowchart of our genetic algorithm, called BRKGA+NSGA-II, that combines BRKGA and NSGA-II. Initially, the first population is made up of randomly generated and pre-determined individuals. In each generation or iteration, the feasible solution of every individual in the population is obtained through a decoder algorithm. This decoding procedure can be parallelized to speed up the overall algorithm execution (GONÇALVES; RESENDE, 2011). After evaluating each feasible solution according to the objective functions, the population is sorted based on the non-dominated and crowding distance operations. Before partitioning the population into elite and non-elite groups, only the N_{pop} best-ranked individuals/solutions are kept, where $N_{\text{pop}} > 0$ is the population size of each generation. A new generation of individuals is then produced by copying the elite members and through mutation and crossover procedures.

Finally, the iterative process is repeated until a stopping criterion is met.

Figure 17 – BRKGA+NSGA-II flowchart


Start
generate initial population
decode each individual
evaluate objective functions
rank solutions by non-dominated and crowding distance sorting
keep the best-ranked individuals and discard the rest
stopping criteria met?
YES
End
NO
classify solutions as elite or non-elite
copy elite solutions to next population
generate mutants in next population
combine elite and non-elite solutions by crossover and add offspring to next population

Source: Author.

In the next subsections, we detail the main evolutionary operations of the proposed genetic algorithm. More specifically, we describe our proposed chromosome representation and decoder in Subsection 4.3.1. Next, Subsection 4.3.2 describes how to form the initial population. Then, the next population generation and stopping criteria procedure are detailed in Subsection 4.3.3. Finally, we analyze the overall complexity of our genetic algorithm in Subsection 4.3.4.

4.3.1 Chromosome Representation and Decoder

In BRKGA as well in BRKGA+NSGA-II, the chromosome representation and decoder algorithm play essential roles as the problem-dependent part of these genetic algorithms. Consequently, we need to design a chromosome representation and decoder algorithm to produce feasible solutions to problem (4.25). Then, our proposed chromosome representation and the description of its parts are given below:

$$\begin{aligned} C = \Big[& C^{\mathrm{I}}_1, C^{\mathrm{I}}_2, \dots, C^{\mathrm{I}}_{|\mathscr{A}|}, \\ & C^{\mathrm{II}}_{1,1}, C^{\mathrm{II}}_{1,2}, \dots, C^{\mathrm{II}}_{1,|\mathscr{V}|}, \dots, C^{\mathrm{II}}_{|\mathscr{A}|,1}, C^{\mathrm{II}}_{|\mathscr{A}|,2}, \dots, C^{\mathrm{II}}_{|\mathscr{A}|,|\mathscr{V}|}, \\ & C^{\mathrm{III}}_1, C^{\mathrm{III}}_2, \dots, C^{\mathrm{III}}_Q \Big] \end{aligned}$$

1. $C^{\mathrm{I}}_a \in [0,1]$ is used to define the number of nodes to be selected as host candidates of an application $a \in \mathscr{A}$. As a result, it upper limits the number of application replicas to be placed.
2. $C^{\mathrm{II}}_{a,n} \in [0,1]$ specifies the priority to place a replica of application $a \in \mathscr{A}$ in a node $n \in \mathscr{V}$.

3. $C_q^{\mathrm{III}} \in [0,1]$ is related to the order of a request q assigned to a replica, where $q \in \{1,2,\ldots,Q\}$ and $Q = \sum_{a\in\mathscr{A}} \sum_{m\in\mathscr{V}} Q_{a,m}$ is the total number of requests.

4.3.1.1 Decoder Algorithm

The proposed Algorithm 2 decodes the above chromosome representation into a feasible solution. Its basic idea is to first select the potential placement locations of each application (lines 4 to 7). For this, it takes the first part of the chromosome (C_a^{I}) to define the number of nodes to be selected as candidates to host an application replica (line 5). The quantity of selected nodes is upper limited by the total number of nodes in the system ($|\mathscr{V}|$) and the maximum allowed number of replicas ($A_a^{\max}$). Then, nodes with high values in the second part of the chromosome ($C_{a,n}^{\mathrm{II}}$) are chosen as potential deployment sites for an application. The cloud node is also added as a possible location to ensure that there are resources to deploy at least one replica of each application on the network.

The second step of Algorithm 2 is the load/request distribution (lines 8 to 21). It creates a sequence of all requests conforming to the third part of the chromosome (C_q^{III}). Following this sequence, requests are assigned one at a time among the nodes selected in the first step of the algorithm. For a request, the decoder looks for a node with short response time and sufficient resources to receive it. When the first envisioned target node is found, it sets to place a replica of the requested application on this node and assigns the request to this replica. As we assume that the cloud node has unlimited capacity and is included in the selected nodes set, it is always possible to find a target node with available resources to receive an additional request. Moreover, Algorithm 2 indicates that a request is assigned to an application replica by incrementing this replica workload (line 14 and 18).After increasing an application replica workload, the decoder updates the free resources on the node hosting this replica and the replica response time. Note that only the processing delay may be affected by a growing workload in our response time estimation. Thus, using the estimated response time to select the target node is a way to distribute load among nodes without ignoring the network delay.

Finally, the decoder algorithm verifies that the maximum number of replicas of an application is respected and replaces surplus replicas with the cloud node (lines 22 to 25).

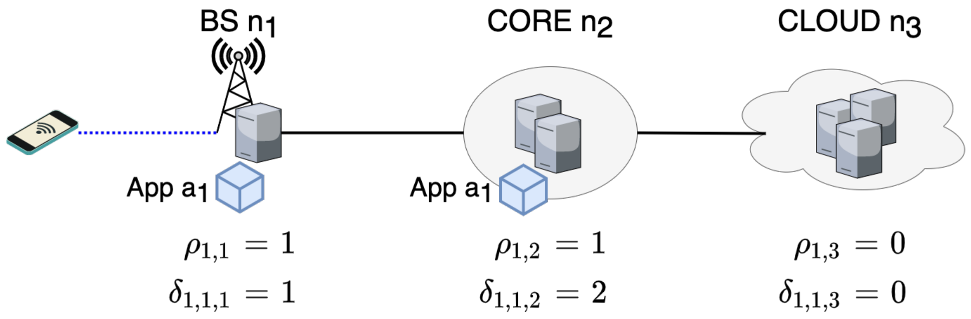
In order to exemplify how Algorithm 2 works, let us define a simple system model with three nodes (one base station, the core, and the cloud), and one application, as shown in Figure 18a. In this system model, the first and only application specifies that a maximum

Algorithm 2: Chromosome decoder for the static service placement problem

```
Data: individual
Result: Decoded solution $(\rho, \delta, \gamma)$
initialize $\rho_{a,n}$, $\gamma_{a,m,n}$, $\delta_{a,m,n} \leftarrow 0$;
initialize $d_{a,m,n} \leftarrow d^{\text{net}}_{a,m,n}$;
$C^{\text{I}}, C^{\text{II}}, C^{\text{III}} \leftarrow$ individual.chromosome;
/* Step I: Node Selection                                              */
forall $a \in \mathscr{A}$ do
    $z \leftarrow \min\left(|\mathscr{V}|, \lceil C^{\text{I}}_a A^{\max}_a \rceil\right)$;
    $V_a \leftarrow$ select $z$ nodes with higher $C^{\text{II}}_{a,n}$, $n \in \mathscr{V}$;
    $V_a \leftarrow V_a \cup \{cloud\}$;
/* Step II: Request Distribution                                       */
$L \leftarrow$ list of requests sorted by $C^{\text{III}}$ in descending order;
forall $r \in L$ do
    $a \leftarrow app_r$;                          // requested application of $r$
    $m \leftarrow source_r$;                                 // source node of $r$
    sort nodes $n$ in $V_a$ by $d_{a,m,n}$ in ascending order;
    forall $n \in V_a$ do
        $l \leftarrow \delta_{a,m,n} + 1$;
        if $(\rho_{a,n} = 1, \gamma_{a,m,n} = 1, \delta_{a,m,n} = l)$ respects constraints (4.10) and (4.11) then
            $\rho_{a,n} \leftarrow 1$;
            $\gamma_{a,m,n} \leftarrow 1$;
            $\delta_{a,m,n} \leftarrow l$;
            update free resources on node $n$ given $(\rho, \delta, \gamma)$;
            update $d_{a,m,n}$ by Equation (4.1) and current $(\rho, \delta, \gamma)$;
            break;
/* Step III: Feasibility Verification                                  */
forall $a \in \mathscr{A}$ do
    $z \leftarrow A^{\max}_a - \sum_{n \in \mathscr{V}} \rho_{a,n}$;
    if $z > 0$ then
        replace $z + 1$ replicas of $a$ with the cloud node;
```

of four replicas can be placed in the system (i.e., $A^{\max}_1 = 4$), and it has three requests to be distributed. Moreover, the base station has resources available to receive a maximum of one request, while the core node can receive two requests, and the cloud has no restriction. Figure 18b shows the chromosome of an individual in this defined system. This chromosome leads the decoder algorithm to select only two nodes as candidates to host the application because $\lceil C^{\text{I}}_1 A^{\max}_1 \rceil = \lceil 0.5 \times 4 \rceil = 2$. More specifically, the base station and core nodes are selected because they have higher values in the second part of the chromosome (i.e., $C^{\text{II}}_{1,1} = 0.7, C^{\text{II}}_{1,2} = 0.4$). The third part of the chromosome indicates that request C^{III}_2 is assigned first and followed by requests C^{III}_1 and C^{III}_3. According to the shortest response time order in the second step of Algorithm 2, a replica of the application with request C_2^{III} is placed in the base station and, then, another replica with the remaining requests C_1^{III} and C_3^{III} are placed in the core node. According, Figure 18c shows the feasible solution decoded from the exemplified chromosome.

Figure 18 – Decoding example for Algorithm 2



(a) System model

C_1^{I}	$C_{1,1}^{\text{II}}$	$C_{1,2}^{\text{II}}$	$C_{1,3}^{\text{II}}$	C_1^{III}	C_2^{III}	C_3^{III}
0.5	0.7	0.4	0.1	0.6	0.8	0.3

(b) Chromosome as an input parameter

$\rho_{1,1}$	$\rho_{1,2}$	$\rho_{1,3}$	$\delta_{1,1,1}$	$\delta_{1,1,2}$	$\delta_{1,1,3}$
1	1	0	1	2	0

(c) Feasible solution decoded

Source: Author.

4.3.1.2 Complexity Analysis

Let $A = |\mathscr{A}|, V = |\mathscr{V}|$ and $R = |\mathscr{R}|$. The first outermost loop of Algorithm 2 (lines 4 to 7) has complexity $O(AV \log V)$ due to the sorting procedure on line 6. Line 8 has complexity $O(Q \log Q)$, where Q is the total number of requests. By maintaining the current amount of free resources on each node, checking the satisfaction of constraints on line 15 can be done in $O(R)$. The update of variables between lines 16 and 20 has complexity $O(1)$. Then the complexity of the second outermost loop (lines 9 to 21) is $O(QV \log V + QVR)$. The loop between lines 22 and 25 has complexity $O(AV)$. Thus, Algorithm 2 has complexity $O((A+Q)V \log V + Q(VR + \log Q))$. As $O(\log n) \in O(n)$ for $n > 0$, then Algorithm 2 has polynomial complexity.

4.3.2 Initial Population

Along with randomly generated individuals, the initial population of BRKGA+NSGA-II also includes a few solutions obtained from simple heuristic methods for the specific optimization problem being solved. This strategy may help speed up the algorithm convergence and improve the quality of the final solutions. Hence, we add to the initial population individuals generated by the following heuristics that encode some feasible solutions to the proposed chromosome representation:

- **Cloud**. A simple solution is to place all applications in the cloud. According to Algorithm 2 (lines 4-7), it is sufficient that $C^{\mathrm{I}}_{a} = 0$ to only select the cloud node. The remaining chromosome parts can have any value, but we set to zero as a default value. Thus, the solution is encoded as:

$$\begin{aligned} C^{\mathrm{I}}_{a} &= 0 & \forall a \in \mathscr{A} \\ C^{\mathrm{II}}_{a,n} &= 0 & \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \\ C^{\mathrm{III}}_{q} &= 0 & \forall q \in [1, Q] \end{aligned}$$

- **Deadline**. Another heuristic is to prioritize requests for applications with shorter deadline requirements. Given $A^{\mathrm{rd}}_{app_q}$ as the application deadline requirement of request q, the request prioritization is established in the third chromosome part by normalizing $A^{\mathrm{rd}}_{app_q}$ against the maximum deadline requirement among all applications and subtracting the normalized result from one. In this way, requests with short deadlines have high priorities. In addition, the heuristic selects as many nodes as possible, i.e., $C^{\mathrm{I}}_{a} = 1$, for an application to reduce response time and deadline violations. The second chromosome part is set to the default value, i.e., $C^{\mathrm{II}}_{a,n} = 0$, which informs that node ranking is not a concern of this heuristic. Therefore, the following encoded solution is produced:

$$\begin{aligned} C^{\mathrm{I}}_{a} &= 1 & \forall a \in \mathscr{A} \\ C^{\mathrm{II}}_{a,n} &= 0 & \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \\ C^{\mathrm{III}}_{q} &= 1 - \frac{A^{\mathrm{rd}}_{app_q}}{\max_{a \in \mathscr{A}} \left\{ A^{\mathrm{rd}}_{a} \right\}} & \forall q \in [1, Q] \end{aligned}$$

- **Net Delay**. Zhao and Liu (2018) propose a heuristic that selects nodes with the lowest network latency to all other nodes as candidates to host an application. In order to prioritize a node according to this heuristic, we assign $C^{\mathrm{II}}_{a,n}$ to the normalized accumulative network delay for application a between node n and all nodes. We then subtract the normalized result from one as the proposed decoder algorithm selects nodes with higher $C^{\mathrm{II}}_{a,n}$ values. Moreover, we set $C^{\mathrm{I}}_{a} = 1$ to pick as many nodes as possible and $C^{\mathrm{III}}_{q} = 0$ so that request sorting is not a heuristic concern. Hence, we encode this heuristic as follows:

$$
\begin{aligned}
&C^{\mathrm{I}}_a = 1 && \forall a \in \mathscr{A} \\
&C^{\mathrm{II}}_{a,n} = 1 - \frac{\sum_{i\in\mathscr{V}} d^{\mathrm{net}}_{a,i,n}}{\max_{j\in\mathscr{V}}\left\{\sum_{i\in\mathscr{V}} d^{\mathrm{net}}_{a,i,j}\right\}} && \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \\
&C^{\mathrm{III}}_q = 0 && \forall q \in [1, Q]
\end{aligned}
$$

- **Cluster**. A heuristic is to place replicas of an application in regions where users of this application are located. We can define a region as a set of close nodes where there are users attached to them. Moreover, we apply the K-medoids clustering technique (PARK; JUN, 2009) to detect these regions. The K-medoids algorithm is a variation of the classical K-means algorithm. Despite the similarity between these two algorithms, the K-medoids algorithm chooses points in the dataset as centers or medoids of the clusters based on any arbitrary distance function. In contrast, in K-means, a cluster center is not necessary a point in the dataset. Moreover, Park and Jun (2009) propose a simple and fast K-metoids algorithm with time complexity of $O(NK)$, where N is the dataset size and K is the number of clusters. For each application $a \in \mathscr{A}$, these K-medoids characteristics allow us to partition nodes $n \in \mathscr{V}$ where $|\mathscr{U}_{a,n}| > 0$ in clusters using the network delay $d^{\mathrm{net}}_{a,m,n}$ to other nodes $m \in \mathscr{V}$ as the distance function. Let $\mathscr{M}_a$ be the set of medoids obtained after performing the clustering algorithm for application a with a maximum of $A^{\max}_a$ clusters. Then, the heuristic prioritizes the application placement in nodes near to a center of $|\mathscr{M}_a|$ regions. By applying the similar normalization procedure of *Deadline* and *Net Delay* heuristics, the *Cluster* heuristic is formally encoded as:

$$
\begin{aligned}
&C^{\mathrm{I}}_a = 1 && \forall a \in \mathscr{A} \\
&C^{\mathrm{II}}_{a,n} = 1 - \frac{\min_{i\in\mathscr{M}_a}\left\{d^{\mathrm{net}}_{a,i,n}\right\}}{\max_{j\in\mathscr{V}} \min_{i\in\mathscr{M}_a}\left\{d^{\mathrm{net}}_{a,i,j}\right\}} && \forall a \in \mathscr{A}, \forall n \in \mathscr{V} \\
&C^{\mathrm{III}}_q = 0 && \forall q \in [1, Q]
\end{aligned}
$$

- **Combined Solution**. Given $\mathscr{S}$ as a set of random-key chromosome vectors and v_i the i-th element of a vector $v \in \mathscr{S}$ with length N, then we can combine two or more heuristic solutions by summing their encoded vectors as specified below:

$$
v^+_i = \frac{1}{|\mathscr{S}|} \sum_{v\in\mathscr{S}} v_i \quad \forall i \in \{1, \ldots, N\}
$$

Indeed, we combine in pairs all vectors created by *Cloud*, *Deadline*, *Net Delay*, and *Cluster* heuristics. Then, the resulted individuals are added to the initial population.

– **Complementary Solution**. In order to add more diversity to the initial population, we can add complementary solutions to those obtained by the above heuristics. As each element in a chromosome v is in the range $[0,1]$, we can then obtain the complementary solution of v by doing the following operation:

$$v_i^- = 1 - v_i \quad \forall i \in \{1,\ldots,N\}$$

4.3.2.1 *Complexity Analysis*

In order to generate the random individuals, it is required $O\left(N_{\text{pop}}N\right)$ computations, where N_{pop} is the population size and N is the chromosome vector length. For our chromosome representation in Subsection 4.3.1, the vector length N is equal to $A + AV + Q$, where $A = |\mathscr{A}|, V = |\mathscr{V}|$, and Q is the total number of requests. Thus, $O\left(N_{\text{pop}}N\right) = O\left(N_{\text{pop}}(AV + Q)\right)$.

Cloud and *Deadline* heuristics have complexity $O(N) = O(AV + Q)$ because they basically iterate over the vector elements. As *Combined* and *Complementary* solutions go over a constant number of individuals, their complexities are also $O(AV + Q)$. Meanwhile, *Net Delay* complexity is $O\left(AV^2 + Q\right)$ due to the summation in the chromosome third part. Before the vector elements iteration, *Cluster* performs K-medoids for each application with a maximum of V clusters and V points in the dataset. Therefore, the complexity of the *Cluster* heuristic is $O\left(AV^2 + N\right) = O\left(AV^2 + Q\right)$.

By considering the randomly and heuristically generated individuals, the overall time complexity of initial population formation is $O\left(N_{\text{pop}}(AV + Q) + AV^2\right)$.

4.3.3 Next Population and Stopping Criteria

In BRKGA+NSGA-II, a new population is created by including (*i*) elite individuals of the current population, (*ii*) mutant individuals, and (*iii*) offspring individuals. The proposed GA generates a mutant individual as a simple vector of random values. Thus, it adds a specified number of mutant individuals in the new population. Then, BRKGA+NSGA-II completes the population with offspring. Each new offspring individual is produced by combining two randomly selected solutions as parents from the current population. In this selection, a parent comes from the elite group and the other solution from the non-elite group. Moreover, the offspring genes (i.e., vector values) are defined by the parameterized uniform crossover described in Section 2.4.1, where P_{elite} is the probability of the offspring inherits a gene of its elite parent.

After generating a new population, BRKGA+NSGA-II decodes the individuals in this population into feasible solutions. Next, it applies non-dominated and crowding distance sorting to rank the individuals based on the evaluation of their decoded solutions. In this sorting procedure, either the Pareto $\prec$ or preferred $\prec^1$ dominance operators can be used. Then, BRKGA+NSGA-II only keeps the best-ranked individuals in the population of the next generation.

The above-discussed procedure for generating the next population is repeated until a stopping criterion is met. For BRKGA+NSGA-II, we select two stopping criteria: (*i*) the maximum number of generations $t_{\max}$ and (*ii*) the MGBM criteria. As discussed in Section 2.4.3, MGBM tries to detect situations where no further progress will be made in the GA by designing a progress estimator $\hat{I}_{\text{mdr}}(t)$. Thus, the proposed GA terminates either when it reaches a maximum number of generations (i.e., $t \geq t_{\max}$) or when the progress estimation falls below a defined threshold (i.e., $\hat{I}_{\text{mdr}}(t) < \hat{I}_{\text{mdr}}^{\min}$). Here, $t_{\max}$ and $\hat{I}_{\text{mdr}}^{\min}$ are parameters to be set, and $t \geq 1$ is the generation/iteration index.

4.3.4 Complexity Analysis of the Genetic Algorithm

In an iteration or generation of BRKGA+NSGA-II, the operations shown in Figure 17 have the following complexities:

- Decoding N_{pop} individuals by Algorithm 2 is $O\left(N_{\text{pop}}D\right)$, where $D = (A+Q)V\log V + Q(VR + \log Q)$.
- Evaluating M objective functions for N_{pop} solutions is $O\left(N_{\text{pop}}MF\right)$, where $O(F)$ is the worst-case complexity of computing an objective function. For instance, function $f_{\text{dv}}(\cdot)$ in Equation (4.12) requires $O\left(AV^2\right)$ computations to obtain the deadline violation of each request flow.
- Sorting all solutions is $O\left(MN_{\text{pop}}^2\right)$ as described in Section 2.4.2.
- As discussed in Section 2.4.3, checking the MGBM stopping criteria is also $O\left(MN_{\text{pop}}^2\right)$.
- After the sorting procedure, keeping the best individuals and classifying them as elite or non-elite is $O\left(N_{\text{pop}}\right)$.
- Generating the next population by mutation and crossover operations has complexity $O\left(N_{\text{pop}}N\right)$, where $N = A + AV + Q$ is the length of chromosome vector presented in Section 4.3.1.

The time complexity of a single iteration is then $O\left(N_{\text{pop}}\left(D + MF + MN_{\text{pop}}\right)\right)$.

Given that there are at most $t_{\max}$ iterations and $O\left(N_{\text{pop}}(AV+Q)+AV^2\right)$ is the complexity of the initial population generation, the overall complexity of BRKGA+NSGA-II is equal to $O\left(t_{\max}N_{\text{pop}}\left(D+MF+MN_{\text{pop}}\right)+AV^2\right)$ when the chromosome representation and decoder described in Section 4.3.1 are used. By assuming $O\left(AV^2\right) \in O(F)$, the time complexity of our genetic algorithm can be reduced to $O\left(t_{\max}N_{\text{pop}}\left(D+MF+MN_{\text{pop}}\right)\right)$. This assumption is valid, for instance, if an objective function, such as $f_{\text{dv}}(\cdot)$, evaluates each request flow.

Therefore, the population decoding, evaluation, and sorting procedures govern the computational complexity of our genetic algorithm. Despite that, the decoding and evaluation procedures involve independent computations for each individual and can then be parallelized to accelerate the algorithm execution.

4.4 Performance Analysis

In this section, we present the performance (i.e., the optimality) results of our proposed GA by comparing it with benchmarking algorithms over a cellular network (5G) with Edge Computing (EC) capabilities.

This section is structured as follows. First, Subsection 4.4.1 presents the performance metrics. Next, Subsection 4.4.2 describes the evaluated algorithms. Then, Subsection 4.4.3 details the experiment setup. In Subsection 4.4.4, we define the values of key parameters of the proposed GA. Finally, we analyze the obtained experimental results in Subsection 4.4.5.

4.4.1 Performance Metrics

We select the deadline violation f_{dv}, operational cost f_{cost}, and service unavailability f_{fail} as performance-related functions to be optimized. We chose these functions because they are relevant in the context of EC, and there are conflicts between them, as discussed in Section 4.2.4. It is important to note that we can use other objectives since the proposed algorithm has no restrictions on this aspect.

4.4.2 Evaluated Algorithms

An overview of the compared algorithms is given below:

– ***MILP*** returns the optimal solution for the relaxed MILP problem (4.24), which it found by the branch and cut technique in the CPLEX linear solver. We also use a timeout parameter equal to 2 hours as a stopping criterion of the solver tool.

– ***Cloud*** simply places everything in the cloud node.
– ***NetDelay+DL*** combines the *Net Delay* and *Deadline* heuristics presented in Section 4.3.2. We combine these two heuristics to have both node and request priorities well defined in the resulting chromosome vector. Then, Algorithm 2 decodes the combined chromosome vector into a feasible solution.
– ***Cluster+DL*** combines the *Cluster* and *Deadline* heuristics presented in Section 4.3.2. Like *NetDelay+DL*, the combined chromosome vector of *Cluster+DL* defines priorities to nodes and requests, and Algorithm 2 decodes this vector into a feasible solution.
– ***MOHGA***$(\prec^1)$ is the proposed GA for multi-objective using the heuristic initialization and the dominance operator $\prec^1$.
– ***MOHGA***$(\prec)$ is similar to *MOHGA*$(\prec^1)$ but using the Pareto dominance operator $\prec$ instead.
– ***MOGA***$(\prec^1)$ is the same as *MOHGA*$(\prec^1)$ but without using the heuristic initialization. That is, the first population is only randomly generated.
– ***SOHGA***(f) uses the proposed GA with heuristic initialization to optimize a single objective function $f \in \{f_{\text{dv}}, f_{\text{cost}}, f_{\text{fail}}\}$.

4.4.3 Analysis Setup

We conduct the experiment in Python with the CPLEX solver (CPLEX, 2009) to evaluate the performance of the above-mentioned algorithms in a 5G network scenario. Moreover, we use a server machine with Intel Xeon E5-2630 @ 2.60GHz, 24 CPUs, and 64 GB of RAM to run the compared algorithms.

In the experiment scenario, Base Stations (BSs) are equally distributed in a grid area, and there is a network link between neighboring BSs. These BSs are also connected to a core node, which is connected to the cloud on the other side. All of these nodes (BSs, core, and cloud) have hosting capabilities, and their total resource capacities are reduced as we descend from cloud to BSs. Table 8 summarizes the major experiment parameters.

We use the three types of applications specified for 5G networks,whose characteristics are described as follows (ALLIANCE, 2015):

– **massive Machine Type Communications (mMTC)** has low resource usage, a high deadline tolerance, and a large number of users;
– **Ultra Reliable Low Latency Communications (URLLC)** has low resource usage, a

Table 8 – Performance evaluation parameters for the static service placement problem

Parameter	**Value**
System	
CPU (MIPS)	Cloud: ∞, Core: 200000, BS: 40000
Storage Disk (MB)	Cloud: ∞, Core: 32000, BS: 16000
RAM (MB)	Cloud: ∞, Core: 8000, BS: 4000
Node Availability N_n^{avail} (%)	Cloud: 99.9, Core: 99.0, BS: 90.0
Usage Cost $N_{n,1}^{\text{cost}}, N_{n,2}^{\text{cost}}$	Cloud: 0.025, Core: 0.05, BS: 0.1
User Proportion (%)	70 mMTC, 20 eMBB, 10 URLLC
App. Proportion (%)	34 mMTC, 33 eMBB, 33 URLLC
Applications	
Max. Replicas $A_a^{\max}$	$[1, \lvert\mathscr{V}\rvert]$
Deadline A_a^{rd} (ms)	mMTC: $[100, 1000]$, URLLC: $[1, 10]$, eMBB: $[10, 50]$
A_a^{req} (requests/ms)	mMTC: $[0.0002, 0.001]$, eMBB: $[0.001, 0.01]$, URLLC: $[0.02, 0.2]$
App. Availability A_a^{avail} (%)	mMTC, eMBB: $[80.0, 90.0]$, URLLC: $[90.0, 99.0]$
CPU Work A_a^{work} (MI)	mMTC, URLLC: $[1, 5]$, eMBB: $[1, 10]$
RAM, Disk $A_{a,1}^r, A_{a,2}^r$	mMTC, URLLC: $[1, 10]$, eMBB: $[1, 50]$
CPU $A_{a,1}^r, A_{a,2}^r$	A_a^{work}, $A_a^{\text{work}}/A_a^{\text{rd}} + 1$
Net. Delay $D_{a,m.n}^{\text{net}}$ (ms) neighbor BS-BS, BS-Core	mMTC, URLLC: $[1, 2]$, eMBB: $[1, 5]$
Net. Delay $D_{a,m.n}^{\text{net}}$ (ms) Core-Cloud	mMTC, URLLC: $[10, 12]$, eMBB: $[10, 15]$

Source: Author.
Note: An interval $[a, b]$ means that a value is chosen randomly within this range.

strict deadline, and a small volume of users; and

– **enhanced Mobile Broadband (eMBB)** has high resource usage, a medium deadline, and an intermediate number of users.

Although IoT applications are generally classified as mMTC or URLLC types, we also include eMBB in our experiments to have a diversity of applications expected in the heterogeneous scenario of Edge Computing.

Then, we randomly assign the value of the application parameters based on the above characteristics and some predictions for 5G discussed by Schulz *et al.* (2017) (response deadline and request rate). For evaluation purposes, we assume that it is sufficient to use relative values for the parameters among different application types instead of applying more realistically accurate values. In addition, we also assume that the application resource demand $A_a^r(\cdot)$ and node usage cost $N_n^{\text{cost}}(\cdot)$ functions are linear according to Equations (4.14) and (4.28), respectively, where

$A^r_{a,1}, A^r_{a,2}, N^{\text{cost}}_{n,1}$, and $N^{\text{cost}}_{n,2}$ are constants. In Equation (4.28), $N^{\text{cost}}_n(\cdot)$ is composed of the cost of placing an application replica plus the cost of allocating each resource for this replica.

$$\begin{aligned} N^{\text{cost}}_n(\lambda_{a,n}) &= N^{\text{cost}}_{n,1}\rho_{a,n} + \sum_{r\in\mathscr{R}} N^{\text{cost}}_{n,2} A^r_a(\lambda_{a,n}) \\ &= N^{\text{cost}}_{n,1}\rho_{a,n} + \sum_{r\in\mathscr{R}} \left(N^{\text{cost}}_{n,2} A^r_{a,1}\lambda_{a,n} + N^{\text{cost}}_{n,2} A^r_{a,2}\right) \end{aligned} \tag{4.28}$$

In order to have different user densities, users are distributed either uniformly or through isotropic Gaussian blobs (SCIKIT-LEARN, 2020) in the grid area. Then, a user is attached to the nearest BS. Finally, each test case is executed 30 times to obtain results with a 95% confidence interval (JAIN, 1991).

4.4.4 Different Parameters Settings

Regarding the parameters of the proposed $MOHGA(\prec^1)$, we analyze its performance in terms of the optimization objectives against different values of these parameters.

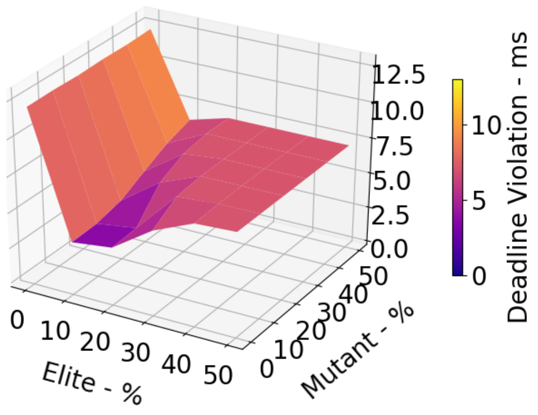
4.4.4.1 Elite and Mutant Group Size

In a GA with elitist strategy and random mutants, some parameters to be defined are the number of individuals in the elite and mutant sets. Figure 19 presents the influence of these parameters in the objective functions for $MOHGA(\prec^1)$ with a population size of 100. We observe that parameter values in the range between 10% and 20% produce better results for all three objectives. These values allow the algorithm to have a diversity of solutions within the search space and still benefit from the elitist strategy. Therefore, we select the number of elite and mutant individuals to be both 10% of all population.

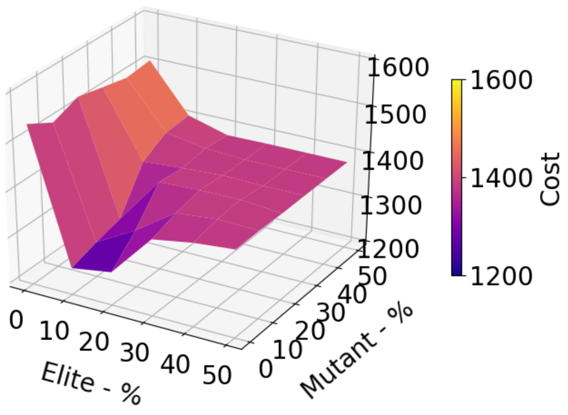
4.4.4.2 Elite Probability

In the crossover operation described in Sections 2.4.1 and 4.3.3, a parameter to be defined is the probability P_{elite} to an offspring inheriting a gene of its elite parent. Figure 20 presents the impact of different values of P_{elite} on the objectives functions. We can see that this parameter does not have much influence on the objectives, but $P_{\text{elite}} = 0.6$ has a slightly better results, specially for service availability in Figure 20c.

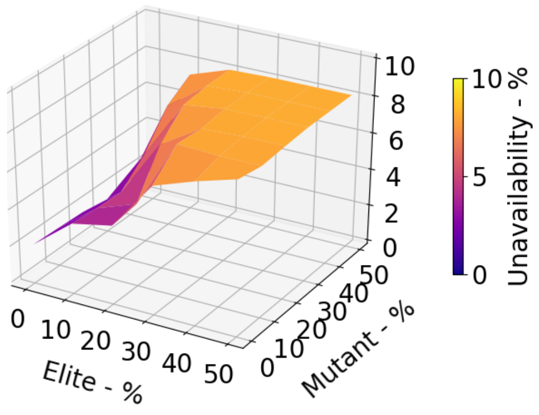
Figure 19 – Performance of different elite and mutant group sizes



(a) Max. Deadline Violation



(b) Overall Operational Cost



(c) Avg. Service Unavailability

Source: Author.

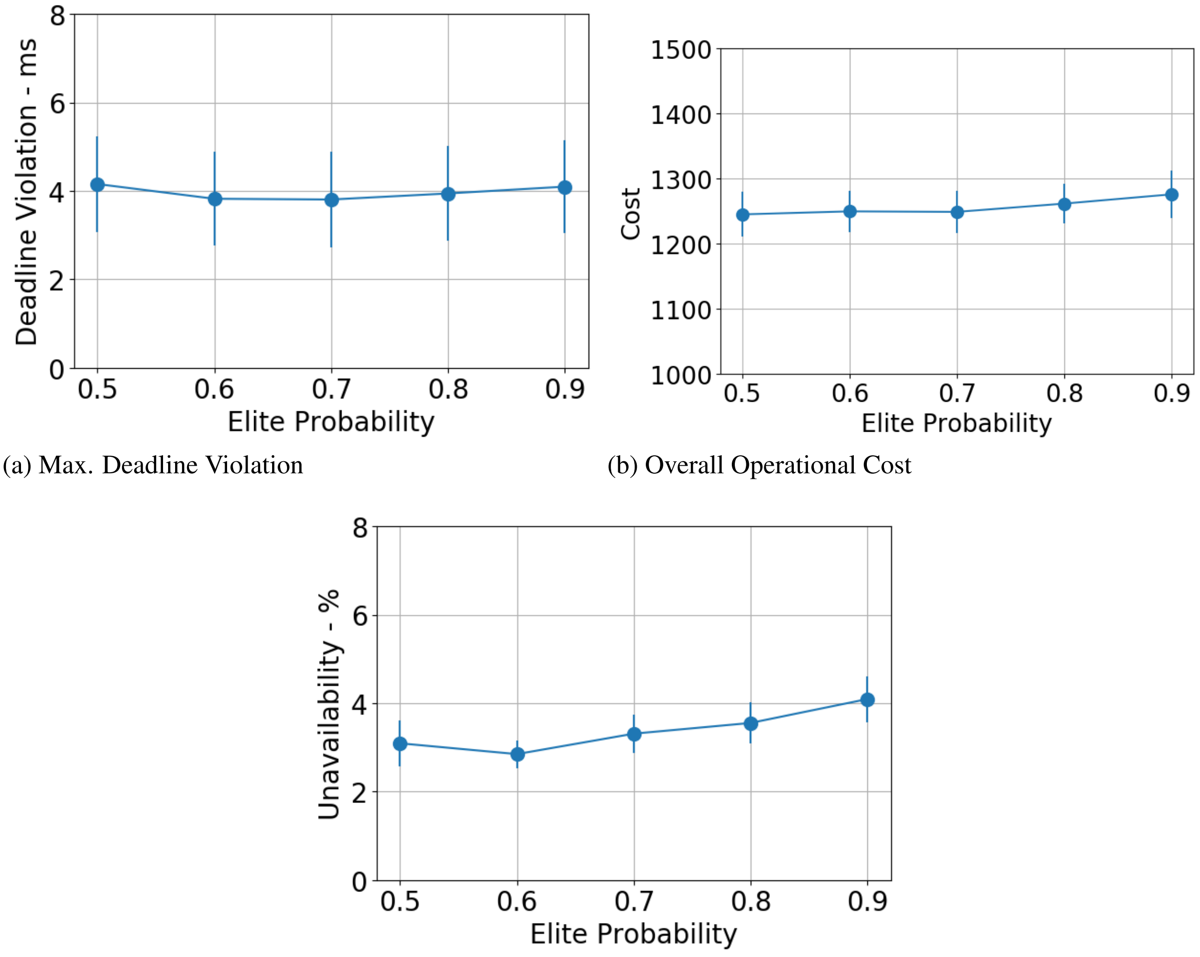
4.4.4.3 Stopping Criteria Threshold

We examine the performance of different stopping threshold $\hat{I}_{\mathrm{mdr}}^{\mathrm{min}} \in [0, 0.5]$ values for *MOHGA*$(\prec^1)$ with a maximum of 100 generations and a population size of 100 for each generation. Regarding the optimization objectives, the algorithm performs better when the threshold is below 0.2. However, a small threshold implies that the algorithm iterates over more generations, consequently, resulting in longer execution time, as shown in Figure 21d. Hence, we select $\hat{I}_{\mathrm{mdr}}^{\mathrm{min}} = 0.1$ as a trade-off between optimality and execution time.

4.4.5 Results and Discussion

We evaluated the performance of the examined algorithms for each metric in scenarios with different amounts of applications, users, and nodes.

Figure 20 – Performance of different elite probability P_{elite} values



(a) Max. Deadline Violation (b) Overall Operational Cost

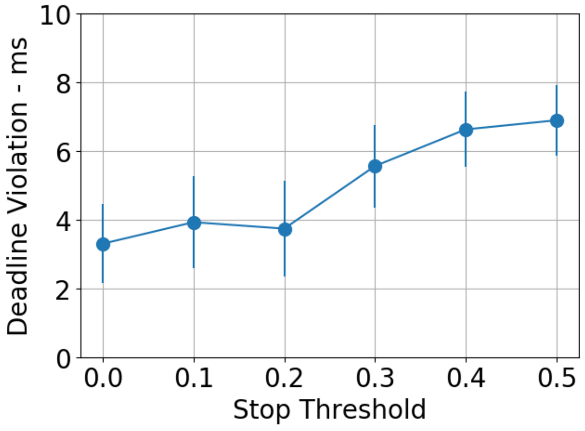
(c) Avg. Service Unavailability

Source: Author.

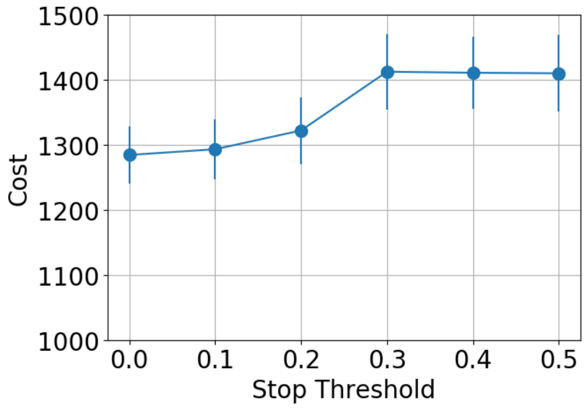
4.4.5.1 Maximum Deadline Violation

Figure 22a presents the impact on the system deadline violation level by increasing the number of deployed applications in a scenario with a two-dimensional grid of 5x5 BSs and $10,000$ users. In all algorithms, the violation level grows as more applications compete for the fixed amount of node resources. More specifically, *Cloud* heuristic has the worst results due to the distance between the users and the cloud node. Meanwhile, the optimum solutions of *MILP* present, as expected, the best results. When we compare *MOHGA*$(\prec^1)$ with *MOGA*$(\prec^1)$ and *MOHGA*$(\prec)$, a drastic performance improvement of the GA is observed due to the inclusion of the heuristics initialization and preferred dominance operator $\prec^1$. *MOHGA*$(\prec^1)$ also outperforms both *NetDelay+DL* and *Cluster+DL* heuristics, which are used at its initialization. Furthermore, the multi-objective *MOHGA*$(\prec^1)$ has similar results obtained by only optimizing the deadline violation in *SOHGA*(f_{dv}) because the dominance operator $\prec^1$ prioritizes time-sensitive applica-

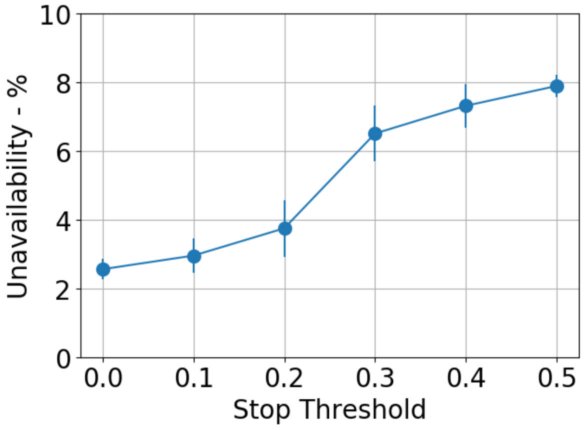
Figure 21 – Performance of different stopping criteria threshold $\hat{I}_{mdr}^{min}$ values



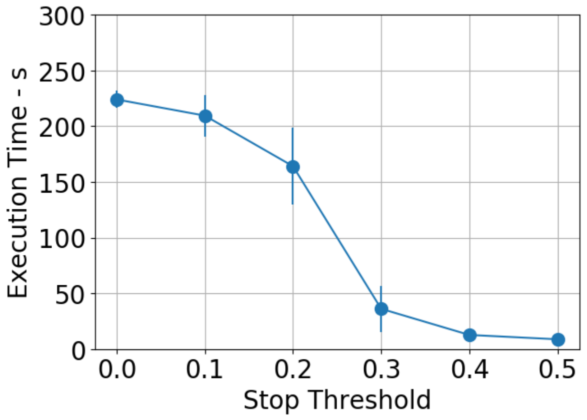
(a) Max. Deadline Violation



(b) Overall Operational Cost



(c) Avg. Service Unavailability



(d) Avg. Execution Time

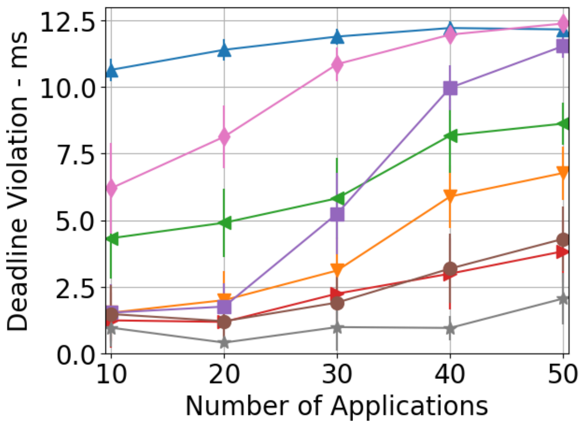
Source: Author.

tions with strict deadline requirements. Both *MOHGA*($\prec^1$) and *SOHGA*(f_{dv}) perform near the optimum results of *MILP*.

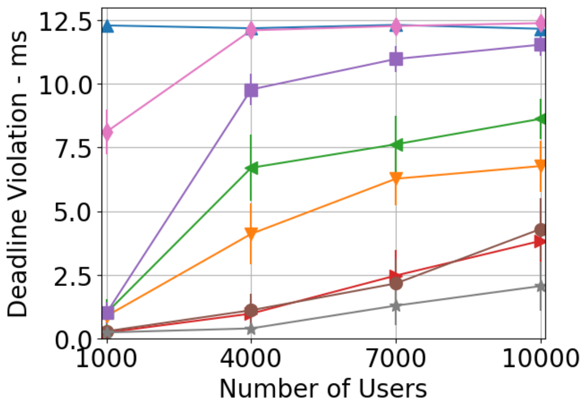
The growth in users number also affects the demand of node resources and, consequently, the level rise of deadline violation, as shown in Figure 22b. *Cloud* heuristic is an exception in this figure with constant results due to the unlimited capacity of the cloud node. Besides that, the growth slope is less steep for *MOHGA*($\prec$), *MOGA*($\prec^1$), *Cluster+DL*, and *NetDelay+DL* algorithms when the number of users is greater than $4,000$ due to surplus requests being processed in the cloud node. The other algorithms, *MILP*, *SOHGA*(f_{dv}), and *MOHGA*($\prec^1$), tend to have a linear growth behavior presumably because resource demand also increases linearly with the number of users in the tested scenario.

On the other hand, more nodes mean more resources available to meet application requirements. Figure 22c shows the deadline violation performance of the algorithms by varying the number of base stations with 50 applications and $10,000$ users. *Cluster+DL*, *NetDelay+DL*,

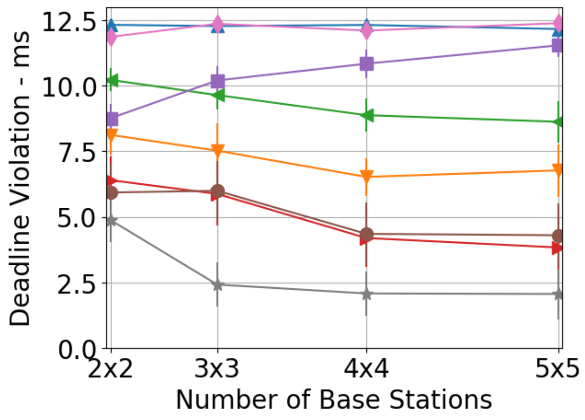
Figure 22 – Maximum deadline violation



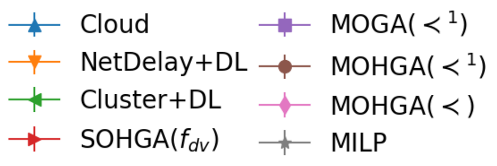
(a) 10,000 users and 5x5 BSs



(b) 50 apps and 5x5 BSs



(c) 10,000 users and 50 apps



(d) Legend

Source: Author.

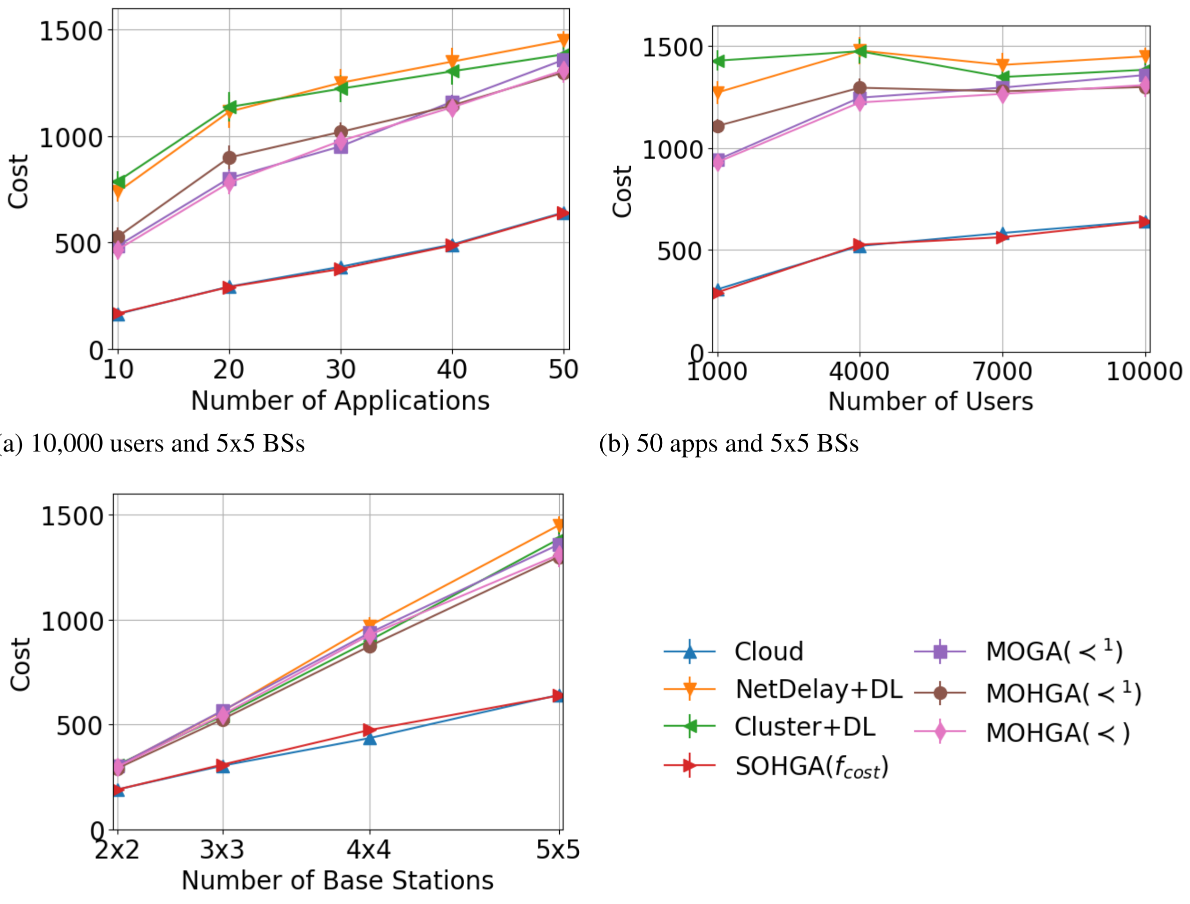
MILP, *SOHGA*(f_{dv}), and *MOHGA*($\prec^1$) exhibit violation decrease by adding more nodes, while *Cloud* and *MOHGA*($\prec$) have similar results in all tested variations. However, *MOGA*($\prec^1$) produces worse results with more base stations, possibly due to many possible chromosome combinations.

4.4.5.2 Overall Operational Cost

Figure 23 presents the performance of the compared algorithms according to cost function f_{cost}. In the tested scenarios, the optimum solution to minimize the overall operational cost is to place only one replica of each application in the cloud node, which has the cheapest resources. Thus, *Cloud* heuristic is the optimum solution in this case. As solution *SOHGA*(f_{cost}) includes the *Cloud* heuristic and only optimizes the cost function f_{cost}, it also achieves an optimal solution. In addition, the multi-objective algorithms *MOHGA*($\prec^1$), *MOHGA*($\prec$), and *MOGA*($\prec^1$) have slightly better results than *NetDelay+DL* and *Cluster+DL* by varying the number of applications and users, as shown in Figures 23a and 23b, respectively.

In Figure 23c, all algorithms present a linear cost increase by varying the number of nodes but with a fixed number of users and applications. This behavior is explained by how we estimate the number of requests generated. More specifically, we assume that at least one request comes from a node with users attached by defining the number of requests as $Q_{a,m} = \lceil |\mathscr{U}_{a,m}| A_a^{\mathrm{req}} \rceil$ in Section 4.2.1.2. Moreover, users will be more distributed in the test scenario by increasing the number of BSs. Consequently, there are more nodes with attached users and, thus, more requests are generated even if the number of users and applications does not change. As a result, increasing the number of requests implies more resource consumption and higher operational costs.

Figure 23 – Overall operational cost



(a) 10,000 users and 5x5 BSs

(b) 50 apps and 5x5 BSs

(c) 10,000 users and 50 apps

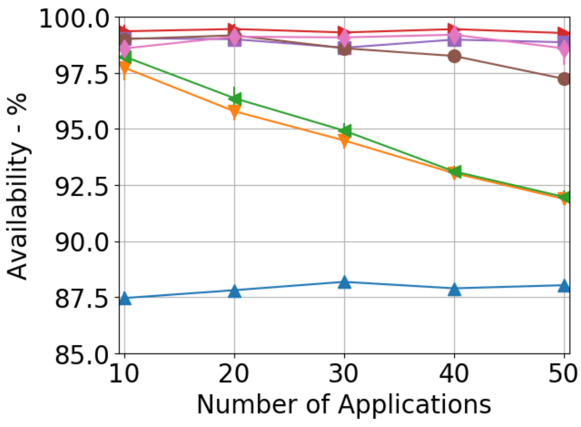
(d) Legend

Source: Author.

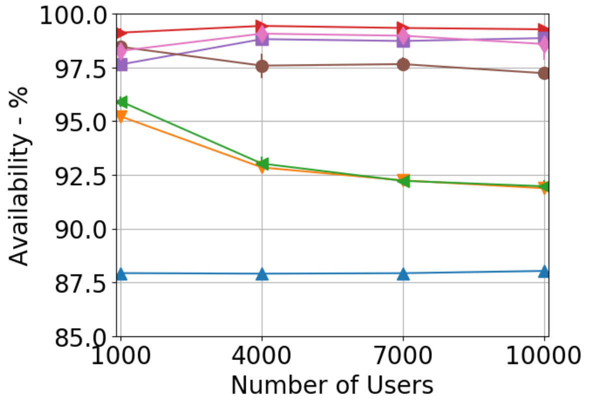
4.4.5.3 *Average Service Availability*

Figures 24a, 24b and 24c relate the impact of service availability or unavailability with the variation of the number of applications, users, and nodes, respectively. In these figures, the *Cloud* heuristic has the worst results due to the placement of a single replica for each application. All compared genetic algorithms outperform *NetDelay+DL* and *Cluster+DL*. Moreover, *MOHGA*($\prec^1$) has slightly lower availability than other GAs in Figures 24a and 24b. Meanwhile, the performance difference between *MOHGA*($\prec^1$) and the other GAs is most notable in Figure 24c. This performance degradation may be explained by the trade-off of solution *MOHGA*($\prec^1$) having less deadline violation.

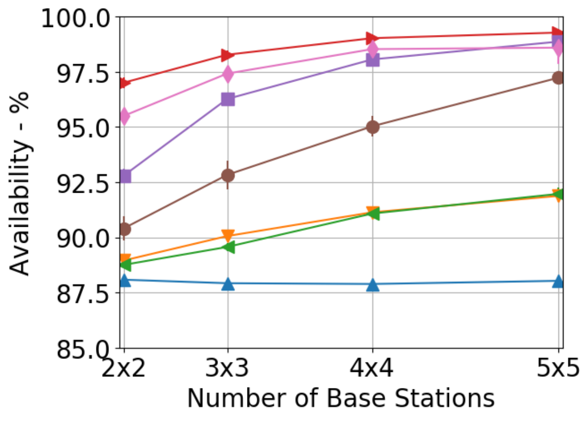
Figure 24 – Average service availability



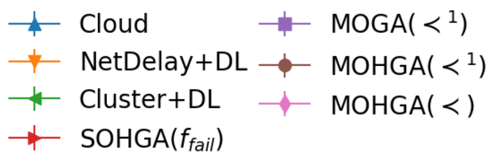
(a) 10,000 users and 5x5 BSs



(b) 50 apps and 5x5 BSs



(c) 10,000 users and 50 apps



(d) Legend

Source: Author.

In Figures 24a and 24b, we observe a considerable loss of availability for *NetDelay+DL* and *Cluster+DL* heuristic by increasing the number of applications or users on a grid of 5x5 BSs. As these heuristics may prioritize the same EC nodes for applications with some similar characteristics (e.g., network latency and user distribution), so there is more competition for these nodes by increasing the number of applications or users. As a result, *NetDelay+DL* and *Cluster+DL* deploy fewer application replicas resulting in decreased service availability. Meanwhile, the other compared algorithms do not have large variations in their performance, which may be explained by the use of the cloud node to receive the increased resource demand.

Figure 24c shows that the majority of compared algorithms improve their performance by adding more hosting nodes in the system. As the number of nodes grows, it is possible to place more application replicas on the system and, thus, increasing service availability. *Cloud* is the only algorithm without improvement as the number of replicas deployed per application is fixed.

4.5 Summary

In this chapter, we addressed the research question RQ1: *"How to make service placement and load distribution decisions to deploy multiple IoT applications or services in an EC infrastructure according to certain infrastructure constraints, application requirements, and performance criteria?"*. In order to answer this question, we jointly formulated the static service placement and load distribution as an optimization problem by considering diverse application characteristics (e.g., response deadline, resource demand, scalability, and availability). The formulated problem aims to minimize SLA infringements caused by violations of the deadline requirement and as well optimize other possibly conflicting objectives (e.g., operational cost and service availability). Then, we proposed a multi-objective genetic algorithm based on BRKGA and NSGA-II to obtain feasible solutions close to the Pareto optimal front. We also modified the Pareto dominance operator to prioritize applications with strict deadline requirements. Furthermore, we included heuristic solutions during the initialization of the proposed genetic algorithm to improve its solution results.

We analyzed the efficiency of the proposed algorithm through simulations. Our experimental results show that the proposed multi-objective GA achieves values close to the optimum of the MILP formulation in terms of deadline violation, and still generally outperforms the benchmark heuristics for the other analyzed objectives (operational cost and service availability). Moreover, we observed that the results of the proposed GA are related to the deadline prioritization and the heuristic initialization.

In the next chapter, we investigate the dynamic (or online) service placement and load distribution problems, which includes application migration, and dynamic incoming load due to user mobility or time-varying workload pattern.

5 A DYNAMIC AND CENTRALIZED APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION

In this chapter, we extend the static service placement and load distribution problem of the previous chapter by taking into account that application loads might vary in both spatial and temporal domains due to, for instance, user mobility and workload patterns. Consequently, a decision-making process needs to adjust the placement and distribution decisions in order to handle load fluctuations. Moreover, this decision-making process should consider the benefits and costs of decision adjustments. For instance, a readjusted decision can be the migration of an application to a new location to keep a short response time. However, a migration operation takes time to be completed, which may negatively affect the application response time.

Therefore, in this chapter, we design a centralized controller responsible for dynamic service placement and load distribution decisions. The designed controller adopts a proactive approach that prepares the Edge Computing (EC) system in advance for predicted load fluctuations. The main contributions of this chapter are the following:

– We include support for dynamic request loads of end-user devices in the Edge Computing system modeled in Chapter 4. In this way, it is possible to migrate applications and redistribute load over time.
– Using the Limited Look-ahead Control (LLC) concept presented in Chapter 2, we estimate how the modeled EC system evolves under controllable (placement and distribution) decisions and uncontrollable (user-generated load) events. Then, we jointly formulate the service placement and load distribution as an optimization problem of multiple performance criteria over a look-ahead prediction horizon while satisfying a set of constraints.
– We use a genetic algorithm to solve the formulated problem in a discrete prediction horizon with length H. We first propose a chromosome representation and a decoder algorithm to obtain a single and valid control decision when $H = 1$. Then, two heuristics extending this genetic solution are proposed to solve the problem when $H > 1$. The first heuristic generates simple sequences of control decisions over the prediction window, while the second one produces more general control sequences.

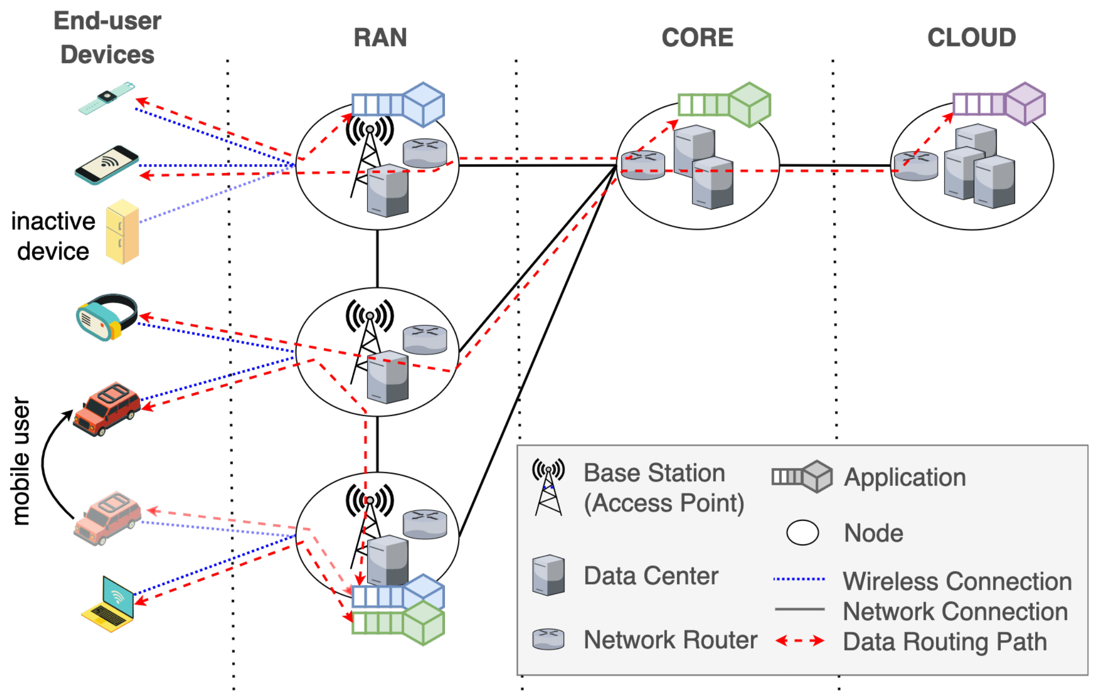
The remainder of this chapter is structured as follows. Section 5.1 presents the EC system model with dynamic loads. In Section 5.2, we formulate the dynamic service placement and load distribution problem using the LLC concept. Then, Section 5.3 presents our proposed algorithms to solve the formulated problem. Next, we conduct performance evaluations of the proposed algorithms in Section 5.4. Finally, Section 5.5 concludes this chapter.

5.1 System Model

Based on the system model presented in Chapter 4, we consider a dynamic Edge Computing system consisting of an Infrastructure Provider (InP), multiple Application Service Providers (ASPs), and several end-user devices. In this way, the InP offers on-demand different types of (virtual) resources to deploy applications of the ASPs within the EC infrastructure. Then, end-user devices attached to (wireless) access points requests services of the deployed applications. Furthermore, the InP is responsible for resource allocation decisions, such as application (service) placement and load distribution.

Unlike the system model in Chapter 4, users may move or be inactive without sending requests in the dynamic system, thereby changing the number of active devices attached to each access point node over time. As a result, the overall request load generated by users can vary in the spatial and temporal domains. Figure 25 illustrates the EC system model in a cellular network where users can be mobile or inactive. Moreover, applications can be deployed on different network regions, and requests from end-user devices are routed over the network to a node hosting the requested application.

Figure 25 – Proposed Edge Computing system with mobile and inactive devices



Source: Author.

In the next subsections, we describe the main features of the infrastructure, application, and user models for an EC system with dynamic application loads. In addition, Table 9 presents the main notations used in this chapter.

Table 9 – Main notations of the dynamic service placement problem

Symbol	**Description**
System Model	
$\mathscr{V},\mathscr{E},\mathscr{R},\mathscr{A},\mathscr{U}_t$	Set of nodes, links, resource types, applications, and users, respectively
$L_l^{\text{bw}},L_l^{\text{pd}}$	Bandwidth and propagation delay of link l, respectively
$N_{n,r}^{\text{cap}}$	Total capacity of resource r on node n
$N_{n,r}^{\text{cost}}(x)$	Cost of allocating a specific amount x of resource r on node n
A_a^{rd}	Response deadline of app a
A_a^{max}	Maximum number of replicas for app a
$A_a^r(\lambda)$	Demand of resource r for a replica of app a with workload λ
A_a^{work}	CPU work size of a request for app a
A_a^{req}	Request generation rate for app a
A_a^{data}	Request data length of app a
$u_{a,n}(t)$	Number of active users connected to node n requesting app a at time t
System Dynamics	
$s(t)$, $e(t)$, $c(t)$	System state, environment input, and control input at time step t, respectively
$\rho_{a,n}(t)$	Whether or not a replica of app a should be placed on node n at time t
$\delta_{a,m,n}(t)$	Fraction of requests for app a that should be distributed from node m to node n at time t
$Q_{a,n}(t)$	Request generation rate from users of app a attached to node n at time t
$d_{a,m,n}(t)$	Response time of requests for app a from node m to n at time t
$q_{a,n}(t)$	Number of requests waiting in a queue of app a placed on node n at time t
Problem Formulation	
$\Lambda_{a,n}(t)$, $\lambda_{a,n}(t)$	Request arrival rate in app a placed on node n before and after load distribution at time t
$\mu_{a,n}^i(t)$	Processing rate in app a placed on node n at time t
$A_a^{\text{D+R}}(\lambda)$	Content size of app a with workload λ
T_s	Time step duration

Source: Author.

5.1.1 Infrastructure Model

Similar to the system model in Chapter 4, graph $G = (\mathscr{V},\mathscr{E})$ represents the EC infrastructure composed of EC nodes and network links connecting them. We also assume that the infrastructure graph and its properties are immutable over time. We justify this assumption by focusing on networks with fixed infrastructure, such as 5G networks, where their properties do not change, or possible changes occur less frequently than a dynamic application load. For instance, an application may have a day/night workload cycle pattern, whereas a cellular network topology may remain unchanged for months.

In graph G, each link $l = (m, n) \in \mathscr{E}$ corresponds to a network connection between two nodes (m and n), and it has the following properties:

- **Bandwidth** L_l^{bw} is the average transmission rate between end-points of link l.
- **Propagation Delay** L_l^{pd} is the time required for bits to reach the other end of link l.

We use the above two properties instead of the one defined in Section 4.1.1 as there are now two types of traffic in the system: (*i*) request traffic and (*ii*) application migration traffic. As a result, the transmission delay for a request or application migration can be estimated using the bandwidth and propagation delay properties.

Regarding a node $n \in \mathscr{V}$, it has a total capacity of $N_{n,r}^{\text{cap}}$ for a resource $r \in \mathscr{R}$, where $\mathscr{R}$ is the set of all resource types provided by the InP. We still assume that the cloud node has unlimited resource capacity. Moreover, $N_{n,r}^{\text{cost}}(x)$ is the (monetary) cost function of allocating a specific amount x of resource $r \in \mathscr{R}$ to an application on the node n. Note that it is possible to include other node attributes, for instance, when required by an optimization metric.

5.1.2 Application Model

Given $\mathscr{A}$ as the set of all applications to be placed in the system, an application $a \in \mathscr{A}$ has the response deadline A_a^{rd}, maximum number of replicas $A_a^{\max}$, resource demand function $A_a^r(\cdot)$, CPU work size A_a^{work}, and request rate A_a^{req} properties defined in Section 4.1.2. Also, let A_a^{data} be the request data length, i.e., the amount of data (in bits or bytes) in a request for the application a sent over the network.

For the sake of simplicity, we consider that set $\mathscr{A}$ and the above-mentioned application properties do not change over time. For instance, this set can include all applications that need to be deployed in the system over an experiment. Nonetheless, as we are interested in the system behavior under a dynamic application load, this dynamism is expressed in the user model.

5.1.3 User Model

Let $\mathscr{U}_t$ be the set of all end-user devices at a particular time t, then a user $u \in \mathscr{U}_t$ has the following characteristics:

- **Requested Application** U_u^{app}. We consider that a user sends requests for the same application at any time, i.e., U_u^{app} is a constant.
- **Attached Node** $U_u^{\text{node}}(t)$. It denotes the node acting as an access point where the user is attached at time t. As a user can be mobile in the dynamic approach, the attached node may change over time.
- **User Status** $U_u^{\text{on}}(t) \in \{0,1\}$. A user can be either active or inactive during a time interval. In an inactive status ($U_u^{\text{on}}(t) = 0$), the user does not send requests to an application. On the other hand, an active user ($U_u^{\text{on}}(t) = 1$) sends requests according to the request rate defined in the application model. This status parameter can also vary over time.

Given the above-mentioned user characteristics, Equation (5.1) defines the total number of active users connected to node n requesting application a at a time t. In other words, this equation specifies the load variation of an application in both spatial and temporal domains in our system model.

$$u_{a,n}(t) = \left| \left\{ u \in \mathscr{U}_t \,\middle|\, U_u^{\text{app}} = a \text{ and } U_u^{\text{node}}(t) = n \text{ and } U_u^{\text{on}}(t) = 1 \right\} \right| \tag{5.1}$$

5.2 Problem Statement and Formulation

As an application load might vary in spatial and temporal domains, the service placement and load distribution decisions should be re-evaluated to maintain satisfactory system performance. The re-evaluated decision may involve redistributing the load or even migrating an application replica to a new location in order to, for example, keep low latency/response time. However, decision adjustments could also lead to additional operational or performance costs. For instance, excessive reallocation, especially for large applications, may result in network overload or even latency degradation due to the migration operation itself (YU *et al.*, 2019; GAO *et al.*, 2019). Therefore, a dynamic approach for service placement should consider the benefits and costs of service migrations.

Given the above context, we intend to mitigate the burden of service migration operations on system performance by adopting a proactive approach that prepares the system for load fluctuations in advance. For example, a proactive decision can be to pre-deploy an application to avoid the impact of migration delay in future application response time.

Therefore, we adopt the Limited Look-ahead Control concept introduced in Section 2.5 to define a proactive approach for service placement and load distribution in an EC system with dynamic application loads. The design of an LLC approach has the following main components: (*i*) a discrete-time model capturing the system behavior under controllable actions and uncontrollable events, (*ii*) the optimization problem to be solved at any time step, and (*iii*) a strategy to solve the formulated problem. We present the dynamic model and problem formulation in Subsections 5.2.1 and 5.2.2, respectively. Then, Section 5.3 presents our genetic-based algorithms to solve the formulated problem.

5.2.1 Modeling the System Dynamics

Following the LLC concept, the dynamics of our EC system are described by the following discrete-time state-space equation:

$$s(t+1) = \Phi(s(t), c(t), e(t)) \tag{5.2}$$

Here, the behavioral model $\Phi(\cdot)$ captures the relationship between the system state $s(t)$ and the inputs (control input $c(t)$ and environment input $e(t)$) that adjust the state parameters at any discrete time step $t \geq 0$. Moreover, a centralized limited look-ahead controller continuously monitors the current state and environment input of the entire system. Then, the controller uses the observation results and the behavioral model to re-adjust the controllable system parameters in order to optimize the forecast system behavior over a limited look-ahead prediction horizon with length $H > 0$.

An important observation is that a centralized controller means a logically single entity responsible for control decisions. In practice, it is possible to add redundancy to this entity in order to mitigate failure issues.

Since we are interested in the service placement and load distribution decisions, we design the following adjustable system parameters as the control input $c(t) = (\rho(t), \delta(t))$:

- **Application Placement** $\rho(t) = \{\rho_{a,n}(t) \mid a \in \mathscr{A} \text{ and } n \in \mathscr{V}\}$ is a set of binary variables, where $\rho_{a,n}(t) \in \{0,1\}$ indicates whether or not a replica of application a should be placed on a node n at time step t.
- **Load Distribution** $\delta(t) = \{\delta_{a,m,n}(t) \mid a \in \mathscr{A} \text{ and } m,n \in \mathscr{V}\}$ is a set of real variables, where $\delta_{a,m,n}(t) \in [0,1]$ establishes the fraction of requests for an application a that should be distributed from a node m to another node n at time t.

Note that the above system parameters as control input are similar to the decision variables for the static service placement approach in Chapter 3. However, unlike the static approach, the decision variables can vary over time in the dynamic approach. Moreover, we use real variables in the $[0,1]$ interval for load distribution instead of integer ones in the static approach to have an upper bound limit when dealing with dynamic loads.

Regarding the environment input as uncontrollable events that also affect the system state, we define this input $e(t) = (Q(t))$ as follows:

- **User-Generated Request Rate** $Q(t) = \{Q_{a,n}(t) \mid a \in \mathscr{A} \text{ and } n \in \mathscr{V}\}$. Here, $Q_{a,n}(t)$ is the observed request generation rate from all users of an application a attached to a node n at time step t.

At the current time t, the environment input value can be inferred from the observed number of active users and the application request rate, i.e., $Q_{a,n}(t) = A_a^{\text{req}} u_{a,n}(t)$. However, the actual value for an environment input within the prediction horizon cannot be obtained until the next measuring samples. Hence, a forecasting technique (e.g., ARIMA) can be applied to predict the environment input, expressed as $\widehat{Q}_{a,n}(k)$, for each time step $k > t$ along the prediction horizon H. Moreover, it is possible to use different forecasting techniques for different applications, but the adequate selection of a forecasting technique is out of the scope of this thesis.

In order to observe the impact of control and environment inputs on the system state, let us define a dynamic request flow as follows:

Definition 5.1 (Dynamic Request Flow) *A request flow $\mathscr{F}_{a,m,n}(t)$ is the requests rate for application a from node m (source node) being handled by a replica of application a on node n (target node) during time step t.*

Furthermore, as the application response time is a relevant performance-related metric to observe in Edge Computing, we define the system state $s(t) = (d(t), q(t))$ at time t as follows:

- **Response Time** $d(t) = \{d_{a,m,n}(t) \mid a \in \mathscr{A} \text{ and } m,n \in \mathscr{V}\}$ is the average response time of each request flow $\mathscr{F}_{a,m,n}(t)$ at the beginning of time step t.
- **Queue Length** $q(t) = \{q_{a,n}(t) \mid a \in \mathscr{A} \text{ and } n \in \mathscr{V}\}$ is the number of requests waiting in a queue to be processed on each application replica deployed on the system at the beginning of time step t.

Based on the above system parameters, we develop the behavioral model $\Phi(\cdot)$ in the remainder of this subsection.

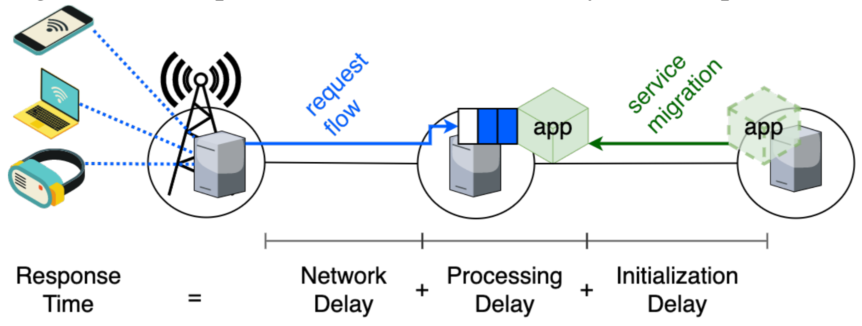
5.2.1.1 System State Estimation

Given a system state $s(k)$ and environment input $e(k)$ at time $k \in [t, t+H-1]$ within the prediction horizon H starting at the current time t, the behavioral model $\Phi(\cdot)$ determines the next system state $s(k+1)$ when a control input $c(k)$ is applied in the system at time step k. Thus, we need to estimate the response time $d(k+1)$ and queue length $q(k+1)$ of system state $s(k+1)$.

As shown in Figure 26, the response time estimation includes the initialization delay in addition to the network and processing delays already considered in the static approach. This initialization delay is related to the time to place an application replica in a new location during a time step. Thus, Equation (5.3) formally expresses the response time of a request flow $\mathscr{F}_{a,m,n}(k+1)$ where $d^{\text{net}}_{a,m,n}(k+1)$, $d^{\text{proc}}_{a,n}(k+1)$, and $d^{\text{init}}_{a,n}(k+1)$ are the network, processing, and initialization delays, respectively.

$$d_{a,m,n}(k+1) = d^{\text{net}}_{a,m,n}(k+1) + d^{\text{proc}}_{a,n}(k+1) + d^{\text{init}}_{a,n}(k+1) \tag{5.3}$$

Figure 26 – Response time estimation for a dynamic request flow



Source: Author.

Next, we specify the network, processing, and initialization delays for a request flow $\mathscr{F}_{a,m,n}(k+1)$.

Network Delay. Equation (5.4) denotes the average time to send requests over the network from the source node m to the target node n, where $\mathscr{P}_{m,n}$ contains the links in a routing path from m to n. Similar to the static approach, this path can be obtained a priori by some shortest routing path algorithm.

$$d^{\text{net}}_{a,m,n}(k+1) = \begin{cases} 0 & \text{if } m = n \\ \sum\limits_{l \in \mathscr{P}_{m,n}} \left(\dfrac{A^{\text{data}}_{a}}{L^{\text{bw}}_{l}} + L^{\text{pd}}_{l} \right) & \text{otherwise} \end{cases} \tag{5.4}$$

Processing Delay. Another similarity with the static approach is using an M/M/1 queue to model the requests processing. Besides that, the load to be distributed in the dynamic approach comprises the requests generated by users during time step k and requests waiting to be processed in the application queues. Thus, let $\Lambda_{a,n}(k)$ and $\lambda_{a,n}(k)$ be the request arrival rate before and after load distribution for application a in node n at time step k, respectively. In Equation (5.5), the arrival rate $\Lambda_{a,n}(k)$ is given by the predicted environment input $\widehat{Q}_{a,n}(k)$ plus the estimated queue length, which is converted to a rate value using T_s as the sampling period, i.e., the time step duration. Meanwhile, the control input $\delta(k)$ regulates the arrival rate $\lambda_{a,n}(k)$, as shown in Equation (5.6).

$$\Lambda_{a,n}(k) = \widehat{Q}_{a,n}(k) + \frac{q_{a,n}(k)}{T_s}\rho_{a,n}(k-1) \tag{5.5}$$

$$\lambda_{a,n}(k) = \sum_{m\in\mathscr{V}} \delta_{a,m,n}(k)\Lambda_{a,m}(k) \tag{5.6}$$

In Equation (5.7), the average processing rate $\mu_{a,n}(k)$ is determined by the CPU speed $A_a^{\text{CPU}}(\cdot)$ allocated to the application replica at time k, and the amount of CPU work A_a^{work} necessary to process a request for this application.

$$\frac{1}{\mu_{a,n}(k)} = \frac{A_a^{\text{work}}}{A_a^{\text{CPU}}(\lambda_{a,n}(k))} \tag{5.7}$$

According to the M/M/1 queueing model, Equation (5.8) and Equation (5.9) estimate the average queue length $q_{a,n}(k+1)$ and the average processing delay $d_{a,n}^{\text{proc}}(k+1)$, respectively.

$$q_{a,n}(k+1) = \frac{\lambda_{a,n}(k)}{\mu_{a,n}(k)-\lambda_{a,n}(k)} - \frac{\lambda_{a,n}(k)}{\mu_{a,n}(k)} \tag{5.8}$$

$$d_{a,n}^{\text{proc}}(k+1) = \frac{1}{\mu_{a,n}(k)-\lambda_{a,n}(k)} \tag{5.9}$$

Initialization Delay. In order to initialize the placement of an application on a selected node, an EC system migrates or replicates an instance of this application over the network from another node, called origin, already hosting it to the selected node. A migration process removes the application instance from the origin node hosting it, whereas a replication procedure keeps it in the origin node. Despite this difference, both migration and replication operations take the same time to complete and, thus, we use the migration term to indicate a migration or replication process. Furthermore, the application migration delay impacts the response time, as requests arriving at the selected node for this application need to wait for the migration conclusion before they can be processed. Hence, we determine this migration delay as the time to transfer the application content, which includes the storage (DISK) and memory

(RAM) volumes, from the nearest node hosting the application, as shown in Equation (5.10). Moreover, we disregard the migration delay at the first time step ($k = 0$) for the sake of simplicity.

$$d^{\text{mig}}_{a,n}(k) = \begin{cases} (1-\rho_{a,n}(k-1))\,\rho_{a,n}(k) \min\limits_{m\in\mathscr{V}} \left\{ d^{\text{mig}}_{a,m,n}(k)\rho_{a,m}(k-1) \right\} & \text{if } k > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.10a}$$

$$d^{\text{mig}}_{a,m,n}(k) = \sum_{l\in\mathscr{P}_{m,n}} \left(\frac{A^{\text{D+R}}_a\left(\lambda_{a,m}(k-1)\right)}{L^{\text{bw}}_l} + L^{\text{pd}}_l \right) \tag{5.10b}$$

$$A^{\text{D+R}}_a(\lambda) = A^{\text{DISK}}_a(\lambda) + A^{\text{RAM}}_a(\lambda) \tag{5.10c}$$

We assume that a migration process starts and finishes at the same time step. Consequently, not all requests and their response time are impacted by this process. Moreover, let us define an application initialization delay as the impact of a migration procedure in the response time. By following the Poisson arrival process in an M/M/1 queue, a node n receives $\lambda_{a,n}(k)\Delta t$ requests on average for an application a during a time interval Δt. We can split the migration delay into $M = \lceil d^{\text{mig}}_{a,m,n}(k)\rceil$ consecutive intervals of one unit of time (e.g., $\Delta t = 1$s). Then, during the i-th migration interval, $\lambda_{a,n}(k)$ requests arrive and wait for $M-i+1$ units of time until the migration is complete. We calculate $d^{\text{init}}_{a,n}(k+1)$ as a weighted average by using the arrived requests of each migration interval against all requests during sampling period T_s. We then approximate it by setting $M \approx d^{\text{mig}}_{a,m,n}(k)$, as shown in Equation (5.11).

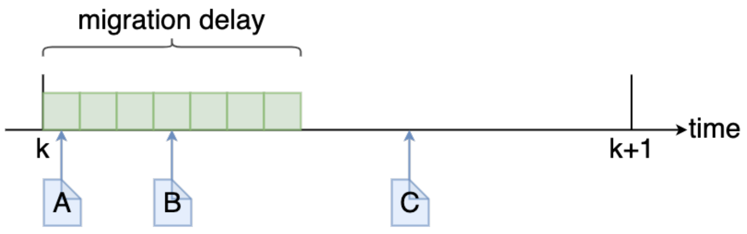
$$d^{\text{init}}_{a,n}(k+1) = \frac{\sum_{i=1}^{M} \lambda_{a,n}(k)\,(M-i+1)}{\lambda_{a,n}(k) T_s} \approx \frac{d^{\text{mig}}_{a,n}(k)\left(d^{\text{mig}}_{a,n}(k)+1\right)}{2T_s} \tag{5.11a}$$

$$M = \lceil d^{\text{mig}}_{a,n}(k)\rceil \tag{5.11b}$$

We exemplify how the migration delay may have different impacts on distinct requests in Figure 27. In this figure, a migration operation starts at the beginning of time step k, and it finishes at the same time step. In addition, three requests (A, B, and C) arrive at different time instants. Request A arrives at the beginning of the migration and waits for the entire migration procedure before being processed. Then, request B arrives in the middle of the migration and only waits half of the entire migration delay. After finishing the migration operation, request C arrives and is not impacted by this operation.

Once we specified the behavioral model $\Phi(\cdot)$ by Equations (5.3) to (5.11), we formulate the optimization problem that adjusts control decisions over time to improve system performance in the next subsection.

Figure 27 – Impact of a migration delay in different requests



Source: Author.

5.2.2 Optimization Formulation

Let $F = (f_1, \dots, f_i, \dots, f_M)$ be a list of M functions and $f_i(s(k+1), c(k), e(k))$ a function that associates some performance for reaching and maintaining a system state $s(k+1)$ given control input $c(k)$ and environment input $e(k)$ applied in the system. Then, at each time step t, the centralized limited load-ahead controller aims to optimize the following problem:

$$\min_{c(k)\in\mathscr{C}} \sum_{k=t}^{t+H-1} F\left(s(k+1), c(k), e(k)\right) \tag{5.12a}$$

$$\text{s.t.} \quad s(k+1) = \Phi\left(s(k), c(k), e(k)\right) \qquad \forall k \in [t, t+H) \tag{5.12b}$$

$$1 \le \sum_{n\in\mathscr{V}} \rho_{a,n}(k) \le A_a^{\max} \qquad \forall a \in \mathscr{A}, \forall k \in [t, t+H) \tag{5.12c}$$

$$\delta_{a,m,n}(k) \le \rho_{a,n}(k) \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V}, \forall k \in [t, t+H) \tag{5.12d}$$

$$\sum_{i\in\mathscr{V}} \delta_{a,n,i}(k)\Lambda_{a,n}(k) = \Lambda_{a,n}(k) \qquad \forall a \in \mathscr{A}, \forall n \in \mathscr{V}, \forall k \in [t, t+H) \tag{5.12e}$$

$$\sum_{a\in\mathscr{A}} \rho_{a,n}(k) A_a^r\left(\lambda_{a,n}(k)\right) \le N_{n,r}^{\text{cap}} \qquad \forall r \in \mathscr{R}, \forall n \in \mathscr{V}, \forall k \in [t, t+H) \tag{5.12f}$$

$$\lambda_{a,n}(k) < \mu_{a,n}(k) \qquad \forall a,n\left(\rho_{a,n}(k) = 1\right), \forall k \in [t, t+H) \tag{5.12g}$$

where $\mathscr{C}$ is the set of all possible control inputs, Equation (5.12a) is the optimization objective, and Equation (5.12b) estimates the next system state according to the behavioral model. Furthermore, a feasible solution for problem (5.12) is associated with a sequence of control decisions $\pi_c = \{c(k) \mid k \in [t, t+H-1]\}$ that satisfies all the following constraints within the prediction horizon. First, Equation (5.12c) constraints the allowed number of application replicas placed in the system. Then, Equation (5.12d) restricts load distribution to nodes that host the requested application, whereas Equation (5.12e) ensures the distribution of all loads. Finally, Constraint (5.12f) assures that the amount of resources allocated on a node does not exceed its capacity, and (5.12g) guarantees the processing queue stability.

After solving problem (5.12) and obtaining a control sequence π_c at the beginning of time step t, the controller only applies the first control input $c(t)$ in this sequence to the system.

This decision-making process is repeated at the next time step $t+1$ when the new measured system state $s(t+1)$ and environment input $e(t+1)$ are available. However, a control algorithm that evaluates all possible control inputs to solve problem (5.12) presents an exponential increase in worst-case complexity with an increasing number of control inputs and longer prediction horizons (ABDELWAHED *et al.*, 2004). Even for a single time step (i.e., $H = 1$), this problem can be seen as an MINLP problem, which is generally NP-Hard. Furthermore, not all control inputs in $\mathscr{C}$ produce a feasible solution satisfying all problem constraints. Therefore, it necessary to develop more efficient control algorithms, which is the focus of the next section.

5.3 Control Algorithms

In order to alleviate the complexity overhead in the centralized controller when solving problem (5.12), we propose two (meta-)heuristic algorithms that find sub-optimal solutions in a reasonable time and, thus, trading optimization for speed. First, we present a genetic-based algorithm that solves the problem for a single time step in Subsection 5.3.1. Then, we extend this algorithm by taking into account a prediction horizon $H > 1$ in Subsection 5.3.2.

5.3.1 One-Time Step Algorithm

In addition to the static service placement problem in Chapter 4, the genetic algorithm BRKGA+NSGA-II can also solve problem (5.12) when the prediction horizon $H = 1$. In BRKGA+NSGA-II, the chromosome representation and decoder need to be specified for a particular optimization problem as the problem-dependent part of the meta-heuristic. However, the chromosome representation and decoder algorithm for the static service placement cannot be directly employed in the dynamic approach. More specifically, the chromosome vector length presented in Section 4.3.1 depends on the number of requests. Consequently, the same vector cannot be used to build a feasible solution at any time step because the number of requests varies along time in the dynamic case. Moreover, the decoder algorithm for the static approach only places an application replica on a node if this replica receives some non-empty load. Nevertheless, we would like to support the pre-deployment of an application to a node with no incoming load at the current time, but which will soon receive requests.

5.3.1.1 Chromosome Representation

To address the above-mentioned issues, we need to design a new chromosome representation and decoder algorithm to (*i*) produce feasible control inputs at any single time step within the prediction horizon and (*ii*) support pre-deployment operations. Therefore, the following one-time step chromosome encode is proposed:

$$\begin{aligned} C = \Big[& C^{\mathrm{I}}_{1}, C^{\mathrm{I}}_{2}, \ldots, C^{\mathrm{I}}_{|\mathscr{A}|}, \\ & C^{\mathrm{II}}_{1,1}, C^{\mathrm{II}}_{1,2}, \ldots, C^{\mathrm{II}}_{1,|\mathscr{V}|}, \ldots, C^{\mathrm{II}}_{|\mathscr{A}|,1}, C^{\mathrm{II}}_{|\mathscr{A}|,2}, \ldots, C^{\mathrm{II}}_{|\mathscr{A}|,|\mathscr{V}|}, \\ & C^{\mathrm{III}}_{1,1}, C^{\mathrm{III}}_{1,2}, \ldots, C^{\mathrm{III}}_{1,|\mathscr{V}|}, \ldots, C^{\mathrm{III}}_{|\mathscr{A}|,1}, C^{\mathrm{III}}_{|\mathscr{A}|,2}, \ldots, C^{\mathrm{III}}_{|\mathscr{A}|,|\mathscr{V}|} \Big] \end{aligned}$$

where each part of this representation is described below:

1. C^{I}_{a} is the fraction of nodes to be candidates for hosting application a.
2. $C^{\mathrm{II}}_{a,n}$ is the priority to place application a in node n.
3. $C^{\mathrm{III}}_{a,m}$ is the priority to distribute requests for application a from a (source) node m.

Compared to the chromosome representation in Section 4.3.1, the above chromosome has its third part (C^{III}) based on the application and node sets instead of request numbers. Furthermore, as we assume that the numbers of applications and nodes do not change over time, the length of the one-time step chromosome representation is the same for any time step.

5.3.1.2 Chromosome Decoder

Algorithm 3 decodes a chromosome with the above representation into a valid control input. This algorithm operates in three steps: (*i*) nodes selection, (*ii*) load distribution, and (*iii*) local search. First, it selects nodes as potential placement locations for each application (lines 3 to 5). For this, the first part of the chromosome (C^{I}) delimits the number of nodes to be selected on line 4. Then, the next line selects nodes with high values in the second part of the chromosome (C^{II}) as host candidates.

The second step (lines 6 to 24) of Algorithm 3 is related to load distribution. It first orders all possible load sources according to the third part of the chromosome (C^{III}) on line 6. Following this order, it distributes the total loads $\Lambda_{a,m}(k)$ of a source in chunks λ^* among the nodes selected in the first step plus the cloud node. The cloud node addition ensures there are enough resources to deploy at least one replica of each application. Note that the chunk size $\lambda^* = \Lambda_{a,m}(k)\lambda_{\%}$ can be an algorithm input by setting parameter $\lambda_{\%} \in (0,1]$. Line 13 orders the selected nodes by response time estimated in Equation (5.3). Then, while there are still loads to be dispatched, the decoder searches in the sorted nodes for one with sufficient resources to receive an additional chunk of load. When the first envisioned target node is found on line 17, it sets to place a replica of the requested application on this node and assigns an additional chunk to the replica.

In the last step, Algorithm 3 performs a local search around the placement decision from line 25 to 32. If the number of replicas deployed by previous steps exceeds the maximum allowed, it replaces surplus replicas with one on the cloud node. Otherwise, the decoder tries to place an application replica on each node selected by the first algorithm step. This former case allows the pre-deployment of an application that may be requested in the next time steps, avoiding future migration burden on response time.

Complexity Analysis. Let $A = |\mathscr{A}|, V = |\mathscr{V}|, R = |\mathscr{R}|$, and $L = \lceil 1/\lambda_{\%} \rceil$. The complexity of Algorithm 3 is upper bounded by its second step (lines 6 to 24). Due to sorting procedures, lines 6 and 13 have complexity $O(AV \log AV)$ and $O(V \log V)$, respectively. By assuming that the constraints checking on line 17 is $O(R)$, the inner loop between lines 14 and 24 is $O(LVR)$. Then, the overall complexity of Algorithm 3 is $O(AV(\log A + V \log V + LVR))$, which is a polynomial complexity.

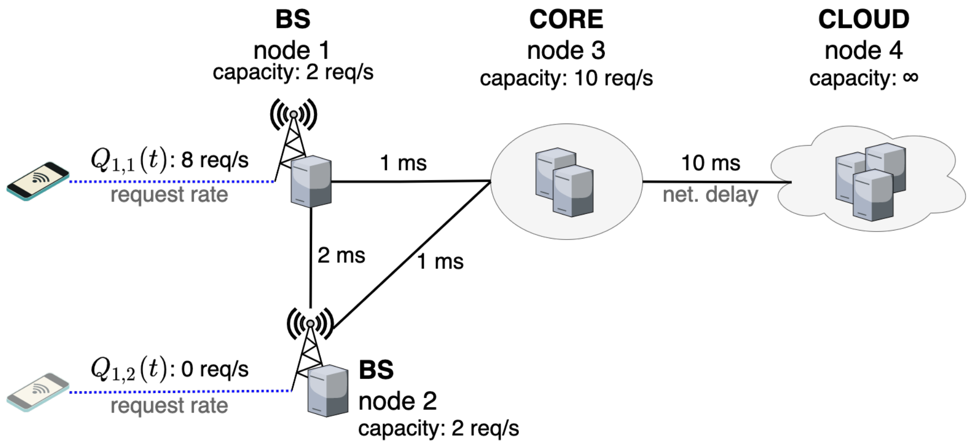
In order to exemplify how Algorithm 3 works, Figure 28a presents a simple scenario composed of 4 nodes and one application. This figure also shows the values considered for the network delay, node capacity, and user-generated request rate parameters. Moreover, let us set $A_1^{\max} = 4$ and $A_1^r(\lambda) = \lambda$ for application 1. For time step $t = 0$, Algorithm 3 decodes the chromosome vector in Figure 28b as follows. In the first step, the decoder selects $\lceil C_a^{\mathrm{I}} A_a^{\max} \rceil = \lceil 0.75 \times 4 \rceil = 3$ nodes as candidates to host application 1, which are nodes 1, 2, and 3 because they have the highest values in the second vector part. According to the third vector part, requests from users attached to node 1 are distributed first. By setting $\lambda_{\%} = 0.25$, these requests are distributed in 4 chunks of $\Lambda_{1,1}(t)\lambda_{\%} = 8 \times 0.25 = 2$ req/s, where $\Lambda_{1,1}(t) = Q_{1,1}(t) = 8$ req/s at $t = 0$. Then, Algorithm 3 first tries to assign these chunks to node 1 because it has the shortest response time among the selected nodes to the source node, which is the node 1 itself. Only one chunk of 2 req/s is actually assigned to node 1 due to its resource capacity. The remaining chunks are then assigned to node 3 because it has the second shortest response time/network delay and enough capacity to receive these chunks. As a consequence of these distributions, the control input is set to $\rho_{1,1}(t) = 1$, $\rho_{1,3}(t) = 1$, $\delta_{1,1,1}(t) = 1/4$, and $\delta_{1,1,3}(t) = 3/4$. In the third step, the decoder determines the pre-deployment of an application replica in node 2 because this

Algorithm 3: Chromosome decoder for the dynamic service placement problem with a single time step

```
Data: individual, $s(k)$, $e(k)$, $\lambda_{\%} \leftarrow 0.25$
Result: Control input $c(k) = (\rho(k), \delta(k))$
1  initialize $\rho_{a,n}(k)$, $\delta_{a,m,n}(k)$, $\lambda_{a,n}(k) \leftarrow 0$;
2  $C^{\mathrm{I}}, C^{\mathrm{II}}, C^{\mathrm{III}} \leftarrow$ individual.chromosome;
   /* Step I: Nodes Selection                                   */
3  forall $a \in \mathscr{A}$ do
4      $z \leftarrow \min(|\mathscr{V}|, \lceil C^{\mathrm{I}}_a A^{\max}_a \rceil)$;
5      $V_a \leftarrow$ select $z$ nodes with higher $C^{\mathrm{II}}_{a,n}$, $n \in \mathscr{V}$;
   /* Step II: Load Distribution                                */
6  $L \leftarrow$ list of pairs $(a,m) \in \mathscr{A} \times \mathscr{V}$ sorted by $C^{\mathrm{III}}_{a,m}$ in descending order;
7  forall $(a,m) \in L$ do
8      $r \leftarrow \Lambda_{a,m}(k)$;              // remaining load to be distributed
9      $\lambda^* \leftarrow \Lambda_{a,m}(k)\lambda_{\%}$;                           // load chunk
10     $V'_a \leftarrow V_a \cup \{cloud\}$;
11     forall $n \in V'_a$ do
12         $d_{a,m,n}(k+1) \leftarrow$ by Equation (5.3) and $c(k) = (\rho(k) = \{1\}, \delta(k))$;
13     sort nodes $n \in V'_a$ by $d_{a,m,n}(k+1)$ in ascending order;
14     while $r > 0$ do
15         forall $n \in V'_a$ do
16             $l \leftarrow \lambda_{a,n}(k) + \lambda^*$;
17             if $(\rho_{a,n}(k) = 1, \lambda_{a,n}(k) = l)$ respects constraints (5.12f) and (5.12g) then
18                 $\rho_{a,n}(k) \leftarrow 1$;
19                 $\lambda_{a,n}(k) \leftarrow l$;
20                 $\delta_{a,m,n}(k) \leftarrow \delta_{a,m,n}(k) + \lambda^*/\Lambda_{a,m}(k)$;
21                 $r \leftarrow r - \lambda^*$;
22                 $\lambda^* \leftarrow \min\{r, \lambda^*\}$;
23                 update free resources on node $n$ given $c(k) = (\rho(k), \delta(k))$;
24                 break;
   /* Step III: Local Search                                    */
25 forall $a \in \mathscr{A}$ do
26     $z \leftarrow A^{\max}_a - \sum_{n \in \mathscr{V}} \rho_{a,n}(k)$;
27     if $z > 0$ then                                          // Surplus nodes
28         replace $z+1$ replicas of $a$ with the cloud node;
29     else
30         forall $n \in V_a$ do                                // Pre-deployment
31             if $(\rho_{a,n}(k) = 1, \lambda_{a,n}(k))$ respects constraints (5.12f) and (5.12g) then
32                 $\rho_{a,n}(k) \leftarrow 1$;
```

node was selected in the algorithm first step. Finally, Figure 28c presents the decoded control input for the chromosome vector.

Figure 28 – Decoding example for Algorithm 3



(a) System scenario

C^{I}_{1}	$C^{\mathrm{II}}_{1,1}$	$C^{\mathrm{II}}_{1,2}$	$C^{\mathrm{II}}_{1,3}$	$C^{\mathrm{II}}_{1,4}$	$C^{\mathrm{III}}_{1,1}$	$C^{\mathrm{III}}_{1,2}$	$C^{\mathrm{III}}_{1,3}$	$C^{\mathrm{III}}_{1,4}$
0.75	0.9	0.4	0.7	0.1	0.8	0.2	0.5	0.6

(b) Chromosome as an input parameter

$\rho_{1,1}(t)$	$\rho_{1,2}(t)$	$\rho_{1,3}(t)$	$\rho_{1,4}(t)$	$\delta_{1,1,1}(t)$	$\delta_{1,1,2}(t)$	$\delta_{1,1,3}(t)$	$\delta_{1,1,4}(t)$	$\delta_{1,n,m}(t), n>1$
1	1	1	0	0.25	0.0	0.75	0.0	0.0

(c) Control input decoded for time step $t = 0$

Source: Author.

5.3.1.3 Genetic Operations

In order to obtain control inputs for a time step $k \in [t, t+H-1]$, the initial population of BRKGA+NSGA-II is composed of random-generated individuals, elite members from the GA results of the previous time step (i.e., $k-1$), and individuals generated by the Cloud, Deadline, Net Delay, Cluster, Combined Solution, and Complementary Solution heuristics described in Section 4.3.2. These heuristics can be easily adapted to the one-time step chromosome representation. For Cloud, Net Delay, and Cluster heuristics, their first and second chromosome parts remain unchanged, and the third part is now $C^{\mathrm{III}}_{a,m} = 0$. The Deadline heuristic also keeps its first two parts, but its last part is set to $C^{\mathrm{III}}_{a,m} = 1 - \left(A^{\mathrm{rd}}_{a} / \max_{i \in \mathscr{A}} \left\{A^{\mathrm{rd}}_{i}\right\}\right)$. Meanwhile, Combined Solution and Complementary Solution heuristics continue the same.

Regarding stopping criteria, we include an execution timeout parameter in addition to the criteria mentioned in Section 4.3.3. For the other genetic operations (e.g., crossover, classification, and selection operations), we can use the same ones utilized in the static approach.

5.3.2 *H-Steps Look-Ahead Algorithm*

For problem (5.12) with a prediction horizon $H > 1$, instead of directly establishing a control input sequence for this prediction horizon, let $\pi_C = \{C(k) \mid k \in [t, t+H-1]\}$ be a sequence of chromosome vectors using the one-time step representation and $C(k)$ be the chromosome selected for a time step k. Then, Algorithm 4 describes how to obtain a control input sequence from π_C. For each time step k, Algorithm 4 firstly decodes the chromosome vector $C(k)$ to a control input $c(k)$ by Algorithm 3, current system state $s(k)$, and environment input $e(k)$. Then, it uses the behavioral model $\Phi(\cdot)$ and a forecasting method to estimate the system state $s(k+1)$ and environment input $e(k+1)$ of the next algorithm iteration (i.e., $k = k+1$). As Algorithm 3 always decodes a chromosome to a valid control input, a control sequence generated by Algorithm 4 is a feasible solution for problem (5.12). Moreover, Algorithm 4 has $O(H)$ times the complexity of Algorithm 3 when considering that environment inputs are predicted before the algorithm execution. That is, the computational complexity of Algorithm 4 is $O\left(HAV\left(\log A + V \log V + LVR\right)\right)$, where A is the number of applications, V is the number of nodes, R is the number of resource types, and L is related to the load chunk size parameter of Algorithm 3.

An evaluation of all possible π_C sequences may present a similar computational complexity issue to the control input sequence case. Therefore, we propose two heuristics in the rest of this section to obtain chromosome sequences that are decoded to solutions for problem (5.12).

Algorithm 4: The procedure of obtaining a control input sequence from a chromosome sequence for the dynamic service placement problem

```
   Data: $\pi_C = \{C(t), \ldots, C(t+H-1)\}$, $s(t)$, $e(t)$
   Result: $\pi_c = \{c(t), \ldots, c(t+H-1)\}$
1  forall $k \in [t, t+1, \ldots, t+H-1]$ do
2      $c(k) \leftarrow$ Algorithm 3 with $C(k)$, $s(k)$, $e(k)$;
3      $s(k+1) \leftarrow \Phi(s(k), c(k), e(k))$;
4      $e(k+1) \leftarrow$ by a forecasting method;
```

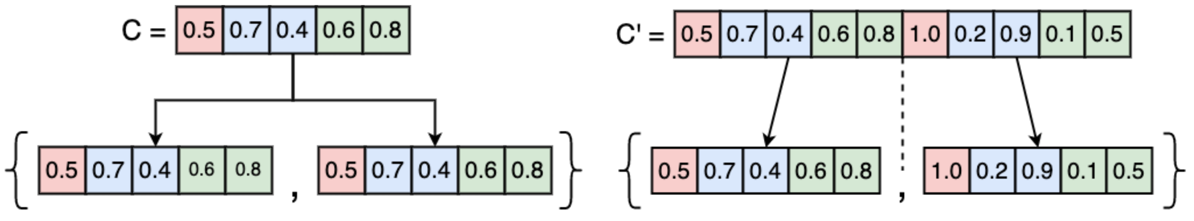
5.3.2.1 *Simple Sequence*

Given a one-time step chromosome vector C, this heuristic creates a simple sequence only containing this chromosome, i.e., $\pi_C = \{C(k) = C \mid k \in [t, t+H-1]\}$. Figure 29a illustrates this simple sequence generation from a chromosome vector C and prediction horizon $H = 2$. The basic idea is to explore how the system state evolves when control inputs generated by the same chromosome are applied to the system through the prediction horizon. In this way, BRKGA+NSGA-II can solve problem (5.12) by using the one-time step chromosome representation and Algorithm 5 as the decoder algorithm. Algorithm 5 creates the simple chromosome sequence of an individual chromosome and, then, it uses Algorithm 4 to obtain a control input sequence as the associated solution for this individual. Consequently, Algorithm 5 has the same computational complexity as Algorithm 4.

Algorithm 5: Chromosome decoder for the dynamic service placement problem with multiple look-ahead time steps and simple sequence generation

```
  Data: individual, $s(t)$, $e(t)$
  Result: $\pi_c = \{c(t), \dots, c(t+H-1)\}$
1 $C \leftarrow$ individual.chromosome;
2 $\pi_C \leftarrow \{C(k) = C \mid k \in [t, t+H-1]\}$;
3 $\pi_c \leftarrow$ by Algorithm 4 with $\pi_C$, $s(t)$, $e(t)$;
```

Figure 29 – Chromosome sequence generation examples



(a) Simple sequence generation

(b) General sequence generation

Source: Author.

5.3.2.2 *General Sequence*

Another heuristic to generate a control input sequence within the prediction horizon H is to encode this sequence in a chromosome and leaving for a GA this generation task. For this, we design a new chromosome representation C' with length $|C'| = H \times |C|$, where $|C|$ is the vector length of the one-time step chromosome representation. By using this new representation and Algorithm 6 as the decoder method, we can perform BRKGA+NSGA-II to find sub-optimal solutions for problem (5.12). The basic idea of Algorithm 6 is to split vector C' into H consecutive parts with length $|C|$, resulting in a sequence of one-time step chromosome vectors. Then, Algorithm 4 is performed to decode this chromosome sequence into a sequence of control inputs. Figure 29b illustrates the general sequence generation by slicing chromosome C' into $H = 2$ parts, creating a sequence of two one-time step vectors. Observe that this new representation can be seen as the concatenation of a sequence of on-time step chromosome vectors. Hence, the new presentation C' allows the generation of any one-time step chromosome vector sequence.

Algorithm 6: Chromosome decoder for the dynamic service placement problem with multiple look-ahead time steps and general sequence generation

```
   Data: individual, $s(t)$, $e(t)$
   Result: $\pi_c = \{c(t), \dots, c(t+H-1)\}$
1  $C' \leftarrow$ individual.chromosome;                    // Vector length $|C'| = H \times |C|$
2  $\pi_C = \{C(k) \mid k \in [t, t+H)\} \leftarrow$ by splitting $C'$ into $H$ consecutive parts with length $|C|$;
3  $\pi_c \leftarrow$ by Algorithm 4 with $\pi_C$, $s(t)$, $e(t)$;
```

Finally, Table 10 summarizes our three control algorithms. They all perform the genetic algorithm BRKGA+NSGA-II to solve problem (5.12) but with different prediction horizon lengths, chromosome representations, and decoder algorithms.

Table 10 – Proposed control algorithms

Control Algorithm		**Horizon Length**	**GA**	**Chromosome Representation**	**Chromosome Decoder**
One-Time Step		$H = 1$	BRKGA+NSGA-II	$C = [C^{\mathrm{I}}, C^{\mathrm{II}}, C^{\mathrm{III}}]$	Algorithm 3
H-Steps Look-Ahead	Simple Sequence	$H > 1$	BRKGA+NSGA-II	$C = [C^{\mathrm{I}}, C^{\mathrm{II}}, C^{\mathrm{III}}]$	Algorithm 5
	General Sequence	$H > 1$	BRKGA+NSGA-II	C', $\lvert C'\rvert = H \times \lvert C\rvert$	Algorithm 6

Source: Author.

5.4 Performance Analysis

This section presents the analytical evaluation of our proposed control algorithms by comparing them with benchmarking algorithms over a cellular network (5G) with Edge Computing capabilities.

This section is organized as follows. First, Subsection 5.4.1 describes the evaluated algorithms. Next, Subsection 5.4.2 presents the performance metrics. Then, Subsection 5.4.3 details the experiment setup. In Subsection 5.4.4, we define the values of key parameters of the control algorithms. Finally, we analyze the obtained experimental results in Subsection 5.4.5.

5.4.1 *Evaluated Algorithms*

An overview of the compared algorithms is given below:

- ***Cloud***. It places all applications in the cloud node.
- ***Net delay + Deadline (N+D)***. It is a combination of Net Delay and Deadline heuristics presented in Section 5.3.1. We combine these two heuristics to have well-defined values for the second and third parts of the one-time step chromosome vector. For the first vector part, we set $C_a^{\mathrm{I}} = 1$ to place as many replicas as possible for any application $a \in \mathscr{A}$. Then, Algorithm 3 decodes the combined chromosome vector into a control input.
- ***One-Time Step (H1)***. We use the genetic algorithm BRKGA+NSGA-II with the one-time step chromosome representation and decoder described in Section 5.3.1 when the prediction horizon $H = 1$. We also include in BRKGA+NSGA-II the preferred dominance operator defined in Section 4.2.4.
- ***Static***. This algorithm outputs a fixed placement decision and load distribution without much change throughout the experiment. For this, it performs the *H1* algorithm only in the first time step of the experiment. Then, the resulted control decision is maintained almost without changes for the remaining time steps. Control decisions may change when an application replica cannot handle a load increase due to a lack of resources and, then, this excess load is sent to the cloud node.
- ***Simple Sequence (SS)***. It is the BRKGA+NSGA-II with the preferred dominance operator and the simple sequence generation heuristic presented in Section 5.3.2 when the prediction horizon $H > 1$, which allows pre-deployment exploration. Moreover, we use ARIMA as the forecasting method and the auto-ARIMA algorithm proposed by Hyndman and Khandakar (2008) to automatically identify the most optimal parameters for an ARIMA model.
- ***General Sequence (GS)***. It is similar to *SS*, but using the general sequence generation heuristic instead.

5.4.2 *Performance Metrics*

We define the following metrics as the optimization objectives upon which we evaluate the compared algorithms:

- **Deadline Violation**. Equation (5.13) expresses the weighted average deadline violation among all request flows at a time step, where $[z]^{+} = max(0,z)$. Each request flow $\mathscr{F}_{a,m,n}(k)$ contributes to this violation according to its request transmission rate $\delta_{a,m,n}(k)\Lambda_{a,m}(k)$ as a weight. Given the importance of keeping response time below its deadline, we select this metric as the primary objective function in the preferred dominance operator. Furthermore, we normalize the result of this objective function by using the *Cloud* solution as the base.

$$f_{\text{dv}}\left(s(k+1),c(k),e(k)\right) = \frac{\sum_{a\in\mathscr{A}}\sum_{m,n\in\mathscr{V}}\left[d_{a,m,n}(k+1) - A_a^{\text{rd}}\right]^{+}\delta_{a,m,n}(k)\Lambda_{a,m}(k)}{\sum_{a\in\mathscr{A}}\sum_{m,n\in\mathscr{V}}\delta_{a,m,n}(k)\Lambda_{a,m}(k)} \tag{5.13}$$

– **Operational Cost**. Equation (5.14) specifies the operational cost during a time step as the total cost of the resources allocation for all deployed application replicas.

$$f_{\text{cost}}\left(s(k+1),c(k),e(k)\right) = \sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}}\rho_{a,n}(k)T_s\sum_{r\in\mathscr{R}}N_{n,r}^{\text{cost}}\left(A_a^r\left(\lambda_{a,n}(k)\right)\right) \tag{5.14}$$

– **Migration Cost** is the ratio of application replicas migrated/replicated in the system at a time step, as shown in Equation (5.15). The application content size $A_a^{\text{D+R}}(\cdot)$ is a weight in this metric as large applications may take longer to be transferred and consume more network resources.

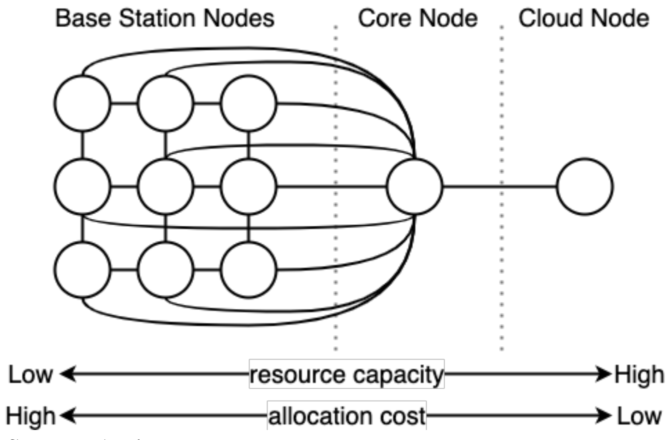
$$f_{\text{mig}}\left(s(k+1),c(k),e(k)\right) = \frac{\sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}}A_a^{\text{D+R}}\left(\lambda_{a,n}(k)\right)\rho_{a,n}(k)\left(1-\rho_{a,n}(k-1)\right)}{\sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}}A_a^{\text{D+R}}\left(\lambda_{a,n}(k)\right)\rho_{a,n}(k)} \tag{5.15}$$

5.4.3 Analysis Setup

We developed the experiments using Python in a server machine with Intel Xeon E5-2630 @ 2.60GHz, 24 CPUs, and 64 GB of RAM to run the compared algorithms. As shown in Figure 30, the experiment scenario consists of a 5G network with nine BSs, forming a 3×3 grid network. These BSs are also connected to a core node, which is connected to the cloud on the other side. All of these nodes (BSs, core, and cloud) have hosting capabilities, and their total resource capacities are lower as we descend from cloud to BSs. On the other hand, the resource allocation cost increases as nodes get closer to end users due to resource scarcity. Besides, we consider a linear function to the node usage cost $N_{n,r}^{cost}(\cdot)$. Table 11 presents the main evaluation parameters.

Similar to the evaluation in Chapter 4, we analyze the placement of the three types of applications specified for a 5G network. First, mMTC applications are characterized by low resource requirements, delay tolerance, and a quite large number of users. Then, eMBB applications have high resource demand, a medium deadline, and an intermediate number of users. Finally, URLLC applications have low resource usage, a strict deadline, and a small number

Figure 30 – Experiment network scenario for the dynamic service placement problem



Source: Author.

of users. Based on these characteristics, we randomly assign the values for the application parameters. Moreover, application resource demands $A_a^r(\cdot)$ are considered to be linear functions. In the CPU demand case, the linear constants are selected based on queue stability and deadline requirements.

Each user is randomly attached to a BS, and it generates requests with an unchanged average rate to a single application selected according to the user proportion parameter in Table 11. Despite this constant rate, we create a synthetic dynamic load by changing the number of active users for each application in each node according to the Stable, Growing, Cycle/Bursting, or On-and-Off workload patterns for cloud environments discussed in (LORIDO-BOTRAN *et al.*, 2014). Along with the cloud patterns, we also include a random load generation. These workload patterns are implemented as follows:

– **Stable Pattern.** A stable workload is characterized by a constant number of requests per time unit. In this way, we apply Equation (5.16) to generate a stable load at each time step t of the experiment, where the constant parameter c is randomly selected.

$$L_{\text{stable}}(t) = c, \quad c \in [0,1] \tag{5.16}$$

– **Growing Pattern.** It shows a load that increases due to, for instance, an application becoming popular. We also support the inverse of this pattern where a load decreases along the time. This pattern is implemented by the linear function in Equation (5.17), where the constant a is randomly chosen and $T_{\max}$ is the total number of time steps in the evaluation.

$$L_{\text{grow}}(t) = t\left(\frac{b-a}{T_{\max}}\right) + a, \quad a \in [0,1], b = \begin{cases} 0 & \text{if } a > 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{5.17}$$

Table 11 – Performance evaluation parameters for the dynamic and centralized service placement problem

Parameter	Value
Infrastructure	
CPU (MIPS)	Cloud: ∞, Core: 2×10^4, BS: 10^4
DISK (GB)	Cloud: ∞, Core: 32, BS: 16
RAM (GB)	Cloud: ∞, Core: 16, BS: 8
Usage Cost for resource/second	$N_{n,r}^{\text{cost}}(x) = a \times 10^{-b}(x+1)$ $a =$ Cloud: 0.25, Core: 0.5, BS: 1 $b =$ CPU: 12, DISK: 18, RAM: 15
Bandwidth (Gbps)	BS-BS:0.1, BS-Core:1, Core-Cloud:10
Propagation Delay (ms)	BS-BS:1, BS-Core:1, Core-Cloud:10
User Proportion (%)	mMTC:70, eMBB:20, URLLC:10
App. Proportion (%)	mMTC:34, eMBB:33, URLLC:33
Applications	
Max. Replicas $A_a^{\max}$	$[1, \lvert\mathscr{V}\rvert]$
Deadline A_a^{rd} (s)	mMTC:$[0.1,1]$, eMBB:$[0.01,0.1]$, URLLC:$[0.001,0.01]$
λ_a (requests/s)	mMTC: $[0.1,1]$, eMBB: $[1,100]$, URLLC: $[1,100]$
Data A_a^{data} (Kb)	mMTC: $[0.1,1]$, eMBB: $[1,10]$, URLLC: $[0.1,1]$
CPU Work A_a^{work} (CPU Instructions)	mMTC, URLLC: $[1,5] \times 10^6$, eMBB: $[5,10] \times 10^6$
RAM, DISK Demand $A_a^r(\lambda) = b\lambda + c$ (MB)	$b =$ mMTC, URLLC: $[0.1,1]$, eMBB:$[1,10]$ $c =$ mMTC, URLLC: $[10,100]$, eMBB: $[100,1000]$
CPU Demand (IPS) $A_a^r(\lambda) = b\lambda + c$	$b = A_a^{\text{work}} + 1$ $c = A_a^{\text{work}}/\alpha A_a^{\text{rd}} + 1$, $\alpha = [0.1, 0.5]$
Genetic Algorithm	
Max. Generations	50
Population Size	100
Execution Timeout (s)	60

Source: Author.
Note: An interval $[x,y]$ means that a value is chosen randomly within this range.

- **Cyclic/Bursting Pattern.** This workload pattern may present regular periods or bursts of loads (e.g., daytime has more workload than nighttime). We split this pattern into cyclic and bursting patterns. The trigonometric function in Equation (5.18) creates the cyclic pattern, where both period p and phase shift θ parameters are randomly picked. Meanwhile, Equation (5.19) produces a bursting pattern using the probability density function of the Beta distribution $\mathscr{B}(\alpha,\beta)$, where the pair (α,β) is randomly selected from set $\{(2,2),(2,3),(3,2),(1,5),(5,1)\}$ to generate different curves. This probability density function is used by the 3rd Generation Partnership Project (3GPP) to model the burst traffic of IoT devices in cellular networks (METZGER *et al.*, 2019).

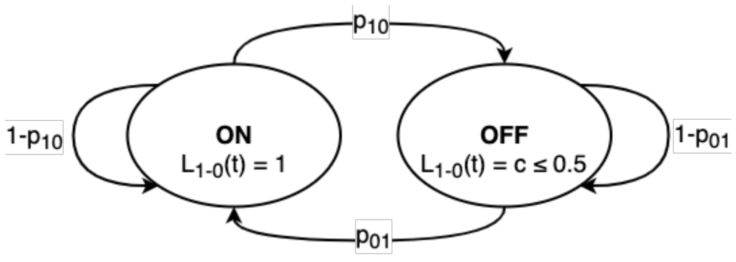
$$L_{\text{cycle}}(t) = \cos(2\pi t/p - \theta), \quad p \in \{T_{\max}/2, T_{\max}\}, \theta \in [0, \pi] \tag{5.18}$$

$$L_{\text{burst}}(t) = \frac{1}{\mathscr{B}(\alpha,\beta)} \left(\frac{t}{T_{\max}}\right)^{\alpha-1} \left(1 - \frac{t}{T_{\max}}\right)^{\beta-1} \tag{5.19}$$

– **On-and-Off Pattern.** In this pattern, workloads are processed periodically or occasionally processed in batches. We implement this pattern using a two-state Markov chain, as shown in Figure 31. In the On state, the normalized load $L_{1-0}(t)$ is equal to 1, whereas the load is a constant $c \in [0, 0.5]$ randomly chose in the Off state. Furthermore, the transition probabilities (p_{10}, p_{01}) are randomly picked from set $\{(0.1,0.1),(0.2,0.2),(0.3,0.3)\}$, which contains low values to produce consecutive time steps with the same state.

– **Random Pattern.** In this pattern, the normalized load $L_{\text{rnd}}(t)$ is uniformly drawn over the interval $[0,1]$ for each experiment time step t.

Figure 31 – On-and-Off workload pattern as a two-state Markov chain



Source: Author.

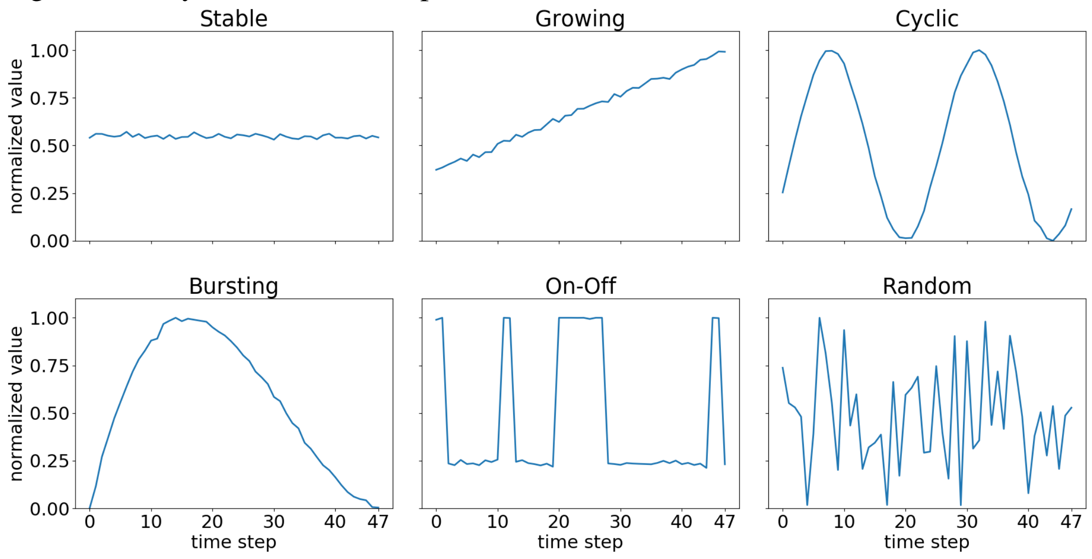
For all samples produced by the above workload patterns, we add to it a white Gaussian noise with zero mean and standard deviation equal to 0.01, i.e., the white noise follows the normal distribution $\mathcal{N}(0, 0.01^2)$. Then, the resulted values are normalized to be in the interval $[0,1]$. Figure 32 illustrates the normalized load with white noise over time steps for all the aforementioned workload patterns. Moreover, let $L_{a,n}$ be the normalized samples from a randomly selected workload pattern, then the load $Q_{a,n}(t)$ generated by users of application a attached to node n is determined for each experiment time step t as follows:

$$Q_{a,n}(t) = A_a^{\text{req}} U_{a,n} L_{a,n}(t)$$

where $U_{a,n}$ is the total number of users of application a attached to node n, and A_a^{req} is the application request rate.

A test case of the evaluated scenario consists of 48 time steps with 30 minutes of duration each, totaling one day. In this way, it is possible to make, apply, and measure the result

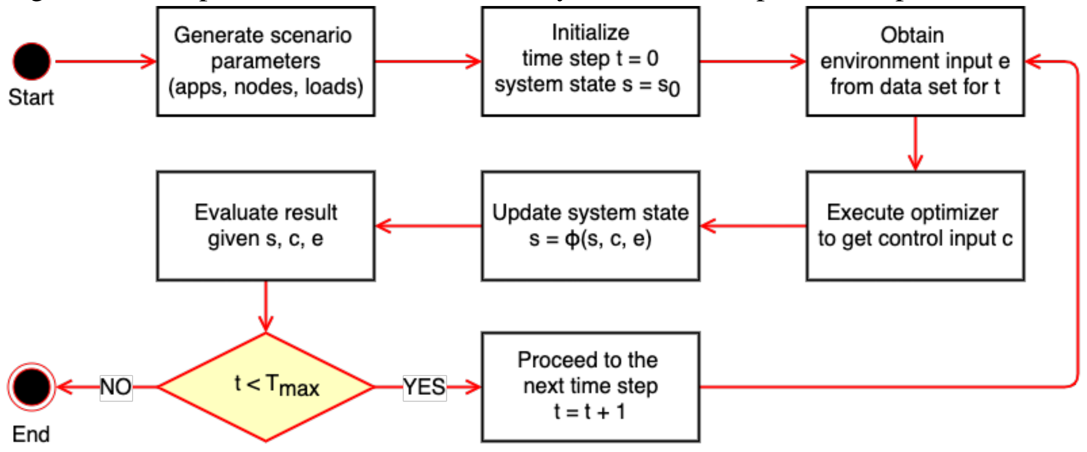
Figure 32 – Synthetic workload patterns



Source: Author.

of a control decision in a single time step. We conduct test case experiments according to the flowchart shown in Figure 33. Initially, all parameters of the evaluation scenario are generated and saved in a data set, including the dynamic loads as environment inputs. For each time step, we get the environment input of this time step to execute the controller optimizer using one of the compared algorithms. After obtaining the control input from the optimizer, we apply the behavioral model to update the system state. Then, we evaluate the algorithm performance according to the optimization objectives and the updated system state. This process is repeated until the maximum number of time steps is reached.

Figure 33 – Experiment flowchart for the dynamic service placement problem


Start
Generate scenario parameters (apps, nodes, loads)
Initialize time step t = 0 system state s = s0
Obtain environment input e from data set for t
Execute optimizer to get control input c
Update system state s = ϕ(s, c, e)
Evaluate result given s, c, e
t < Tmax
YES
NO
Proceed to the next time step t = t + 1
End

Source: Author.

Finally, the results presented in the following sub-sections come from an average of 30 different random runs for each test case and compared algorithm.

5.4.4 Different Parameters Settings

This subsection analyzes the performance in terms of the optimization objective against different values for key parameters (load chunk size and prediction horizon length) of the proposed control algorithms.

5.4.4.1 Load Chuck Size

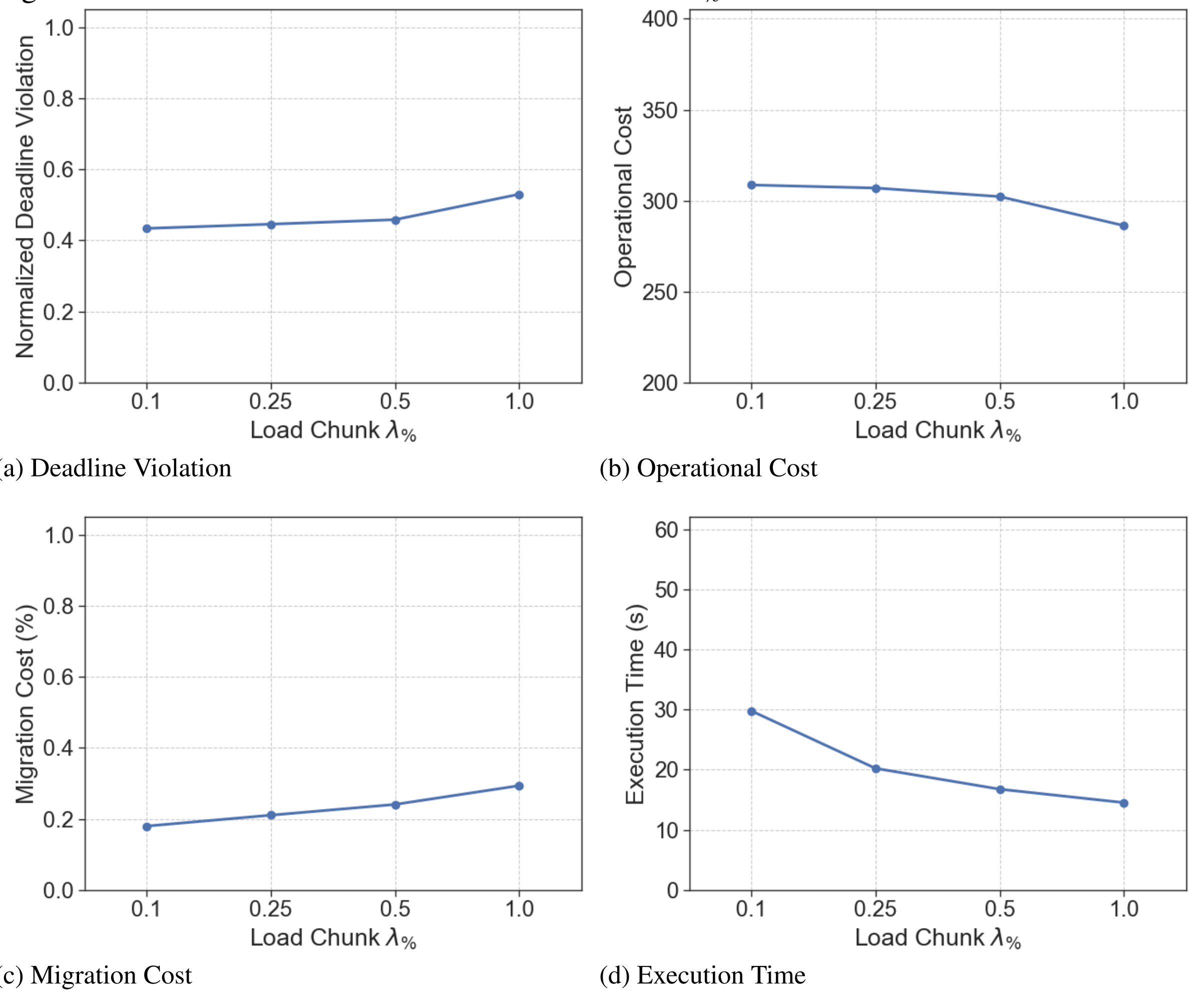
In Algorithm 3, the total application request load from a source node is distributed in chunks whose size depends on the parameter $\lambda_{\%} \in (0,1]$. A small value for $\lambda_{\%}$ means a small chunk size and, thus, a more fine-grained load distribution. On the other hand, a large $\lambda_{\%}$ implies a coarse-grained distribution. For example, $\lambda_{\%} = 1$ means that Algorithm 3 tries to find a single target node to receive the total application load from a source node. Meanwhile, $\lambda_{\%} = 0.5$ means that the entire load is split in half, and each half load can be distributed to a different target node. However, as each loop iteration assigns a single load chunk to a target node, a small chunk requires more iterations than a large one to distribute all load.

Figure 34 presents the performance influence of different chunk load sizes in a scenario with 10 applications and 10,000 users for *H1*, which uses Algorithm 3 as the chromosome decoder. In Figure 34a, we observe slight increases in deadline violation from $\lambda_{\%} = 0.1$ to 0.5, respectively, and a more pronounced increase from 0.5 to 1. Similar result pattern is presented for migration cost in Figure 34c. On the other hand, Figure 34b shows that the operational cost slightly decreases from $\lambda_{\%} = 0.1$ to 0.5 and, then, this cost drops more accentuated when $\lambda_{\%} = 1$. These result patterns can be due to the cloud node being more used to receive large chunks of load per time by having unlimited and inexpensive resources. However, a small chunk implies more execution time, as shown in Figure 34d. Hence, we select $\lambda_{\%} = 0.25$ as a trade-off between performance, especially for deadline violation, and execution time.

5.4.4.2 Prediction Horizon Length

We also examine the performance of different prediction horizon lengths for *SS* and *GS* algorithms in a scenario with 10 applications and 10,000 users. *SS* does not present significant performance changes by increasing the horizon length for the metrics in Figure 35. For *GS*, the results between $H = 2$ and 4 differ less than 7% on average for deadline violation, operation cost, and migration cost in Figures 35a, 35b, and 35c, respectively. Regarding the

Figure 34 – Performance of different load chunk size $\lambda_{\%}$ values



(a) Deadline Violation (b) Operational Cost

(c) Migration Cost (d) Execution Time

Source: Author.

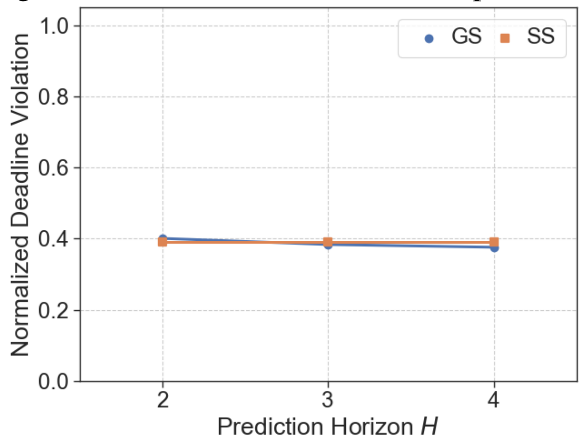
execution time, Figure 35d shows a 12% time increase from $H = 2$ to 3, reaching the specified maximum execution time. This increase in time can be due to the growth of the chromosome vector length used in *GS*, which depends on the prediction horizon length.

Nonetheless, it is sufficient to set the prediction horizon H to 2 as we specify the time step duration to be large enough to start and complete a migration operation. In this way, applications can be pre-deployed in the first time step in the prediction horizon. Then, in the second time step, loads can be distributed to the pre-deployed applications without the migration drawbacks.

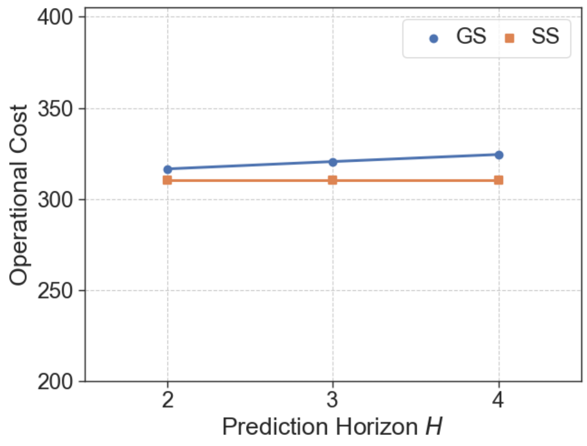
5.4.5 *Results and Discussion*

We evaluated the performance of the examined algorithms for each optimization objective in scenarios with different amounts of applications and users.

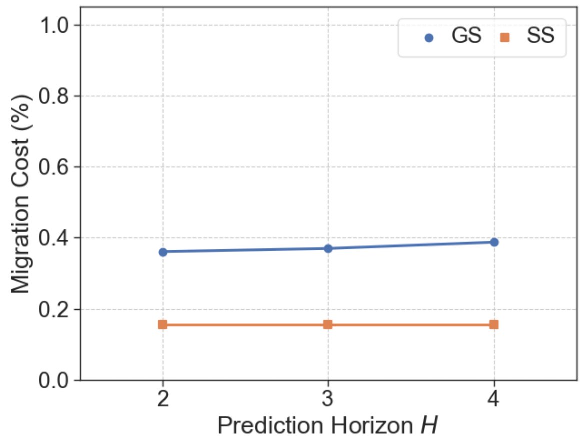
Figure 35 – Performance of different prediction horizon H values



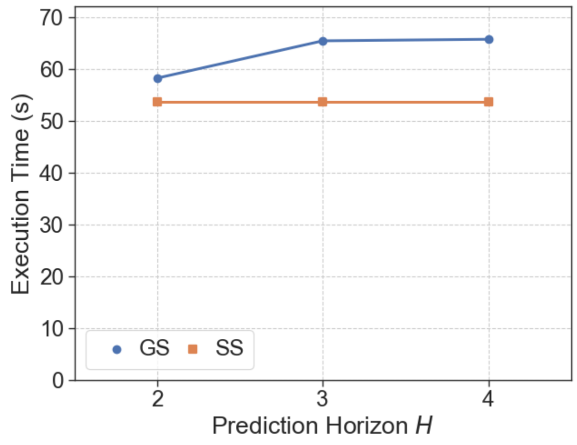
(a) Deadline Violation



(b) Operational Cost



(c) Migration Cost



(d) Execution Time

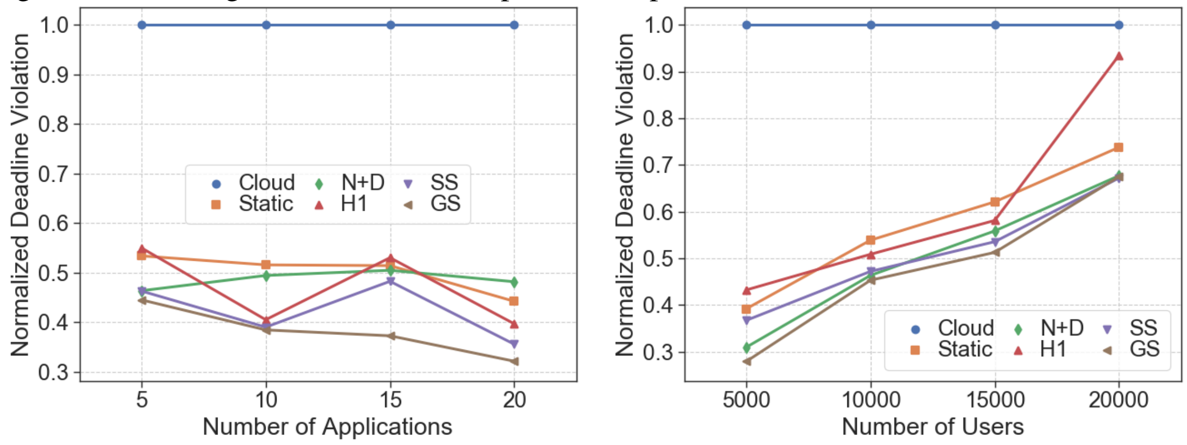
Source: Author.

5.4.5.1 Deadline Violation

Figures 36a and 36b show the normalized deadline violation per time step when increasing the number of applications and users, respectively. This normalization uses *Cloud* results as the base. In both figures, we can see that *GS* achieves lower violation levels than the other algorithms. On average, *GS* has 24% and 21% fewer violations than the other algorithms when varying the number of applications and users, respectively. Meanwhile, *SS* presents on average 15% and 8% fewer deadline violations than *H1*, *Static*, and *N+D* solutions when varying the number of applications and users, respectively.

We also observe that *GS* reduces deadline violations in Figure 36a when there are more applications but keeping the total number of users fixed. This drop can be an effect of having fewer users per application, especially for the URLLC type that is characterized by low user percentages and strict deadlines. On the other hand, violations rise when the number of users increases with a fixed number of applications in Figure 36b. This is caused by the growth of requests traffic to the remote cloud node when there is more competition for resources on the other nodes.

Figure 36 – Average deadline violation per time step



(a) 10,000 users (b) 5 applications

Source: Author.

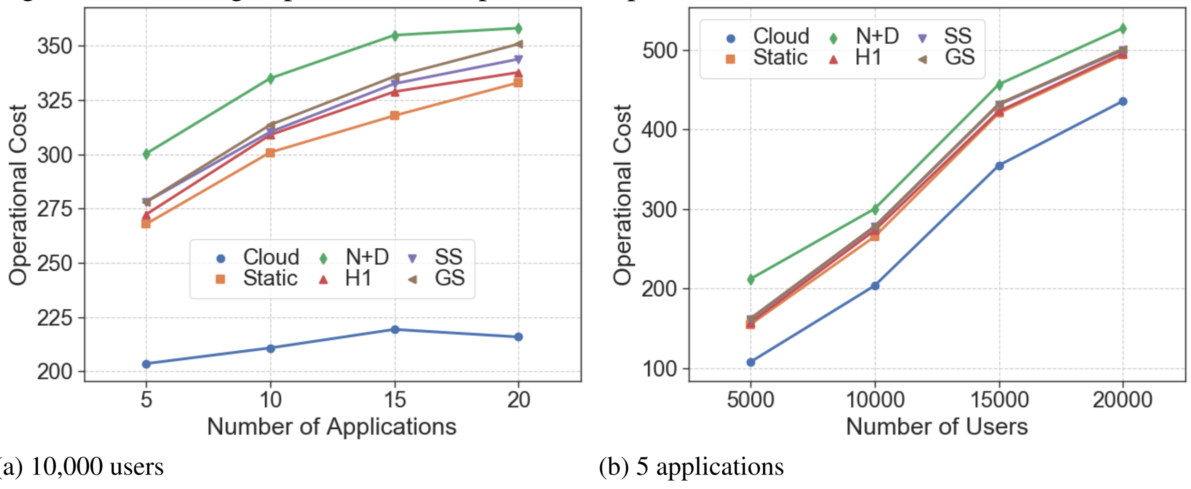
5.4.5.2 *Operational Cost*

We observe an augmentation of operational costs per time step by having more applications or users in Figures 37a and 37b, respectively. *Cloud* exhibits the lowest costs because only one replica of each application is placed in the system, and cloud resources are cheaper than in other locations. Meanwhile, *N+D* tends to place as many replicas as possible, having the highest cost. The other compared algorithms have similar costs when varying the number of applications or users.

5.4.5.3 *Migration Cost*

Regarding migration costs in Figures 38a and 38b, *Cloud* and *Static* present no migration as expected. In both figures, *GS* has the highest costs, which is a trade-off of prioritizing deadline violation. In addition, migration costs for *H1*, *SS*, and *GS* initially decrease from 5 to 10 applications and then increase after ten applications in Figure 38a. This cost turning point happens because of two aspects that impact the volume of migration traffic: (*i*) the total number of application replicas that increase with more applications, and (*ii*) their content size that shrinks when there are fewer users per application. In Figure 38b, migration costs fall when

Figure 37 – Average operational cost per time step

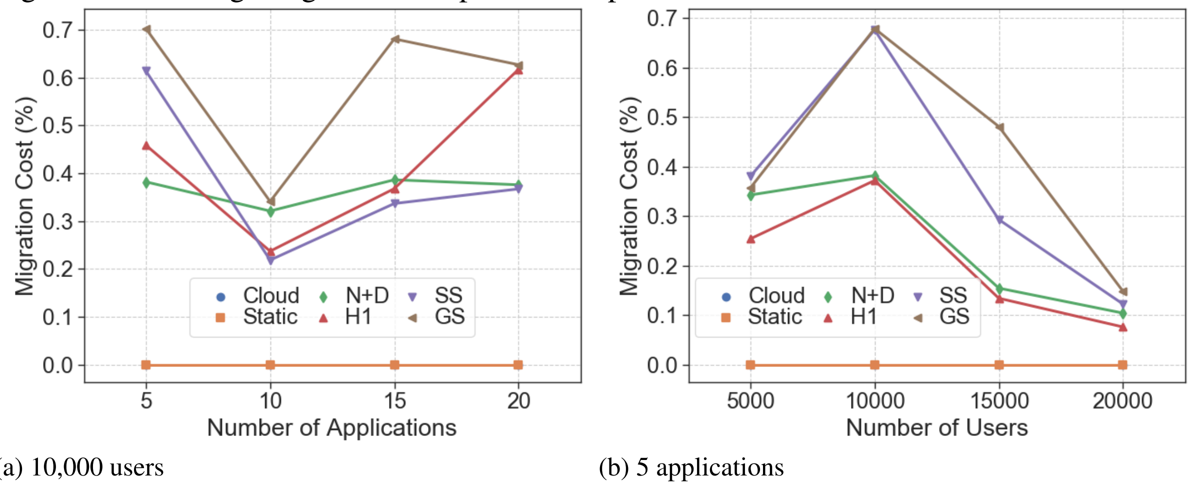


(a) 10,000 users (b) 5 applications

Source: Author.

there are more than 10,000 users for a fixed number of applications. As well as the rise of deadline violation in Figure 36b, this migration cost reduction is also a consequence of cloud traffic growth.

Figure 38 – Average migration cost per time step



(a) 10,000 users (b) 5 applications

Source: Author.

Although *GS* has the highest migration costs among the algorithms in Figures 38a and 38b, its results indicate that less than 0.7% of all application contents in a time step comes from migration operations. Hence, these results are still low and can be considered an acceptable compromise for *GS* to have the lowest deadline violations. Furthermore, our other dynamic control algorithms (*SS* and *H1*) also present low migration costs and have deadline violations less than or equal to *Static*, *N+D*, and *Cloud* solutions.

5.5 Summary

In this chapter, we aimed to answer the research question RQ2: *"How to reassess the service placement and load distribution decisions due to dynamic application loads by taking into account the benefits and costs of service migrations?"*. We addressed this question by proposing a centralized controller that uses a limited look-ahead prediction strategy to handle the impact of service migration on application response time while optimizing multiple performance-related objectives. In this strategy, we designed a genetic algorithm to solve the formulated problem for a single time step, and two extensions of this algorithm, *SS* and *GS*, when looking at more time steps within a prediction horizon.

Evaluations has shown that our proposed *SS* and *GS* algorithms outperform other benchmark algorithms in terms of deadline violations while having similar operational cost. A consequence of prioritizing deadline violation minimization by our proposals was the occurrence of more service migrations.

In the next chapter, we address the scalability issue of a centralized controller in a large Edge Computing environment by designing a distributed control strategy.

6 A DYNAMIC AND DISTRIBUTED APPROACH FOR SERVICE PLACEMENT WITH LOAD DISTRIBUTION

This chapter addresses the scalability issue of the centralized controller presented in the previous chapter when taking control decisions in a large-scale Edge Computing (EC), i.e., an infrastructure with many EC nodes. Hence, we propose a hierarchical distributed limited look-ahead control approach to reduce the dimensionality of the overall control problem by decomposing this problem into a set of local control problems solved in a hierarchical cooperative fashion. The main contributions of this chapter are the following:

– We design a two-layer hierarchical control architecture. At the upper control layer, a global controller receives system-wide information and provides local control restrictions for the lower control layer, which is composed of local controllers that may exchange information to coordinate their control decisions.

– For the global controller, we propose to use a simplified or aggregated view of the system to reduce the dimensionality of taking control decisions for the entire system. Based on this simplified system model, we formulate a global control decision problem to optimize the overall system performance. We then apply the BRKGA+NSGA-II to solve the formulated problem, in which the obtained solution is communicated to local controllers as additional constraints in their control decision problems.

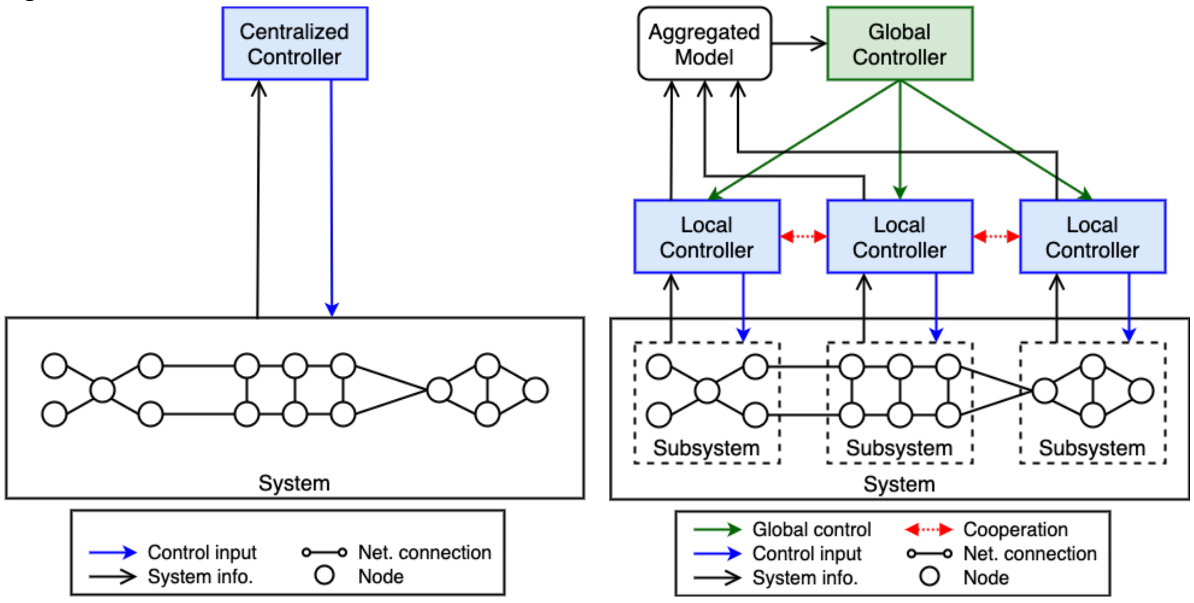
– We develop a local controller to make control decisions for a subset of EC nodes based on dynamics, constraints, objectives, and environment disturbances in this node subset. As a control decision in a local controller may affect the performance on nodes controlled by another controller, we implement a cooperative strategy where local controllers exchange information to coordinate their control actions.

The remainder of this chapter is organized as follows. In Section 6.1, we present our hierarchical distributed control architecture. Section 6.2 describes the global controller, whereas Section 6.3 defines the local controllers and their cooperation. Next, we conduct performance evaluations in Section 6.4. Then, Section 6.5 concludes this chapter.

6.1 Hierarchical Distributed Control Design

As shown in Figure 39a, the centralized controller presented in Chapter 5 receives information about the entire system and must decide the control inputs for each node in the Edge Computing infrastructure. Moreover, control decisions involve solving the optimization control problem (5.12) at each time step. However, the centralized approach may suffer scalability issues to solve this problem within a time step duration in a large-scale Edge Computing system. In order to address this issue, we adopt a distributed control approach wherein the overall control problem is decomposed into a set of local control problems to be solved in a hierarchical cooperative fashion.

Figure 39 – Control architectures



(a) Centralized control

(b) Hierarchical distributed control

Source: Author.

The adopted distributed control architecture is illustrated in Figure 39b, which is based on a distributed control structure proposed by Abdelwahed *et al.* (2004) for load distribution in a Cloud Computing system. In the adopted distributed architecture for an EC system, all EC nodes are partitioned into disjoint subsets called subsystems. For instance, nodes can be grouped in the same subsystem if they are geographically close. However, it is out of the scope of this thesis to define how to do this partition. In addition, the distributed architecture contains two layers of limited look-ahead controllers, which are described as follows.

At the lowest level, a subsystem has its own local controller responsible for control decisions (e.g., application placement and load distribution) only on its underlying nodes, which are fewer than the total number of nodes in the entire system. Consequently, there are fewer decision variables in a local controller compared to the centralized approach. Each local controller makes its own control decisions based on dynamics, constraints, objectives, and environment disturbances of the subsystem under consideration. Unlike the distributed architecture proposed by Abdelwahed *et al.* (2004), local controllers may also exchange information (e.g., control inputs and subsystem states) in our distributed control system to coordinate their decision-making processes as subsystems interactions may influence their performance. For example, a fraction of request load from a subsystem can be distributed to another subsystem, affecting application workloads in both subsystems and, thus, application response time. In this way, the local controller of a subsystem can use the load distribution information from other controllers to make its own load distribution decision in order to improve its subsystem performance.

On top of local controllers, a global controller receives system-wide information and takes control decisions based on dynamics, constraints, objectives, and environment disturbances of the overall system. These global control decisions are then communicated to local controllers as additional restrictions on their decision-making processes towards optimizing the overall system performance while satisfying global constraints. For instance, a global constraint can be the maximum number of application replicas to be placed over the entire system. However, the global controller has a more simplified view of the entire system instead of considering a detailed system model like the one used by the centralized approach. This simplified view is used to reduce the dimensionality of taking control decisions and is provided by the aggregated model that aggregates or averages information from the subsystems over some time interval.

Furthermore, controllers at different levels in the hierarchy can operate at different time scales. As the global controller uses aggregated information about the subsystems, it typically operates on a longer time scale when compared to the local controllers. Hence, global decisions can be seen as coarse-grained control actions that impose long-term restrictions while local controllers have short-term and fine-grained control of their subsystems.

In the next two sections, we detail the distributed control approach in a top-bottom fashion, where Table 12 summarizes the main notations used in this chapter. More specifically, Sections 6.2 and 6.3 describe the global and local controllers, respectively.

6.2 Global Controller

The global controller delimits the number of application replicas placed in each subsystem and the amount of load dispatched between subsystems. By having a global view of the entire system, the global controller can specify these delimit values as control actions to optimize the overall system performance while satisfying global constraints, such as the maximum number of application replicas allowed to be placed in the entire system.

Table 12 – Main notations of the distributed service placement problem

Symbol	**Description**
$L_l^{\text{bw}}, L_l^{\text{pd}}$	Bandwidth and propagation delay of link l, respectively
$N_{n,r}^{\text{cap}}, N_{n,r}^{\text{cost}}(x)$	Total capacity and allocation cost of resource r on node n, respectively
$A_a^{\text{rd}}, A_a^{\text{max}}, A_a^{\text{data}}$	Response deadline, max. number of replicas, and request data length of app a, respectively
$A_a^{\text{work}}, A_a^{\text{req}}$	CPU work size of a request and request generation rate for app a, respectively
$A_a^r(\lambda)$	Demand of resource r for a replica of app a with workload λ
$A_a^{\text{D+R}}(\lambda)$	Content size of app a with workload λ
$u_{a,n}(t)$	Number of active users connected to node n requesting app a at time t
Global Controller	
$\mathscr{V}_G, \mathscr{V}_i$	Set of all subsystems and nodes in subsystem i, respectively
$\rho_{a,i}^G(t_G)$	Number of replicas of app a to be placed in subsystem i at global time step t_G
$\delta_{a,i,j}^G(t_G)$	Fraction of requests for app a that should be distributed from subsystem i to j at time t_G
$Q_{a,i}^G(t_G)$	Request generation rate from users of app a in subsystem i at time t_G
$d_{a,i,j}^G(t_G)$	Response time of requests for app a from subsystem i to j at time t_G
$q_{a,i}^G(t_G)$	Number of requests waiting in the queue of a single app replica a in subsystem i at time t_G
T_G, T_L	Global and local time step duration, respectively
$\Lambda_{a,i}^G(t_G), \lambda_{a,i}^G(t_G)$	Request arrival rate in an app a on subsystem i before and after load distribution at time t_G
i_C	Central node of subsystem i
$D_{\text{net}}(x,m,n)$	Network delay between nodes m and n for data with length x
$A_{a,i}^{\text{min}}, A_{a,i}^{\text{max}}$	Min. and max. number of app replicas a that can be placed in subsystem i, respectively
$\Lambda_{a,i,j}^{\text{max}}$	Max. load of app a that can be dispatched from subsystem i to j
Local Controller	
$\mathscr{V}_{-i}, \mathscr{V}_i'$	Set of neighbor subsystems and interactable entities of subsystem i, respectively
$\rho_{a,n}^i(t_L), \delta_{a,m,n}^i(t_L)$	App placement and load distribution variables in subsystem i at local time step t_L
$Q_{a,n}^i(t_L)$	Request generation rate from users of app a in entity $n \in \mathscr{V}_i'$ at time t_L
$d_{a,m,n}^i(t_L)$	Response time of requests for app a from entity m to $n \in \mathscr{V}_i'$ at time t_L
$q_{a,n}^i(t_L)$	Number of requests waiting in the app a queue on node $n \in \mathscr{V}_i$ at time t
$\Lambda_{a,n}^i(t_L), \lambda_{a,n}^i(t_L)$	Request arrival rate in an app a in $n \in \mathscr{V}_i'$ before and after load distribution at time t_L
$\alpha_{a,i}(t_L)$	Load spreading factor in subsystem i for app a at time t_L
$\lambda_{a,j}^{-i}(k)$	Load from outside subsystem i to an app a in subsystem j at time t_L
$P_{a,i}(t_L)$	Total number of replicas of app a placed in subsystem i at time t_L
$\Lambda_{a,i,j}^{\text{max}}(t_L)$	Max. load that can be dispatched from subsystem i to j at time t_L

Source: Author.

According to the Limited Look-ahead Control (LLC) concept, we first model the system dynamics used by the global controller in Subsection 6.2.1. Then, we formulate the global control optimization problem in Subsection 6.2.2. Finally, we present a genetic-based algorithm to solve the formulated problem in Subsection 6.2.3.

6.2.1 System Dynamics

Instead of considering all nodes in the Edge Computing, the global controller uses a simplified system model where all EC nodes in a subsystem are seen as a single entity representing the subsystem. Let $\mathscr{V}$ be the set of all EC nodes in the system, $\mathscr{V}_G$ be the set of all subsystems, and $\mathscr{V}_i \subseteq \mathscr{V}$ be the set of nodes in a subsystem $i \in \mathscr{V}_G$. Then, the total capacity for resource $r \in \mathscr{R}$ in a subsystem i is the aggregated capacity of all nodes in the subsystem, i.e., $N_{i,r}^{\text{cap}} = \sum_{n \in \mathscr{V}_i} N_{n,r}^{\text{cap}}$. Meanwhile, the resource allocation cost in subsystem i is determined by the average allocation cost among all nodes in this subsystem, i.e., $N_{i,r}^{\text{cost}}(x) = \frac{1}{|\mathscr{V}_i|} \sum_{n \in \mathscr{V}_i} N_{n,r}^{\text{cost}}(x)$.

Regarding system dynamics, we also use a simplified model to describe how the system behaves when applications are placed in the subsystems and application loads are distributed among subsystems. Thus, the global controller uses the following behavioral model $\Phi_G(\cdot)$ to characterize the dynamics of an EC system partitioned into subsystems:

$$s_G(t_G + 1) = \Phi_G\left(s_G(t_G), c_G(t_G), e_G(t_G)\right) \tag{6.1}$$

where t_G is the time step index of the global controller. The parameters $s_G(\cdot)$, $c_G(\cdot)$, and $e_G(\cdot)$ are the system state, control input, and environment input, respectively.

For the behavioral model $\Phi_G(\cdot)$, the control input $c_G(t_G) = (\rho_G(t_G), \delta_G(t_G))$ is given by the following decision variables:

– **Application Placement** $\rho_G(t_G) = \{\rho_{a,i}^G(t_G) \mid a \in \mathscr{A} \text{ and } i \in \mathscr{V}_G\}$. Unlike the centralized approach that uses binary variables, here, $\rho_{a,i}^G(t_G) \geq 0$ is an integer variable specifying the number of replicas of application a to be placed inside subsystem i at time step t_G as a subsystem can be composed of various nodes with hosting capability.

– **Load Distribution** $\delta_G(t_G) = \{\delta_{a,i,j}^G(t_G) \mid a \in \mathscr{A} \text{ and } i, j \in \mathscr{V}_G\}$. Similar to the centralized approach, $\delta_{a,i,j}^G(t_G) \in [0,1]$ is the fraction of requests for an application a to be distributed from a source subsystem i to a target subsystem j at time t_G.

The environment input $e_G(t_G) = (Q_G(t_G))$ is defined as:

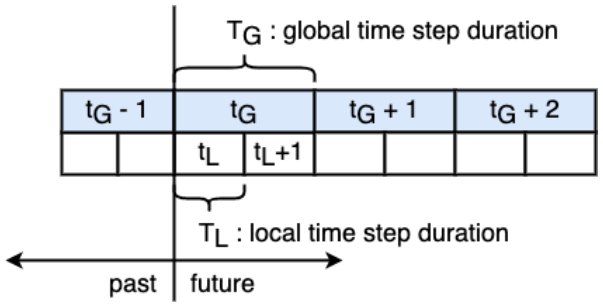
– **User-Generated Request Rate** $Q_G(t_G) = \{Q_{a,i}^G(t_G) \mid a \in \mathscr{A} \text{ and } i \in \mathscr{V}_G\}$, where $Q_{a,i}^G(t_G)$ is the aggregated average load generated by all users of an application a attached to nodes in subsystem i during global time step t_G.

As the global controller operates at a slower time scale when compared to local controllers, the global time step duration T_G is τ times longer than a time step of a local controller, i.e., $T_G = \tau T_L$, where T_L is the local time step duration and $\tau > 1$ is a positive integer. This time scale difference between global and local controllers is illustrated in Figure 40 when $\tau = 2$. Let $Q^G_{a,i}(t_G)$ be the user-generated load in a subsystem i seen by the global controller. We can define $Q^G_{a,i}(t_G)$ as the average of aggregated loads estimated by the local controller of this subsystem over a period of τ local time steps. That is, $Q^G_{a,i}(t_G)$ is given as:

$$Q^G_{a,i}(t_G) = \frac{1}{\tau}\sum_{k=0}^{\tau-1}\sum_{n\in\mathscr{V}_i}\widehat{Q}^i_{a,n}(t_L+k) \tag{6.2}$$

where t_L is the local time step at the beginning of global time step t_G and $\widehat{Q}^i_{a,n}(t_L+k)$ is the estimated load for an application a from users attached to a node n in subsystem i at local time step t_L+k, $k \in [0, \tau-1]$.

Figure 40 – Example of time scale difference between global and local controllers


T_G : global time step duration
t_G - 1
t_G
t_G + 1
t_G + 2
t_L
t_L+1
T_L : local time step duration
past
future

Source: Author.

Similar to the centralized approach, the system state $s_G(t_G) = (d_G(t_G), q_G(t_G))$ at time step t_G is given as:

- **Response Time** $d_G(t_G) = \{d^G_{a,i,j}(t_G) \mid a \in \mathscr{A} \text{ and } i, j \in \mathscr{V}_G\}$, where $d^G_{a,i,j}(t_G)$ is the average response time of requests for application a from source subsystem i to target subsystem j at the beginning of time step t_G.
- **Queue Length** $q_G(t_G) = \{q^G_{a,i}(t_G) \mid a \in \mathscr{A} \text{ and } i \in \mathscr{V}_G\}$, where $q^G_{a,i}(t_G)$ is the average number of requests for a replica of application a waiting to be processed on subsystem i at the beginning of time step t_G.

At the current global time step t_G, we estimate $q^G_{a,i}(t_G)$ as the averaged values observed by the local controller of subsystem i. Equation (6.3) presents this estimation, where $q^i_{a,n}(t_L)$ is the observed queue length at beginning of local time step t_L for a replica of application a placed on a node n inside subsystem i, and $\rho^i_{a,n}(t_L-1)$ is a binary variable specified by the local controller related to the application placement.

$$q^G_{a,i}(t_G) = \begin{cases} \dfrac{\sum_{n\in\mathcal{V}_i} q^i_{a,n}(t_L)\rho^i_{a,n}(t_L-1)}{\sum_{n\in\mathcal{V}_i} \rho^i_{a,n}(t_L-1)} & \text{if } \sum_{n\in\mathcal{V}_i} \rho_{a,n}(t_L-1) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

Next, we estimate system states within the prediction horizon with length H_G employed by the global controller.

6.2.1.1 System State Estimation

Given a system state $s_G(k)$ and environment input $e_G(k)$ at time $k \in [t_G, t_G + H_G - 1]$, we estimate the next system state $s_G(k+1)$ for the behavioral model $\Phi_G(\cdot)$ when a control input $c_G(k)$ is applied to the system at time step t_G.

In a system state $s_G(k+1)$, the response time $d^G_{a,i,j}(k+1)$ comprises the network delay $d^{G,\text{net}}_{a,i,j}(k+1)$ and processing delay $d^{G,\text{proc}}_{a,j}(k+1)$, as shown in Equation (6.4). We do not consider the initialization delay as the migration delay can be neglected due to the longer duration of a global time step. We estimate both network and processing delays in the remainder of this subsection.

$$d^G_{a,i,j}(k+1) = d^{G,\text{net}}_{a,i,j}(k+1) + d^{G,\text{proc}}_{a,j}(k+1) \tag{6.4}$$

Network delay $d^{G,\text{net}}_{a,i,j}(k+1)$. Regarding the source and target subsystems, we have two cases to consider: they are the same (i.e., $i = j$), or they are different subsystems (i.e., $i \neq j$). For the network delay between two different subsystems, we estimate it as the network delay between the central nodes of these subsystems. The central node i_C of a subsystem i is defined as the node with the lowest network delay between all other nodes inside the subsystem. We adopt this network delay estimation based on central nodes to have an approximate distance between any two nodes in different subsystems. Moreover, Equation (6.5) determines this central node i_C, where $\mathscr{P}_{m,n}$ is the shortest path between nodes m and n, and L^{bw}_l and L^{pd}_l are the bandwidth and propagation delay of a link l in this path, respectively.

$$\begin{aligned} i_C &= \arg\min_{n\in\mathcal{V}_i} \left\{ \sum_{m\in\mathcal{V}_i} D_{\text{net}}(1,m,n) \right\} \\ D_{\text{net}}(x,m,n) &= \begin{cases} 0 & \text{if } m = n \\ \displaystyle\sum_{l\in\mathscr{P}_{m,n}} \frac{x}{L^{\text{bw}}_l} + L^{\text{pd}}_l & \text{otherwise} \end{cases} \end{aligned} \tag{6.5}$$

For the case where the source and target subsystems are the same, we estimate the network delay as being inversely proportional to the number of application replicas placed in the subsystem. The reason for this estimation is as follows. On the one hand, the network delay is zero when an application replica is placed in every node inside the subsystem. The idea is that loads are processed in their source nodes and, thus, there is no request transmission over the network between nodes inside the subsystem. On the other hand, the network delay is the average network delay between the central node and other nodes inside the subsystem when there is at most one application replica placed in this subsystem. For other numbers of application replicas placed in the subsystem, the network delay follows a linear relationship between these two extreme values.

Finally, Equation (6.6) expresses the network delay based on the above-discussed two cases for the source and target subsystems.

$$d_{a,i,j}^{G,\text{net}}(k+1) = \begin{cases} D_\text{net}(A_a^\text{data}, i_C, j_C) & \text{if } i \neq j \\ \left(1 - \frac{\rho_{a,i}^G(k)}{|\mathscr{V}_i|}\right)\left(\frac{1}{|\mathscr{V}_i|-1}\right)\sum_{m\in\mathscr{V}_i} D_\text{net}(A_a^\text{data}, m, i_C) & \text{if } i = j \text{ and } |\mathscr{V}_i| \geq 2 \\ 0 & \text{otherwise} \end{cases} \quad (6.6)$$

Processing delay $d_{a,j}^{G,\text{proc}}(k+1)$. Like the centralized approach, the aggregated request arrival rate before load distribution $\Lambda_{a,j}^G(k)$ for application a in all nodes inside subsystem j is also given by the predicted environment input $\widehat{Q}_{a,j}^G(k)$ and the estimated queue length $q_{a,j}^G(k)$, as shown in Equation (6.7). Moreover, we simplify the load after distribution by considering the request are uniformly distributed among all application replicas placed in a subsystem. In this way, Equation (6.8) describes the request arrival rate after load distribution in a single replica of application a placed in subsystem j, and Equation (6.9) gives the processing rate $\mu_{a,j}^G(k)$ of this replica.

$$\Lambda_{a,j}^G(k) = \widehat{Q}_{a,j}^G(k) + \frac{q_{a,j}^G(k)}{T_G}\rho_{a,j}^G(k-1) \quad (6.7)$$

$$\lambda_{a,j}^G(k) = \begin{cases} \frac{1}{\rho_{a,j}^G(k)}\sum_{i\in\mathscr{V}_G} \delta_{a,i,j}^G(k)\Lambda_{a,i}^G(k) & \text{if } \rho_{a,j}^G(k) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (6.8)$$

$$\frac{1}{\mu_{a,j}^G(k)} = \frac{A_a^\text{work}}{A_a^\text{CPU}\left(\lambda_{a,j}^G(k)\right)} \quad (6.9)$$

As we assume that every replica of application a placed in subsystem j has the same arrival and processing rate, then each of these replicas has the same processing time and queue length. According to the M/M/1 queueing model, Equations (6.10) and (6.11) determine the average processing time and queue length of an application replica, respectively.

$$d_{a,j}^{G,\text{proc}}(k+1) = \frac{1}{\mu_{a,j}^{G}(k) - \lambda_{a,j}^{G}(k)} \tag{6.10}$$

$$q_{a,j}^{G}(k+1) = \frac{\lambda_{a,j}^{G}(k)}{\mu_{a,j}^{G}(k) - \lambda_{a,j}^{G}(k)} - \frac{\lambda_{a,j}^{G}(k)}{\mu_{a,j}^{G}(k)} \tag{6.11}$$

Finally, the behavioral model $\Phi_G(\cdot)$ is specified by Equations (6.4) to (6.11). That is, it estimates the next system state $s_G(k+1)$ given the current state $s_G(k)$, a control input $c_G(k)$, and an environment input $e_G(k)$.

6.2.2 Optimization Formulation

Let F_G be a list of performance-related functions to be optimized. Then, the global controller solves problem (6.12) at the beginning of each time step t_G. In this problem, Equations (6.12b) to (6.12g) are similar to the constraints of optimization problem (5.12) in the centralized approach. Additionally, Equation (6.12h) ensures that the number of application replicas placed in a subsystem does not exceed the number of nodes inside this subsystem. The reason for this is that only a single replica of an application is placed per node. Constraint (6.12i) guarantees that all applications are deployed in the cloud subsystem, which is the subsystem containing the cloud node. This constraint allows that local controllers can always dispatch load to the cloud node. Equation (6.12j) limits that resources allocated to a single application replica do not surpass the average node capacity in a subsystem.

$$\min_{c_G(k)\in\mathscr{C}} \sum_{k=t_G}^{t_G+H_G-1} F_G\left(s_G(k+1), c_G(k), e_G(k)\right) \tag{6.12a}$$

$$\text{s.t.} \quad s_G(k+1) = \Phi_G\left(s_G(k), c_G(k), e_G(k)\right) \qquad \forall k \in [t_G, t_G+H_G) \tag{6.12b}$$

$$1 \leq \sum_{j\in\mathscr{V}_G} \rho^G_{a,j}(k) \leq A^{\max}_a \qquad \forall a \in \mathscr{A}, \forall k \in [t_G, t_G+H_G) \tag{6.12c}$$

$$\delta^G_{a,i,j}(k) \leq \rho^G_{a,j}(k) \qquad \forall a \in \mathscr{A}, \forall i,j \in \mathscr{V}_G, \forall k \in [t_G, t_G+H_G) \tag{6.12d}$$

$$\sum_{j\in\mathscr{V}_G} \delta^G_{a,i,j}(k)\Lambda^G_{a,i}(k) = \Lambda^G_{a,i}(k) \qquad \forall a \in \mathscr{A}, \forall i \in \mathscr{V}_G, \forall k \in [t_G, t_G+H_G) \tag{6.12e}$$

$$\sum_{a\in\mathscr{A}} \rho^G_{a,j}(k) A^r_a\left(\lambda^G_{a,j}(k)\right) \leq N^{\text{cap}}_{j,r} \qquad \forall r \in \mathscr{R}, \forall j \in \mathscr{V}_G, \forall k \in [t_G, t_G+H_G) \tag{6.12f}$$

$$\lambda^G_{a,j}(k) < \mu^G_{a,j}(k) \qquad \forall a,j \left(\rho^G_{a,j}(k) > 0\right), \forall k \in [t_G, t_G+H_G) \tag{6.12g}$$

$$0 \leq \rho^G_{a,j}(k) \leq |\mathscr{V}_j| \qquad \forall a \in \mathscr{A}, \forall j \in \mathscr{V}_G, \forall k \in [t_G, t_G+H_G) \tag{6.12h}$$

$$\rho^G_{a,cloud}(k) \geq 1 \qquad \forall a \in \mathscr{A}, \forall k \in [t_G, t_G+H_G) \tag{6.12i}$$

$$\rho^G_{a,j}(k) A^r_a\left(\lambda^G_{a,j}(k)\right) \leq \rho^G_{a,j}(k)\frac{N^{\text{cap}}_{j,r}}{|\mathscr{V}_j|} \quad \forall a,j,r \in \mathscr{A}\times\mathscr{V}_G\times\mathscr{R}, \forall k \in [t_G, t_G+H_G) \tag{6.12j}$$

There are fewer decision variables in problem (6.12) than in centralized optimization problem (5.12) as long as the number of subsystems is much less than the number of nodes in the entire system, i.e., $|\mathscr{V}_G| \ll |\mathscr{V}|$. Hence, the global controller is more scalable than the centralized controller and can still maintain some control of the entire system. We describe this control in detail in the rest of this subsection.

Once the problem (6.12) is solved, the global controller uses the obtained solution as a sequence $\pi_c = \{c_G(k) \mid k \in [t_G, t_G+H_G-1]\}$ of control inputs to indirectly control the system by delimiting control actions of local controllers. In the obtained control sequence, we use the first control input $c_G(t_G) = (\rho_G(t_G), \delta_G(t_G))$ to define upper and lower limits related to application placement and load distribution decisions for each local controller. More specifically, these limits are described as follows:

– **Maximum Application Replicas** $A^{\max}_{a,i}$. For each subsystem i and application a, Equation (6.13) specifies the maximum number of application replica that a local controller can place in this subsystem based on the control input selected by the global controller.

$$A^{\max}_{a,i} = \rho^G_{a,i}(t_G) \tag{6.13}$$

– **Minimum Application Replicas** $A^{\min}_{a,i}$. It is the minimum number of replicas of application a that should be placed in subsystem i. As shown in Equation (6.14), this lower limit is just used to ensure that all applications are deployed in the cloud subsystem.

$$A_{a,i}^{\min} = \begin{cases} 1 & \text{if } i = cloud \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

– **Maximum Load Distribution** $\Lambda_{a,i,j}^{\max}$. Equation (6.15) specifies $\Lambda_{a,i,j}^{\max}$ as the maximum load (i.e., requests rate) of an application a that can be dispatched from a subsystem i to another subsystem j. Moreover, we do not impose limits to loads dispatched to the cloud or the same subsystem.

$$\Lambda_{a,i,j}^{\max} = \begin{cases} \infty & \text{if } j = cloud \text{ or } i = j \\ \delta_{a,i,j}^{G}(t_G)\Lambda_{a,i}^{G}(t_G) & \text{otherwise} \end{cases} \tag{6.15}$$

The global controller then informs the aforementioned limits to local controllers as constraints that should be satisfied by their control decisions until the next global time step when the above decision-making process may adjust these limits.

6.2.3 Control Algorithm

Although problem (6.12) is reduced in terms of decision variables when compared with the centralizer optimization problem, it is yet an MINLP problem and, consequently, NP-Hard. Hence, we can apply the genetic algorithm BRKGA+NSGA-II to find sub-optimal solutions for problem (6.12). In the same fashion as the centralized approach, we first propose a genetic-based algorithm to solve the problem for a single time step. Then, this algorithm can be extended to consider a prediction horizon with multiple time steps.

In order for algorithm BRKGA+NSGA-II to solve problem (6.12) when the prediction horizon $H_G = 1$, we need to define a new chromosome representation and decoder algorithm. This new design is necessary because the global controller employs integer variables for the application placement decisions as multiple replicas of an application can be deployed in a subsystem. Hence, the chromosome for the one-time step algorithm used by the global controller is a random-key vector represented as follows:

$$\begin{aligned} C = \Big[& C^{\mathrm{I}}_1, C^{\mathrm{I}}_2, \dots, C^{\mathrm{I}}_{|\mathscr{A}|}, \\ & C^{\mathrm{II}}_1, C^{\mathrm{II}}_2, \dots, C^{\mathrm{II}}_{|\mathscr{A}|}, \\ & C^{\mathrm{III}}_{1,1}, C^{\mathrm{III}}_{1,2}, \dots, C^{\mathrm{III}}_{1,|\mathscr{V}_G|}, \dots, C^{\mathrm{III}}_{|\mathscr{A}|,1}, C^{\mathrm{III}}_{|\mathscr{A}|,2}, \dots, C^{\mathrm{III}}_{|\mathscr{A}|,|\mathscr{V}_G|}, \\ & C^{\mathrm{IV}}_{1,1}, C^{\mathrm{IV}}_{1,2}, \dots, C^{\mathrm{IV}}_{1,|\mathscr{V}_G|}, \dots, C^{\mathrm{IV}}_{|\mathscr{A}|,1}, C^{\mathrm{IV}}_{|\mathscr{A}|,2}, \dots, C^{\mathrm{IV}}_{|\mathscr{A}|,|\mathscr{V}_G|} \\ & C^{\mathrm{V}}_{1,1}, C^{\mathrm{V}}_{1,2}, \dots, C^{\mathrm{V}}_{1,|\mathscr{V}_G|}, \dots, C^{\mathrm{V}}_{|\mathscr{A}|,1}, C^{\mathrm{V}}_{|\mathscr{A}|,2}, \dots, C^{\mathrm{V}}_{|\mathscr{A}|,|\mathscr{V}_G|} \Big] \end{aligned}$$

where the five parts of the above representation are detailed below:

1. C^{I}_a is related to the total number of replicas of application a to be placed in the entire system.
2. C^{II}_a is the fraction of subsystems to be selected as candidates for deployment places of application a.
3. $C^{\mathrm{III}}_{a,j}$ is the priority to select a subsystem j to be a deployment place of application a.
4. $C^{\mathrm{IV}}_{a,j}$ is related to the number of replicas of application a to be placed in subsystem j.
5. $C^{\mathrm{V}}_{a,i}$ is the priority to distribute requests for application a from a subsystem i.

Compared with the one-time step chromosome for the centralized approach presented in Section 5.3.1, the above chromosome representation has two additional parts, C^{I} and C^{IV}. These additional parts address the issue that the number of subsystems selected to host application replicas does not imply knowing the total number of replicas to be placed in the entire system because a single subsystem can host multiple replicas. Therefore, it is necessary to specify the total number of application replicas placed in the entire system and each subsystem.

Algorithm 7 decodes a vector with the above chromosome representation into a control input for problem (6.12). This algorithm is an adaptation of Algorithm 3 to take into account integer variables for the placement decisions. The main difference between these two algorithms is the first step. For each application, the first step of Algorithm 7 not only selects a subset of subsystems as candidates to host the application but also defines the number of application replicas that should be placed in each selected subsystem. For this, lines 4 and 5 determine the number of replicas and subsystems based on C^{I} and C^{II}, respectively. Then, line 6 selects the subsystems as candidates to host a specific application by C^{III}. The cloud subsystem is also added to the selected subsystems set in line 7. Next, it is ensured that each selected subsystem can host at least one replica of the specified application between lines 8 to 10. Finally, from lines 12 to 16, the total amount of replicas is split among the selected subsystems based on

Algorithm 7: Chromosome decoder for the global one-time step representation

```
Data: individual, $s_G(k)$, $e_G(k)$
Result: Control input $c_G(k) = (\rho_G(k), \delta_G(k))$
initialize $\rho^G_{a,j}(k)$, $\delta^G_{a,i,j}(k)$, $\lambda^G_{a,j}(k) \leftarrow 0$;
$C^{\mathrm{I}}, C^{\mathrm{II}}, C^{\mathrm{III}}, C^{\mathrm{IV}}, C^{\mathrm{V}} \leftarrow$ individual.chromosome;
forall $a \in \mathscr{A}$ do /* Step I: Subsystems Selection                  */
    $r \leftarrow \min(\sum_{i \in \mathscr{V}_G} |\mathscr{V}_i|, \lceil C^{\mathrm{I}}_a (A^{\max}_a - 1) \rceil)$;
    $z \leftarrow \min(r, \lceil C^{\mathrm{II}}_a (|\mathscr{V}_G| - 1) \rceil)$;
    $V_a \leftarrow$ select $z$ subsystems with higher $C^{\mathrm{III}}_{a,j}$, $j \in \mathscr{V}_G \setminus \{cloud\}$;
    $V_a \leftarrow V_a \cup \{cloud\}$;
    forall $j \in V_a$ do
        $P_{a,j} \leftarrow 1$;
        $C^{\mathrm{IV}}_{a,j} \leftarrow (C^{\mathrm{IV}}_{a,j} + 1)/(|V_a| + \sum_{i \in V_a} C^{\mathrm{IV}}_{a,i})$;      // normalized value
    $r \leftarrow \min(r, \sum_{i \in V_a} |\mathscr{V}_i|) - |V_a|$;    // remaining replicas to be distributed
    while $r > 0$ do
        forall $j \in V_a$ do
            $p \leftarrow \max(1, \lfloor r C^{\mathrm{IV}}_{a,j} \rfloor)$;
            if $P_{a,j} + p \leq |\mathscr{V}_j|$ and $r > 0$ then
                $P_{a,j} \leftarrow P_{a,j} + p$; $r \leftarrow r - p$;

/* Step II: Load Distribution                                                  */
$L \leftarrow$ list of pairs $(a,i) \in \mathscr{A} \times \mathscr{V}_G$ sorted by $C^{\mathrm{V}}_{a,i}$ in descending order;
forall $(a,i) \in L$ do
    $r \leftarrow \Lambda^G_{a,i}(k)$; $\lambda^* \leftarrow \Lambda^G_{a,i}(k) \lambda_{\%}$;
    forall $j \in V_a$ do
        $d^G_{a,i,j}(k+1) \leftarrow$ by Equation (6.4) and $c_G(k) = \big(\rho_G(k) = \{P_{a,j}\}, \delta_G(k)\big)$;
    sort subsystems $j \in V_a$ by $d^G_{a,i,j}(k+1)$ in ascending order;
    while $r > 0$ do
        forall $j \in V_a$ do
            $l \leftarrow \lambda^G_{a,j}(k) + \lambda^*/P_{a,j}$;
            if $\big(\rho^G_{a,j}(k) = P_{a,j}, \lambda^G_{a,j}(k) = l\big)$ respects (6.12f), (6.12g), and (6.12j) then
                $\rho^G_{a,j}(k) \leftarrow P_{a,j}$; $\lambda^G_{a,j}(k) \leftarrow l$;
                $\delta^G_{a,i,j}(k) \leftarrow \delta^G_{a,i,j}(k) + \lambda^*/\Lambda^G_{a,j}(k)$;
                $r \leftarrow r - \lambda^*$; $\lambda^* \leftarrow \min\{r, \lambda^*\}$;
                update free resources on subsystem $j$ given $c_G(k) = (\rho_G(k), \delta_G(k))$;
                break;

forall $a \in \mathscr{A}$ do /* Step III: Local Search                        */
    forall $j \in V_a$ do                                         // Pre-deployment
        if $\big(\rho^G_{a,j}(k) = P_{a,j}, \lambda^G_{a,j}(k)\big)$ respects (6.12f), (6.12g), and (6.12j) then
            $\rho^G_{a,j}(k) \leftarrow P_{a,j}$;
```

C^{IV}. The basic idea of this splitting is to give more application replicas to subsystems with high values for C^{IV}.

The second and third steps of Algorithm 7 work similarly to the corresponding steps of Algorithm 3. Nevertheless, instead of placing a single application replica in a subsystem, the number of replicas to be placed in a subsystem is defined by the first step of Algorithm 7.

Let $A = |\mathscr{A}|$, $G = |\mathscr{V}_G|$, $V = |\mathscr{V}| = \sum_{i \in \mathscr{V}_G} |\mathscr{V}_i|$, $R = |\mathscr{R}|$, and $L = \lceil 1/\lambda_{\%} \rceil$. Then, the first step of Algorithm 7 has complexity equals to $O(A(G \log G + V))$ due to the sorting procedure in line 6, and the number of replicas to split (lines 12 to 16) can be in the worst-case equal to the total number of nodes in the system. For the same reason used for Algorithm 3, the rest of Algorithm 7 (i.e., second and third steps) has complexity $O(AG(\log A + G \log G + LGR))$. Therefore, the overall complexity of Algorithm 7 is $O(AG(\log A + G \log G + LGR) + AV)$.

For a prediction horizon $H_G > 1$, the global controller can apply the simple or general sequence heuristic described in Section 5.3.2 to obtain a sequence of control inputs for problem (6.12).

6.3 Local Controller

A local controller can be seen initially as a centralized controller that only considers control actions regarding application placement and load distribution on nodes inside a subsystem. However, subsystems do not work, in general, isolated in the Edge Computing infrastructure and, thus, control decisions in a subsystem may affect other subsystems and vice versa. For instance, a local controller can decide to dispatch application requests from a subsystem to another due to the lack of available resources in the former subsystem to handle these requests. Therefore, a local controller should also take into account interactions between external subsystems and control restrictions imposed by the global controller.

We then design in this section a local controller based on the consideration mentioned above. First, we describe the dynamics of a subsystem in Subsection 6.3.1. Then, we formalize the optimization problem solved by each local controller in Subsection 6.3.2. Subsection 6.3.3 presents the proposed algorithm that obtains solutions for the formulated problem. Finally, we design how local controllers cooperate with each other to decide their control actions in Subsection 6.3.4.

6.3.1 System Dynamics

For a subsystem $i \in \mathscr{V}_G$, we incorporate the interaction with its neighbor subsystems to model the dynamic behavior of this subsystem. A subsystem j is said to be a neighbor of subsystem i if the global controller allowed them to exchange request loads for some application. Equation (6.16) determines the set $\mathscr{V}_{-i}$ of neighbor subsystems of subsystem i, where $\Lambda^{\max}_{a,i,j}$ and $\Lambda^{\max}_{a,j,i}$ are load distribution limits imposed by the global controller.

$$\mathscr{V}_{-i} = \left\{ j \in \mathscr{V}_G \setminus \{i\} \mid \exists a \in \mathscr{A},\, \Lambda^{\max}_{a,i,j} + \Lambda^{\max}_{a,j,i} > 0 \right\} \tag{6.16}$$

Given $\mathscr{V}_i$ as the set of all EC nodes inside a subsystem i, we define $\mathscr{V}_i' = \mathscr{V}_i \cup \mathscr{V}_{-i}$ as the extension of this set of nodes by including each neighbor subsystem represented as a single entity. Then, we use $\mathscr{V}_i'$ to model the dynamic behavior of subsystem i, including neighbor subsystems interactions, as follows:

$$s_i(t_L+1) = \Phi_L\left(s_i(t_L), c_i(t_L), e_i(t_L)\right) \tag{6.17}$$

where t_L is the time step index of a local controller and $\Phi_L(\cdot)$ is the subsystem behavioral model. For the subsystem state $s_i(t_L)$, control input $c_i(t_L)$, and environment input $e_i(t_L)$, we define them in the same way as in the centralized approach but using $\mathscr{V}_i'$ instead of $\mathscr{V}$. Control input $c_i(t_L) = (\rho_i(t_L), \delta_i(t_L))$ is related to application placement $\rho_i(t_L) = \{\rho^i_{a,n}(t_L) \in \{0,1\} \mid a \in \mathscr{A}$ and $n \in \mathscr{V}_i'\}$ and load distribution $\delta_i(t_L) = \{\delta^i_{a,m,n}(t_L) \in [0,1] \mid a \in \mathscr{A}$ and $m,n \in \mathscr{V}_i'\}$. Subsystem state $s_i(t_L) = (d_i(t_L), q_i(t_L))$ comprises the response time $d_i(t_L) = \{d^i_{a,m,n}(t_L) \mid a \in \mathscr{A}$ and $m,n \in \mathscr{V}_i'\}$ and queue length $q_i(t_L) = \{q^i_{a,n}(t_L) \mid a \in \mathscr{A}$ and $n \in \mathscr{V}_i'\}$, and the application load $Q_i(t_L) = \{Q^i_{a,n}(t_L) \mid a \in \mathscr{A}$ and $n \in \mathscr{V}_i'\}$ is the environment input $e_i(t_L)$. For a neighbor subsystem $n \in \mathscr{V}_{-i}$, $Q^i_{a,n}(t_L)$ can be seen as the request load for application a dispatched from this subsystem to be handle in subsystem i.

An important observation is that a local controller does not actually decide the placement of applications in neighbor subsystems, but we include this decision to simplify our model. For the same simplification reason, we represent a neighbor subsystem as a single entity instead of including all nodes inside this neighbor subsystem to reduce decision variables without disregarding interaction with neighbors.

Next, we describe how the behavioral model $\Phi_L(\cdot)$ estimates subsystem states within a prediction horizon with length H_L.

6.3.1.1 System State Estimation

By disregarding neighbor subsystems, a subsystem state $s_i(k+1)$ at a time step within a prediction horizon H_L, i.e., $k \in [t_L, t_L + H_L - 1]$, is estimated in the same fashion as a system state in the centralized approach. However, we need to extend this estimation to consider external influence from these neighbor subsystems on a local subsystem state. Hence, we further detail this extension in the rest of this subsection.

The response time $d^i_{a,m,n}(k+1)$ in a state of subsystem i involves, such as in the centralized approach, the network, processing, and initialization delays. For nodes inside subsystem i, i.e., m and $n \in \mathcal{V}_i$, these three factors are estimated as described for the centralized approach in Section 5.2.1. When a neighbor subsystem is involved, i.e., m or $n \in \mathcal{V}_{-i}$, we use its central node to represent it and still continue adopting the network and initialization delays estimation defined for the centralized approach. However, a processing delay in a subsystem is more complicated to estimate by the local controller of another subsystem because it does not know the dynamic behavior of external subsystems. Hence, a local controller uses an approximation of this behavior to determine processing delays in neighbor subsystems.

For a neighbor subsystem $n \in \mathcal{V}_{-i}$, its request rate before the distribution $\Lambda^i_{a,n}(k)$ is the load dispatched from this subsystem to subsystem i, as shown in Equation (6.18). Then, this load can be distributed among nodes inside subsystem i or to the cloud subsystem. After the load distribution, the request rate arriving in a neighbor subsystem includes not only loads dispatched from nodes inside subsystem i but also loads from other nodes over the Edge Computing infrastructure. Moreover, as a subsystem can host multiple replicas of an application, request arriving in this subsystem must be distributed among these replicas. Equation (6.19) determines the request arrival rate in a single application replica placed in the neighbor subsystem n, where $\lambda^{-i}_{a,n}(k)$ is the load from nodes outside subsystem i to this application replica, and $\alpha_{a,n}(k) \in [0,1]$ is a spreading factor. This spreading factor is applied to approximate how requests for application a are distributed among all replicas of this application placed in subsystem n at time step t_L.

$$\Lambda^i_{a,n}(k) = \widehat{Q}^i_{a,n}(k) \qquad n \in \mathcal{V}_{-i} \tag{6.18}$$

$$\lambda^i_{a,n}(k) = \lambda^{-i}_{a,n}(k) + \alpha_{a,n}(k) \sum_{m \in \mathcal{V}'_i} \delta^i_{a,m,n}(k) \Lambda^i_{a,m}(k) \qquad n \in \mathcal{V}_{-i} \tag{6.19}$$

Given the above arrival rate after load distribution, the processing rate, queue length, and processing delay for a neighbor subsystem can be estimated in the same fashion as for a node inside the subsystem. That is, we use the estimations presented in Section 5.2.1.1 for the centralized approach.

6.3.2 Optimization Formulation

Give a list F_L of performance-related functions to be optimization in each subsystem, the local controller of a subsystem $i \in \mathscr{V}_G$ independently solves the following problem at each local time step t_L:

$$\min_{c_i(k)\in\mathscr{C}} \sum_{k=t_L}^{t_L+H_L-1} F_L\left(s_i(k+1), c_i(k), e_i(k)\right) \tag{6.20a}$$

$$\text{s.t.}\, s_i(k+1) = \Phi_L\left(s_i(k), c_i(k), e_i(k)\right) \qquad \forall k \in [t_L, t_L+H_L) \tag{6.20b}$$

$$\delta^i_{a,m,n}(k) \le \rho^i_{a,n}(k) \qquad \forall a \in \mathscr{A}, \forall m,n \in \mathscr{V}_i', \forall k \in [t_L, t_L+H_L) \tag{6.20c}$$

$$\sum_{n\in\mathscr{V}_i'} \delta^i_{a,m,n}(k)\Lambda^i_{a,m}(k) = \Lambda^i_{a,m}(k) \qquad \forall a \in \mathscr{A}, \forall m \in \mathscr{V}_i', \forall k \in [t_L, t_L+H_L) \tag{6.20d}$$

$$\sum_{a\in\mathscr{A}} \rho^i_{a,n}(k)A^r_a\left(\lambda^i_{a,n}(k)\right) \le N^{\text{cap}}_{n,r} \qquad \forall r \in \mathscr{R}, \forall n \in \mathscr{V}_i, \forall k \in [t_L, t_L+H_L) \tag{6.20e}$$

$$\lambda^i_{a,n}(k) < \mu^i_{a,n}(k) \qquad \forall a,n\left(\rho^i_{a,n}(k) = 1\right), \forall k \in [t_L, t_L+H_L) \tag{6.20f}$$

$$A^{\min}_{a,i} \le \sum_{n\in\mathscr{V}_i} \rho^i_{a,n}(k) \le A^{\max}_{a,i} \qquad \forall a \in \mathscr{A}, \forall k \in [t_L, t_L+H_L) \tag{6.20g}$$

$$\sum_{m\in\mathscr{V}_i'} \delta^i_{a,m,n}(k)\Lambda^i_{a,m}(k) \le \Lambda^{\max}_{a,i,n}(k) \qquad \forall a \in \mathscr{A}, \forall n \in \mathscr{V}_{-i}, \forall k \in [t_L, t_L+H_L) \tag{6.20h}$$

$$\delta^i_{a,m,n}(k) = 0 \qquad \forall a \in \mathscr{A}, \forall m \in \mathscr{V}_{-i}, \forall n \in \mathscr{V}_{-i}\setminus\{cloud\}, k \ge t_L \tag{6.20i}$$

$$\rho^i_{a,n}(k) = P_{a,n}(k) \qquad \forall a \in \mathscr{A}, \forall n \in \mathscr{V}_{-i}, \forall k \in [t_L, t_L+H_L) \tag{6.20j}$$

where Equation (6.20c) limits load distribution to nodes or neighbor subsystems hosting the requested application, whereas Equation (6.20d) ensures the distribution of all loads. Equation (6.20e) assures that the maximum resource capacity of any node inside subsystem i is not violated, while Equation (6.20f) ensures queue stability for each application replica. Equations (6.20g) and (6.20h) set the constraints imposed by the global controller regarding application placement and load distribution, respectively. Constraint (6.20i) specifies that loads from neighbor subsystems are handled in nodes inside subsystem i or the cloud subsystem. As a result, this constraint prevents requests from getting stuck in a load redistribution loop between subsystems. Finally, Equation (6.20j) establishes that the decision variables for application placement in neighbor subsystems are actually constants. We discuss later in Section 6.3.4 how to obtain the values of these constants $P_{a,n}(k)$.

6.3.3 Control Algorithm

Due to the similarity between problem (6.20) and problem (5.12) defined for the centralized approach, the genetic algorithm BRKGA+NSGA-II can still solve the former problem with a one-time or multiple-time step prediction horizon, i.e., $H_L = 1$ or $H_L > 1$. Moreover, problem (6.20) has fewer decision variables than problem (5.12) as $|\mathcal{V}_i| \ll |\mathcal{V}|$ and $|\mathcal{V}_{-i}| < |\mathcal{V}_G| \ll |\mathcal{V}|$, which may result in faster decision making by a local controller. However, Algorithm 3, the chromosome decoder for the one-time step representation, needs fewer adaptations to include the new constraints introduced by problem (6.20). These adaptations are described below.

Algorithm 8 is the modification of Algorithm 3 that incorporates problem (6.20) specificity to decode a one-time step chromosome vector into a feasible control input. The main modifications are the following. For each application, the first step of Algorithm 8 selects a set of nodes inside a subsystem i as hosting candidates for the application whose set size is upper limited by the global controller. Then, the second step adds to the set of hosting candidates for an application, the cloud subsystem and neighbor subsystems containing at least one replica of this application. Next, application loads are distributed among this expanded candidates set while respecting constraints of problem (6.20). Finally, the last step of Algorithm 8 tries to place an application replica in every node selected in the first step. An important observation is that these modifications do not increase the algorithm complexity and, thus, Algorithm 8 has the same time complexity as Algorithm 3.

Furthermore, either the simple or general heuristic of Section 5.3.2 can be applied to solve problem (6.20) when $H_L > 1$ and Algorithm 8 as the one-time step decoder.

6.3.4 Cooperative Control Design

In order to solve problem (6.20), a local controller of a subsystem needs certain information regarding the control decisions in neighbor subsystems. For instance, this information includes the number of application replicas placed in a subsystem and the amount of request load dispatched to other subsystems. However, these control decisions are not known in advance by a local controller because local controllers make their decisions in parallel, i.e., at the same time.

The above issue can be addressed by allowing cooperation among local controllers through exchanges of information in their decision-making processes. Christofides *et al.* (2013)

Algorithm 8: Chromosome decoder for the local one-time step representation

```
Data: individual, $s_i(k)$, $e_i(k)$
Result: Control input $c_i(k) = (\rho_i(k), \delta_i(k))$
initialize $\rho^i_{a,n}(k)$, $\delta^i_{a,m,n}(k)$, $\lambda^i_{a,n}(k) \leftarrow 0$;
initialize $\rho^i_{a,n}(k) \leftarrow P_{a,n}(k)$, $n \in \mathscr{V}_{-i}$;
$C^{\mathrm{I}}, C^{\mathrm{II}}, C^{\mathrm{III}} \leftarrow$ individual.chromosome;
forall $a \in \mathscr{A}$ do /* Step I: Nodes Selection */
    $z \leftarrow \min(|\mathscr{V}_i|, \lceil C^{\mathrm{I}}_a A^{\max}_{a,i} \rceil)$;
    $V_a \leftarrow$ select $z$ nodes with higher $C^{\mathrm{II}}_{a,n}$, $n \in \mathscr{V}_i$;
/* Step II: Load Distribution */
$L \leftarrow$ list of pairs $(a,m) \in \mathscr{A} \times \mathscr{V}_i'$ sorted by $C^{\mathrm{III}}_{a,m}$ in descending order;
forall $(a,m) \in L$ do
    $r \leftarrow \Lambda^i_{a,m}(k)$;                // remaining load to be distributed
    $\lambda^* \leftarrow \Lambda^i_{a,m}(k)\lambda_{\%}$;                // load chunk
    $V_a' \leftarrow V_a \cup \{cloud\} \cup \{n \in \mathscr{V}_{-i} \mid \rho^i_{a,n}(k) = 1\}$;
    forall $n \in V_a'$ do
        update response time $d^i_{a,m,n}(k+1)$ given $c_i(k) = (\rho_i(k) = \{1\}, \delta_i(k))$;
    sort nodes $n \in V_a'$ by $d^i_{a,m,n}(k+1)$ in ascending order;
    while $r > 0$ do
        forall $n \in V_a'$ do
            $l \leftarrow \lambda^i_{a,n}(k) + \lambda^*$;
            if $\left(\rho^i_{a,n}(k) = 1, \lambda^i_{a,n}(k) = l\right)$ respects (6.20e), (6.20f), (6.20h)–(6.20j) then
                $\rho^i_{a,n}(k) \leftarrow 1$;
                $\lambda^i_{a,n}(k) \leftarrow l$;
                $\delta^i_{a,m,n}(k) \leftarrow \delta^i_{a,m,n}(k) + \lambda^*/\Lambda^i_{a,m}(k)$;
                $r \leftarrow r - \lambda^*$;
                $\lambda^* \leftarrow \min\{r, \lambda^*\}$;
                update free resources on node/subsystem $n$ given $c_i(k) = (\rho_i(k), \delta_i(k))$;
                break;
forall $a \in \mathscr{A}$ do /* Step III: Local Search */
    forall $n \in V_a$ do                // Pre-deployment
        if $\left(\rho^i_{a,n}(k) = 1, \lambda^i_{a,n}(k)\right)$ respects constraints (6.20e) and (6.20f) then
            $\rho^i_{a,n}(k) \leftarrow 1$;
```

present an iterative cooperation strategy for distributed Model Predictive Control. At each iteration of this strategy, each controller optimizes its own set of control inputs assuming that control inputs of other controllers are fixed to the last agreed value at the previous iteration. Then, all controllers share their resulting optimal control input sequences. Based on the newly received control sequences, the iteration process is repeated until a stopping criterion is satisfied. However, Christofides *et al.* (2013) specify that control decisions are exchanged among all controllers, which may cause scalability issues for the cooperative strategy if there are a large number of controllers. In our distributed control approach, a local controller only needs to send information about its control decisions to controllers of neighbor subsystems, generally a subset of all subsystems. Therefore, our implementation of the iterative cooperation strategy is the following:

1. At time step t, all controllers receive the monitored state $s(t)$ and environment input $e(t)$ from their subsystems.
2. At iteration it ($it \geq 1$):
   2.1 If it is the first iteration, each controller guesses the control information of its neighbor controllers[1].
   2.2 Each controller evaluates its own future control input sequence based on its local subsystem information ($s(t)$ and $e(t)$) and the last received control information of the neighbor controllers.
   2.3 Neighbor controllers exchange their latest future control information.
3. If a stopping criterion is satisfied, each controller applies its control input $c(t)$ into its subsystem. Otherwise, go to Step 2 and $it \leftarrow it + 1$.

In order to use the above iterative cooperation strategy in the decision-making process of each local controller in our EC system,, we need to specify these three aspects: (*i*) what information is shared, (*ii*) how to guess this information at the first iteration, and (*iii*) what is the stopping criterion.

As discussed in Sections 6.3.1 and 6.3.2, $P_{a,i}(k), \alpha_{a,i}(k), \widehat{Q}^{j}_{a,i}(k), \lambda^{-j}_{a,i}(k)$, and $\Lambda^{\max}_{a,j,i}(k)$ are parameters used by the local controller of a subsystem j that depend on neighbor subsystems $i \in \mathscr{V}_{-j}$. These parameters can be determined by neighbor subsystem controllers sharing information related to their control decisions. Thus, after solving problem (6.20), the local controller of subsystem i sends to each controller of its neighbor subsystems the information below:

– **Amount of Application Replicas** $\{A_{a,i}(k) = \sum_{n \in \mathscr{V}_i} \rho^{i}_{a,n}(k)\}$. For an application $a \in \mathscr{A}$, $A_{a,i}(k)$ is the total number of application replicas to be placed in nodes inside subsystem i at a time step k within the prediction horizon. According to this data, neighbor controllers of subsystem i update the application placement $P_{a,i}(k)$ and spreading factor $\alpha_{a,i}(k)$

[1] A neighbor controller is the local controller of a neighbor subsystem.

parameters as follows:

$$P_{a,i}(k) = \begin{cases} 1 & \text{if } A_{a,i}(k) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.21}$$

$$\alpha_{a,i}(k) = \begin{cases} \frac{1}{A_{a,i}(k)} & \text{if } A_{a,i}(k) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.22}$$

– **Amount of Load Dispatched** $\{\Lambda_{a,i,j}(k) = \sum_{n\in\mathscr{V}_i} \delta^i_{a,n,j}(k)\Lambda^i_{a,n}(k)\}$. Here, $\Lambda_{a,i,j}(k)$ is an estimation of the amount of load generated by users of application a in subsystem i that will be dispatched to a neighbor subsystem $j \in \mathscr{V}_{-i}$ at a time step k within the prediction horizon. The local controller of this neighbor subsystem j uses this information to update its environment inputs as below:

$$\widehat{Q}^j_{a,i}(k) = \Lambda_{a,i,j}(k) \tag{6.23}$$

– **Application Replica Workload** $\{\lambda_{a,i}(k) = \sum_{n\in\mathscr{V}_i} \lambda^i_{a,n}(k)/A_{a,i}(k)\}$. At a time step k within the prediction horizon, $\lambda_{a,i}(k)$ is the average load received by each replica of application a placed in subsystem i. Based on this data, the controller of a neighbor subsystem j updates the following parameter:

$$\lambda^{-j}_{a,i}(k) = \max\{0, \lambda_{a,i}(k) - \alpha_{a,i}(k) \sum_{n\in\mathscr{V}'_j} \delta^j_{a,n,i}(k)\Lambda^j_{a,n}(k)\} \tag{6.24}$$

– **Amount of External Load Not Handled Locally** $\{\Lambda_{a,j,-i}(k) = \sum_{n\in\mathscr{V}_{-i}} \delta^i_{a,j,n}(k)\Lambda^i_{a,j}(k)\}$. As the name suggests, $\Lambda_{a,j,-i}(k)$ is the amount of load for application a from a neighbor subsystem j to subsystem i that is not then distributed to nodes inside subsystem i at a time step k. The local controller of subsystem j uses this received information to update the maximum load that can be dispatched to other subsystems, as shown below:

$$\Lambda^{\max}_{a,j,i}(k) = \max\{0, \Lambda^{\max}_{a,j,i}(k) - \Lambda_{a,j,-i}(k)\} \tag{6.25}$$

At the first iteration, local controllers guess the values of the above-mentioned parameters as follows:

– $A_{a,i}(k) = A^{\max}_{a,i}$, where $A^{\max}_{a,i}$ is a parameter defined by the global controller, as described in Section 6.2.2.

– $\Lambda_{a,i,j}(k) = \Lambda^{\max}_{a,i,j}$, where $\Lambda^{\max}_{a,i,j}$ is another parameter determined by the global controller. The idea here is that a local controller assumes the worst case in which a neighbor subsystem dispatches the maximum load allowed by the global controller.

- $\lambda_{a,i}(k) = 0$. A local controller takes a simple assumption that application replicas in other subsystems are not busy, i.e., they do not have workloads.
- $\Lambda_{a,j,-i}(k) = 0$ and $\Lambda^{\max}_{a,j,i}(k) = \Lambda^{\max}_{a,j,i}$. Another simple assumption is that all external loads are handle on nodes inside a subsystem. Moreover, the maximum load that can be distributed to other subsystem is initially set to values defined by the global controller. Then, these values can only be reduced in the next iterations according to Equation (6.25). The idea behind this is to prevent a subsystem from sending more load to another subsystem that this latest one can handle.

Regarding the stopping criteria, we use the maximum number of iterations $it_{\max}$. Moreover, this parameter value should be smaller in order not to increase the decision-making time. When the maximum number of iterations is reached, each local controller applies the first control input of its latest obtained control sequence into its subsystem and also sends this input to the global controller.

6.4 Performance Analysis

This section presents the preliminary evaluation of our distributed control proposal when comparing its results with the centralized approach. First, Subsection 6.4.1 describes the evaluated algorithms. Next, Subsection 6.4.2 presents the performance metrics. Then, Subsection 6.4.3 details the experiment setup. Finally, we analyze the obtained experimental results in Subsection 6.4.4.

6.4.1 Evaluated Algorithmic Solutions

We consider the following algorithmic solutions for the experiment conducted in this chapter:

- ***Cloud***. It places all applications in the cloud node/subsystem.
- ***Centralized***. It is our centralized approach using BRKGA+NSGA-II and the general sequence heuristic detailed in Section 5.3.2. For this centralized algorithm, we use the same prediction horizon length and time step duration of local controllers in the distributed approach, i.e., $H = H_L$ and $T_S = T_L$.
- ***Coop***. It is our distributed approach. For the global controller, we set the prediction horizon $H_G = 1$, the time step duration $T_G = 2T_L$, and BRKGA+NSGA-II with Algorithm 7 as

the chromosome decoder. For the local controller, we also use BRKGA+NSGA-II but with Algorithm 8 as the control input decoder and the general sequence heuristic to obtain control sequences within the prediction horizon $H_L = 2$. Moreover, we allow that local controller cooperates by distributing load among subsystems and using the iterative cooperation strategy explained in Section 6.3.4. In this strategy, we set the maximum number of iterations to $it_{\max} \in \{1,2\}$.

– ***Non-Coop***. It is similar to *Coop* but without allowing loads to be distributed among subsystems, except for the cloud subsystem. That is, the global controller sets $\Lambda^{\max}_{a,i,j} = 0$ when $i \neq j$ and $j \neq cloud$. Thus, local controllers do not cooperate in their decision-making processes. Besides, application requests from a subsystem are handled either inside this subsystem or in the cloud subsystem.

6.4.2 Performance Metrics

Regarding the global controller, we use the deadline violation, operation cost, and weighted average response time defined in Equations (6.26), (6.27), and (6.28), respectively. The $f^G_{\text{dv}}(\cdot)$ and $f^G_{\text{cost}}(\cdot)$ functions are similar to the functions defined in Equations (5.13) and (5.14), respectively, but adapted for the global control context. Moreover, we select $f^G_{\text{dv}}(\cdot)$ as the primary objective in the preferred dominance operator, and we add $f^G_{\text{rt}}(\cdot)$ to have a non-primary conflicted objective for the operation cost function.

$$f^G_{\text{dv}}(s_G(k+1), c_G(k), e_G(k)) = \frac{\sum_{a\in\mathscr{A}}\sum_{i,j\in\mathscr{V}_G}\left[d^G_{a,i,j}(k+1) - A^{\text{rd}}_a\right]^+ \delta^G_{a,i,j}(k)\Lambda^G_{a,i}(k)}{\sum_{a\in\mathscr{A}}\sum_{i,j\in\mathscr{V}_G}\delta^G_{a,i,j}(k)\Lambda^G_{a,i}(k)} \tag{6.26}$$

$$f^G_{\text{cost}}(s_G(k+1), c_G(k), e_G(k)) = \sum_{a\in\mathscr{A}}\sum_{j\in\mathscr{V}_G}\rho^G_{a,j}(k)T_G\sum_{r\in\mathscr{R}}N^{\text{cost}}_{j,r}(A^r_a(\lambda^G_{a,j}(k))) \tag{6.27}$$

$$f^G_{\text{rt}}(s_G(k+1), c_G(k), e_G(k)) = \frac{\sum_{a\in\mathscr{A}}\sum_{i,j\in\mathscr{V}_G}d^G_{a,i,j}(k+1)\delta^G_{a,i,j}(k)\Lambda^G_{a,i}(k)}{\sum_{a\in\mathscr{A}}\sum_{i,j\in\mathscr{V}_G}\delta^G_{a,i,j}(k)\Lambda^G_{a,i}(k)} \tag{6.28}$$

For a local controller, we select the same objective functions (deadline violation, operation cost, and migration cost) as the centralized approach experiment in Chapter 5 but adjusted for the subsystem context. Equations (6.29), (6.30), and (6.31) present the deadline violation, operation cost, and migration cost adaptations for a subsystem $i \in \mathscr{V}_G$, respectively.

$$f^L_{\text{dv}}\left(s_i(k+1), c_i(k), e_i(k)\right) = \frac{\sum_{a\in\mathscr{A}}\sum_{m,n\in\mathscr{V}_i'}\left[d^i_{a,m,n}(k+1) - A^{\text{rd}}_a\right]^+ \delta^i_{a,m,n}(k)\Lambda^i_{a,m}(k)}{\sum_{a\in\mathscr{A}}\sum_{m,n\in\mathscr{V}_i'}\delta^i_{a,m,n}(k)\Lambda^i_{a,m}(k)} \tag{6.29}$$

$$f^L_{\text{cost}}\left(s_i(k+1), c_i(k), e_i(k)\right) = \sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}_i}\rho^i_{a,n}(k)T_L\sum_{r\in\mathscr{R}}N^{\text{cost}}_{n,r}\left(A^r_a\left(\lambda^i_{a,n}(k)\right)\right) \tag{6.30}$$

$$f^L_{\text{mig}}\left(s_i(k+1), c_i(k), e_i(k)\right) = \frac{\sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}_i'}A^{\text{D+R}}_a\left(\lambda^i_{a,n}(k)\right)\rho^i_{a,n}(k)\left(1-\rho^i_{a,n}(k-1)\right)}{\sum_{a\in\mathscr{A}}\sum_{n\in\mathscr{V}_i'}A^{\text{D+R}}_a\left(\lambda^i_{a,n}(k)\right)\rho^i_{a,n}(k)} \tag{6.31}$$

In order to compare the system performance between the centralized and distributed approaches, we need first to transform the distributed control decisions of all local controllers into a control decision of a centralized controller. In this way, we can use the deadline violation $f_{\text{dv}}(\cdot)$, operational cost $f_{\text{cost}}(\cdot)$, and migration cost $f_{\text{mig}}(\cdot)$ defined in Equations (5.13), (5.14), and (5.15), respectively, to evaluate both centralized and distributed approaches.

Equation (6.32) describes how to compose a control input $c(k) = (\rho(k), \delta(k))$ for the centralized approach given the control inputs selected by each local controller. In this equation, we form the application placement decision $\rho(k) = \{\rho_{a,n}(k)\}$ for the centralized approach by simply joining the placement variables of all subsystems. On the other hand, the load distribution $\delta(k) = \{\delta_{a,m,n}(k)\}$ between any two nodes needs to consider two main cases. If these two nodes are inside the same subsystems, we use the load distribution decision of the local controller responsible for this subsystem. Otherwise, if two nodes belong to different subsystems, the load distribution is the combination of a load distribution chain between neighbor subsystems.

$$\rho_{a,n}(k) = \rho^i_{a,n}(k) \quad a\in\mathscr{A}, n\in\mathscr{V}_i \tag{6.32a}$$

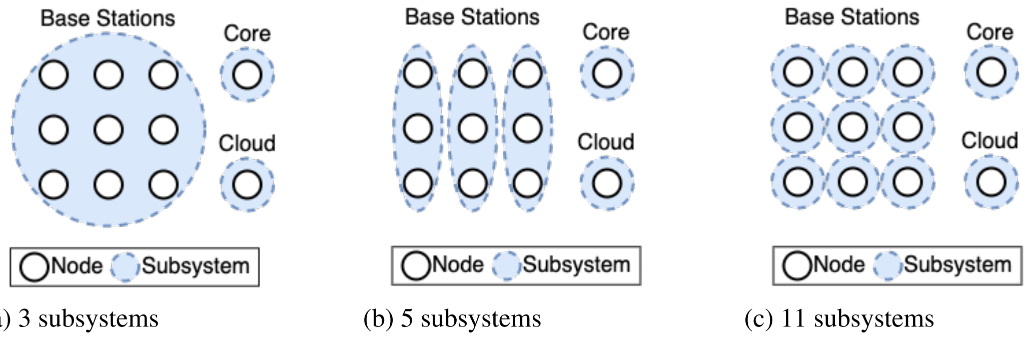
$$\delta_{a,m,n}(k) = \begin{cases} \delta^i_{a,m,n}(k) & \text{if } m,n\in\mathscr{V}_i \\ \delta^i_{a,m,j}(k)\delta^j_{a,i,n}(k) + \sum\limits_{l\in\mathscr{V}_{-i}\setminus\{j\}}\delta^i_{a,m,l}(k)\delta^l_{a,i,j}(k)\delta^j_{a,l,n}(k) & \text{otherwise}\end{cases} \tag{6.32b}$$

6.4.3 Analysis Setup

We use the same experiment scenario and parameters defined for the centralized approach analysis in Section 5.4.3. We adopt this scenario due to the computational limitation when executing distributed decision-making processes of several subsystems in a single machine. Nevertheless, it is still possible to evaluate the performance of the compared algorithms in this scenario.

For the distributed approach, we partitioned the Base Station nodes with hosting capabilities into different subsystems cases, as shown in Figure 41. In the first case, all 3×3 Base Stations are included in a single subsystem. The second case divides BSs into three subsystems containing the same number of nodes. In the last case, each BS has an exclusive subsystem. In each partition case, both core and cloud nodes also have their particular subsystem, and all subsystems have their own local controller.

Figure 41 – Subsystems scenario cases



(a) 3 subsystems (b) 5 subsystems (c) 11 subsystems

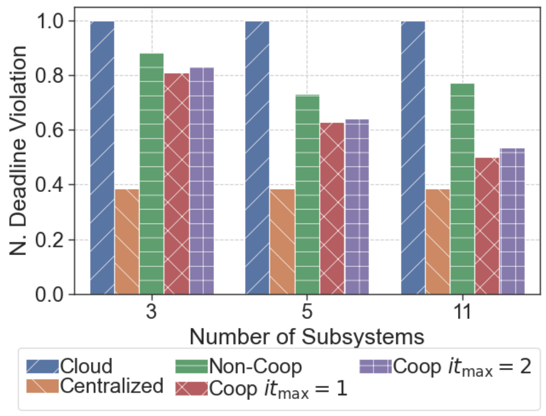
Source: Author.

6.4.4 Results and Discussion

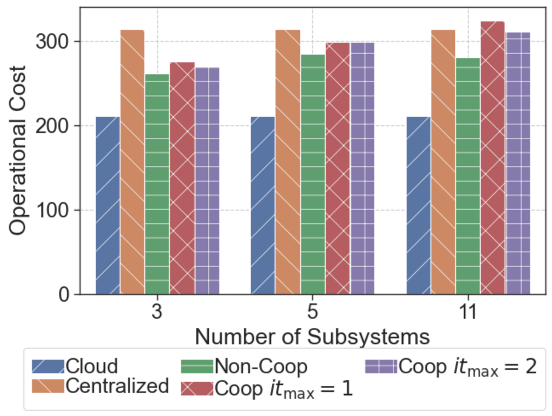
We evaluated the performance of the examined algorithms in a scenario with 10 applications and 10,000 users.

Figure 42a presents the obtained results for the average normalized deadline violations per time step in the three subsystems cases. To normalize the values of function $f_{\mathrm{dv}}(\cdot)$, we use the *Cloud* solution as the base. In this figure, we can observe that *Cloud* and *Centralized* solutions have the same results for different subsystem cases on the x-axis because they do not consider that system partition into subsystems. However, the *Coop* performance is significantly impacted by the number of nodes in a subsystem. In the case of three subsystems where all BS nodes are in a single subsystem, *Coop* with maximum iterations equal to 1 or 2 has about double of deadline violations than the *Centralized* solution, i.e., a 100% increase. On the other hand, in the case of 11 subsystems where each BS has its own subsystem, *Coop* presents a 35% average increase in deadline violations than the *Centralized* solution. Regarding the *Non-Coop* solution, it is also affected by the number of nodes in a subsystem, but its results do not improve as much as those of *Coop* when decreasing the number of BS nodes in a subsystem. Moreover, *Non-Coop* has worse results than *Coop* in every subsystem case. This outcome can be explained by the fact that *Non-Coop* does not distribute loads between subsystems, except for the cloud subsystem.

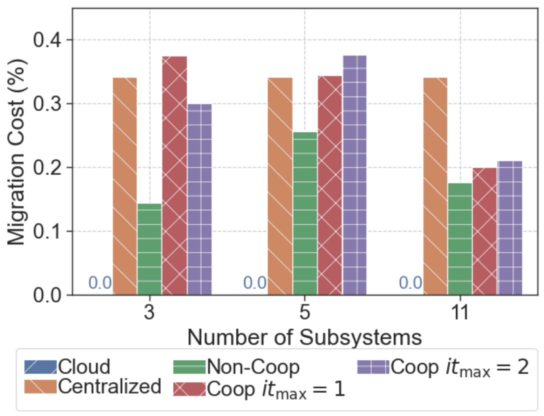
Figure 42 – Performance for different numbers of subsystems with 10 applications and 10,000 users



(a) Normalized Deadline Violation



(b) Operational Cost



(c) Migration Cost

Source: Author.

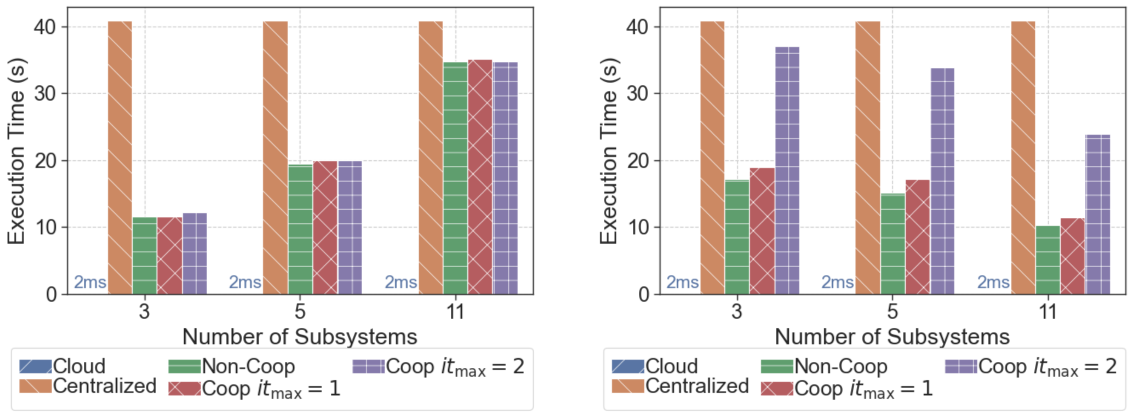
The average operational cost $f_{\text{cost}}(\cdot)$ per time slot is shown in Figure 42b. As expected, *Cloud* presents the lowest cost due to cloud resources being cheaper than in other locations. In contrast, the *Centralized* solution has the highest cost in almost all subsystem cases because it allocates more resources in BSs close to users, which we assume to be more expensive. *Coop* raises its operational cost by increasing the number of subsystems. This cost increase can be explained by *Coop* allocating more resources in the BS nodes to deploy applications. Another consequence of using more resources from BSs is the deadline violation reduction, as shown in Figure 42a. *Non-Coop* also presents a cost increase from 3 to 5 subsystems due to the increase of allocation of BS resources. However, we observe a slight cost reduction from 5 to 11 subsystems. As a subsystem can only dispatch load to itself or the cloud subsystem in *Non-Coop*, more loads will be dispatched to this cloud subsystem from a subsystem with fewer nodes and more resource competition to handle its user-generated load. Consequently, *Non-Coop* uses more cloud resources for the 11 subsystem case, where each subsystem contains a single node, than for 5 subsystems case.

In Figure 42c, we can see that the *Cloud* solution has no migration cost $f_{\text{mig}}(\cdot)$ as its placement decisions do not change over time. *Non-Coop* presents an increase then decrease pattern due to the same cloud usage reason explained for its operational cost results. Instead of changing a placement decision, *Non-Coop* tends to send surplus load to the cloud node and, thus, it has the second-lowest migration cost. We observe that the migration cost of *Coop* declines, especially for $it_{\max} = 1$, when the system is partitioned into more subsystems. This migration reduction can be caused by having fewer placement options in a subsystem with few nodes and, thus, a local controller does not change so much its placement decisions during a global time step duration.

Figure 43a shows the average execution time per global time step of the global controller in the distributed solutions, i.e., *Non-Coop* and *Coop*. For comparison reason, we also include in this figure the execution time of *Cloud* and Centralized *solutions*, which have a centralized control architecture. We can observe an execution time increase for the global controller in both *Non-Coop* and *Coop* solutions when partitioning the system into more subsystems. This time increase is due to the global chromosome decoder (Algorithm 7) complexity, which depends on the total number of subsystems. As the *Cloud* and *Centralized* do not rely on the system partition, it has constant execution times when increasing the number of subsystems. The *Cloud* solution has the best results, 2ms on average, as it does not search for control decisions, but instead, it has a fixed decision. In contrast, the *Centralized* solution presents the worst results because it has more control decision variables than the global controller in *Non-Coop* and *Coop* solutions as $|\mathscr{V}| = 11 \geq |\mathscr{V}_G| \in \{3, 5, 11\}$ and $H = 2 > H_G = 1$.

Figure 43b depicts the average execution time per local time step among all local controllers in a distributed solution. In this figure, the execution time decreases for local controllers in *Non-Coop* and *Coop* when there are more subsystems and, thus, fewer EC nodes per subsystem. Moreover, *Coop* with $it_{\max} = 2$ presents double execution time than *Coop* with $it_{\max} = 1$ because local controllers perform BRKGA+NSGA-II twice according to the iterative cooperation strategy when the maximum number of iterations is equal to 2. Similar to Figure 43a, we included the execution time results of *Cloud* and *Centralized* solutions, which are also the best and worst results in Figure 43b, respectively.

Figure 43 – Execution time of the hierarchical distributed controllers

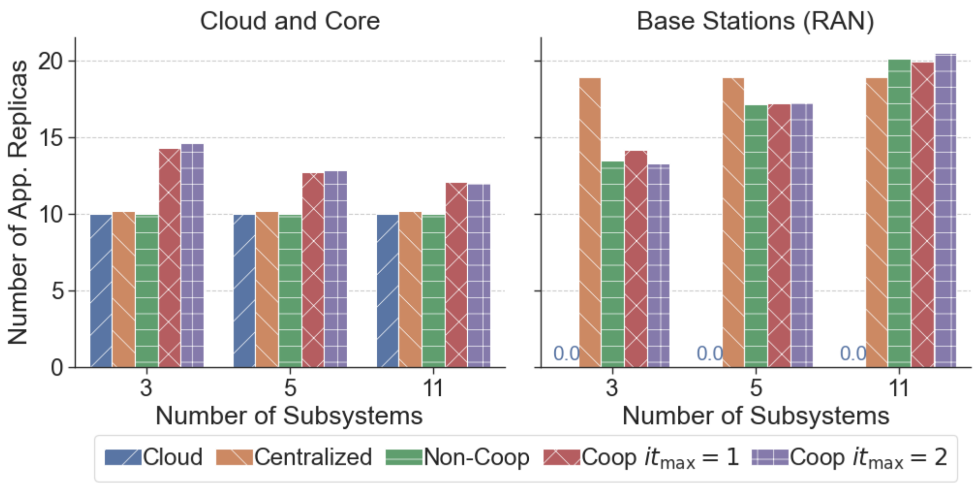


(a) Global Controller (b) Local Controllers

Source: Author.

Figure 44 shows the average overall number of application replicas placed in different network parts per time step. In this figure, we can see an increase in the number of application replicas placed on BS nodes for the distributed solutions (*Coop* and *Non-Coop*) by partitioning the system into more subsystems. Along with the results in Figure 42a, we can conclude that the global controller underestimates the necessary number of application replicas inside a subsystem to have low deadline violations if this subsystem contains many nodes. However, a large number of subsystems containing fewer nodes results in increasing the global controller execution time, as shown in Figure 43a. Therefore, how an EC system is partitioned affects the trade-off between system performance and decision scalability.

Figure 44 – Average number of application replicas placed in different network parts



Source: Author.

Another observation is that *Coop* with $it_{max} = 1$ and $it_{max} = 2$ show similar results in Figures 42, 43a and 44 for almost all compared cases. These results can be an indication that local controllers are dispatching the maximum load allowed by the global controller to other subsystems, which is the value estimated at the first iteration of the cooperation strategy described in Section 6.3.4.

6.5 Summary

In this chapter, we addressed the research question RQ3: *"How to make scalable and optimized (service placement, load distribution, and service migration) decisions in a large EC environment?"*. In order to answer this question, we proposed a hierarchical distributed limited look-ahead control approach that reduces the dimensionality of a centralized control decision. In this hierarchical distributed control, all EC nodes are partitioned into clusters called subsystems. Each subsystem is managed by a local controller responsible for control decisions (service placement, load distribution, and service migration) regarding nodes within this subsystem. On top of local controllers, the global controller receives simplified system-wide information and provides additional control restrictions for local controllers towards optimizing the overall system performance while satisfying global constraints. Furthermore, neighbor local controllers exchange subsystem information to coordinate their control decisions.

Preliminary evaluations shows that the performance, especially in terms of deadline violations, of our distributed approach that solves the control problem in a hierarchical cooperative fashion depends on how the EC system is partitioned into subsystems. In a subsystem with many nodes, the simplified system model used by the global controller simplifies or disregards some interaction aspects between nodes inside the subsystem that may affect system performance. In contrast, a system split into many subsystems containing fewer nodes has less interactions between nodes in a subsystem, but it may cause scalability issues in control decisions of the global controller. Therefore, the challenge is to consider detailed system dynamics while having scalable control decisions.

7 CONCLUSION

Edge Computing and 5G networks are promising technologies to enable a myriad of IoT applications, especially time-sensitive applications, by providing Cloud Computing capabilities, low latency, reduced core traffic load, low energy consumption, and local-context awareness. However, Edge Computing faces several challenges to become a reality. In particular, an important issue is to decide the ideal places (edge nodes or cloud servers) to deploy multiple applications or services while meeting their demands, respecting specified constraints, and optimizing desired performance metrics. Moreover, service placement is a non-trivial problem due to the vast distributed, heterogeneous, and dynamic EC environments.

In this thesis, we were interested in the service placement problem, including load distribution and service migration sub-problems, in the context of cellular networks with EC capabilities, such as 5G networks. Hence, we first studied related works according to their problem formulation, system model, controller design, and algorithm solution technique in order to identify research gaps. We then proposed decision-making approaches to cover the identified aspects not fully considered by related works, such as time-sensitive application requirements and performance metrics, multi-objective optimization, proactive control decisions, and distributed control architecture.

The remainder of this chapter is organized as follows. We summarize the main contributions of this thesis in Section 7.1. In Section 7.2, we discuss possible future work.

7.1 Contributions

Given the difficult task of jointly deciding service placement, load distribution, and service migration in a large-scale EC system, we addressed this problem throughout this thesis in increased complexity steps. First, we only considered service placement and load distribution in a static decision approach. Then, we included service migration in a dynamic and centralized decision approach. At last, we distributed the decision-making process to handle with the large-scale characteristic of an EC system. Therefore, these steps are related to our three main contributions.

In our first main contribution, we jointly formulated the static service placement and load distribution in Edge Computing as an optimization problem that takes into account diverse application characteristics (e.g., response deadline, resource demand, scalability, and availability) and nodes' resource constraint. The formulated problem aims to minimize SLA infringements caused by violations of the deadline requirement and as well as to optimize other conflicting objectives, such as operational cost and service availability. Then, we proposed an approach based on genetic algorithms called BRKGA+NSGA-II to obtain feasible solutions close to the Pareto optimal front. Moreover, we modified the Pareto dominance operator to prioritize time-sensitive applications. Our analytical analysis has shown that the proposed algorithm achieves deadline violation results close to the optimum of the MILP formulation and still outperforming the compared heuristics for the other analyzed objectives (operational cost and service availability).

In the second main contribution, we considered that application workload might vary in spatial and temporal domains. To handle this dynamic load, we proposed a centralized proactive controller that makes application placement, load distribution, and service migration decisions over time to optimize multiple performance-related objectives, while regarding decision readjustment costs. Then, we adapted BRKGA+NSGA-II to this dynamic control approach. Specifically, the adapted BRKGA+NSGA-II produces sequences of control actions over a look-ahead prediction horizon that optimize system performance based on the specified multi-objectives. Moreover, evaluations has shown that our proposal outperformed other benchmark algorithms in terms of deadline violations prevention while having similar operational costs, but it presented more service migrations. Nevertheless, the presented migration results are still low.

Our third contribution addressed the scalability issue of a centralized decision-making process in a large EC system. To this end, we designed a hierarchical distributed limited look-ahead control approach that reduces the dimensionality of the overall control problem by decomposing this problem into a set of local control problems solved in a hierarchical cooperative fashion. At the upper control layer, the global controller receives system-wide information and provides local control restrictions for the lower control layer towards optimizing the overall system performance while satisfying global constraints. The lower control layer is composed of local controllers that exchange information to coordinate their control decisions. Moreover, the global controller considers slower time scales and larger systems more abstractly, whereas local controllers consider faster time scales and smaller (sub)systems in a more detailed way. Preliminary evaluations showed that the proposed distributed control performance could be close to the centralized control performance depending on how the EC system is partitioned into subsystems. However, the trade-off between system performance and scalable decisions needs further analysis.

In summary, we proposed in this thesis decision-making approaches for service placement and its sub-problems (load distribution and service migration) that take into account distinct characteristics of EC (a large, distributed, heterogeneous, and dynamic environment) and time-sensitive application requirements.

Finally, Table 13 shows the publications as a primary or secondary result of this thesis contributions. A secondary publication means that it is not directly related to our contributions, but it helped us acquire knowledge during this doctoral research. Moreover, all codes of the experiments carried out in this thesis are available at https://bit.ly/2M062CF.

7.2 Perspectives

The work carried out throughout this thesis allows the emergence of some insights for potential future works, which are described below.

Network resource provisioning: In this thesis, we focus on allocating resources provided by computer-based nodes, such as computing, memory, and storage, to deploy applications efficiently. Moreover, some network aspects, such as bandwidth and routing path, can also be considered in resource management to improve application QoS. In this regard, the Network Slicing paradigm can play an essential role by providing on-demand end-to-end virtual network, including virtual network links and nodes, tailored to the requirements of a specific application or service. Therefore, the service placement decision-making process can be improved by integrating it with network slicing technologies.

Dynamic infrastructure and applications: In our dynamic service placement approaches, we consider that end-user devices are responsible for the EC system dynamics by changing their overall generated load. The dynamics of a system can also be related to infrastructure and application characteristics. For instance, an EC infrastructure can have a dynamic topology where nodes enter and leave the system over time, such as in a Vehicle-to-everything (V2X) network where vehicles act as nodes with hosting capabilities. Similarly, applications may enter and leave the system, and their requirements can also change over time. Therefore, a service placement approach can be improved by taking into account these dynamic infrastructure and application aspects.

Multi-component placement: In our service placement approaches, we consider that all functional components of an application required to handle user requests are placed

Table 13 – Primary and secondary publications of this thesis

Reference	Status	BR Qualis	AU CORE
Primary Publications			
A. M. Maia, Y. Ghamri-Doudane, D. Vieira and M. F. de Castro, "Optimized Placement of Scalable IoT Services in Edge Computing," 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Arlington, VA, USA, 2019, pp. 189-197.	published	A2	A
A. M. Maia, Y. Ghamri-Doudane, D. Vieira and M. F. de Castro, "A Multi-Objective Service Placement and Load Distribution in Edge Computing," 2019 IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, 2019, pp. 1-7.	published	A1	B
A. M. Maia, Y. Ghamri-Doudane, D. Vieira and M. F. de Castro, "Dynamic Service Placement and Load Distribution in Edge Computing," 2020 16th International Conference on Network and Service Management (CNSM), Izmir, Turkey, 2020, pp. 1-9.	published	A2	B
A. M. Maia, Y. Ghamri-Doudane, D. Vieira and M. F. de Castro, "An Improved Multi-Objective Genetic Algorithm with Heuristic Initialization for Service Placement and Load Distribution in Edge Computing", Computer Networks, v. 194, p. 108146, 2021.	published	A1	A
A Hierarchical Distributed Limited Look-ahead Control for Service Placement in Edge Computing. Under preparation to be submitted to a journal.	to be submitted		
Secondary Publications			
A. M. Maia, D. Vieira, M. F. de Castro, and Y. Ghamri-Doudane, "A fair QoS-aware dynamic LTE scheduler for machine-to-machine communication," Computer Communications, Volumes 89–90, 2016, Pages 75-86.	published	A2	C
R. M. Carvalho, R. M. Andrade, J. Barbosa, **A. M. Maia**, B. A. Junior, P. A. Aguilar, C. I. M. Bezerra, K. M. Oliveira, "Evaluating an IoT application using software measures," International Conference on Human-Computer Interaction (HCII), Vancouver, Canada, 2017, pp. 22-33	published	B2	B

Source: Author.

jointly as a single piece. We assumed this application design to avoid additional delays in response time when these components are placed in different locations over the EC infrastructure. For some time-tolerant applications, some of their functional components can be independently placed and then shared by application replicas or instances to reduce operational cost. Hence, we can enhance the decision-making process by having a fine-grained placement of an application composed of multiple functional components.

Multi-domain resource allocation: Another assumption along this thesis is that a single Infrastructure Provider owns and maintains the entire EC infrastructure from nodes near to end-user devices to cloud servers. In a more realistic scenario, multiple InPs offer resources in different parts of the network or in different geographic locations. For example, there are several mobile operators in different countries that can provide EC capabilities in their cellular networks in addition to diverse cloud providers, such as Google, Amazon, and Microsoft. Hence, we can consider the resource allocation for service placement in a multi-domain scenario, i.e., a scenario with multiple InPs.

AI-based control: In a large-scale EC system, it is hard to accurately model the dynamic behavior of the entire system given the complex interaction among infrastructure, applications, and end-user devices. On the one hand, a detailed model can imply many control decision options, which is not scalable. On the other hand, a simplified model reduces the control decision dimensionality, but it may omit some aspects that affect system performance. Therefore, we can alleviate the burden of explicitly model the system dynamics and still having scalable control decisions by using Artificial Intelligence (AI) techniques. For example, some Machine Learning techniques that combine Reinforcement Learning and Deep Learning, such as Deep Q-learning, are model-free and can provide real-time proactive control by offline training. Furthermore, we can also use AI techniques in our distributed approach to partition the EC system and its EC nodes in subsystems according to local contextual information (e.g., user density).

Experiments through test-bed or EC simulator: In order to have more realistic data, we can conduct experiments to evaluate the performance of our proposals through test-beds or Edge Computing simulator that supports simulation of both network and processing parts of an EC system.

BIBLIOGRAPHY

ABDEL-BASSET, M.; ABDEL-FATAH, L.; SANGAIAH, A. K. Chapter 10 - metaheuristic algorithms: A comprehensive review. In: SANGAIAH, A. K.; SHENG, M.; ZHANG, Z. (Ed.). **Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications**. [*S. l.*]: Academic Press, 2018, (Intelligent Data-Centric Systems). p. 185 – 231. ISBN 978-0-12-813314-9.

ABDELWAHED, S.; KANDASAMY, N.; NEEMA, S. Online control for self-management in computing systems. In: **Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.** [*S. l.*: *s. n.*], 2004. p. 368–375.

AL-FUQAHA, A.; GUIZANI, M.; MOHAMMADI, M.; ALEDHARI, M.; AYYASH, M. Internet of things: A survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys Tutorials**, v. 17, n. 4, p. 2347–2376, 2015.

ALLIANCE, N. 5g white paper. **Next generation mobile networks, white paper**, p. 1–125, 2015.

ASHTON, K. That ‘internet of things’ thing. **RFID journal**, v. 22, n. 7, p. 97–114, 2009.

BAI, J.; ABDELWAHED, S. Efficient algorithms for performance management of computing systems. In: **Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks FeBID. IEEE**. [*S. l.*: *s. n.*], 2009.

BAKTIR, A. C.; OZGOVDE, A.; ERSOY, C. How can edge computing benefit from software-defined networking: A survey, use cases, and future directions. **IEEE Communications Surveys Tutorials**, v. 19, n. 4, p. 2359–2391, Fourthquarter 2017.

BALAJI, S.; NATHANI, K.; SANTHAKUMAR, R. Iot technology, applications and challenges: A contemporary survey. **Wireless Personal Communications**, v. 108, n. 1, p. 363–388, Sep 2019. ISSN 1572-834X.

BARB, G.; OTESTEANU, M. 4g/5g: A comparative study and overview on what to expect from 5g. In: **2020 43rd International Conference on Telecommunications and Signal Processing (TSP)**. [*S. l.*: *s. n.*], 2020. p. 37–40.

BELOTTI, P.; KIRCHES, C.; LEYFFER, S.; LINDEROTH, J.; LUEDTKE, J.; MAHAJAN, A. Mixed-integer nonlinear optimization. **Acta Numerica**, Cambridge University Press, v. 22, p. 1, 2013.

BERTSEKAS, D. P. Nonlinear programming. **Journal of the Operational Research Society**, Taylor & Francis, v. 48, n. 3, p. 334–334, 1997.

BILAL, K.; KHALID, O.; ERBAD, A.; KHAN, S. U. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. **Computer Networks**, v. 130, p. 94 – 120, 2018. ISSN 1389-1286.

BONOMI, F.; MILITO, R.; ZHU, J.; ADDEPALLI, S. Fog computing and its role in the internet of things. In: **Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing**. New York, NY, USA: ACM, 2012. (MCC ’12), p. 13–16. ISBN 978-1-4503-1519-7.

BRANKE, J.; BRANKE, J.; DEB, K.; MIETTINEN, K.; SLOWIŃSKI, R. **Multiobjective optimization: Interactive and evolutionary approaches**. [*S. l.*]: Springer Science & Business Media, 2008. v. 5252.

BROGI, A.; FORTI, S.; GUERRERO, C.; LERA, I. How to place your apps in the fog: State of the art and open challenges. **Software: Practice and Experience**, v. 50, n. 5, p. 719–740, 2020.

BURER, S.; LETCHFORD, A. N. Non-convex mixed-integer nonlinear programming: A survey. **Surveys in Operations Research and Management Science**, v. 17, n. 2, p. 97 – 106, 2012. ISSN 1876-7354.

CARVALHO, J. O. de; TRINTA, F.; VIEIRA, D.; CORTES, O. A. C. Evolutionary solutions for resources management in multiple clouds: State-of-the-art and future directions. **Future Generation Computer Systems**, v. 88, p. 284 – 296, 2018. ISSN 0167-739X.

CHONG, E. K.; ZAK, S. H. **An introduction to optimization**. [*S. l.*]: John Wiley & Sons, 2004.

CHRISTOFIDES, P. D.; SCATTOLINI, R.; PEÑA, D. M. de la; LIU, J. Distributed model predictive control: A tutorial review and future research directions. **Computers & Chemical Engineering**, v. 51, p. 21 – 41, 2013. ISSN 0098-1354. CPC VIII.

COELLO, C. A. C. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. **Computer Methods in Applied Mechanics and Engineering**, v. 191, n. 11, p. 1245 – 1287, 2002. ISSN 0045-7825.

CPLEX, I. I. V12. 1: User's manual for cplex. **International Business Machines Corporation**, v. 46, n. 53, p. 157, 2009.

CUI, Y.; GENG, Z.; ZHU, Q.; HAN, Y. Review: Multi-objective optimization methods and application in energy saving. **Energy**, v. 125, p. 681 – 704, 2017. ISSN 0360-5442.

DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. **IEEE Transactions on Evolutionary Computation**, v. 6, n. 2, p. 182–197, April 2002. ISSN 1089-778X.

DOLUI, K.; DATTA, S. K. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In: **2017 Global Internet of Things Summit (GIoTS)**. [*S. l.*: *s. n.*], 2017. p. 1–6.

ETSI. **Mobile Edge Computing (MEC); Framework and reference architecture**. [*S. l.*], 2016. Accessed: 2020-10-02. Available at https://www.etsi.org/technologies/multi-access-edge-computing.

FARHADI, V.; MEHMETI, F.; HE, T.; PORTA, T. L.; KHAMFROUSH, H.; WANG, S.; CHAN, K. S. Service placement and request scheduling for data-intensive applications in edge clouds. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. [*S. l.*: *s. n.*], 2019. p. 1279–1287.

FILHO, M. C. S.; MONTEIRO, C. C.; INÁCIO, P. R.; FREIRE, M. M. Approaches for optimizing virtual machine placement and migration in cloud environments: A survey. **Journal of Parallel and Distributed Computing**, v. 111, p. 222 – 250, 2018. ISSN 0743-7315.

FLOYD, R. W. Algorithm 97: Shortest path. **Commun. ACM**, ACM, New York, NY, USA, v. 5, n. 6, p. 345–, jun. 1962. ISSN 0001-0782.

GAO, B.; ZHOU, Z.; LIU, F.; XU, F. Winning at the starting line: Joint network selection and service placement for mobile edge computing. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. [*S. l.*: *s. n.*], 2019. p. 1459–1467.

GEDEON, J.; BRANDHERM, F.; EGERT, R.; GRUBE, T.; MüHLHäUSER, M. What the fog? edge computing revisited: Promises, applications and future challenges. **IEEE Access**, v. 7, p. 152847–152878, 2019.

GOLDBERG, D. E.; HOLLAND, J. H. Genetic algorithms and machine learning. **Machine Learning**, v. 3, n. 2, p. 95–99, Oct 1988. ISSN 1573-0565.

GONÇALVES, J. F.; RESENDE, M. G. C. Biased random-key genetic algorithms for combinatorial optimization. **Journal of Heuristics**, v. 17, n. 5, p. 487–525, Oct 2011. ISSN 1572-9397.

GU, L.; ZENG, D.; GUO, S.; BARNAWI, A.; XIANG, Y. Cost efficient resource management in fog computing supported medical cyber-physical system. **IEEE Transactions on Emerging Topics in Computing**, v. 5, n. 1, p. 108–119, 2017.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of things (iot): A vision, architectural elements, and future directions. **Future Generation Computer Systems**, v. 29, n. 7, p. 1645 – 1660, 2013. ISSN 0167-739X.

HOLLAND, J. H. *et al.* **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. [*S. l.*]: MIT press, 1992.

HU, P.; DHELIM, S.; NING, H.; QIU, T. Survey on fog computing: architecture, key technologies, applications and open issues. **Journal of Network and Computer Applications**, v. 98, p. 27 – 42, 2017. ISSN 1084-8045.

HYNDMAN, R.; KHANDAKAR, Y. Automatic time series forecasting: The forecast package for r. **Journal of Statistical Software, Articles**, v. 27, n. 3, p. 1–22, 2008. ISSN 1548-7660.

HYNDMAN, R. J.; ATHANASOPOULOS, G. **Forecasting: principles and practice**. [*S. l.*]: OTexts, 2018.

ITU-R. **IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond**. [*S. l.*], 2015. Accessed: 2020-10-02. Available at http://www.itu.int/rec/R-REC-M.2083.

JAIN, R. **The art of computer systems performance analysis**. [*S. l.*]: John Wiley & Sons Chichester, 1991. v. 182.

KANDASAMY, N.; ABDELWAHED, S.; KHANDEKAR, M. A hierarchical optimization framework for autonomic performance management of distributed computing systems. In: **26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)**. [*S. l.*: *s. n.*], 2006. p. 9–9.

KATSALIS, K.; PAPAIOANNOU, T. G.; NIKAEIN, N.; TASSIULAS, L. Sla-driven vm scheduling in mobile edge computing. In: **2016 IEEE 9th International Conference on Cloud Computing (CLOUD)**. [*S. l.*: *s. n.*], 2016. p. 750–757.

KEKKI, S.; FEATHERSTONE, W.; FANG, Y.; KUURE, P.; LI, A.; RANJAN, A.; PURKAYASTHA, D.; JIANGPING, F.; FRYDMAN, D.; VERIN, G. *et al.* **MEC in 5G networks**. [*S. l.*], 2018. Accessed: 2020-10-02. Available at https://www.etsi.org/technologies/multi-access-edge-computing.

KHAN, W. Z.; AHMED, E.; HAKAK, S.; YAQOOB, I.; AHMED, A. Edge computing: A survey. **Future Generation Computer Systems**, v. 97, p. 219 – 235, 2019. ISSN 0167-739X.

KITANOV, S.; MONTEIRO, E.; JANEVSKI, T. 5g and the fog — survey of related technologies and research directions. In: **2016 18th Mediterranean Electrotechnical Conference (MELECON)**. [*S. l.*: *s. n.*], 2016. p. 1–6. ISSN 2158-8481.

LIU, H.; ELDARRAT, F.; ALQAHTANI, H.; REZNIK, A.; FOY, X. de; ZHANG, Y. Mobile edge cloud system: Architectures, challenges, and approaches. **IEEE Systems Journal**, v. 12, n. 3, p. 2495–2508, Sep. 2018. ISSN 1932-8184.

LORIDO-BOTRAN, T.; MIGUEL-ALONSO, J.; LOZANO, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. **Journal of grid computing**, Springer, v. 12, n. 4, p. 559–592, 2014.

MACH, P.; BECVAR, Z. Mobile edge computing: A survey on architecture and computation offloading. **IEEE Communications Surveys Tutorials**, v. 19, n. 3, p. 1628–1656, 2017.

MARABISSI, D.; MUCCHI, L.; FANTACCI, R.; SPADA, M.; MASSIMIANI, F.; FRATINI, A.; CAU, G.; YUNPENG, J.; FEDELE, L. A real case of implementation of the future 5g city. **Future Internet**, MDPI AG, v. 11, n. 1, p. 4, Dec 2018. ISSN 1999-5903.

MARTÍ, L.; GARCÍA, J.; BERLANGA, A.; MOLINA, J. M. A stopping criterion for multi-objective optimization evolutionary algorithms. **Information Sciences**, v. 367-368, p. 700 – 718, 2016. ISSN 0020-0255.

MASDARI, M.; NABAVI, S. S.; AHMADI, V. An overview of virtual machine placement schemes in cloud computing. **Journal of Network and Computer Applications**, v. 66, p. 106 – 127, 2016. ISSN 1084-8045.

MCCORMICK, G. P. Computability of global solutions to factorable nonconvex programs: Part i — convex underestimating problems. **Mathematical Programming**, v. 10, n. 1, p. 147–175, Dec 1976. ISSN 1436-4646.

MELL, P. M.; GRANCE, T. **SP 800-145. The NIST Definition of Cloud Computing**. Gaithersburg, MD, USA, 2011.

METZGER, F.; HOßFELD, T.; BAUER, A.; KOUNEV, S.; HEEGAARD, P. E. Modeling of aggregated iot traffic and its application to an iot cloud. **Proceedings of the IEEE**, v. 107, n. 4, p. 679–694, 2019.

MOURADIAN, C.; NABOULSI, D.; YANGUI, S.; GLITHO, R. H.; MORROW, M. J.; POLAKOS, P. A. A comprehensive survey on fog computing: State-of-the-art and research challenges. **IEEE Communications Surveys Tutorials**, v. 20, n. 1, p. 416–464, Firstquarter 2018.

NAHA, R. K.; GARG, S.; GEORGAKOPOULOS, D.; JAYARAMAN, P. P.; GAO, L.; XIANG, Y.; RANJAN, R. Fog computing: Survey of trends, architectures, requirements, and research directions. **IEEE Access**, v. 6, p. 47980–48009, 2018. ISSN 2169-3536.

NEELY, M. J. Stochastic network optimization with application to communication and queueing systems. **Synthesis Lectures on Communication Networks**, v. 3, n. 1, p. 1–211, 2010.

OUYANG, T.; ZHOU, Z.; CHEN, X. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. **IEEE Journal on Selected Areas in Communications**, v. 36, n. 10, p. 2333–2345, 2018.

PAN, J.; MCELHANNON, J. Future edge cloud and edge computing for internet of things applications. **IEEE Internet of Things Journal**, v. 5, n. 1, p. 439–449, Feb 2018. ISSN 2327-4662.

PARK, H.-S.; JUN, C.-H. A simple and fast algorithm for k-medoids clustering. **Expert Systems with Applications**, v. 36, n. 2, Part 2, p. 3336 – 3341, 2009. ISSN 0957-4174.

PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. Context aware computing for the internet of things: A survey. **IEEE Communications Surveys Tutorials**, v. 16, n. 1, p. 414–454, First 2014. ISSN 1553-877X.

PIETRI, I.; SAKELLARIOU, R. Mapping virtual machines onto physical machines in cloud computing: A survey. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 49, n. 3, p. 49:1–49:30, oct. 2016. ISSN 0360-0300.

PIRES, F. L.; BARÁN, B. A virtual machine placement taxonomy. In: **2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. [*S. l.*: *s. n.*], 2015. p. 159–168.

ROMAN, R.; LOPEZ, J.; MAMBO, M. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. **Future Generation Computer Systems**, v. 78, p. 680 – 698, 2018. ISSN 0167-739X.

SALAHT, F. A.; DESPREZ, F.; LEBRE, A. An overview of service placement problem in fog and edge computing. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 3, jun. 2020. ISSN 0360-0300.

SATYANARAYANAN, M.; BAHL, P.; CACERES, R.; DAVIES, N. The case for vm-based cloudlets in mobile computing. **IEEE Pervasive Computing**, v. 8, n. 4, p. 14–23, Oct 2009. ISSN 1536-1268.

SCHRIJVER, A. **Theory of linear and integer programming**. [*S. l.*]: John Wiley & Sons, 1998.

SCHULZ, P.; MATTHE, M.; KLESSIG, H.; SIMSEK, M.; FETTWEIS, G.; ANSARI, J.; ASHRAF, S. A.; ALMEROTH, B.; VOIGT, J.; RIEDEL, I.; PUSCHMANN, A.; MITSCHELE-THIEL, A.; MULLER, M.; ELSTE, T.; WINDISCH, M. Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture. **IEEE Communications Magazine**, v. 55, n. 2, p. 70–78, February 2017. ISSN 0163-6804.

SCIKIT-LEARN. **Generate isotropic Gaussian blobs for clustering**. 2020. Accessed: 2020-10-14. Available at https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html.

SHAFI, M.; MOLISCH, A. F.; SMITH, P. J.; HAUSTEIN, T.; ZHU, P.; SILVA, P. D.; TUFVESSON, F.; BENJEBBOUR, A.; WUNDER, G. 5g: A tutorial overview of standards, trials, challenges, deployment, and practice. **IEEE Journal on Selected Areas in Communications**, v. 35, n. 6, p. 1201–1221, 2017.

SHORTLE, J. F.; THOMPSON, J. M.; GROSS, D.; HARRIS, C. M. **Fundamentals of queueing theory**. [*S. l.*]: John Wiley & Sons, 2018. v. 399.

SINGH, A.; PAYAL, A.; BHARTI, S. A walkthrough of the emerging iot paradigm: Visualizing inside functionalities, key features, and open issues. **Journal of Network and Computer Applications**, v. 143, p. 111 – 151, 2019. ISSN 1084-8045.

SKARLAT, O.; NARDELLI, M.; SCHULTE, S.; BORKOWSKI, M.; LEITNER, P. Optimized iot service placement in the fog. **Service Oriented Computing and Applications**, Springer, v. 11, n. 4, p. 427–443, 2017.

SKARLAT, O.; NARDELLI, M.; SCHULTE, S.; DUSTDAR, S. Towards qos-aware fog service placement. In: **2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)**. [*S. l.*: *s. n.*], 2017. p. 89–96.

SPEARS, W. M.; JONG, K. D. D. **On the virtues of parameterized uniform crossover**. [*S. l.*], 1995.

SPINNEWYN, B.; MENNES, R.; BOTERO, J. F.; LATRÉ, S. Resilient application placement for geo-distributed cloud networks. **Journal of Network and Computer Applications**, v. 85, p. 14 – 31, 2017. ISSN 1084-8045. Intelligent Systems for Heterogeneous Networks.

SULTAN, S.; AHMAD, I.; DIMITRIOU, T. Container security: Issues, challenges, and the road ahead. **IEEE Access**, v. 7, p. 52976–52996, 2019.

SUN, X.; ANSARI, N. Green cloudlet network: A sustainable platform for mobile cloud computing. **IEEE Transactions on Cloud Computing**, v. 8, n. 1, p. 180–192, 2020.

TALEB, T.; SAMDANIS, K.; MADA, B.; FLINCK, H.; DUTTA, S.; SABELLA, D. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. **IEEE Communications Surveys Tutorials**, v. 19, n. 3, p. 1657–1681, thirdquarter 2017.

TÄRNEBERG, W.; MEHTA, A.; WADBRO, E.; TORDSSON, J.; EKER, J.; KIHL, M.; ELMROTH, E. Dynamic application placement in the mobile cloud network. **Future Generation Computer Systems**, v. 70, p. 163 – 177, 2017. ISSN 0167-739X.

TOCZÉ, K.; NADJM-TEHRANI, S. A taxonomy for management and optimization of multiple resources in edge computing. **Wireless Communications and Mobile Computing**, Hindawi, v. 2018, 2018.

URGAONKAR, R.; WANG, S.; HE, T.; ZAFER, M.; CHAN, K.; LEUNG, K. K. Dynamic service migration and workload scheduling in edge-clouds. **Performance Evaluation**, v. 91, p. 205 – 228, 2015. ISSN 0166-5316. Special Issue: Performance 2015.

VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: Towards a cloud definition. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 1, p. 50–55, dec. 2009. ISSN 0146-4833.

VERMESAN, O.; FRIESS, P.; FRIESS, P. **Internet of Things: Global technological and societal trends**. [*S. l.*]: River Publishers Aalborg, Denmark, 2011. v. 37.

WANG, S.; URGAONKAR, R.; HE, T.; CHAN, K.; ZAFER, M.; LEUNG, K. K. Dynamic service placement for mobile micro-clouds with predicted future costs. **IEEE Transactions on Parallel and Distributed Systems**, v. 28, n. 4, p. 1002–1016, 2017.

WANG, S.; ZHANG, X.; ZHANG, Y.; WANG, L.; YANG, J.; WANG, W. A survey on mobile edge networks: Convergence of computing, caching and communications. **IEEE Access**, v. 5, p. 6757–6779, 2017.

WARSHALL, S. A theorem on boolean matrices. **J. ACM**, ACM, New York, NY, USA, v. 9, n. 1, p. 11–12, jan. 1962. ISSN 0004-5411.

XU, M.; TIAN, W.; BUYYA, R. A survey on load balancing algorithms for virtual machines placement in cloud computing. **Concurrency and Computation: Practice and Experience**, v. 29, n. 12, p. e4123, 2017. Available at https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4123.

YANG, L.; CAO, J.; LIANG, G.; HAN, X. Cost aware service placement and load dispatching in mobile cloud systems. **IEEE Transactions on Computers**, v. 65, n. 5, p. 1440–1452, 2016.

YOUSEFPOUR, A.; FUNG, C.; NGUYEN, T.; KADIYALA, K.; JALALI, F.; NIAKANLAHIJI, A.; KONG, J.; JUE, J. P. All one needs to know about fog computing and related edge computing paradigms: A complete survey. **Journal of Systems Architecture**, v. 98, p. 289 – 330, 2019. ISSN 1383-7621.

YU, W.; LIANG, F.; HE, X.; HATCHER, W. G.; LU, C.; LIN, J.; YANG, X. A survey on the edge computing for the internet of things. **IEEE Access**, v. 6, p. 6900–6919, 2018.

YU, X.; GUAN, M.; LIAO, M.; FAN, X. Pre-migration of vehicle to network services based on priority in mobile edge computing. **IEEE Access**, v. 7, p. 3722–3730, 2019.

YU, Y.; CHIU, T.; PANG, A.; CHEN, M.; LIU, J. Virtual machine placement for backhaul traffic minimization in fog radio access networks. In: **2017 IEEE International Conference on Communications (ICC)**. [*S. l.*: *s. n.*], 2017. p. 1–7.

ZHANG, F.; LIU, G.; FU, X.; YAHYAPOUR, R. A survey on virtual machine migration: Challenges, techniques, and open issues. **IEEE Communications Surveys Tutorials**, v. 20, n. 2, p. 1206–1243, 2018.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of internet services and applications**, Springer, v. 1, n. 1, p. 7–18, 2010.

ZHAO, L.; LIU, J. Optimal placement of virtual machines for supporting multiple applications in mobile edge networks. **IEEE Transactions on Vehicular Technology**, v. 67, n. 7, p. 6533–6545, 2018.

ZHOU, A.; QU, B.-Y.; LI, H.; ZHAO, S.-Z.; SUGANTHAN, P. N.; ZHANG, Q. Multiobjective evolutionary algorithms: A survey of the state of the art. **Swarm and Evolutionary Computation**, v. 1, n. 1, p. 32 – 49, 2011. ISSN 2210-6502.