Migration et instanciation dynamique des microservices avec garantie de performance de bout en bout

Kiranpreet Kaur

HESAM UNIVERSITÉ

le cnam

ÉCOLE DOCTORALE SCIENCES DES MÉTIERS DE L'INGÉNIEUR
Centre d'études et de recherche en informatique et communications

THÈSE

présentée par : **Kiranpreet KAUR**
soutenue le : **19 Septembre 2023**

pour obtenir le grade de : **Docteur d'HESAM Université**

préparée au : **Conservatoire National des Arts et Métiers**

Discipline : **Informatique**

Migration et instanciation dynamique des microservices avec garantie de performance de bout en bout

Dynamic migration and instantiation of microservices with end-to-end service chain performance guarantee

THÈSE DIRIGÉE PAR :
SECCI Stefano Professeur, Cnam

ET CO-ENCADRÉE PAR :
GUILLEMIN Fabrice Responsable programme, Orange
SAILHAN Françoise Professeure, IMT-Atlantique

Jury

Géraldine TEXIER	Professeure, IMT Atlantique	Présidente
André-Luc BEYLOT	Professeur, INP/ENSEEIHT	Rapporteur
Yassine HADJADJ AOUL	Professeur, Université de Rennes	Rapporteur
Guillaume PIERRE	Professeur, Université de Rennes	Examinateur
Ravi MAZUMDAR	Professeur, Université de Waterloo	Examinateur

T H È S E

Affidavit

Je soussignée, Kiranpreet Kaur, déclare par la présente que le travail présenté dans ce manuscrit est mon propre travail, réalisé sous la direction scientifique de Fabrice Guillemin, Françoise Sailhan et Stefano Secci, dans le respect des principes d'honnêteté, d'intégrité et de responsabilité inhérents à la mission de recherche. Les travaux de recherche et la rédaction de ce manuscrit ont été réalisés dans le respect de la charte nationale de déontologie des métiers de la recherche. Ce travail n'a pas été précédemment soumis en France ou à l'étranger dans une version identique ou similaire à un organisme examinateur.

Fait à Paris, le 12/05/2023

Signature

Affidavit

I, undersigned, Kiranpreet Kaur, hereby declare that the work presented in this manuscript is my own work, carried out under the scientific direction of Fabrice Guillemin, Françoise Sailhan and Stefano Secci, in accordance with the principles of honesty, integrity and responsibility inherent to the research mission. The research work and the writing of this manuscript have been carried out in compliance with the French charter for Research Integrity. This work has not been submitted previously either in France or abroad in the same or in a similar version to any other examination body.

Place Paris, date 12/05/2023

Signature

Acknowledgements

In addition to my own efforts, the successful journey towards obtaining my doctorate degree has greatly relied on the encouragement and guidance of many individuals. Therefore, I would like to take this opportunity to express my heartfelt gratitude to those who have played a crucial role in the completion of this significant milestone.

First and foremost, I would like to extend my deepest gratitude to Fabrice Guillemin and Françoise Sailhan, my supervisors and co-directors, for their invaluable support, insightful guidance and unwavering availability. Their mentorship has been instrumental in shaping my research journey and I am truly grateful for their contributions. I would also like to express my heartfelt appreciation to my director, Stefano Secci, for his kind cooperation, prompt responsiveness and continuous assistance throughout these three years. Working under their guidance has been an immense privilege and I am humbled and honored to have had the opportunity to collaborate with such exceptional researchers. Their expertise has profoundly enriched my experience, fostering both my personal and professional growth.

I would like to extend my sincere appreciation to Prof. Yassine Hadjadj Aoul and Prof. André-Luc Beylot for their invaluable time, dedication and effort in reviewing and evaluating this manuscript. I would also like to express my gratitude to Prof. Guillaume Pierre, Prof. Géraldine Texier and Prof. Ravi Mazumdar for accepting the invitation to be part of my evaluation committee as examiners. Their expertise and feedback have been instrumental in enhancing the quality of my work and I am truly grateful for their contributions.

I would like to express my heartfelt gratitude to Eric Debeau, the head of the OSONS team at Orange Lannion campus, Aroussia Maadi, the director of Automation for Network department at Orange and the entire HR team for warmly welcoming me from the very first day and providing unwavering support throughout my journey. Their guidance and assistance have been invaluable in reaching this significant stage of my life.

I would also like to extend my deep appreciation to my team at OSONS, who not only welcomed me with

open arms but also integrated me seamlessly into their processes. Their support and collaboration have allowed me to develop and enhance my skills during my PhD journey. I am especially grateful for the humbleness and dedication of Alexandre Ferrieux, Nicolas Edel, Stéfane Tuffin and Sylvain Desbureaux, to whom I hold great respect. They always made time to answer my questions and provided assistance during our work on the implementation of the 4G/5G chain of private networks using open-source technologies in our laboratory at Orange Lannion site. Collaborating with them has been an invaluable opportunity for learning and expanding my expertise.

Furthermore, I would like to express my sincere appreciation to all my colleagues at the CEDRIC laboratory at CNAM Paris. Even though most interactions were virtual, they made my PhD journey a memorable experience. The seminars featuring a wide range of advanced research topics provided a platform for deepening our knowledge and fostering fruitful exchanges. Their contributions have been significant and I am grateful for their support.

I would like to express my immense gratitude to my beloved family. To my mother, Jasvir Kaur, my father, Sarbjit Singh, my sister, Komalpreet Kaur and my brother, Gurkirat Singh, I am eternally grateful for their unwavering love, unwavering belief in me and constant support throughout every stage of my career. Words cannot adequately express my appreciation for their consistent trust and encouragement.

Lastly, I would like to extend my heartfelt thanks to my close friends and loved ones for the wonderful times we've shared and the unforgettable memories we've created together. Your presence and support have been a source of joy and inspiration and I am deeply grateful for your friendship.

Abstract

The microservices and containerization plays an important role in the design of Virtualized Network Functions (VNFs), which are widely adopted by the telco-cloud industry. In practice, 5G/6G services are packaged as small and loosely coupled microservices that are deployed in containers and scaled (up and down) on distributed cloud servers/ data-centers. This is still a challenge to carefully orchestrate the allocation and rearrangement of (micro)services to avoid an im-balanced and a largely segmented solution space in a dynamic environment where services are arriving and leaving the network.

In this regards, this PhD thesis aims to tackle the challenge of migrating and dynamic instantiation of containerized microservices in a distributed cloud architecture while maintaining the end-to-end service chain performance with the objective of low-latency exchange.

To meet this objective, our first proposed contribution enables the latency-aware placement strategy for 5G/6G services over a substrate network. The strategy has been investigated from the perspective of their inter-dependency and the amount of traffic among microservices, which increased the service latency due to the delay associated with messages transiting through the transport network connecting data centers. The proposed approach tends to minimize the global end-to-end latency, which is further solved using a hybrid heuristic algorithm and evaluated through the simulation experiments.

Further, to allow the run-time placement of microservices and attain its optimality in a dynamic system, we introduce an approach that tends to trigger the dynamic migration and management of CNFs while considering the whole life cycle of containers on the basis of a driving use-case: an open-source 5G core network namely Magma. Here, we introduced three heuristic strategies to solve the formalized optimization model on migrating microservices across heterogeneous data center architecture. The evaluated results executed through the simulation show better performance of the migration approach that tends to minimize the global latency.

Moving towards a more practical point-of-view, we performed an extensive study on various container-based migration techniques utilizing the popular orchestration tools (such as: Kubernetes, Docker Compose and

Docker swarm etc.) to develop a Kubernetes based test-bed. A final work considered as a Proof-Of-Concept (PoC) has been developed to illustrate live migration of pods between remote Kubernetes clusters. As a use case, we consider the migration of a network function belonging to an open source 5G core network (namely, Magma).

Keywords : Network virtualization; Microservices placement; Multi-cluster migration; Live migration; Kubernetes; Containerized network function.

Résumé

Les microservices et la conteneurisation jouent un rôle prédominant au niveau de la conception des fonctions de réseau virtualisées (VNF) largement adoptées par l'industrie du cloud des télécommunications. L'utilisation croissante des microservices pour les services basés sur la 5G/6G pousse l'industrie des télécommunications à trouver des moyens efficaces d'exploiter les nouvelles technologies de communication et d'informatique. En pratique, l'approche cloud-native implique de petits microservices faiblement couplés qui sont déployés dans des conteneurs et mis à l'échelle (à la hausse et à la baisse) sur des serveurs/centres de données distribués dans le cloud. Orchestrer soigneusement l'allocation et la réorganisation des (micro)services pour éviter un espace de solution déséquilibré et largement segmenté dans un environnement dynamique où les services arrivent et quittent le réseau, reste un défi.

Par conséquent, cette thèse de doctorat vise à relever le défi de la migration et de l'instanciation dynamique de microservices conteneurisés dans une architecture cloud distribuée tout en maintenant les performances de la chaîne de services de bout en bout dans un objectif d'échange à faible latence.

Pour atteindre cet objectif, notre première contribution correspond à une stratégie de placement sensible à la latence pour les services 5G/6G sur un réseau de substrat. La stratégie a été étudiée du point de vue de leur interdépendance et de la quantité de trafic entre les microservices, ce qui a augmenté la latence du service en raison du retard associé aux messages transitant par le réseau de transport reliant les centres de données. L'approche proposée tend à minimiser la latence globale de bout en bout, à l'aide d'une heuristique hybride évaluée grâce à une simulation.

De plus, pour permettre le placement lors de l'exécution des microservices et atteindre son optimalité dans un système dynamique, nous nous appuyons sur une approche qui tend à déclencher la migration et la gestion dynamiques des CNF tout en considérant l'ensemble du cycle de vie des conteneurs sur la base d'un cas d'utilisation: un cœur de réseau 5G open-source nommé Magma. Ici, nous avons introduit trois heuristiques pour résoudre le modèle d'optimisation formalisé en ce qui concerne la migration des microservices à travers

une architecture de centre de données hétérogène. Les résultats évalués grâce à une simulation montrent de meilleures performances de l'approche de migration qui tend à minimiser la latence globale.
D'un point de vue plus pratique, nous avons réalisé une étude approfondie de diverses techniques de migration basées sur des conteneurs en utilisant les outils d'orchestration populaires (tels que: Kubernetes, Docker Compose et Docker Swarm, etc.) pour développer un banc d'essai basé sur Kubernetes. Ce travail final considéré comme une preuve de concept (PoC) a été développé pour illustrer la migration de pods entre des clusters Kubernetes distants. Comme cas d'utilisation, nous considérons la migration d'une fonction réseau appartenant à un cœur de réseau 5G open source (à savoir, Magma).

Mots-clés : Virtualisation de réseau; Placement de microservices; Migration multi-cluster; Migration live; Kubernetes; Fonction de réseau conteneurisé.

Résumé étendu de la thèse

Une forte tendance des réseaux actuels est ladite *softwarisation* des réseaux qui favorise l'adoption de technologies virtualisées et containerisées pour soutenir le développement rapide de nouveaux services s'adaptant facilement aux besoins changeants des clients. La softwarisation du réseau conduit au remplacement progressif des fonctions réseau assurées par des équipements propriétaire dédiés, par des fonctions réseau virtualisées qui sont assurées par du matériel en étagère. En pratique, une fonction réseau peut offrir une large gamme de fonctionalités de mise en réseau qui fonctionnent sur l'équipement du client, jusqu'au réseau coeur prenant en charge par ex. un pare-feu ou une fonction applicative.

Les microservices sont devenus déterminants dans la conception de fonctions résau virtualisées complexes qui nécessitent une décomposition en de nombreux services, par exemple plusieurs centaines de services pour les fonctions du réseau coeur. Dans ce cas, les microservices sont de petits services implémentant un nombre limité de fonctionnalités qui peuvent être exécutées indépendamment (même si elles sont logiquement dispersées à la périphérie, dans le brouillard ou dans le cloud); chaque microservice exécute ses propres processus/fonctionnalités et communique via des protocoles légers. Dans l'ensemble, cette conception orientée cloud offre une approche différente car il s'agit d'une approche agile qui porte à l'échelle et supporte une orchestration efficace des fonctions de réseau distribuées.

L'approche orientée conteneur est également de plus en plus privilégiée, car un microservice conteneurisé peut être rapidement instancié selon les besoins et peut également être répliqué indépendamment, pour répondre à la demande croissante de traitement ou de stockage supplémentaire.

Contexte et motivations

L'introduction de la virtualisation des fonctions réseau vise à faciliter la gestion et la fourniture de fonctionnalités réseau en utilisant des applications logicielles virtualisées hébergées sur des serveurs commerciaux en

étagère [36]. Initialement, les fonctions virtualisées (VNF) basées sur les machines virtuelles (VM) étaient destinées à remplacer les fonctions réseau matérielles. Cependant, avec l'évolution des conteneurs, une fonction réseau virtualisée tend à être déployée dans des conteneurs.

De plus, la tendance technologique actuelle montre une adoption croissante des approches cloud par les opérateurs de réseaux de télécommunication, qui intègrent de petits microservices faiblement couplés, déployés dans des conteneurs et redimensionnés (à la hausse et à la baisse) selon les besoins [10, 36]. Ainsi, cette montée en puissance des microservices amplifie l'utilisation des conteneurs, qui offrent un environnement idéal pour les microservices petits et autonomes.

Les opérateurs de réseaux, les fournisseurs de cloud (par exemple, AWS, Google) et les fournisseurs de contenu (par exemple, Netflix, BBC) adoptent le style architectural des microservices [15, 86] et traitent avec des applications pouvant comprendre des centaines voire des milliers de conteneurs. Bien que les conteneurs présentent l'avantage de regrouper toutes les dépendances d'une fonction réseau en une seule unité, la gestion, le déploiement et la migration de ces conteneurs dans une infrastructure multi-cloud de grande envergure à l'aide d'outils ou de scripts personnalisés deviennent de plus en plus complexes et difficiles à gérer.

Dans le domaine des télécommunications, l'architecture de base 5G/6G basée sur les microservices pousse l'industrie à identifier des moyens efficaces d'exploiter les nouvelles technologies de communication et de calcul. Dans un système dynamique, il est nécessaire d'améliorer en continu les services initialement mis en place en réinitialisant ou en réallouant les microservices afin de maintenir les performances au fil du temps et assurer la qualité de service requise.

Les défis et les objectifs principaux

Alors que les attentes sont élevées quant à la large applicabilité de la virtualisation des fonctions réseau, la mise en œuvre des fonctions réseaux virtualisées est loin d'être une tâche anodine, surtout compte tenu du fait que le réseau virtualisé doit s'adapter à la dynamique du réseau (par exemple, malgré des conditions changeantes) et aux besoins des microservices.

Avec la virtualisation des fonctions réseau, un service réseau est composé d'une série de fonctions réseau (également appelées microservices) caractérisées par un ordre prédéfini, connu sous le nom de chaine de service. Du point de vue de la conception, les fonctions réseau virtualisées prennent en charge une fonctionnalité spécifique dédiée et restent souvent dépendantes de l'état, c'est-à-dire que les états sont stockés et mis à jour

localement. En suivant la chaîne de service, le trafic passe par la série de fonctions réseau dans un ordre précis, de sorte que le trafic peut circuler de manière bidirectionnelle entre des fonctions virtualisées distantes lorsque ces dernières résident sur des serveurs physiques ou des centres de données distincts.

Pendant le fonctionnement des fonctions réseaux virtualisées, le trafic, la bande passante du réseau, le stockage disponible et les ressources de calcul fluctuent généralement au fil du temps, ce qui entraîne une utilisation possiblement déséquilibrée des liens/ressources. Par conséquent, l'allocation efficace et la gestion continue des fonctions réseaux virtualisées deviennent plus complexes, compte tenu de l'hétérogénéité et de la dynamique des ressources physiques, ainsi que de la nature éphémère des services.

Pour surmonter ce problème, un nombre croissant d'efforts de recherche a été consacré à la prise en charge de la migration des services réseau vers d'autres serveurs physiques/centres de données, ce qui est essentiel pour préserver la qualité de service et répondre aux attentes de l'utilisateur en termes de performance.

Alors que des sujets connexes tels que la migration de machines virtuelles/conteneurs dans les centres de données cloud ont atteint une certaine maturité, le découplage et la réallocation élastique de petites fonctions réseau entre les centres de données couvrant le réseau périphérique jusqu'au cœur restent un défi.

Dans ce cadre, les objectifs de la thèse visent à :

1. Identifier les facteurs clés permettant de réaliser des microservices déployés de manière indépendante (découplés) lors de la conception de réseaux.

2. Proposer de nouveaux placements statiques et dynamiques de microservices et évaluer les performances de la fonction réseau globale associée ainsi que l'impact sur les autres fonctions réseau.

3. Analyser et modéliser le comportement des cycles de déploiement-placement continus des réseaux en se basant sur un cas d'utilisation clé (par exemple, le plan de contrôle 5G).

4. Développer des politiques de gestion des ressources structurées pour les microservices.

Principales problématiques de recherche abordées dans cette thèse

Au cours des dernières décennies, de nombreux travaux de recherche ont été proposés pour améliorer les techniques de migration des machines virtuelles. Ensuite, les efforts se sont concentrés sur l'application de ces techniques aux conteneurs en raison de leurs avantages incontournables.

Plus précisément, les études ont tenté de résoudre certains des problèmes nouveaux auxquels est confrontée la migration des conteneurs (virtualisation basée sur le système d'exploitation), qui ne sont pas concernés par la migration des machines virtuelles (virtualisation orientée matériel). Différentes approches ont également été proposées pour gérer la migration des conteneurs avec ou sans état tout en réduisant le temps de migration, les interruptions de service et la taille des données transférées. Comme détaillé ci-dessous, il reste des défis non résolus, notamment relatifs à la gestion des centres de données distribués, le déploiement et la gestion de la chaîne de microservices conteneurisés pendant la migration afin d'éviter les perturbations de service, la baisse de la qualité de service et les perturbations des échanges en cours :

1. **Gestion du réseau multi-cloud** - Les orchestrateurs populaires tels que Kubernetes (K8s) [109] sont principalement orientés cloud, alors que l'on prévoit que 75% des données générées seront traitées en dehors d'un cloud centralisé d'ici 2025 [94]. En particulier, Kubernetes gère le déploiement et l'évolutivité horizontale en permettant la création et l'arrêt d'instances de microservices en fonction de la charge de travail, de la récupération en cas de panne ou de la continuité du service. Actuellement, la demande augmente pour un orchestrateur Edge Multi Cloud (EMCO) [106, 107] qui gère le déploiement d'une chaîne de microservices mettant en œuvre des services 5G et MEC à travers un réseau multi-cluster composé de clouds de différents types (tels que ceux de l'edge, du brouillard et du cœur). En particulier, l'exécution de microservices soigneusement placés au sein de l'infrastructure réseau (c'est-à-dire les centres de données à la périphérie, dans le brouillard et dans le cloud) implique des aspects totalement différents.

2. **Gestion d'un grand ensemble de microservices** - Les EMCO récemment apparus facilitent la gestion et le déploiement de services géo-distribués sur plusieurs clusters K8s distribués. Cependant, la gestion automatisée des microservices composés pendant l'ensemble de leur cycle de vie, y compris leur instantiation, leur migration et leur terminaison, est assez complexe pour un grand ensemble de services devant être déployés sur des centres de données distincts à travers un réseau multi-cluster. En particulier, cela nécessite la définition de plusieurs contraintes de placement basées sur l'affinité, l'anti-affinité ou le coût. Cependant, il reste encore plusieurs défis à relever. L'outil de conception doit être capable de lier étroitement les microservices. De plus, il est difficile de maintenir la connexion active pendant la terminaison et la réinitialisation, car une chaîne de microservices communique non seulement avec les utilisateurs finaux, mais également avec les microservices respectifs qui peuvent être placés sur des serveurs dans différents clusters.

3. **Sélection de la cible optimale** - En plus des problèmes mentionnés précédemment, il est nécessaire de gérer la procédure de sélection d'un hôte cible approprié, car la complexité augmente avec les migrations multiples. Il est nécessaire de migrer la charge de travail près des utilisateurs finaux afin de répondre à diverses exigences (telles que la latence et la continuité du service) pour les centres de données décentralisés dans le brouillard et à la périphérie, qui distribuent et mettent à l'échelle la charge de travail.

4. **Gestion du placement en fonction du type de service** - Les tailles variables des fonctions réseaux conteneurisées nécessitent également d'examiner les exigences du service lors de la mise en correspondance : le service composé d'une chaîne de microservices doit répondre à toutes les exigences des microservices, y compris la sensibilité au temps, la latence ou l'efficacité de charge. La conception du modèle doit être capable de distinguer les services afin de placer efficacement l'ensemble spécifique de microservices sur les centres distribués de périphérie et les autres sur les clouds centralisés. Dans le but de sauvegarder autant que possible les ressources en périphérie, car celles-ci sont critiques.

5. **Gestion de la charge dynamique d'un système** - De plus, en passant à une stratégie en ligne - où les services arrivent ou quittent continuellement le système et posent le problème d'un déséquilibre des ressources avec une incertitude du poit de vue du moment d'arrivée/de départ, la recherche doit prendre en compte le problème de savoir quand déclencher la migration et la sélection du conteneur à migrer de manière à obtenir un taux de migration plus faible. Le taux de migration influe directement sur la consommation d'énergie du système.

Résumé des contributions

Dans cette thèse, nous avons introduit quatre contributions qui relèvent de trois domaines : la gestion et l'orchestration des services 5G/6G, les infrastructures distribuées (Edge/Fog/Cloud) et la conteneurisation en environnement dynamique, multi-cluster (Figure 1). L'intersection de ces domaines constitue les aspects clés étudiés au cours de cette thèse, allant de l'état de l'art à l'analyse des nombreux algorithmes d'optimisation de placement et de migration des microservices conteneurisés, ainsi que des techniques d'orchestration multi-cluster.

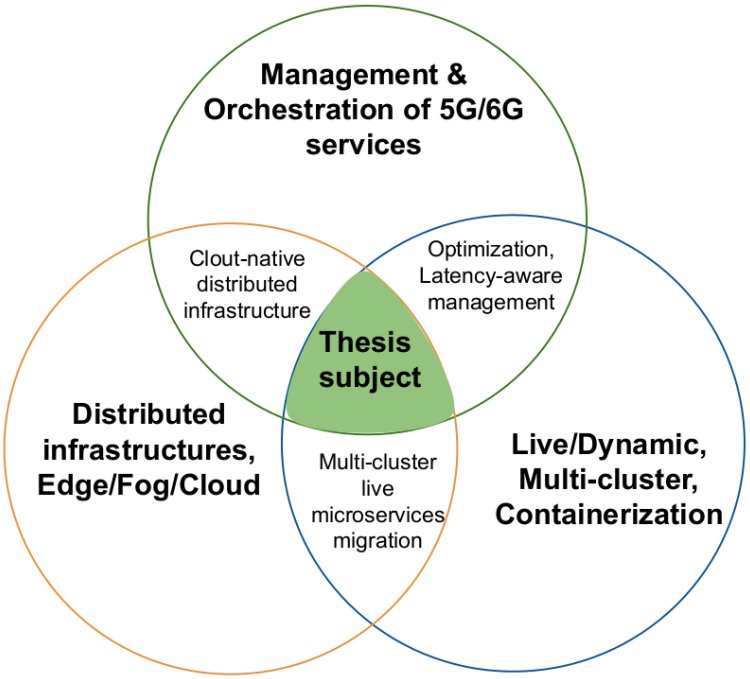


Figure 1: Portée de la thèse

1. Placement prenant en compte la latence et le réseau pour les services 5G/6G natifs du cloud : Solution hybride basée sur des heuristiques

Pour répondre à des exigences de plus en plus strictes en termes de latence, les réseaux 5G/6G évoluent vers des architectures distribuées, pour lesquelles le paradigme natif du cloud avec des services décomposés en microservices est extrêmement pertinent. Cela soulève à son tour la question de la distribution des fonctions réseau.

Cependant, malgré la pléthore de travaux existants sur la placement des fonctions réseaux virtualisées, la plupart d'entre eux se concentrent sur l'équilibrage de la charge et l'examen des besoins en ressources (CPU, RAM, disque) des fonctions réseau par rapport à la disponibilité des ressources dans les centres de données. Il existe quelques travaux traitant du problème de placement des microservices du point de vue des exigences de latence, en tenant compte soit du délai de traitement, soit de la capacité de liaison dans un centre de données distribué. À travers notre étude approfondie et détaillée (présentée dans le chapitre 3) portant sur diverses stratégies de placement et de migration des conteneurs, nous cherchons également à fournir une connaissance approfondie sur les technologies récentes.

Dans le but d'approfondir cette question, la première contribution vise à aborder le déploiement des fonc-

tions réseau à partir de différents scénarios, en tenant compte notamment de l'interdépendance des microservices et de la quantité de trafic entre les microservices, ce qui entraîne une augmentation de la latence du service en raison du délai associé aux messages transitant à travers le réseau de transport connectant les centres de données. Les contributions sont détaillées dans le chapitre 4 et ont été été publiés à la conférence IEEE CCNC 2022 [42]. Elles comprennent les éléments suivants :

1. Un modèle d'optimisation par programmation linéraire en nombres entiers pour placer les services décomposés en microservices sur un réseau substrat représentant l'architecture à trois niveaux des nœuds Cloud-Fog-Edge.

2. Un placement efficace en termes de latence qui prend en compte les messages échangés entre les microservices afin de répartir les microservices communiquant fortement sur le même centre de données et à proximité de l'utilisateur final afin de minimiser la latence et le délai de bout en bout.

3. Nous envisageons deux variantes pour sélectionner le nœud le plus proche dans un cloud voisin :

   (i) en ignorant le délai de transmission entre les clouds ; dans ce cas, la sélection est **indépendante du réseau**;

   (ii) en tenant compte du délai de transmission entre les centres de données (c'est-à-dire du nombre de messages et du délai de transmission) ; dans ce cas, la sélection **prend en compte le réseau**.

4. Pour résoudre le problème d'optimisation, une approche par heuristique hybride a été proposée, combinant une méthode gloutonne et un algorithme génétique avancé.

2. Algorithmes et mécanismes de migration dynamique des microservices tout en garantissant la continuité du service : Approches par heuristiques et méta-heuristiques

Avec l'adoption du paradigme des microservices par l'industrie des télécommunications dans la conception des réseaux 5G/6G, les fonctions réseau complexes sont décomposées en ensembles de sous-fonctions chaînées, qui sont ensuite déployées à l'aide de technologies conteneurisées sur des clusters cloud répartis géographiquement. Les applications sensibles à la latence nécessitent d'orchestrer soigneusement l'allocation et la réorganisation des (micro)services afin d'éviter un placement largement segmenté des microservices.

Les études de recherche existantes sur le placement et le réarrangement de VNF ignorent le problème commun du chaînage des microservices: en pratique, les fonctions réseau sont placées en se concentrant sur

la disponibilité/le besoin de ressources et/ou le temps de migration tout en omettant de prendre en compte la latence associée à la communication entre les microservices chaînés et l'utilisateur final qui permettrait d'optimiser la latence de bout en bout. Par conséquent, par rapport aux travaux précédents, le travail proposé aborde conjointement la stratégie optimale de placement et de migration pour les scénarios en temps réel, où l'arrivée et le départ des services sont insignifiants, visant à minimiser le retard du réseau et la latence de bout en bout entre les utilisateurs et les services.

Les contributions suivantes sont détaillées dans le chapitre 5 et ont été publiées dans le cadre de la conférence IEEE ICC 2023 [44] et dans le cadre d'une version étendue dans Journal of Network and Systems Management 2023 [45].

1. Nous avons formalisé le problème d'optimisation de la migration des microservices sur plusieurs centres de données en veillant à déplacer le moins de microservices tout en conservant un placement optimal.

2. La logique de conception met d'avantage l'accent sur l'utilisateur final et les microservices fortement actifs tout en les sélectionnant et en les migrant verticalement (de la périphérie vers le cloud) ou horizontalement (entre les centres de données d'une même couche).

3. Nous avons introduit trois algorithmes heuristiques et méta-heuristiques pour résoudre le problème d'optimisation qui réduisent considérablement le temps d'exécution de l'algorithme de migration.

4. Nous avons implémenté une évaluation basée sur la simulation qui montre l'efficacité de la migration des services en utilisant notre algorithme proposé et minimise la latence.

3. Migration dynamique des microservices conteneurisés entre clusters Kubernetes distants: PoC

De nombreuses techniques de migration dynamique des machines virtuelles ont fait l'objet de recherches approfondies au cours des dernières décennies. Cependant, les avantages liés à l'utilisation des conteneurs poussent l'industrie à se tourner vers des techniques orientées conteneurs. De nombreux travaux de recherche tentent de gérer la migration des conteneurs pour différents cas d'utilisation tout en garantissant la Qualité d'expérience /Qualité de service pour l'infrastructure cloud. Pourtant, il existe une demande clé pour un orchestrateur capable de gérer les multiples clusters ainsi que différents types de clouds (tels que edge/fog/core) et pour déployer des applications pour les services 5G et MEC. Le travail final implémenté sous la forme de

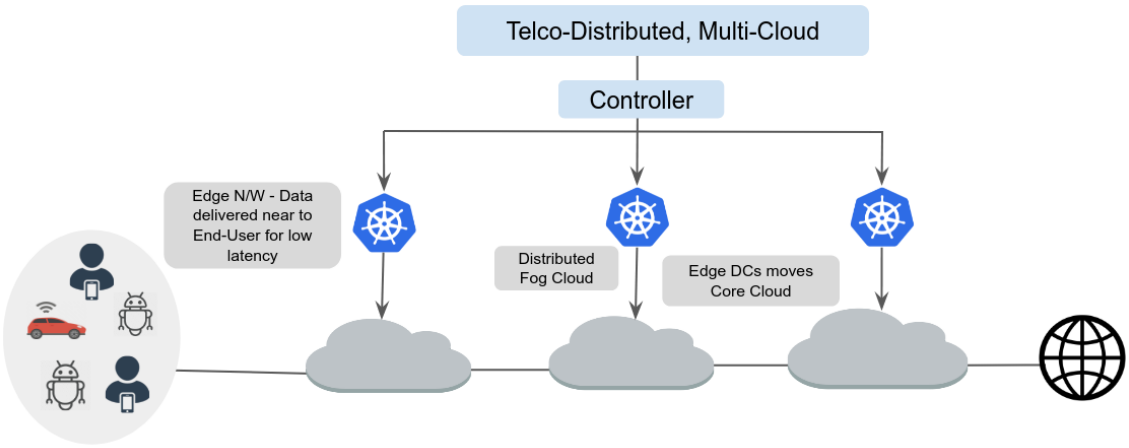


Figure 2: Extension du paradigme Kubernetes à d'autres domaines

Proof-of-Concept (PoC) dans le cadre de cette thèse est détaillé dans le chapitre 6 qui rassemble deux contributions:

1. Le déploiement d'un banc d'essai illustrant le mécanisme implémenté de migration de pod entre des clusters K8 distants. Le cas d'utilisation considéré est un réseau central 5G open source (à savoir, Magma).

2. À l'aide d'un contrôleur, l'ensemble du processus de migration peut être exécuté automatiquement et est capable de gérer une connexion active lors de la résiliation et du rétablissement des connexions.

L'infrastructure cloud à trois niveaux considérée (Figure 2), qui représente aujourd'hui raisonnablement les réseaux traditionnels des fournisseurs d'accès à Internet impliquant des réseaux à 2/3 niveaux. Ces réseaux, qui ont une empreinte nationale et régionale, interconnectent les utilisateurs finaux aux réseaux dorsaux de niveau 1 utilisés pour échanger le trafic international. L'infrastructure cloud associée reflète l'architecture des réseaux opérateurs avec leurs services fournis.

4. Déploiement de la chaîne complète du réseau mobile 4G/5G

Dans le cadre de travaux de recherche, nous avons également trois grands principes sur l'utilisation de logiciels open source pour le déploiement et l'orchestration des services 4G/5G, comme indiqué dans le schéma de principe 3.

1. Tester une solution open source pour un réseau autonome 5G de bout en bout.

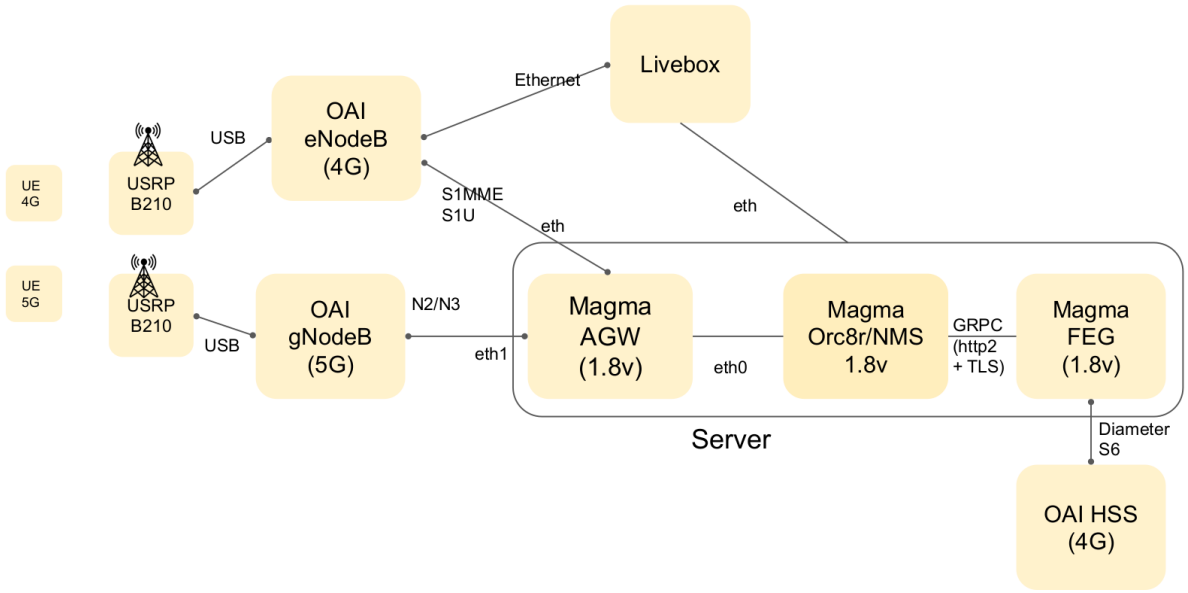


Figure 3: Schéma de configuration de la chaîne 4G/5G

2. Implémenter et évaluer un scénario d'hôte neutre à l'aide d'une solution réseau open source (à savoir, MAGMA).

3. Étudier l'orchestration des services à l'aide d'opérateurs dans Kubernetes.

Pour répondre à ces exigences de la solution E2E, deux composants logiciels open source ont été utilisés, notamment Magma (1.8v-5G) et OAI (eNodeB, gNodeB, HSS). Le réseau central open source considéré, mis en œuvre à l'aide de Magma [32], prend en charge diverses technologies radio, notamment LTE, 5G et WiFi. Magma a été conçu à l'origine pour étendre la couverture des réseaux mobiles, mais aujourd'hui, Magma est considéré comme une solution efficace pour construire des réseaux 5G privés. Grâce à la capacité multi-opérateur offerte par le Federation Gateway (FEG), Magma pourrait également être avantageusement utilisé dans le cadre de TowerCos [30].

Comme le montre la figure 4, les principaux composants de l'architecture Magma sont les suivants :

- **Orchestreur (Orc8r)**: l'ochestrateur est un service en nuage qui fournit un moyen simple et cohérent de configurer et de surveiller le réseau sans fil en toute sécurité. L'orchestrateur a trois fonctions principales : un système de gestion de réseau (NMS) qui prend en charge, par exemple, la configuration et les capacités de surveillance de base, les indicateurs de performance clés (KPI) exposés via un point d'accès REST, et un canal de communication sécurisé pour la communication entre les différentes passerelles.

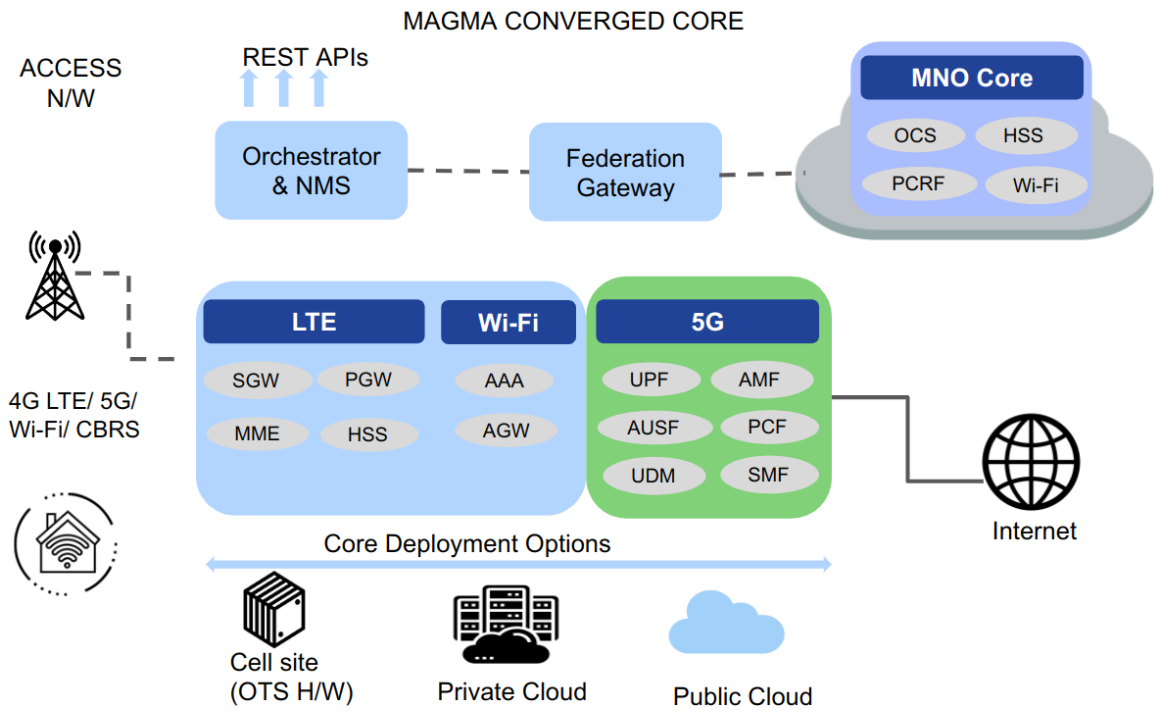


Figure 4: Magma Architecture et ses composants.

- **Passerelle d'accès (AGW) :** Cette fonction fournit un noyau pour les services 4G et 5G. Il s'ensuit une architecture distribuée permettant une mise à l'échelle horizontale avec un réseau d'accès radio (RAN) comprenant, par exemple, des eNodeB et des gNodeB. Avec la 5G, AGW gère les fonctionalités associées au plan utilisateur (UPF), celles liées à la gestion de session (SMF) et celles relevant de la gestion de l'accès et de la mobilité (AMF). Ces trois fonctions constituent le Minimal Viable Core (MVC), qui est l'ensemble minimal de fonctions requises pour établir des sessions en 5G. Dans le cas de la 4G, le MVC comprend les fonctions MME et S/PGW. Il n'y a pas de fonction d'authentification (AUSF, UDM, UDR) : l'authentification est simulée en fournissant via le NMS les IMSI aux UE autorisés à se connecter au réseau.

- **Federation Gateway (FeG):** Cette fonction intègre le réseau central MNO au sein de Magma en fournissant des interfaces 3GPP standard aux composants MNO existants (notamment le HSS en 4G et l'AUSF en 5G). Il agit comme un proxy entre le Magma AGW et le réseau de l'opérateur et facilite la fourniture des fonctions de base, telles que l'authentification, les fonctions relevant du plan de données, l'application des politiques et la facturation pour être conforme à un réseau MNO existant et au réseau étendu utilisant le noyau Magma.

Publications

Le tableau 1 présente la liste des travaux de doctorat publiés - leur titre (avec la citation), le lieu de publication, le type de publication et le statut actuel.

Table 1: Résumé des publications

#	**Titre de la publication**	**Lieu de publication**	**Type de publication**	**Statut**
1	Latency and network aware placement for cloud-native 5G/6G services [42]	IEEE 19th Annual Consumer Communications & Networking Conference (CCNC 2022)	Article complet	Publié
2	Container placement and migration strategies for cloud, fog, and edge data centers: A survey [43]	International Journal of Network Management 2022	Article de journal	Publié
3	A microservice migration approach to controlling latency in 5G/6G networks [44]	IEEE International Conference on Communications (ICC) 2023	Article complet	Publié
4	Live Migration of containerized microservices between remote Kubernetes Clusters [46]	IEEE INFOCOM 2023 Workshop	Article complet	Publié
5	Dynamic migration of microservices for end-to-end latency control in 5G/6G networks [45]	Journal of Network and Systems Management 2023	Article de journal	Publié

Perspectives et travaux futurs

L'ensemble des solutions proposées dans cette thèse permet évidemment un placement efficace des microservices conteneurisés en automatisant le processus de migration ou de réaffectation en fonction des besoins, à proximité de l'utilisateur final, afin de garantir une communication plus rapide. En outre, il est démontré que la solution cloud-native basée sur Kubernetes pour le réseau central 5G open-source constitue un mécanisme abordable pour un scénario en temps réel. Cependant, nous pensons qu'il serait possible d'étendre ce travail, comme nous le verrons dans la section suivante.

Migration de microservices basée sur ML

Comme indiqué dans les travaux de recherche liés au placement des VNF basés sur ML (tels que [11, 12, 82, 95, 114, 116, 117]), l'algorithme d'aprentissage par renforcement profond a été largement utilisé.

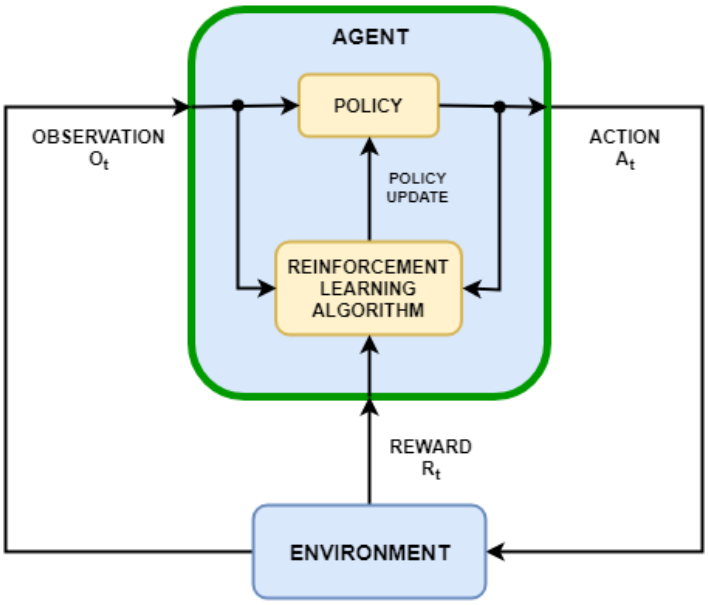


Figure 5: Configuration standard de l'algorithme DRL. Source [5]

Avec cette dernière approche, l'agent reçoit les récompenses ou les pénalités à chaque intervalle de temps en fonction de ses actions et continue d'apprendre à prendre des décisions en fonction de cela. L'apprentissage est ici basé sur des expériences passées dans l'optique de de former l'agent à accomplir une tâche dans un environnement incertain. Les récompenses sont utilisées pour quantifier la qualité des actions exécutées par rapport à la réalisation des objectifs.

De plus, l'agent est composé de deux composants: la politique et l'algorithme d'apprentissage, comme illustré par la figure 5. La politique est définie comme une fonction qui renvoie une action réalisable pour un problème. Il cartographie chaque action et état en fonction d'une probabilité d'agir dans un état particulier. L'algorithme d'apprentissage se concentre sur la recherche d'une politique optimale pour maximiser la récompense cumulée à long terme attendue et reçue au cours de la tâche. Sur la base des actions, des observations et des récompenses, l'algorithme d'apprentissage continue de mettre à jour les paramètres de la politique. Par conséquent, inspiré par le travail [12], où les auteurs ont proposé une approche basée sur l'apprentissage par renforcement profond qui accélère l'apprentissage proposé et améliore les performances par rapport à l'étude de pointe, nous pensons également que cette combinaison d'une approche basée sur l'apprentissage intelligent avec notre algorithme de migration heuristique est capable d'améliorer le placement des VNF/CNF et de sa migration en déterminant un nœud cible optimal.

Intégration de l'algorithme de migration proposé dans le banc d'essai mis en œuvre

La direction la plus intéressante est d'intégrer le banc d'essai implémenté avec nos algorithmes proposés. Même si nous l'avons testé pour une vraie solution de base 5G, l'étape d'évaluation et de compréhension de la façon dont l'algorithme fonctionne lorsqu'il est utilisé dans un système réel est relativement importante. Cela nous permet d'évaluer les performances d'un algorithme dans un mécanisme réel.

Deuxièmement, il est possible d'évaluer le même prototype pour d'autres réseaux centraux 5G open source disponibles (tels que free5GC, open5gs et OAI-CN) car ils sont également basés sur une architecture cloud native. Actuellement, de nombreuses entreprises de télécommunications proposent et investissent dans la production de leur propre solution de réseau central pour répondre aux besoins du réseau cellulaire 5G/6G. Par conséquent, cette comparaison pour diverses solutions apportées et l'exécution simultanée de la migration de plusieurs conteneurs aideraient à fournir une vue d'ensemble et à reconnaître le comportement de composants distincts en temps réel.

Organisation du manuscrit

Le reste du manuscrit est organisé comme suit: Le chapitre 2 présente le contexte relevant de la conteneurisation et des différentes techniques de migration; le chapitre 3 détaille l'état de l'art; le chapitre 4 décrit la première contribution de cette thèse de doctorat, intitulée, *Un placement prenant en compte la latence et le réseau pour des services 5G/6G*; Le chapitre 5 présente la deuxième contribution de cette thèse de doctorat, intitulée *Migration dynamique des microservices pour le contrôle de latence de bout en bout dans les réseaux 5G/6G*; Le chapitre 6 représente la preuve de concept et la contribution finale de cette thèse de doctorat, intitulée *Migration en direct des microservices conteneurisés entre les clusters Kubernetes distants*; et le chapitre 7 conclut le manuscrit.

Contents

List of Tables

List of Figures

Acronyms

A3C Asynchronous Advantage Actor-Critic. 30

CNF Cloud-Native Network Function. 15, 26, 38, 41, 60, 86

COTS Commercial off-the-shelf. 1

DQN Deep Q-Network. 30

DRL Deep Reinforcement Learning. 28, 30, 99

EMCO Edge Multi Cloud Orchestrator. 60

GCN Graph Convolutional Network. 30

ILP Integer Linear Programming. 6, 39, 40

KVM Kernel-based Virtual Machine. 23

LXC Linux Containers. 23

MEC Mobile Edge Computing. 32, 37, 40, 41, 60, 62, 67

MILP Mixed Integer Linear Programming. 38

NF Network Function. 2, 4, 13, 14, 15, 60, 62

NFV Network Function Virtualization. 1, 28, 30, 44, 60, 85

ONAP Open Network Automation Platform. 61

QoE Quality of Experience. 7

QoS Quality of Service. 2, 7

RAN Radio Access Network. 61

RL Reinforcement Learning. 99

SFC Service Function Chain. 28, 38, 60, 61

SLA Service Level Agreement. 2, 60

VNF Virtualized Network Function. xxi, 5, 7, 14, 15, 28, 38, 39, 44, 60, 61, 66

Chapter 1

General Introduction

Content

This chapter summarizes the context and motivation of this PhD (Section 1.1), the key challenges along with main objectives (Section 1.2), key research perspectives (Section 1.3), the summary of contributions (Section 1.4), the published work (Section 1.5) and organization of manuscript in final (Section 1.6).

1.1 Context and motivations

The introduction of Network Function Virtualization (NFV) facilitates the management and provision of network capabilities using virtualized software applications hosted on Commercial off-the-shelf (COTS) servers [36]. Initially, the Virtual Machines (VMs) based VNFs were aimed to replace the hardware-based physical networking functions. However, with the evolution of containers, NFV have been deployed in containers to support the so-called cloud-native network functions (CNFs). Moreover, the current technology

trend showed a high beam in the adoption of cloud-native approaches by the telecom network operators that contain small and loosely-coupled microservices that are deployed in containers and scaled (up and down) as needed [10, 36]. Therefore, this rise of microservice architecture amplifies the usage of containers that offer an ideal host for the small and self-contained microservices. The network operators, cloud providers (e.g., AWS, Google) and content providers (e.g., Netflix, BBC) are adopting the microservice architectural style [15, 86] and deal with applications that may comprise even hundreds or thousands of containers. Even though containers come up with the benefit of packing all the dependencies of a Network Function (NF) into a single unit, managing, deploying and migrating these containers in a large and multi-cloud infrastructure using self-made tools or scripts becomes increasingly complex and difficult to manage.

In telecom, the microservices based 5G/6G core architecture is driving the industry to identify efficient ways of exploiting new communication and computing technology. In a dynamic system, the continuous improvement of initially placed services is required by re-initializing or re-allocating the microservices to maintain the performance across time and reach the Service Level Agreement (SLA).

1.2 Challenges and main objectives

While expectations are high for the wide applicability of network softwarization, putting the deployment of VNFs into practice is far from being a trivial task especially considering that softwarized network should adapt to network dynamics (e.g., despite changing conditions) and microservices needs. With NFV, a network service is composed of series of network functions (a.k.a microservices) characterised by a predefined order, which is known as *service chaining*. By design, VNF supports a dedicated specific functionality and often remains state-dependant, i.e., in practice, states are stored and updated locally with the associated VNFs. Following the chain, traffic goes through the series of ordered network functions such that traffic may flow back and forth among distant VNFs as VNFs reside on distinct physical servers or data centers. During the operation of the VNF, traffic, network bandwidth, available storage and computational resources typically fluctuate over time, which results in imbalanced links/ resource usage. Thus, the efficient allocation and the continuous management of NFVs become more complex, considering the heterogeneity and the dynamics of the physical resources as well as the ephemeral nature of the services. To overcome this issue, a growing number of research effort has been devoted to support the migration of network services possibly to other physical server(s)/data center(s), which is key to preserve the Quality of Service (QoS) and meet the expectation of the user in terms of performance. While related topic including VM/container migration in cloud data centers has matured, the decoupling and

elastic re-allocation of small networking functions across data centers spanning the edge to the core remains challenging. Therefore, the thesis objectives include:

1. Identifying key enablers for achieving independently deployed (decoupled) microservices when designing cloud native networks.

2. Proposing novel static & dynamic placements microservices and evaluating the performance of the associated global network function and the impact on other network functions.

3. Analysing and modeling the behavior of continuous deployment-placement cycles of native cloud networks on the basis of a driving use case (e.g., 5G control plane).

4. Developing structured microservice resource management policies.

1.3 Key research issues addressed in this PhD

During last decades, extensive research works have been actively proposed to improve VMs live migration techniques. Then, effort shifted towards applying these techniques on containers due to their unavoidable advantages. In particular, the aforementioned studies tried to solve some of the novel issues faced by container migration (OS-based virtualization) that are not concerned by VM migration (hardware-aware virtualization). Different approaches were also proposed to handle stateful & stateless container migration while reducing the migration time, downtime and size of transferred data. As detailed in the following, there remains unresolved challenges, including dealing with dis-aggregated data centers, deploying and managing the chain of containerized microservices during the migration while avoiding the service disruption, drop in the QOS & disturbance of the ongoing exchanges:

1. **Handling multi-cloud network** - popular orchestrators such as Kubernetes (K8s) [109] are mostly cloud-oriented while 75% of the data generated is expected to be processed outside a centralized cloud by 2025 [94]. In particular, Kubernetes manages horizontal deployment and scaling by allowing a set of microservices instances to be created and stopped based on e.g. workload or to ensure fault recovery or service continuity. Currently, the demand is increasing for an Edge Multi Cloud Orchestrator (EMCO) [106, 107] that manages the deployment of a chain of microservices implementing 5G and MEC services accross a multi-cluster network that consists of clouds of different types (such as edge/fog/core). In

particular, running microservices carefully placed within the network infrastructure (i.e. edge/fog/cloud data centers) involves completely different aspects.

2. **Handling a large set of microservices** - Recently emerged EMCOs facilitate the management and deployment of geo-distributed services across multiple distributed K8s clusters. Still, the automated management of composed microservice during their whole life cycle, including their instantiation, migration and termination, is quite complex for a large set of services supposed to be deployed on distinct data centers across a multi-cluster network. In particular, it requires the definition of multiple placement constraints based on affinity, anti-affinity or cost.
   Nevertheless, there are still some remaining challenges that require focus on. The design tool must stick the microservices together. Also, it is hard to manage the connection alive during termination and re-establishment as a chain of microservices not only communicate with end-users but also with respective microservices that may be placed on the servers in different clusters.

3. **Selection of optimal target** - Apart from the above mentioned issues, the need is to handle the selection procedure for an appropriate target host as complexity gets enlarged with multiple migrations. Along with migrating the workload near to end users to meet various requirements (e.g., latency and service continuity) for the dis-aggregated fog and edge data centers that distribute and scale the workload.

4. **Handling the placement based on service type** - The varying sizes of containerized NFs also require to examine service requirements while mapping: the service composed of a chain of microservices must fulfill any microservice requirements including time-sensitivity, latency or load efficiency. The design of the model must be able to distinguish the services in order to efficiently place the particular set of microservices on distributed edge centers and others on centralized clouds. Aiming to save as much as possible the resources at edge as these are the critical one.

5. **Handling the dynamic load of a system** - Moreover, moving towards online strategy - where services are continuously arriving or departing the system and raising the issue of resource imbalance with uncertainty of arrival/departure time, the research must consider the problem of when to trigger the migration and selection of container to be migrated in a way to attain the lower migration rate. The migration rate directly influences the system's energy consumption.

1.4 Summary of contributions

Here, we summarized the four accomplished contributions in Sections 1.4.1, 1.4.2 1.4.3 and 1.4.4 respectively. This falls into three domains represented through the diagram 1.1 : Management & Orchestration of 5G/6G services; Distributed infrastructures, Edge/Fog/Cloud; and Live/Dynamic, Multi-cluster, containerization. The intersection of these domains are key investigated aspects that we explored during this Thesis work - from the state-of-the-art to the study analysis on numerous containerized microservices placement & migration optimization algorithms and multi-cluster orchestration techniques.

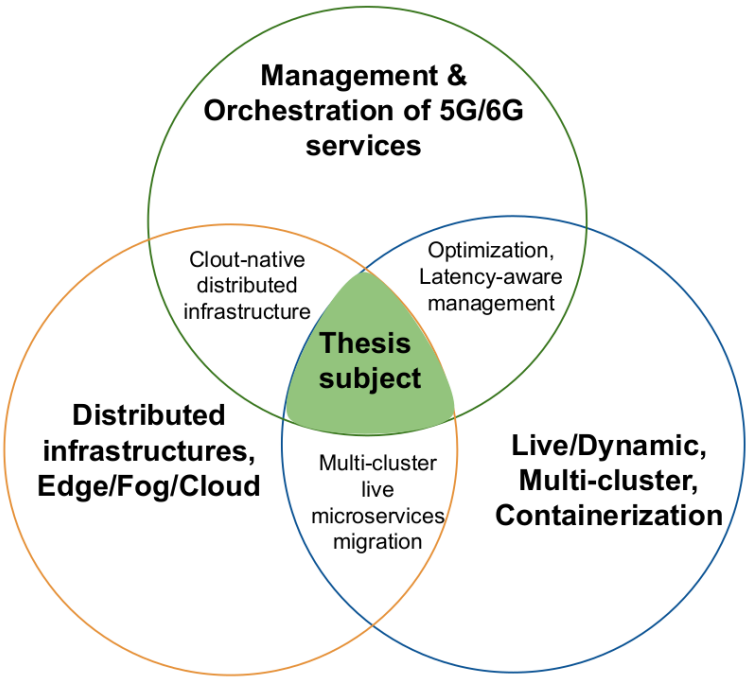


Figure 1.1: Thesis scope

1.4.1 Latency and network aware placement for cloud-native 5G/6G services: Hybrid Heuristic solution

To meet ever more stringent requirements in terms of latency, 5G/6G networks are evolving from centralized to distributed architectures, for which the cloud-native paradigm with services decomposed into microservices is utmost relevant. This in turn raises the issue related to the distribution of network functions. Even so, there are a plethora of existing works on the placement of Virtualized Network Functions (VNFs), most of them put forth on balancing the load and examining the resource (CPU, RAM, disk) needs of network functions with

respect to the resource availability in data centers. There are quite a few works addressing the microservices placement problem from the perspective of latency requirements by considering either the processing delay or the link capacity over a distributed data-center. Through our detailed survey study (provided in Chapter 3) on various placement and container migration strategies, we also tend to provide in-depth knowledge about recent technologies. Aiming to investigate, the first contribution tends to address the deployment of network functions from the different scenario notably by considering the microservice inter-dependency and the amount of traffic among microservices that led to increase the service latency due to the delay associated with messages transiting through the transport network connecting data-centers. The proposed contributions detailed in Chapter 4 that has been published & presented in 2022 IEEE CCNC conference [42] gathers following contributions :

1. an Integer Linear Programming (ILP) optimization model to place the services decomposed into microservices on a substrate network representing the three-tier architecture of Cloud-Fog-Edge nodes.

2. Latency-effective placement that considers the message exchanged between the microservices in order to map highly communicable microservices on the same data center & near to the end-user so as to minimize the E2E latency and delay.

3. We envisage two variants to select the closest computing node in a neighbouring cloud :

   (i) by ignoring the transmission delay between clouds; in that case, the selection is **network agnostic**;

   (ii) by taking into the account the transmission delay between data center (i.e. number of messages and transmission delay); in that case, the selection is **network aware**.

4. To solve the optimization problem a fast hybrid heuristic approach has been proposed - greedy and an advanced genetic algorithm.

1.4.2 Dynamic microservices migration algorithms and mechanism while guarantee the service continuity: Heuristic & Meta-heuristic approaches

With the adoption of the microservice paradigm by the telecom industry in the design of 5G/6G networks, complex network functions are decomposed into sets of chained sub-functions, which are further deployed using containerized technologies over geographically distributed cloud clusters. Latency-sensitive applications require to carefully orchestrate the allocation and re-arrangement of (micro)services to prevent from a largely segmented placement of microservices.

The existing research studies on VNFs placement and re-arrangement ignore the joint problem of chaining the microservices: in practice, network functions are placed focusing on the resource availability/need and/or migration time while omitting to consider the latency associated with the communication between the chained microservices and the end-user that would allow the end-to-end latency to be optimized. Therefore, compared to previous works, the proposed work jointly tackles the optimal placement and migration strategy for real-time scenarios, where arrival and departure of services are insignificant, aiming at minimizing the network delay and end-to-end latency between users and services.

The following gathered contributions detailed in Chapter 5 has been published in IEEE ICC 2023 conference [44] where the extended version in Journal of Network and Systems Management 2023 [45].

1. We formalized the optimization problem of migrating microservices across several data centers by ensuring the lesser number of microservices are moved while keeping the placement optimal.

2. The design rationale puts the more forth on user-centric and highly-active microservice while selecting and migrating them vertically (from the edge up to the cloud) or horizontally (between the data centers at the same layer).

3. We introduced three heuristic and meta-heuristic algorithms to solve the optimization problem that considerably reduce run time of the migration algorithm.

4. We implemented a simulation-based evaluation that shows the efficiency of migration of services using our proposed algorithm and minimizes the latency.

1.4.3 Live migration of containerized microservices between remote Kubernetes clusters: PoC

Many of the VMs' live migration techniques have been extensively researched in recent decades. However, the unavoidable factors of containers are pushing the industry to shift into container-oriented techniques. Many of the aforementioned research works try to handle the container migration for different use-cases while ensuring the respective Quality of Experience (QoE)/QoS for the cloud infrastructure. Still, there is a key demand of an orchestrator that can handle the multiple clusters along with different types of clouds (such as, edge/fog/core) to deploy applications for 5G and MEC services. The final work contributed as a Proof-of-Concept (PoC) of this PhD detailed in Chapter 6 gathers two contributions :

1. Testbed deployment illustrating the implemented mechanism of pod migration between remote K8s clusters. The considered use-case is an open-source 5G core network (namely, Magma).

2. Using a controller, the whole migration process can be executed automatically and able to tackle the connection alive during termination and re-establishment of connections.

1.4.4 Deployment of complete chain of 4G/5G mobile network

As part of internal research work, we also have three main principles on the usage of open-source software for the 4G/5G service deployment and orchestration as shown in schema diagram 1.2.

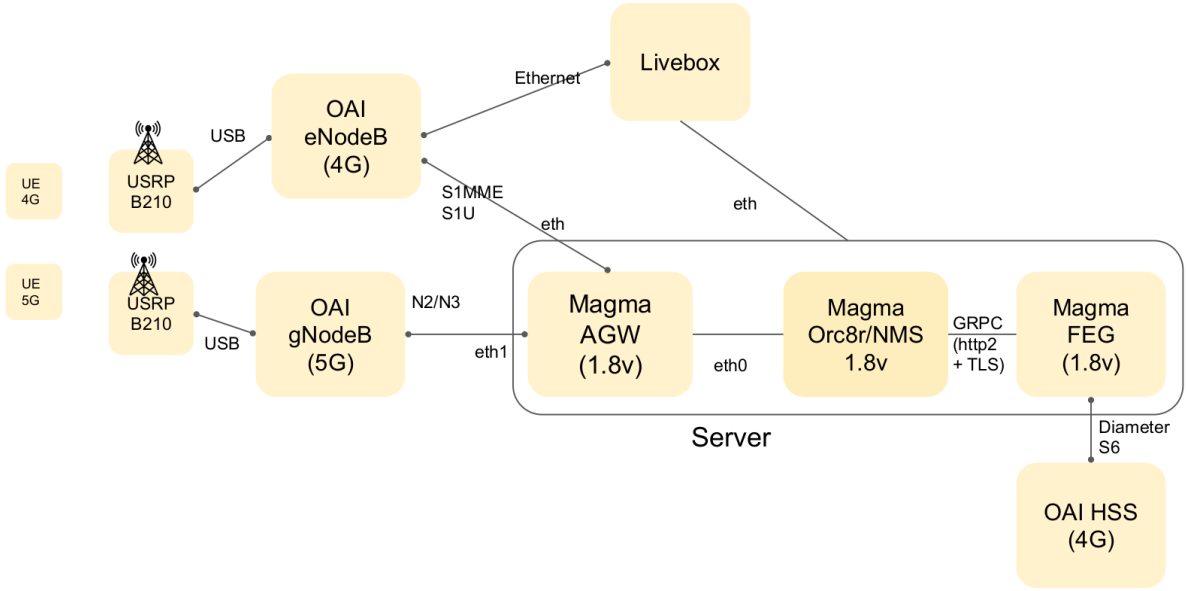


Figure 1.2: 4G/5G Chain Setup Schema

1. Test an open-source solutions for an end-to-end 5G standalone network.

2. Implement and evaluate a Neutral Host scenario using an open-source core network solution (namely, MAGMA).

3. Study service orchestration using operators in Kubernetes.

To meet these requirements of E2E solution, two open-source software components were used including Magma (1.8v-5G) and OAI (eNodeB, gNodeB, HSS). Magma has three components namely - Access gateway,

Orchestrator and Federation Gateway. The detailed description of an architecture has been provided in the Chapter 6. This further has been included as a use-case in our Proof-of-Concept.

1.5 Publications

The table 1.1 represents the list of published PhD works - their title (along with citation), publication venue, publication type and the current status.

Table 1.1: Summary of publications

#	**Publication title**	**Publication venue**	**Publication type**	**Status**
1	Latency and network aware placement for cloud-native 5G/6G services [42]	IEEE 19th Annual Consumer Communications & Networking Conference (CCNC 2022)	Full paper	Published
2	Container placement and migration strategies for cloud, fog, and edge data centers: A survey [43]	International Journal of Network Management 2022	Journal paper	Published
3	A microservice migration approach to controlling latency in 5G/6G networks [44]	IEEE International Conference on Communications (ICC) 2023	Full paper	Published
4	Live Migration of containerized microservices between remote Kubernetes Clusters [46]	IEEE INFOCOM 2023 Workshop	Full paper	Published
5	Dynamic migration of microservices for end-to-end latency control in 5G/6G networks [45]	Journal of Network and Systems Management 2023	Journal paper	Published

1.6 Organization of the manuscript

The rest of the manuscript is organized as follows: Chapter 2 introduces the background on containerization and various migration techniques; Chapter 3 details the state-of-the-art study analysis; Chapter 4 describes the first contribution of this PhD thesis, titled, *Latency and network aware placement for cloud-native 5G/6G services*; Chapter 5 presents the second contribution of this PhD thesis, titled, *Dynamic migration of microservices for end-to-end latency control in 5G/6G networks*; Chapter 6 represents the Proof-of-Concept and the final contribution of this PhD thesis, titled, *Live Migration of containerized microservices between remote Kubernetes Clusters*; and Chapter 7 concluded the manuscript.

Chapter 2

Background on Containerization and Migration

Content

2.1 Introduction

The last decade has witnessed important development of network softwarization that has revolutionized the practice of networks. Virtualized networks bring novel and specific requirements for the control and orchestration of containerized network functions that are scattered across the network. In this regard, the migration of virtualized network functions plays a pivotal role to best meet the requirements of optimal resource utilization, load balancing and fault tolerance. The purpose of this chapter is to offer a detailed overview of the progress on container migration so as to provide a better understanding of thesis background. Following, a taxonomy of the migration techniques that perform the transfer of the containerized microservices is proposed.

2.2 Microservices as an IT paradigm

Traditionally, monolithic architecture has been used in building applications where all the components are put into a single process. The advent of the microservice architecture has the potential to overcome the cons of typical monolithic approaches. Indeed, using the microservices paradigm when implementing network services brings various advantages. Intrinsically, the microservices are resilient, decoupled, scalable and independent. However, some challenges need to be addressed when conceiving network functions as a chain of interconnected (and even distributed) microservices.

2.2.1 Microservices principles

Further, we detailed the key principles of microservices-based architecture, also illustrated through figure 2.1.

- **Resilient:** In microservice-based systems providing resiliency for the elasticity is one of the pivotal components that ensures rebuilding and continuous serving even in case of disruption of one or more services. Hence, failure of one microservice will not lead to the whole system to crash.
- **Complete:** Each microservice is focused on the particular task unit of the system and facilitates the complete features for which it is owned.
- **Scalable:** In the context of modern infrastructures, scalability is another key aspect to handle the enormous amount of workload by appending resources in the network. As per demand or requirement, it is possible to create a several replicas of microservices together with the use of containers to balance the workload.
- **Independent development & deployment:** The distributed microservices communicate via message passing or a lightweight mechanism (such as, HTTP resource API) which is developed, maintained and managed independently. More complex and interdependent systems could enhance the chance of failure. Therefore, due to independent deployment the implementation of new microservices features tends to be faster. Also, the efficiency of programming in any programming language leads to more flexibility.
- **Decoupled:** Microservices are loosely coupled which means they perform and execute their tasks without knowing the architecture or implementation of other microservices in the system.

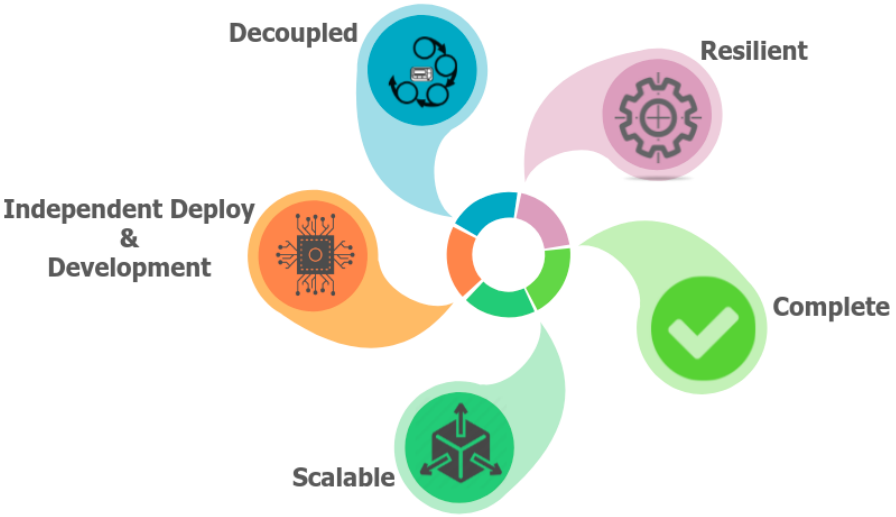
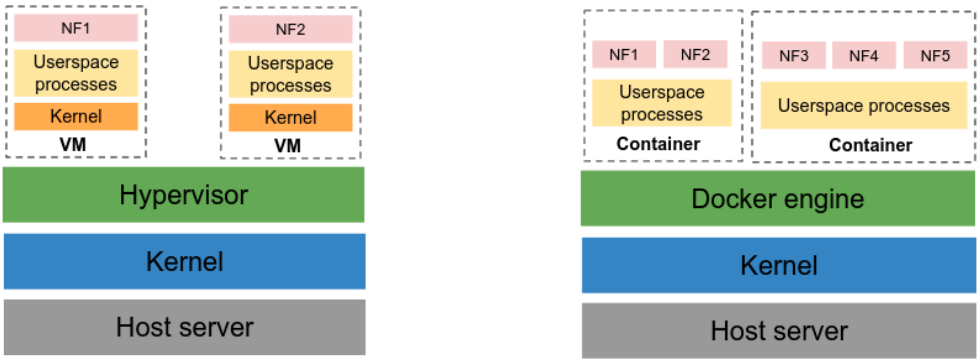


Figure 2.1: Key principles of microservices-based architecture

2.3 Virtualization vs Containerization of network function

Virtualization [21, 36] is a technology intended to create a software-based (i.e. virtual—representation) of applications, servers, storage and networks. In practice, virtualization consists of running multiple operating systems (OSs) or applications on top of a single physical infrastructure. It enables these applications or OSs to run in isolation while sharing the same hardware resources. VM-based virtualization (a.k.a hardware-level virtualization) and containerization (a.k.a operating system level virtualization) are two principle technologies used to facilitate the hosting and deploying of a large set of NFs/applications across multiple and distributed servers. They share some common features. Still, some of the significant factors differentiate VM-based virtualization and containerization from each other.

VM-based virtualization involves a hypervisor which virtualizes the resources and provides an abstracted version of the entire hardware of a physical machine, including e.g. the CPU, memory, and storage. Multiple VMs may run on a single physical host to encapsulate the several network functions running on their own operating systems. As illustrated in Figure 2.2, a VM runs its own operating system and applications. By contrast, containerization (operating system virtualization) virtualizes resources at the OS level: it virtualizes the operating system kernel in a way such that applications running on shared kernel are unaware of other competitors. Containerization encapsulates the application/service along with its dependencies and configuration into lightweight solutions called containers. Multiple containers can be hosted on a single OS which overcomes the need of creating different OS for deploying applications as in virtualization.



(a) Virtualization (b) Containerization

Figure 2.2: Virtualization vs Containerization

Further, the term VNF corresponds to an application operating on a VM to deliver several NF related to e.g. routing, filtering (firewalls) and load balancing. VM is often referred to as “monolithic" since it corresponds to a single, all-in-one unit (Fig. 2.2-a) running a full operating system along with several applications on top of a virtualized hardware. As such, VM tends to consume a high percentage of host resources and is not that portable. In particular, VM migration involves a time-consuming workload transmission and a significant deployment time.

The shift from the traditional architecture to the cloud-native approach is intended to overcome the limitations of VNF by distributing small functions across many out-of-the-box, loosely coupled microservices. Henceforth, the CNF approach is widely adopted in recent network designs (by e.g., Rakuten in Japan or Dish in the United States) to offer a more scalable, flexible and portable solution. In particular, the 5G core NF should be packaged as a chain of small units (i.e. microservices) that are subsequently distributed. Using platforms such as OpenStack and Kubernetes, containerized functions can automate container migration, deployment and recreation comparatively faster while efficiently responding to demands and dealing with failure recovery.

Below in table 2.1, we compared the main characteristics to represent the difference in terms of migration based on VM vs container.

2.4 Classification of Container migration

Container migration refers to the process of transferring or moving the components of a network function hosted within a container from one physical server (source node) to another one (destination node), possibly

Table 2.1: Comparison of VM and container live migration

VM migration	**Container migration**
NF deployed on VM (i.e. VNFs) are migrated considering their dependencies.	Migration of Cloud-Native Network Function (CNF) involves migrating the NF deployed on containers.
Given the large size of VM (expressed gigabytes in general), VM migration/backup is time consuming.	Container images are light-weighted and comparatively smaller in size (megabytes in general) and are thus quicker to relocate or migrate.
VM migration encloses migrating/transferring the CPU state, memory content, network connections/configuration and disk image.	Container migration involves the transfer of NF memory, file system and network connectivity.
Spin-up time takes minutes.	Spin up time takes milliseconds.
Each VM integrates its own OS which serves as a virtual server and requires significant resource usage. Though, fewer VMs can be deployed on physical servers.	Multiple containers can be deployed on top of a single OS resulting in less resources consumed to deploy and run containers on a physical server.
During migration, larger dump size transfer takes time.	While migrating container, small dump size is transferred.
While creating a VNF of a physical component, vendors usually create a large VM of the entire physical component. VNF is hence heavy and the approach lacks of scalability that would be attained with a distributed deployment across the cloud infrastructure.	On the contrary, CNF make use of (many) microservices easily distributed on many lightweight containers that leverage portability, increased flexibility and scalability.

interrupting the network function operation. Network functions that are migrated are either *stateless* (i.e., no past data nor state needs to be persistent or stored) or *stateful* (i.e., the application state lasts and is stored, e.g., on disk). With stateless network function, migration is quite straightforward [81] because the container operates in an isolated manner and is hence portable: the stateless container is simply re-allocated and restarted from scratch without conserving the existing state. As depicted in Figure 2.3, there exists several techniques for moving the container from the source to the destination. They subdivide into cold and live migration depending on whether the containerized service should remain active and network-accessible during the whole migration.

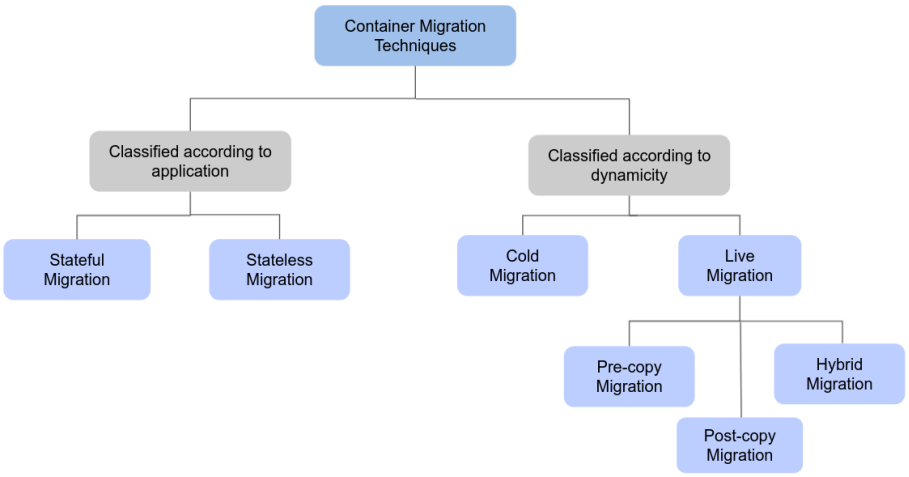


Figure 2.3: Container migration Techniques

2.4.1 Cold and Live Migration

There exists two ways of migrating a container: during the migration the containerized application is inactive (cold migration) or remains active (live migration).

2.4.1.1 Cold migration

This is the trivial form of migration in which the container is simply suspended and migrated between hosts. As illustrated in Figure 2.4, cold migration involves the freeze-transfer-resume steps: First, the container is freezed to ensure its associated state is not modifiable. Second, the dump state is transferred while the container is stopped. After the reception of the state at the destination node, the container is finally re-started and its state is resumed. Overall, cold migration involves a service downtime and thereby should be used in specific cases only, for instance when users are not using the service for a given time period or when the

downtime is planned and users are informed.

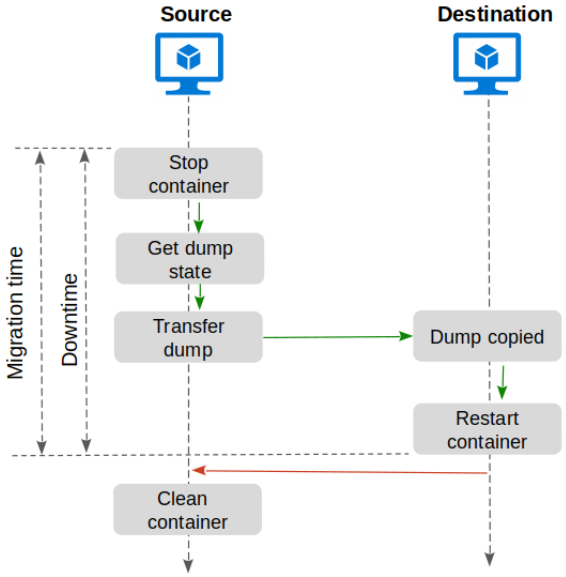


Figure 2.4: Cold migration

2.4.1.2 Live migration

Live migration consists of migrating a running container without service interruption, i.e., container migrates from one node to another while it is running. The main portion of the state is transferred while the container is running; the container is stopped only during the transmission of the execution state. Therefore, service downtime is quite negligible for the end-user.

Both cold and live migration entail the transfer of the original container. In practice, migrating an inactive service (cold migration) involves shutting down the running instance and thereby eliminating the need to handle the memory state. Instead, moving an active service (live migration) necessitates maintaining state consistency during the migration. In particular, in-memory state (including both kernel-internal and application-level state) should be moved in a consistent and efficient fashion. With live migration, the main concern lies in maintaining state consistency (as will be shown) while keeping to a minimum downtime (i.e. time between the container stops and resume) and total migration time (duration between when migration is initiated and when the container may be finally discarded at the source.

2.4.2 Handling State Consistency with Live Migration

Live migration can be approached in several ways: memory state can be sent ahead of time before the container is transferred (pre-copy) or later, i.e., after the container is transferred (post-copy) or combining the pre-copy and post-copy migration techniques (hybrid).

2.4.2.1 Pre-copy Live Migration

As shown in Figure 2.5, the container at source continues to run while pre-dump states are transmitted from source node to destination node. Therefore the service stays responsive during the transmission phase. At that time of copying and transferring of the pre-dump state, memory is kept modifiable at the source node. Then, the container is stopped and restarted at the destination node. The dump state and the memory content (memory pages) that have been modified are transferred. The service downtime (i.e., time between when the container is halted and resumed) is minimized because the container is stopped after the transmission of its state while the memory is also changeable.

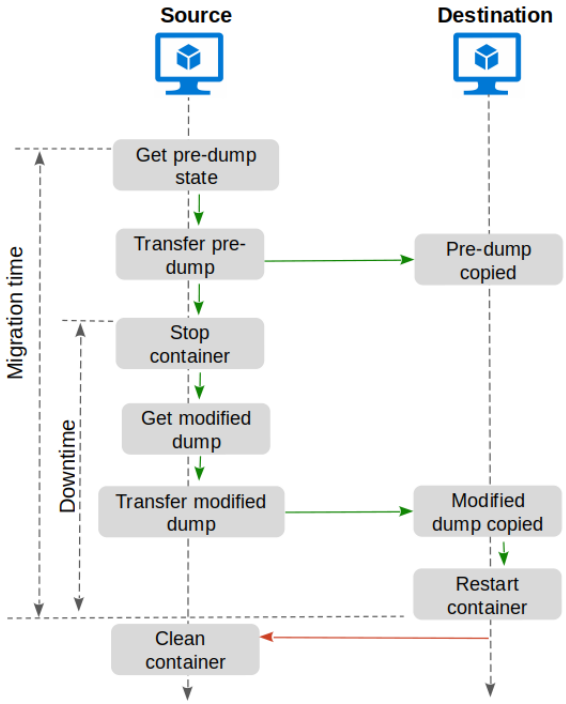


Figure 2.5: Pre-copy live migration

2.4.2.2 Post-copy Live Migration

As Figure 2.6 depicts, the process is initiated by first halting the container at source node, the (minimal subset of) execution state is transmitted to the destination node and the container is resumed as soon as possible based on its latest execution state. Later on, the remaining state (including memory pages) is transferred to the destination node before deleting the container at source node. At the destination, if the restarted container attempts to access a memory page that is not yet available, fault page is demanded to the source node, hence causing an additional delay.

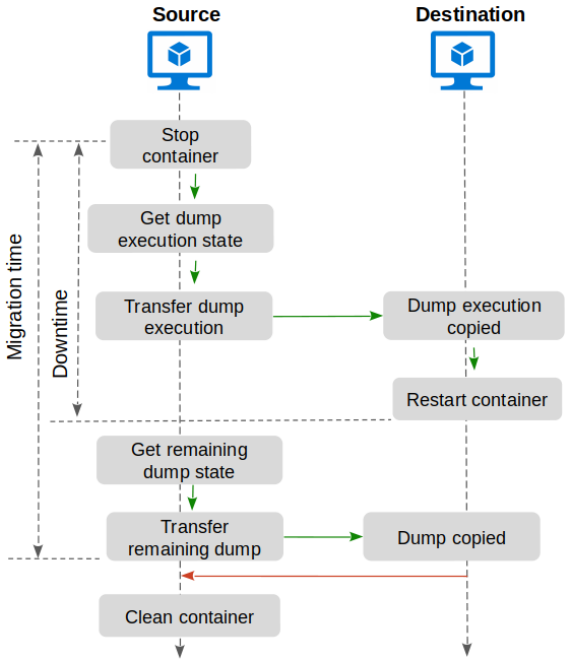


Figure 2.6: Post-copy live migration

2.4.2.3 Hybrid Live Migration

As shown in Figure 2.7, hybrid approach advents by combining the pre-copy and post-copy migration techniques. Following the pre-copy approach, the pre-dump state is transmitted while the container is still alive at the source node. After halting the container, the full dump state (modified and execution state together) is transmitted. Then, the container is restarted using the full dump state. Final step proceeds to transfer the memory contents (faulted pages) that were caused during the pre-copy phase. Hybrid migration addresses the issues related to non-deterministic downtime with pre-copy migration and performance degradation by dint of faulted pages in the post-copy migration approach.

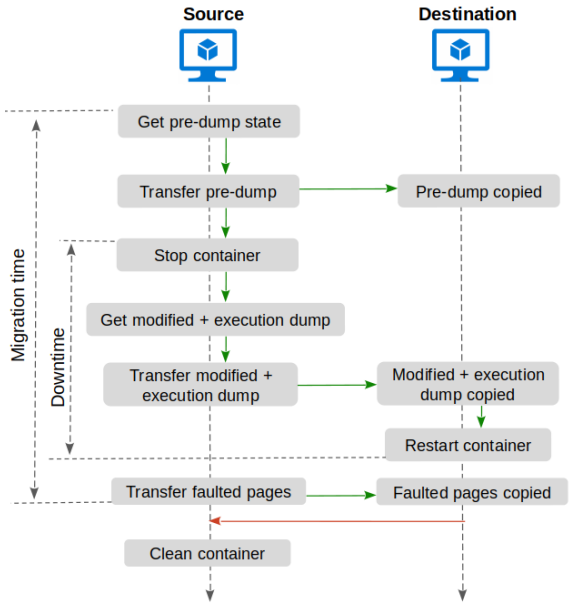


Figure 2.7: Hybrid live migration

In practice, pre-copy, post-copy or hybrid migration is performed using a snapshot/restore tool such as CRIU[1], which has become a de facto standard to handle migration of linux container with OpenVZ, LXC, and Docker. CRIU is an open source tool that dumps the state of processes/containers into a collection of image files on disk and makes it possible to further resume an app (i.e., to restore an app) from exactly where it was suspended. Nonetheless, CRIU has some limitations. CRIU focuses on the internal state of the containerized application, which includes the states of the CPU, registers, signals and memory that are associated with the container. CRIU does not transfer any file/state across physical nodes. To this aim, complementary techniques shall be used to dispose of the files/information necessary for recovery at the destination node. In practice, files are transferred using the rsync primitive or a shared and possibly distributed file-system such as NFS, GlusterFS, or Virtuozzo[2] that are used to store files and avoid transferring them.

2.4.3 Storage Migration

Typically, the state of a network function is local (i.e., accessed by the container by a virtual local disk) if the state is frequently accessed. For example, per-flow state (such as state for individual TCP connections) is local, as long as the traffic is distributed on a flow basis. In the container, the internal state is stored with the network function instance and thereby achieves good performance (e.g. fast read, write). Early work, e.g., [76], on

[1]https://www.criu.org/
[2]https://wiki.openvz.org/Virtuozzo_Storage

NFV management assume that the state is internal; this assumption permits easy migration and elastic scaling of network functions. In practice, the state is then migrated as part of the container image.

Nevertheless, the transfer of the whole container file system results in a high network overload. In order to optimize and reduce the size of the container file system that is transferred from the source to the destination node, a number of works [40,62] take advantage of the layered structure of Docker. Docker storage is formed of several layers: base image layers are read-only while upper layer is read-write. Read-write layer encapsulates all the file system updates issued by the container since its creation, which encompasses (i) the files created by the containerized application as well as (ii) the files corresponding to the updated versions of the read-only layers. Thus, read-only layers can be fetched before the migration from a Docker repository (e.g., public cloud repository such as Docker Hub[3] or self-hosted image hubs) while the thin top writable layer is transferred from the source to the destination node. Following, [40] goes one step further and also checkpoint the current state of the read-write container layer, which further reduces the container's migration downtime.

Another line of research breaks the tight coupling between the NF state from the processing that network functions need to perform by externalizing the storage leveraging a resilient data store that is either central [8,27,41] or distributed [113] and that can be accessed by any NF. Nonetheless, any access (read, write, delete) to the externalized datastore involves a significant communication overhead. To reduce the communication overhead, in-memory data store is privileged in [53, 113]: the state is stored in DRAM leveraging RAMCloud [75] which corresponds to a key-value in-memory datastore with low latency access or Redis [4]).

Another approach introduced a variant of CRIU named VAS-CRIU that avoids costly file system operations that dominate the runtime costs and impact the potential benefits of manipulating in-memory process state. Contrary to CRIU that suffers from expensive filesystem write/read operations on image files containing memory pages, VAS-CRIU saves the checkpointed state in memory (as a separate snapshot address space in DRAM) rather than disk. This accelerates the snapshot/restore of address spaces by two orders of magnitude, and restore time by up to 9 times.

2.4.4 Applicability and Performance Evaluation

Few empirical studies evaluate the performance of container migration such as [51, 81, 84]. They compare the performance of various container migration techniques (e.g. cold, live migration) to that of VM migration

[3] https://hub.docker.com
[4] https://redis.io/

and consider multiple virtualization platforms. First, the referred work [81] analyzes the performance of cold and live - pre-copy, post-copy and hybrid - migration to identify the best techniques while transmitting stateful containers from one node to another. The comparison between cold and live migration indicates that, as expected, cold migration has the lowest total migration time and highest downtime in comparison to various live migration techniques because cold migration transmits the whole state at once after the suspension of the container at source node.

The delay associated to post-copy migration is high migration compared to that of cold migration as it passes on the faulted pages served on request from source node after resuming the container at destination node. Likewise, pre-copy migration depicts better results than post-copy migration when the network has sufficient throughput to convey changed pages quickly, which is the case if network throughput is greater than or close to the page change rate. Otherwise, pre-copy migration is less efficient compared to post-copy. On the other hand, the hybrid migration always involves higher migration time as it results from the combination of pre-copy and post-copy techniques.

Significantly, live migration keeps the container active during the migration process to reduce downtime and maintain responsiveness of the containerized service throughout the communication exchange. The evaluation of the downtime shows that the downtime is lower for the post-copy technique compared to the pre-copy technique and remains comparable to the hybrid technique. The evaluations concerning the amount of transferred data is also showing better results for post-copy, wherein the quantity of transferred data is always lower than for pre-copy and hybrid, but remains competitive to cold migration.

In [84], authors provide a detailed comparison of the performances associated with a VM-enabled and container-enabled live migration supporting the functions of core network functions, including the Home Subscriber Server (HSS), Mobility Management Entity (MME), and Serving and Packet Gateway (SPGW).

First, the analysis of the migration time associated with the HSS VM is comparatively twice that of the HSS container. It takes a modest amount of additional time to complete the VM and container migration process while using a longer path. On the other hand, containers incur a higher downtime than VMs because the containerized HSS is stopped on the source host when checkpointing is initiated and is resumed only once after the complete restoration at the destination host.

Second, the MME VM has a migration time six/seven times higher than the MME container, as the network load and metadata size of the container is comparatively smaller than the VM. Therefore, the large image

size and longer path clearly have an impact on the migration time of the VM. Conversely, the analysis of the container downtime shows double that of the VM because the migration process has to be stopped at the checkpoint stage and restarted only after restoration.

Finally, the SPGW VM also implies much higher migration time than the container due to the large size of the metadata for the VM. However, an interesting result can be observed: the downtime improves for the SPGW VM compared to the container migration, which was not the case with HSS and MME. During the SPGW migration, the UE recovery time is affected by the new UE connection that has to be successfully re-established by updating the sockets after the temporary failure occurred.

The work [51] analyse the real-time behaviour of containers in the cloud environment, under two distinct workloads (100% and 66%). With regard to total migration time, downtime and disk utilization, Linux Containers (LXC) exhibits better outcomes compared to Kernel-based Virtual Machine (KVM) except for the CPU utilization which is better with KVM. In particular, the downtime of KVM is increased by 1.6 and resp. 1.75 times with the workload of 66% and resp. 100% in comparison to LXC. Similarly, the migration time of KVM is 1.35 and 1.45 times higher compared to LXC at the workload of 66% and 100% respectively. Similarly, the live migration with KVM and LXC which has an impact on on their disk utilization. The highest disk utilization of KVM is 455,555 writes/sec and 482,672 writes/sec at the workload of 66% and 100% respectively. Whereas, LXC has a maximum disk utilisation of 301,192 writes/sec and 330,528 writes/sec for a workload of 66% and 100% respectively. Moreover, the evaluations related to CPU utilization shows that LXC has a maximum CPU usage of 78.12% and 86.24% for a workload of 66% and 100% consecutively. However, KVM on the other hand performs better outcomes by lowering upto 73.09% and 74.07% for 66% and 100% workload respectively.

2.5 Conclusion

This trend of application architecture to find a better way of building and establishing the system has been driving the industry towards more of cloud-native solutions. Also, the continuous advancement of this domain always provides a plethora of opportunities to learn new things and contribute to it. Therefore, before going into detail of state-of-the-art methodologies and practical implementations explained in the following chapter, we first introduced the chapter focused on background knowledge on methodologies - virtualization & containerization, microservice architecture, along with various types of migration techniques.

Chapter 3

State-of-the-art on placement and container migration strategies

Content

3.1 Introduction

A notable trend in current networks is the network softwarization that promotes the adoption of virtualized and containerized technologies to support the rapid development of new services that readily adapt to the evolving customer needs. Network softwarization leads to the gradual replacement of hardware network function operating on purpose-built & proprietary network equipment by Virtualized Network Functions (VNFs) that are consolidated on commodity hardware. In practice, a network function (NF) may offer a wide range of networking capabilities that operate on the Universal Customer Premise Equipment (uCPE), up to the core

network supporting e.g. tunneling, firewalling or application-level functions. Microservices have become instrumental in the design of complex NFVs that necessitate a decomposition into many of services, e.g., several hundreds services for core network functions. In such case, microservices are small services implementing a limited amount of functionalities that can be executed independently (even if they are logically dispersed at the edge, fog or in the cloud); each microservice executes its own processes/functionalities and communicates via lightweight protocols. Overall, cloud-native design offers a different approach to the development of softwarized networks, an approach that is suited to the agility and that supports an efficient scaling up and orchestration of the distributed network functions. Container-oriented approach is also increasingly privileged as a containerized microservice can be rapidly instantiated as required and also can be scaled-out independently, to support the increasing demand for more processing or storage, without unnecessarily scaling the overall network function.

Following that, this chapter spanned around the container placement & migration strategies for Cloud, Fog and Edge computing levels. The objective of placement strategies are to identify the appropriate target server(s) to allocate the migrated services. The proposed classified container migration techniques reflect the way the service components hosted in a container are moved from one (or several) physical server(s) to another one(s). Subsequently, we perform a holistic review of the strategies for container migration over a geographically spanned network (edge, fog, core and cloud levels) and describe the frameworks and algorithms that have been used to migrate container-based services.

The organization of this chapter is as follows: In Section 3.2, we present the placement strategies, the subsequent Section 3.3, detailed the container migration strategies for networks geographically spanned and followed by Section 3.4, detailed the comparison on migration strategies based on model and algorithms. The concluding remarks are presented in 3.5.

3.2 Container placement strategies

The migration of a set of CNFs is known to effectively bring more elasticity and scalability to (mission/latency-critical) applications. On the other hand, migration may entail service disruption and may come at the cost of intensive use of computing and communication resources, even though there is a strong practical need for migration. The service migration entails taking a decision concerning the service placement, which consists in determining whether, when and where to migrate. The service placement problem is an optimization problem

that involves a balanced trade-off between the cost associated with the migration and the expected benefits.

Generally speaking, the service placement problem (Table 3.1) is usually framed as a mathematical optimization (Integer Linear Programming and Mixed Integer Linear Programming), which is further solved by an optimization solver, heuristic methods or Machine Learning (ML) approaches. ILP and MILP solvers typically find nearly optimal solutions but are quite time-consuming and hence are not practically viable for large and complex problem instances.

Table 3.1: Classification of placement methods

Methods	**Reference**
Interger Linear Programming (ILP)	[17, 35, 42, 56, 60, 90]
Mixed Interger Linear Programming (MILP)	[33, 34, 49, 55, 57, 91]
Heuristic Method	[17, 34, 49, 55–57, 60, 90]
Machine Learning (ML)	[11, 12, 67, 78, 83]

Instead, heuristics (e.g., greedy algorithms) and meta-heuristics produce comparatively faster but sub-optimal results that usually achieve less objectives (e.g., low response time or reduced communication delay or load balancing or limited energy consumption). On the other hand, ML-based approaches (e.g., genetic algorithm, ant colony) are known to be more accurate solutions [12] thanks to their interactive learning and decision making abilities.

As detailed in Table 3.2, the above container placement strategies can be further categorized based on target architecture (cloud, fog, edge), type of placement (static versus dynamic), key objectives, algorithm to solve and evaluation method. In the case of static placement, an initial placement is typically proposed only once (at start). Instead, dynamic placement involves multiple reallocation decisions that are made over time. A new placement is proposed e.g., in case of overuse/under-use of computing or network resources, inflow/outflow of service instances [54, 118]. Contrary to the static placement which is done based on initial constraints (e.g., expected delay/latency, initial bandwidth usage and initial resource availability), the dynamic placement involves a continuous monitoring of the physical resources and network to support the selection of the appropriate hosting server(s) and/or data center(s) despite changing resources/requirements. Static and dynamic placement strategies attempts to enhance the service quality and/or reduce the operational cost by means of various strategies, particularly: *1) Resource-aware placement:* to avoid unwanted overuse/under-use of resources and decrease operational cost by balancing the load among distributed data-centers and hosts; *2) Latency-aware placement:* to facilitate the fast inter-communication considering processing and migration

delay, transmission and queuing delay; *3) Security-aware placement:* to avoid the allocation on container owned by an adversary user, identifying the unexpected threat or failure and cross-container attacks. Once the placement decision has been made, migration must be carried out.

Furthermore, we detail in Table 3.3 a set of NFVs/ VNFs allocation approaches that exploit machine-learning strategies. Deep Reinforcement Learning (DRL), which is the most popular algorithm, consists in training the agent relying on heuristic approach. It turns out that the A3C algorithms DRL and DQN algorithms proved to be very efficient for the placement of Service Function Chain (SFC) as shown in [11, 48].

3.3 Strategies for container migration techniques

As shown in Figure 3.1, container migration schemes can be classified into three computing layers, which form the underlying virtualization infrastructure.

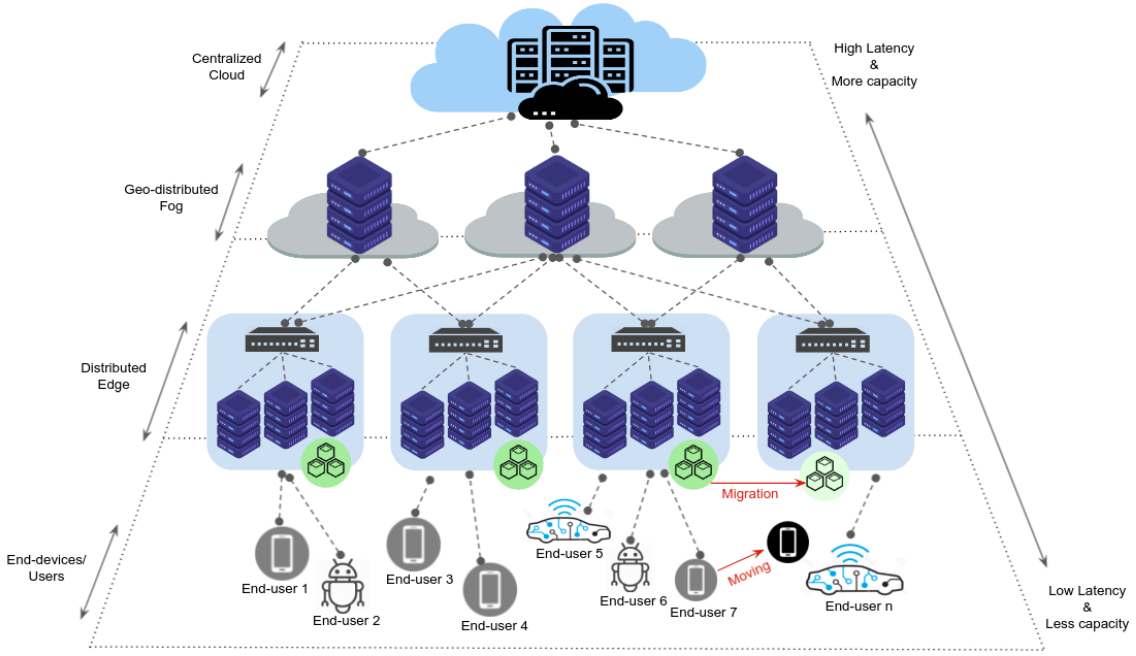


Figure 3.1: Three-layered Cloud-Fog-Edge Infrastructure

The topmost cloud layer constitutes the largest centralized storage and computing resource along with high scalability that is persuasively acquired by end-users in an on-demand manner. The utilization of container-based infrastructures for large-sized environments evidently constitutes a popular choice by dint of its key characteristics - lightweight, scalability, and high portability. Moreover, the cloud-native principle enables

Table 3.2: Comparison of existing works related to placement of containers/instances

Ref.	Architecture	Placement Type	Objectives	Algorithm	Evaluation
[18]	Fog	Static	Response time, Inter-container network communication	Greedy & Genetic Algorithm	Comparison to 3 approaches
[28]	Fog	Static	Response time of task	Ant colony optimization	Simulation
[74]	Edge	Dynamic	Scheduling	Reviewed heuristic-based algorithms	Case study
[80]	Cloud	Dynamic	Rebalancing, Load balancing	Scheduling & Rebalancing process	Simulation (with real-time load)
[38]	Cloud	Dynamic	Resource utilization, Number of instances	Best Fit (BF), Max Fit (MF) & Ant Colony Optimization based on Best Fit (ACO-BF)	Simulation with real-time workload (compare three algorithms)
[120]	Three-tier (Container-VM-PM)	Dynamic	Resource utilization	Best-fit	Use-case
[121]	Cloud	Dynamic	Traffic flow, Placement cost, Resources	One-shot, Rounding and heuristic algorithm	Theoretical Analysis and trace-driven simulations
[61]	Cloud	Dynamic	Communication cost, Load balancing	Communication Aware Worst Fit Decreasing (CA-WFD), Sweep&Search	Extensive evaluation on Baidu's data centers (Comparison with exiting SOA strategies)
[22]	Edge	Static	Container images' retrieval time	KCBP (k-Center-Based Placement), KCBP-WC (KCBP-Without-Conflict)	Trace-driven simulations (Compared with Best-Fit and Random)
[72]	Edge-Fog-Cloud	Dynamic	Service delay, Resource management	Particle-swarm-optimization (PSO)-based metaheuristic, Greedy heuristic	Use-case benchmarking (comparison of 4 approaches)
[119]	Containers as a service (CaaS)	Static	Energy consumption	Improved genetic algorithm	Compared with other 6 algorithms
[50]	Edge-Fog-Cloud	Dynamic	Automate database container placement decision	Markov Decision Processes (MDP)	Testbed
[42]	Edge-Fog-Cloud	Static	End-to-end service Latency	Greedy & Genetic Algorithm	Evaluation of proposed strategy solved using 2 algorithms

Table 3.3: Comparison of existing machine learning based placement strategies of VNFs/instances

Ref.	Archi.	Placement Type	Objectives	Algorithm	Evaluation
[116]	Edge	Dynamic	Automatic embedding of virtual networks to optimize resources	DRL-Asynchronous Advantage Actor-Critic (A3C) algorithm, Graph Convolutional Network (GCN)	Compared to 5 algorithms
[88]	Cloud	Dynamic	Orchestration cost & monitoring load	Machine-learning-based method ; unsupervised & reinforcement learning	Simulation
[48]	-	Dynamic	Placement cost, Services acceptance rate	Deep Q-Network (DQN) approach	Simulation
[117]	Edge-Fog-Cloud	Dynamic	Resource utilization, acceptance ratio among total arrived requests	DRL algorithm	Simulation results compared to 2 existing algorithms
[95]	Edge-Fog-Cloud	Dynamic	Resource scheduling and allocation while preserving user privacy	DRL	Proposed algorithm compared to 3 schemes
[114]	Edge	Dynamic	Operation cost of NFV providers & acceptance rate of requests	Policy Gradient based DRL	Extensive trace-driven results performed against SoA solutions
[82]	Edge	Static	Inter-domain load balancing	DRL & Cost-based First Fit Algorithm (CFF)	Evaluated using Internet topology (Zoo) considering 4 configurations
[12]	Edge-Fog-Cloud	Static	Resource utilization, placement request acceptance ratio, load balancing	Heuristically assisted DRL-A3C & GCN	Simulation
[11]	Edge-Fog-Cloud	Dynamic	Realistic and non-stationary network load & traffic changes	hybrid DRL-heuristic algorithm	Simulation, 4 versions of proposed scheme evaluated
[83]	Edge-Fog-Cloud	Static	Resource allocation, acceptance ratio	Enhanced Exploration Deep Deterministic Policy Gradient (DDPG) - Heuristic Fitting Algorithm (HFA)	Simulation results compared to other approaches
[85]	Edge	Static	Resource allocation	DRL - Trained RL agent & modeled policy using Relational Graph Convolutional Neural-based architecture	Results compared to First-Fit and Best-Fit strategies

network services to be implemented as a bundle of microservices interconnected to each other and deployed on distributed and container-based infrastructures (e.g., Kubernetes) [39] in the cloud. Nonetheless, there exists an inherent limitation associated with cloud computing: the long communication distance results in excessively long delay and the security factors in public cloud models risk the users privacy and unauthorized access to databases [58].

Fog computing provides a promising solution by decreasing the distance between end user's devices and cloud data centers. Cloud functions can be moved towards the end user device in the event of low-latency interactivity. In practice, containerized microservices migrate from centralized cloud to geo-distributed fog nodes [6, 81], which share the workload and lessen the network traffic. Therefore, strategies under fog perform the migration among geo-distributed and heterogeneous data centers. In such case, careful migration of data volumes plays a significant role especially for live and stateful containers. Nonetheless, microservice requesting more computing/storage resources can be offloaded from fog nodes to cloud data centers.

Further, the edge nodes located near the end users provide comparatively lower latency at a cost of limited resource capacity in comparison to cloud and fog servers. Edge clouds enable the deployment of servers near to the user to fulfill the demand of latency-critical applications.In particular, migration techniques map/migrate the containers from one location to another depending on the user moves. That, later on optimize the quality-of-experience (QoE) and network-related requirements by dynamically mapping the containerized services on container-based virtualized environment [66].

While a cloud-fog-edge architecture has the potential to unlock tangible opportunities for industry, it remains pivotal to rely on a mature container migration strategy. In the following, we consider the migration techniques that can be followed to support the migration at any layer of the virtualization infrastructure. Table 3.4 compares the proposed approaches based on their migration type, architecture, scope and considered factors to be handled during migration and the detailed explanation is also provided in the proceeding section. Compared to VM Migration that has attracted considerable interest, there are not so much works that address container migration within the cloud (§ 3.3.1), the fog (§ 3.3.2) or the edge (§ 3.3.3).

Table 3.4: Comparison of various container migration techniques

Ref.	**Type**	**Live/ Cold**	**Archi.**	**Scope**	**Factors to handle**
[115]	Pre-copy	Live	Cloud	Avoid duplicate Docker image layers transmission, manage container context	Migration downtime

[16]	Pre-copy	Live	Cloud	Automate live migration using Ansible along with traffic redirection	Migration time
[68]	Stateful and stateless	Live	Cloud	Protect from malicious attack	Migration time, Application downtime
[14]	-	Live	Cloud	Protection from malicious attack	-
[37]	-	Live	Cloud	Defensive approach against information leakage attack	Time & space migration
[13]	Pre-copy	Live	Cloud	Migrate VM/containers accross physical hosts and complicate the attacker process of placing VM/containers in the same victim/host	-
[40]	Pre-copy	Live	Fog	Transmit the least modified files before the actual migration from one fog node to another	Downtime
[24]	Stateless	Live	Fog	Support both horizontal and vertical migration	-
[63]	Pre-copy	Live	Edge	Reduce size of the file(s) to transfer, consider user's movement while migration	Migration time
[64]	Pre-copy	Live	Mobile Edge Computing (MEC)	Consider users location and select the nearest node to map container/VM	Service downtime, Migration time
[73]	Post-copy	Live	Cloud	Provide Just-In-Time (JIT) migration to access the data at target host during lazy data copying process running in background	Downtime, Performance overhead - read, write, update, scan workload
[70]	Pre-copy	Live	Cloud	Perform check-pointing and restart procedure for containers at the kernel-level ; facilitate the check-point and restoration of the running container state	Downtime
[58]	Post-copy	Live	Cloud	Allow migration of Intel SGX-enabled container used to protect data from untrusted access)	Migration time
[79]	-	Live	Cloud	Migration for RDMA-enabled containerized application	Analyse required modification in implementation, Migration time
[102]	Pre-copy	Live	Fog	Can integrate with Kubernetes clusters; allow backing up, restoring of states and migration from one Kubernetes cluster to another	Backup and restoration of resources

[84]	Stateful	Live	Edge	Container migration of the following network functions that are not supported by current CRIU and OpenAirInterface: Home Subscriber Server (HSS), Mobility Management Entity (MME), and Serving and Packet Gateway (SPGW)	Migration time, Downtime
[9]	-	-	Edge	Mathematical model to handle scalability issue where decision variable is considered to decide migration or re-instantiation	Downtime, latency
[92]	-	-	Fog	MDP problem to handle delay, power consumption and migration cost, solved using DQL & DNN algorithms	Migration cost

3.3.1 Container Migration on Cloud

CloudHopper [16] supports live migration of multiple interdependent containerized applications across multiple clouds over a wide network. The automated solution (relying on Ansible [1]) offers multi-cloud support for three commercial clouds providers (namely, Amazon Web Services, Google Cloud Platform, and Microsoft Azure). The migration of multiple interdependent containers necessitates a network migration to (i) easily locate the other containers and (ii) hold the incoming traffic during the effective migration and eventually redirect when the service gets restored and ready. For this purpose, an IPsec VPN is set up between the source and target and a TCP/HTTP load balancer (HAProxy [2]) is used and tuned to (i) redirect the http traffic and (ii) return unavailability message (HTTP 503 Service Unavailable Response) if timeout occurs during the migration. To support memory pre-copy, the CRIU iterative migration capability is leveraged. Rather than supporting a parallel transfer of the multiple containers, migration is scheduled: containers are ordered by size and large-size containers are migrated first. The next container starts its migration when the previous container has a remaining transfer size that is equal to its transfer size. This scheduling approach uses more efficiently the network bandwidth and enables to start all containers almost immediately upon arrival at the target.

Further, the work [115] adopts the pre-copy algorithm for docker migration across data centers of a cloud network. Different from VM, Docker has a layered image and Docker containers share the same OS kernel, which makes live migration of a Docker container more complex as image, runtime state and context should be migrated. The migration starts by transferring the base layers of the docker image that are read-only by disconnecting the storage volume at the source and re-attaching it at the target node. Then, CRIU performs incremental memory checkpoint and supports the iterative migration of the upper layer which is read-write and thereby possibly updated during the whole migration process. The experimental results show 57% lessened

total migration time, 55% lower image migration time, and 70% of downtime on average in comparison to mentioned state-of-the-art.

The work presented in [73] proposes a solution for live container migration named Voyager, which follows the design principle specified by Open Container Initiative (OCI) [4]. OCI is a consortium initiated by industry leaders (e.g. Docker, CoreOS) to encourage the common and open specifications of container technology. Voyager provides stateful container migration by using the CRIU-based memory migration and union mounts so as to retrieve source container data on the target node without copying container data in advance. As a result, migration downtime is minimized. Voyager supports the so-called just-in-time zero-copy migration where container restarts before transmission of whole states at destination node. This allows Voyager containers to instantly restart at destination host during disk state transmission by means of on-demand copy-on-write and lazy replication.

The live migration model ESCAPE [14] focuses on defense mechanisms for cloud containers by modeling the interactions between the attackers and respective victim hosts as a prey game. The container acts as a prey whose aim is to evade attacks/predator. For the checkpointing of a running containerized application while migration, the model employs an experimental version of Docker that includes the CRIU checkpoint tool. ESCAPE detects and circumvents attacks by either preventing any migration during an attack or migrating the container(s) far away from the potential attacker(s).

In [37], authors propose the frequent relocation of docker containers to reduce the impact of data leakage. Inspired by Moving Target Defense (MTD) technology, the approach promotes the container migration to shorten the container life cycle and thereby guarantee the security of large-sized multi-tenant service deployment. Similarly, the defense framework introduced in [13] offers fast and high frequency migration of VMs/containers so as to obscure the migration process for the attackers. In particular, the destination hosts are chosen randomly, which may degrade the performance by means of load and latency.

MigrOS [79] enables the transparent live migration of RDMA-enabled containers, which require specialised circuitry of the network interface controllers (NICs) and thereby are not transparently supported so far. The OS-level migration strategy requires a modification to the RDMA protocol but still supports full backwards-compatibility and interoperability with the existing RDMA protocol, without compromising RDMA network performance. In order to evaluate the solution, the modified RDMA communication protocol has been integrated with SoftRoCE, a Linux kernel-level open-source implementation of the RoCEv2 protocol. In addition, the solution is implemented in NIC hardware.

In [58], the first migration framework of Intel Software Guard Extensions (SGX)-enabled containers is presented. SGX provides a trusted execution environment named enclave for containers. An *enclave* [3] corresponds to a secure separate encrypted area used by a process to store code or data. The key challenge behind migrating SGX-enabled containers relates to the SGX security model that prevents the states of the enclaves, which are encrypted, to be accessed during the migration process. In order to support the migration of the enclave, the solution encrypts the persistent data stored of the enclave using a symmetric key that is shared by the source and destination node. An empirical evaluation shows that the migration of SGX-enabled containers introduces about 15% overhead. In [68], author secures the live migration of container for both stateful and stateless applications. Application server acts as a control manager that orchestrates the migration process. Also, a secure migration path is established using SSH/SFTP that support authentication, communication confidentiality and integrity.

3.3.2 Container Migration on Fog

The container migration strategy [40] within Kubernetes for stateful services in geo-distributed fog computing environments attempts to minimize the downtime. In case of stateful migration, it is required to migrate the disk state along with the container, which is a time consuming process with large-sized and distributed migration. To address this issue, the layered structure provided by the OverlayFS file system [71] is used to transparently snapshot the pod volumes and transfer the snapshot content prior to the actual container migration. At the source server, the snapshot content becomes read-only and a new empty read/write layer is added on top. Overall, the approach supports the check-pointing of the current state of the container layer. If needed, several snapshots transfers may be performed, which leads to minimizing the container's migration downtime: experiments on a real fog computing test-bed show up to factor 4 downtime reduction during migration in comparison to a baseline with no volume checkpoint.

In [24], the migration framework supports both horizontal migration where containerized IoT functions are migrated from one gateway to another gateway and vertical migration in which IoT function containers are migrated from the gateway located at the edge to the Cloud. The strategy is quite straightforward: the stateless container is re-created at the target node and then deleted from the source node.

The formerly known Heptio Ark project, currently stands out as Velero [102] to leverage the migration of Kubernetes applications and their persistent volumes. Compared to existing tools, it utilizes the Kubernetes API instead of Kubernetes etcd to extract and restore the states. This is advantageous when users do not have access to etcd databases. The resources exposed by API servers are simple to backup and restore even for

several etcd databases. Further, additional functionalities of backing up and restoring of any type of Kubernetes volume is provided by activating the restic [104]. The release of Velero is available on GitHub [103].
In [92], the migration strategy is modeled as multiple dimensional Markov Decision Process (MDP), which is solved using a combination of deep Q-learning and Deep Neural Networks (DNN) algorithms. The system states that are considered are delay, power consumption and migration cost. Further, the actions is based on a selection policy and follow a greedy approach that consists in choosing the source container to be migrated considering the under-utilization versus over-utilization of nodes. All containers on an underutilized node are transferred to allow that node to stop operating unnecessarily and thus minimize the number of nodes operating and thereby the total energy consumed by the nodes. At over-utilized node, the container with minimum migration cost is selected to migrate relying on an allocation policy that selects the target node for each migrated container. To enhance the performance of agents in terms of faster learning speed, they optimize the Double DQN and Prioritized Experience Replay (PER) during the training process. The resulting outcome showed better results in comparison with existing baseline strategies.

3.3.3 Container Migration on Edge

The work presented in [63] designs a third party tool to perform a live migration of services on edge infrastructure. The goal is to reduce the migration time by leveraging the layer structure of the docker container storage system. Docker image is composed of several layers. During the container's whole life cycle, only the top storage layer is changeable. The layers underlying the top layers remain unchanged. Therefore, the proposed strategy transmits only the top layer during the migration process, rest underlying layers have been transmitted before commencing the process. Moreover, authors consider the migration of the service to the end server located near the actively moving end-user: when a user shifts at a new location, then the offloading computation service also passes on to the edge server, which is closer to the end user's new location. In order to attain the fast migration and lessen network traffic, the proposed framework already starts preparing the target edge node before the commencement of the migration process and parallelizes & pipelines the following steps:

1. Parallelize the downloading of the images from a centralized registry at the target nearest edge node and pre-dump/send base memory images from source to target node while starting the container.

2. Reload the docker daemon on the target host (after halting the container at source node). The reload can also be parallelized with the dirty memory transmission from source to target host or could be trigger just after the transmission of latest container layer. Note that container layer can be compressed before

to transmit. Also, container layer compression and transmission can be pipelined. Similarly, the process of acquiring the memory difference at the target server could be pipelined.

The work [66] supports live migration based on Linux Container Hypervisor (LXD) and CRIU and introduces a novel heuristic scheme. The proposed heuristic follows these steps: First, a source node shortlists the containers that are characterised by high latency. For each high-latency container, the source node finds the neighbor node that is geographically closed and that is characterised by good resources availability (e.g. load, CPU, RAM, bandwidth) to migrate the container.

In order to perform the live migration of containers for latency critical industrial applications, the work [29] leverages the redundancy migration approach for edge computing. The approach skipped the stop-and-copy phase of traditional live migration that followed the snapshot and checkpointing, transmission and restoration of state image at the target node. Therefore, the key four composed phases are - 1) Buffer and routing initialization phase, 2) Copy and restore phase, 3) Replay phase and 4) switch phase. This approach significantly minimizes the downtime by a factor of 1.8 in comparison to LXD (Linux containers Daemon) stock live migration as per the evaluation.

In [9], another container-based solution has been proposed for edge applications. First, an ILP model aims at minimizing downtime and latency while handling the placement and migration in a cloud-edge environment. A decision variable is further considered to trigger migration or re-instantiation in case of failure. Also, a heuristic-based model with the primary focus on minimizing downtime has been implemented to address the lack of scalability faced by mathematical models.

In [64], authors present the migration framework that follows the three-layered architecture - Base layer, Application layer and Instance layer to relocate containers or VMs across MEC. Aiming to enhance the performance by placing the service near to the user, the paper considers the stateful migration of applications and induces to minimize the overall migration time and service downtime. The main steps of the procedure are the following: First, the primary components (i.e., guest OS, kernel, etc.) included in the base layer (excluding the application) is transmitted on each MEC in order to avoid the transmission of the base layer for each migration request. Second, the idle application and its data that are both included in the application layer, are passed on when migration is triggered while keeping the service running. Finally, the running states included in the instance layer is transmitted after suspending the service. Therefore, only the transmission time related to the transmission of the instance layer is considered as service downtime. Cloudify [97], an open-source multi-cloud and edge orchestration platform, supports the pod migration without interrupting the containerized service from one node

to another within the Kubernetes cluster.

KubeVirt [98] is a service-oriented architecture that relies on Kubernetes and supports additional functionality. It allows live migration of VM instances (acted as a pod) from one host to another host. Therefore, it could be profitable to relocate the containerized applications (running inside the VM) from one one node to another within a cluster.

A release of KubeVirt is available on GitHub [99]. Another prototype release [100, 101] includes the additional commands <kubectl migrate> and <kubectl checkpoint> with the help of modified kubelet and customized container/cri. In this way, running pods can be checkpointed and migrated within a single or multi-clusters. Despite the fact that the work is considered to be a rough prototype, this contribution is appreciable as it enables the pod migration feature of stateful containers in Kubernetes.

3.4 Comparison based on model and algorithms for migration strategies

The difference between the framework considered in this section and other frameworks relative to virtual network embedding and reconfiguration as well as circuit repacking has been noticed in the Introduction; we review in this section the existing literature on microservice migration, which is instrumental to solve the problem of placing VNFs or CNFs. We further summarise in Table 3.5 the main characteristics of research works. In particular, Table 3.5 compares the various approaches in terms of their ability of handling dynamicity, network distribution, NF chaining, as well as accounting of user location, resource and latency constraints, and application type (stateless or stateful). The management of “stateless" or “stateful" applications has a significant impact on the way migration is carried out. With stateless application, the stateless container is typically migrated (i.e. re-allocated and restarted from scratch) without conserving the application state. On the other hand, stateful container migration involves transferring the application state. In particular, the migration of an inactive application (cold migration) is straightforward as it involves shutting down the running container before initiating the migration, which eliminates the need to handle the memory state. Alternatively, migrating a container from one node to another while it is running (live migration) and without service interruption, necessitates maintaining state consistency during the whole migration.

3.4.1 Optimization based approaches

The authors of [33] propose an Mixed Integer Linear Programming (MILP) model for VNF migration problem to reduce the SFC delays while considering resource constraints (such as CPU and memory), network delay,

Table 3.5: Characteristics of various migration strategies

Ref.	Dyn.	Dist.	Chaining	User loc.	Resource utilization	Latency	App. type
[33]		✓	✓		✓	✓	Stateless
[9]	✓	✓	✓	✓	✓	✓	Stateless
[92]		✓		✓	✓		Stateful
[61]	✓	✓	✓		✓		Stateless
[121]	✓				✓		Stateless
[80]	✓				✓		Stateful
[20]		✓	✓				Stateful
[66]	✓	✓		✓	✓	✓	Stateful
[87]	✓	✓	✓		✓	✓	Stateless
[40]		✓			✓		Stateful
[19]	✓	✓	✓	✓	✓	✓	Stateful
[59]		✓		✓		✓	Stateful
[63]	✓	✓		✓		✓	Stateful
[25]		✓	✓	✓		✓	Stateless
[84]	✓	✓	✓		✓		Statelful

affinity & anti-affinity factors and migration delay (time required to discover service and to propose new placement). They use greedy algorithm to place VNF and analyse the impact of VNF migration or re-instantiation using their proposed model.

In [9], an edge based migration strategy is proposed for containerized applications. An ILP model aims at minimizing the service downtime and latency occurring while performing the migration across edge nodes. Further, authors implement a heuristic approach to overcome the limitation of mathematical models such as lack of scalability and time consumption, and compare it with greedy approaches.

In [92], a multiple dimensional Markov Decision Process (MDP) migration strategy is used for fog networks and is further solved by using two combined algorithms, namely Deep Q-learning and Deep Neural Networks (DNNs). The considered states of the system are delay, power consumption and migration cost. The action contains the selection policy based on greedy methods that choose the containers to migrate. In particular, containers hosted at under-utilized node, are migrated to other nodes to minimize the power consumption. Whereas, at the over-utilized node, the containers involving the least migration costs are migrated. The allocation policy selects the target node for each migrated container. The empirical evaluation shows that the proposed solution performs better comparing to existing baseline strategies.

The approach in [25] formulates two different optimization problems. The first one aims at mitigating the QoE

degradation during user handover. The second one is intended to control the cost of service replicas. To solve the migration problem the authors exploit the replication mechanism while respecting the creation of replicas for each user.

3.4.2 Algorithmic approaches

In [61], a re-assignment strategy is introduced. Containers belonging to the same type of services are required to be placed closer to each other. The placement of new containers is intended to reduce the load and communication cost and is performed by a customized version of Worst Fit Decreasing (WFD) algorithm. Then, the re-assignment of initially placed containers is performed by the Sweep&Search algorithm to minimize the total cost. An online container based placement strategy for managing the inter-container traffic is presented in [121]. An offline ILP model is formulated to fetch the traffic flow along with quadratic constraints. The online scheme follows the primal-dual method and proposes the placement at the arrival of each new container request.

The scheduling mechanism proposed in [80] migrates only long-lived containers as they occupy the resources for a long time. First, long-lived containers are arranged with respect to the CPU resources they consume. Then, the containers on highly occupied hosts are swapped with those hosted by less occupied hosts. The proposed algorithm is based on a random-first-fit algorithm which is continuously executed until the load is uniform.

In [20], the proposed migration algorithm handles the migration of shared VNFs that are deployed on a multi-domain federated network. The proposed algorithm coordinates with each domain orchestrator and migrates the shared and chained VNFs using the information provided by each orchestrator in case of failure.

3.4.3 Migration in the context of MEC

The authors of [74, 96] reviewed various container-based placement and migration strategies. They investigated a set of previously proposed frameworks and algorithms used to build the scheduling models for edge computing, notably MEC.

The dynamic container migration strategy introduced in [66] for MEC focuses on minimizing the workload and migration time, and handles the user-mobility using a heuristic method. The proposed method first shortlists the containers at source nodes based on their total latency. Then, for each container selected for migration, the node that is (i) geographically closer to the end-user and (ii) less utilized is selected as a destination node.

Likewise, the MEC-enabled approach in [87] aims at providing flexible placement and migration of VNFs. The orchestrator dynamically manages the resources on the fly in order to handle the requirements of an application

across a heterogeneous network that spans the core and edge networks.

In [40], a Kubernetes-based container migration approach is presented to migrate stateful services over the fog network and minimize the downtime. For stateful services, transmission of disk states is very time consuming. Hence, the authors put forth on managing the transmission process of container layers from source to destination node.

The work in [19] addresses the migration problem for distributed data centers to manage latency-sensitive applications. Taking into account various parameters (e.g. resource allocation and load) that are related to container usage, they propose three algorithms. First, containers characterised by a high total latency are short listed. Further, suitable neighbours of each container are selected based on utilization level of the region and its location with respect to the user. At the final stage, containers are placed near to their neighbors, which is the final destination that has been chosen based on the number of migrations from source to destination location taking the inter-edge bandwidth and memory load into account. The authors of [59] implemented a testbed based on Kubernetes that redeploys pods near end-user. The design rationale is to relocate services while performing adaptive handoff in MEC architecture. Another container-oriented approach [63] considers the layer structure of the Docker storage system to reduce the migration time. The top storage layers may be modified anytime while underneath layers remain unchanged. Thus, underneath layers may be transmitted in advance before commencing the migration process ; latter the top layers are transmited. End-user's location is also taken into account so as to place the service near the user.

In [84], live migration mechanism for mobile networks overcomes the limitations of current CRIU tool (Checkpoint/Restore In Userspace)[1], which is frequently used for process checkpointing during cold and live linux container migration. Experimental plateform includes the SCTP protocol to ensure message delivery between MME and CU in LTE networks and a tool to manage the GTP (GPRS Tunnelling Protocol) device-specific information. A detailed evaluation of the migration of core network functions such as Home Subscriber Server (HSS), Mobility Management Entity (MME), and Serving and Packet Gateway (SPGW) for VM-enabled and container-enabled live migration is also provided.

The above mentioned research works solve the migration problem by taking into account various critical factors such latency or computing resources and merely focus on network edge with MEC and the migration of user application. In this work, we rather consider network operator applications (mainly CNFs) which can spread over the complete network infrastructure. Contrary to previous works, we pay attention to the communication

[1] https://www.criu.org/

between the chained NFs/microservices and the end-user while optimizing the end-to-end latency. We tackle the placement and migration problem in such a way that chained microservices of the same service co-join on an optimal target while satisfying the resource load, latency and end-user location.

3.5 Conclusion

Majority of companies and open-source communities are adapting the cloud-native approaches as of their performance efficiency which further lays together on technologies - container, orchestration and microservices that are capable of providing the highly scalable, light-weighted, portable and flexible solutions. Through this work, we aimed to evaluate study analysis on the techniques focused on container migration starting from centralized followed to geo-distributed infrastructure.

The proposed taxonomy states the importance of re-allocating the containerized services in larger-cloud data centers in case of more resource requirements or placing them on latency-efficient fog/edge data centers in the event of latency-critical highly communicable application.

Also, depending on the service dynamics, some nodes/clouds provided a heterogeneous capacity may get overloaded. In order to balance the load between nodes and data centers, it is necessary to support migration. Which also requires ensuring resiliency in case of failure in the system and re-deploy the components in an efficient way to avoid any communicational delay. Containerization is capable of dealing with these tasks effectively in comparison to VM-based hardware-aware virtualization techniques due to its light-weight nature. Therefore, the development of a real-time migration model considering the telco infrastructure as a whole induces some challenges to address concerning the application downtime and migration time.

To meet this requirement, we initialized with the proposition of a novel microservice placement strategy considering the internal service composition, notably the communication between the microservices. This contribution is presented in the following chapter where the formulated optimization problem has been solved using a proposed heuristic algorithm with an objective to minimize the end-to-end latency.

Chapter 4

Latency and network aware placement for cloud-native 5G/6G services

Content

In this chapter, we present the first contribution of this PhD thesis. Which is organized as follows: Based on a realistic model of a ISP infrastructure (§ 4.2), we formalize the problem of allocating microservices (§ 4.3) and we introduce an approximate problem-solving solution (§4.4). We then evaluate the algorithms (§ 4.5) and conclude with a summary of our contributions 4.6.

4.1 Introduction

The advent of NFV [31] and the recent adoption of cloud-native principles have deeply modified network operations by dissociating the hosting hardware from network functions and by decomposing network functions into a large number of smaller and ready-to-run microservices. Services are then implemented as bundles of microservices interconnected to each other and distributed on container-based infrastructures (e.g., Kubernetes). In parallel, the ever growing performance requirements in terms of latency and throughput of new services (involved e.g. in virtual reality) pave the way towards architectures, where the network edge plays a primary role. As a matter of fact, latency sensitive network functions need to be deployed as close as possible to end users to meet real time requirements. Closeness of microservices is however not always feasible due to capacity constraints of the hosting data centers, especially when considering the relative resource scarcity of those data centers located at network edge. It is therefore necessary to design an efficient placement strategy of microservices while considering both resources availability and service demand.

The placement of VNFs has been widely studied in the literature. Most research works focused on load balancing, examine the resource (CPU, RAM, disk) needs of network functions with respect to the resource availability in data centers [23, 93]. Only a few works on microservice placement address latency requirements by considering either the processing delay or the link capacity. To address the deployment of network functions across distributed data-centers, microservice placement should be reexamined from different viewpoints, notably by considering the microservice inter-dependency and the amount of traffic among microservices, which increase the service latency due to the delay associated with messages transiting through the transport network connecting data-centers.

In order to minimize the latency of new 5G/6G services, we propose two approaches that consider the communication affinity, i.e., the number of internal communication messages within a distributed service. The first one minimizes the delay associated with messages transiting between data centers. This first approach is network aware as it explicitly depends on latency along transmission links. The second approach is network agnostic and introduces a metric that accounts for (i) communication affinity among microservices and (ii) the location of the user. This metric relying on a weighting strategy, privileges a placement near to the user and the collocation of microservices that involve heavy message exchanges.

To the best of our knowledge, the present work is one of first works considering these aspects. We formalise the problem of placing microservices via Integer Linear Programming (ILP), and we propose an approximate problem-solving solution combining two heuristics (a greedy and a genetic algorithm), which considerably

reduces run time of the placement algorithm. In particular, the fast greedy algorithm provides an initial allocation of services, which is subsequently improved by an advanced genetic algorithm; this latter one rearranges through mutation and crossover the microservices to reduce the global latency, while preserving the placement of microservices achieving a high latency gain.

4.2 Model description

The model under consideration (Fig. 5.1) is made of two key elements: the Substrate Network (6.2.1) and the Services (4.2.2). The substrate network reasonably represents the network of an European Operator [89], which is represented as a tree composed of a central cloud datacenter, fog nodes located at regional Points of Presence and edge nodes near to users.

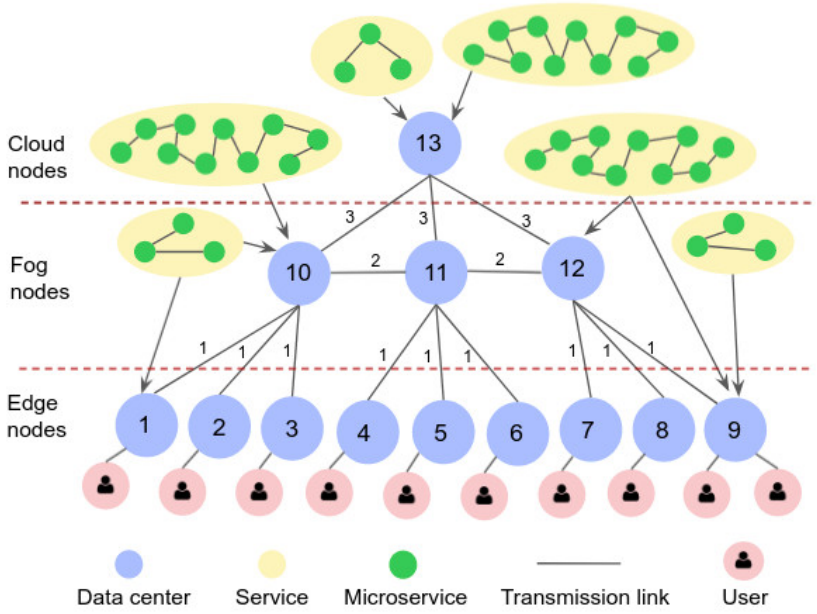


Figure 4.1: Mapping a set of services onto a substrate network

4.2.1 Substrate network

The network of data centers is represented by means of a graph $G = (V, E, W)$, where the set V of vertices is composed of data centers, the set of edges E corresponds to the bidirectional links interconnecting the data centers, and the set of weights W reflects the links characteristics. While an edge weight may represent diverse factors (e.g. bandwidth, the length, the transmission delay along the link), we herein focus on the latency in the execution of a service and hence we assume that the weight $w(e(v_i, v_j))$ equals to the transmission delay

$\delta(v_i, v_j)$ between the two data centers v_i and v_j.

A vertex v offers the IT resources to execute the hosted services and is characterized by a capacity $C(v)$ that may reflect various types of resources (e.g., CPU, RAM, disk). As the most limited resource is either CPU or RAM, the allocation will be based on a single resource type, even if the algorithms presented in § 4.3 could handle various types of resources.

4.2.2 Services

The substrate network hosts services that are decomposed into microservices, each of them having specific requirements in terms of IT resources. A service Σ_j corresponds to a graph composed of K_j vertices, denoted $\sigma_1^j, \ldots, \sigma_{K_j}^j$, representing the microservices, and edges representing the communication between microservices. A microservice σ_i requires an amount of IT resources equal to $c(\sigma_i)$ and may send messages to another microservice σ_j. The number of messages exchanged between the two microservices, on both way, is denoted by $\nu(\sigma_i, \sigma_j)$, with $\nu(\sigma_i, \sigma_j) = \nu(\sigma_j, \sigma_i)$.

Each service is associated with a user lying at the edge of the operator network (see the leaves in Fig. 5.1) and exchanging traffic with the microservices. It is important to consider the latency associated with the user traffic as the access network usually contributes a significant amount to the global latency budget. For that purpose, we introduce a dummy service σ_0 located, where the user stands and which has no resource requirements, i.e., $c(\sigma_0^j) = 0$. The service Σ_j is thus composed of microservices $\sigma_0^j, \ldots, \sigma_{K_j}^j$.

4.3 Problem formulation

We consider a substrate network G composed of N nodes (data centers), in which each node v_n (with $n = 1, \ldots, N$) has a capacity $C(v_n)$. The network G accommodates J services and each service Σ_j is decomposed into $K_j + 1$ microservices (including the end user). The J services can be allocated to the substrate network if and only if the total resource demand is not greater than total resource capacity of the substrate network, that is,

$$\sum_{j=1}^{J} \sum_{i=1}^{K_j} c(\sigma_i^j) \leq \sum_{n=1}^{N} C(v_n).$$

If this condition is violated, then some services are rejected. In the present contibution, we do not address this issue and focus on how perform placement so as to globally reduce latency. We precisely consider a static configuration to better understand issues related to latency when placing microservices. In practice, we should consider the case when services randomly arrive and leave the system. We would then obtain a stochastic

knapsack, which will be addressed in a further study. In order to optimize the microservices placement, we propose to either:

- minimize the overall latency accounting for the transmission delays, thereby considering the structure-of – and delay induced by - the network substrate (§ 6.2.1),
- maximize the amount of microservices that are collocated and placed near to the user, ignoring the substrate network (network-agnostic approach § 4.3.2)

4.3.1 Network-aware approach

The latency $\mathcal{L}_j$ of a service Σ_j is

$$\mathcal{L}_j = \sum_{i=0}^{K_j} \sum_{i'=i}^{K_j} \nu(\sigma_i^j, \sigma_{i'}^j) L(\sigma_i^j, \sigma_{i'}^j), \tag{4.1}$$

where the latency between microservices σ_i^j and $\sigma_{i'}^j$ deployed across distinct data centers verifies:

$$L(\sigma_i^j, \sigma_{i'}^j) = \sum_{n=1}^{N} \sum_{m=n}^{N} \delta(v_n, v_m) \mathbb{1}(v_n, \sigma_i^j) \mathbb{1}(v_m, \sigma_{i'}^j). \tag{4.2}$$

The objective of placement is then to minimize the global latency, which leads to the following optimization problem:

$$\min_{\substack{\mathbb{1}(v_n, \sigma_i^j) \\ n \in [\![1, N]\!],\ j \in [\![1, J]\!],\ i \in [\![1, K_j]\!]}} \mathcal{L} \tag{4.3}$$

$$i \in [\![1, K_j]\!],\ j \in [\![1, J]\!],\ \sum_{n=1}^{N} \mathbb{1}(v_n, \sigma_i^j) \leq 1, \tag{4.4}$$

$$n \in [\![1, N]\!],\ \sum_{j=1}^{J} \sum_{i=0}^{K_j} c(\sigma_i^j) \mathbb{1}(v_n, \sigma_i^j) \leq C(v_n), \tag{4.5}$$

where

$$\mathcal{L} = \sum_{j=1}^{J} \mathcal{L}_j \tag{4.6}$$

As defined in Equation (4.4), a microservice should be allocated at most once. If a microservice σ_i^j cannot be placed (the sum in equation (4.4) equals to 0), then the entire service σ_j is removed.

4.3.2 Network agnostic approach

If several microservices that exchange some messages are collocated (resp. hosted on distant data centers), then the latency of the service tends to decrease (resp. increase). We intuitively introduce a metric quantifying the amount of messages exchanged by the microservices collocated on the same data center since those ones do not increase the global latency of the service. For service Σ_j, the metric is defined by

$$\tilde{L}_j = \sum_{n=1}^{N} \sum_{i=0}^{K_j} \sum_{i'=i}^{K_j} \tilde{\nu}(\sigma_i^j, \sigma_{i'}^j) \mathbb{1}(v_n, \sigma_i^j) \mathbb{1}(v_n, \sigma_{i'}^j).$$

where $\tilde{\nu}(\sigma_i^j, \sigma_{i'}^j) = \nu(\sigma_i^j, \sigma_{i'}^j)$ for $1 \leq i \leq i'$ (i.e., the actual number of messages exchanged between microservices). The product $\mathbb{1}(v_n, \sigma_i^j)\mathbb{1}(v_n, \sigma_{i'}^j)$ is equal to 1 only if microservices σ_i^j and $\sigma_{i'}^j$ are on the same data center. Moreover, to force the placement of microservices close to users, we give more weight to the communications between users and microservices. Thus, $\tilde{\nu}(\sigma_0^j, \sigma_i^j) = \kappa_i(\sigma_0^j, \sigma_i^j)$, where κ_i is a parameter giving more weight to the messages exchanged between the user and any microservice i (with $i > 0$). Overall, the metric $\tilde{L}$ quantifying the "network agnostic latency" gives more weight to collocated microservices:

$$\tilde{L} = \sum_{j=1}^{J} \alpha^{\tilde{L}_j} \tag{4.7}$$

for some $\alpha > 1$. There is thus an exponential discrimination between services with many collocated microservices and those with more distributed microservices. With this latency metric, the optimization problem then reads:

$$\max_{\mathbb{1}(v_n, \sigma_i^j)} \quad \tilde{L} \qquad n \in [\![1, N]\!],\ j \in [\![1, J]\!],\ i \in [\![1, K_j]\!] \tag{4.8}$$

subject to the constraints (4.4), (4.5).

4.3.3 Scalability of the Latency-aware placement

The optimization problem previously introduced could be solved by using a classical ILP solver (e.g. CPLEX[1]) or by performing an exhausting search (i.e., enumerating all the solutions and selecting the optimal one). Nonetheless, these options are not practically viable for large problem instances. Computing the optimal placement of microservices that minimizes latency, is actually a variation of the bin-packing problem, which is known as to be combinatorial NP-Hard problem in which items (i.e., microservices) of varying sizes (e.g., resource capacity) should be packed into a finite number of bins (i.e., computing nodes) with finite capacities so that the number of bins is minimized.

[1] https://www.ibm.com/analytics/cplex-optimizer

Our placement of microservices across several data centers is actually a multi-dimensional bin-packing problem in which (i) the resource usage and microservices latency are considered and (ii) evaluating the latency constraints adds to the complexity of the solution. Overall, the resolution of our NP-hard problem with a solver requires a prohibitive processing before reaching the optimal solution. To remedy this problem, we introduce a family of heuristic approaches.

4.4 Placement algorithms

In order to address the scalability issue of the latency-aware placement problem previously formulated, we introduce an heuristic approach which combines a greedy algorithm and a genetic algorithm. The greedy algorithm (see § 4.4.1) is used to generate an initial (and possibly not optimal) distribution of microservices on the substrate network. Then, the genetic algorithm (see § 4.4.2) further refines the initial placement and attempts to find the best solution [7] by iteratively improving the quality of the result and helping the search process to escape from local optima. We could have opted for either approach, but by combining these two algorithms we get a faster solution without sacrificing the quality.

4.4.1 Finding an initial placement using a greedy heuristic

The design rational of our greedy algorithm is to consider the substrate network characteristics (occupancy and underlying structure) to smartly place the microservices in a fair manner, i.e., without discriminating some services over others based on their internal characteristics (e.g., number of microservices or their demand in terms of resources). The algorithm places services without discrimination, considering the substrate network characteristics (occupancy and underlying structure). In particular, the greedy algorithm (see Algorithm 1) favours the placement of the chain of microservices composing one service within the same data center, ideally at the edge. This placement permits to minimize the service completion time and the latency perceived by the end user while favouring short-distance communications between microservices and hence avoiding as much as possible long-distance communications between data centers.

As pointed earlier, the problem-solving heuristic makes the locally optimal choice (which is to place microservices as much as possible in the same data center, near the end user) at each step of the algorithm. Unfortunately, this selection may not produce the optimal solution (in term of latency). In order to improve the quality of the solution, we rely on a genetic algorithm that improves the initial and fairly good assignment provided by the greedy algorithm.

Algorithm 1: Placement of microservices using greedy approach

Input : Set of J services to place: $\Sigma = \{\Sigma_1, \cdots, \Sigma_j, \cdots, \Sigma_J\}$
Each service Σ_j is constituted of a set of μservices: $\sigma^j = \{\sigma^j_{k_1} \cdots \sigma^j_{k_i} \cdots \sigma^j_{k_I}\}$
Related set of capacity requirements: $c(\sigma^j_{k_1}) \cdots c(\sigma^j_{k_i}) \cdots c(\sigma^j_{k_I})$
N data centers: $v_1, \cdots, v_n, \cdots, v_N$
Related set of capacities: $C(v_1), \cdots, C(v_1), \cdots C(v_N)$
Output: Set of placed μservices

```
while Σ ≠ ∅ do
    while σ^j ≠ ∅ do
        v_n = getDatacenter(σ^j_{k_i})
        if c(σ^j_{k_1}) <= C(v_n) then
            placedμservices.add (σ^j_{k_i}, v_n)
            C(v_n) = C(v_n) − c(σ^j_{k_i})
            σ^j ← σ^j − σ^j_{k_i}
        end
    end
    Σ ← Σ − Σ_j
    return placedμservices
end
```

4.4.2 Enhancing the initial placement using genetic algorithm

The Genetic Algorithm (GA) mimics evolutionary processes, i.e., at each iteration of the GA, the population of individuals evolves using three genetic operators: selection, crossover, and mutation.

4.4.2.1 Population Encoding

The population (Figure 4.2) consists of a set of data centers $C_1 \cdots C_N$. Within the population, each individual (i.e. data center) is represented by a *chromosome* that consists of series of genes. GAs rely on the specification of genes and chromosomes and the steps of GA include population generation along with crossover and mutation which affect genes. A *gene* is a binary variable representing the presence or absence of a microservice on a data center. We thus define the nth chromosome C_n (with $n = 1, \ldots, N$) as a binary series $\mathbb{1}(v_n, \sigma^j_i)$ for $i = 1, \ldots, K_j$. Each chromosome (data center) is characterised by its own resource capacity – in this model we only consider CPU capacity – and each gene by its required resource capacity. A microservice is allocated on a data center if there remains enough resource in the data center to run the microservice.

Once the initial population is generated using the greedy approach (4.4.1), the population evolves iteratively: at each iteration of the genetic algorithm, the population evolves by applying the selection, crossover, and mutation operators.

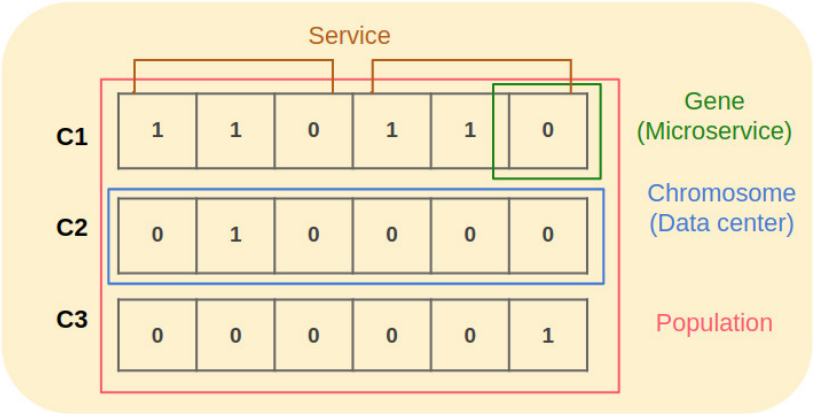


Figure 4.2: Population encoding - Each chromosome consists of 6 genes encoding the presence/absence of a microservice in a data center.

4.4.2.2 Selection process

Among the population, the selection operator selects the best individuals to let them reproduce and have an *offspring* (through crossover and mutation as detailed below), i.e., a set of new individuals, which composes the subsequent generation. The chromosomes with the best fitness value envisaged through two ways of fitness value calculation defined in equations (4.7) and (4.6) - network agnostic and network-aware metrics, respectively. The total fitness of the population is expected to rise (for network agnostic metric) or to decrease (for the network aware metric) with the algorithm. Fitness function plays a pivotal role in order to evaluate the efficiency of obtained solutions.

4.4.2.3 Crossover

After selecting certain individuals for reproduction, the crossover operator swaps genes (bit strings) between two selected parent chromosomes ($P1$, $P2$) to create two new off springs (Figure 4.3). As depicted in Figure 4.3, once the crossover random point has been chosen, then the first half of parent ($P1$) is combined with the second half part of $P2$. Likewise, the first part of $P2$ is combined with the second part of $P2$ to produce two off-springs. Based on our analysis and literature study, the crossover probability value ranges between 0.1 and 0.8 resulted in a better solution. Evidently, this value is subject to particular problems. Each problem statement could have their own optimal range of crossover probability.

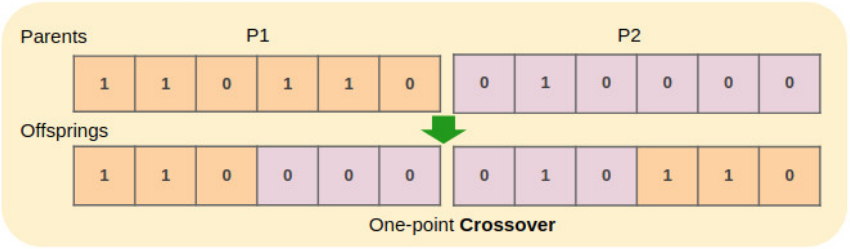


Figure 4.3: One-point Crossover - first half of parent ($P1$) is combined with the second half part of $P2$. Likewise, the first part of $P2$ is combined with the second part of $P2$

4.4.2.4 Mutation

The mutation operator performs a mutation of certain individuals from the new offspring to diversify the generations. The mutation consists of changing a random number of genes in the chromosome of an individual (see Figure 4.4). Consequently, an old generation is evolving into a new generation with a population filled by both the unaltered elite and offspring. The value of mutation probability is suggested to be lesser than crossover probability, i.e., within a range of 0.01 to 0.2 (as per our sensitivity analysis carried out by multiple runs of algorithms with different probability).

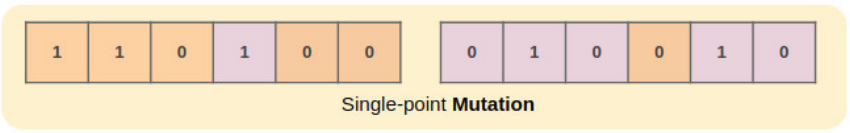


Figure 4.4: Single-point Mutation

4.4.3 Stop condition

Now, the fitness of the new generation has been evaluated and passed through the termination condition of maximum number of iterations and at each new total fitness of generation verifying if the generated total fitness value is improved or not. The resulting generation with highest or lowest total fitness value (depending on the criterion under consideration) is an optimal placement of μservices. The Figure 4.5 depicts the flow of the proposed Genetic algorithm.

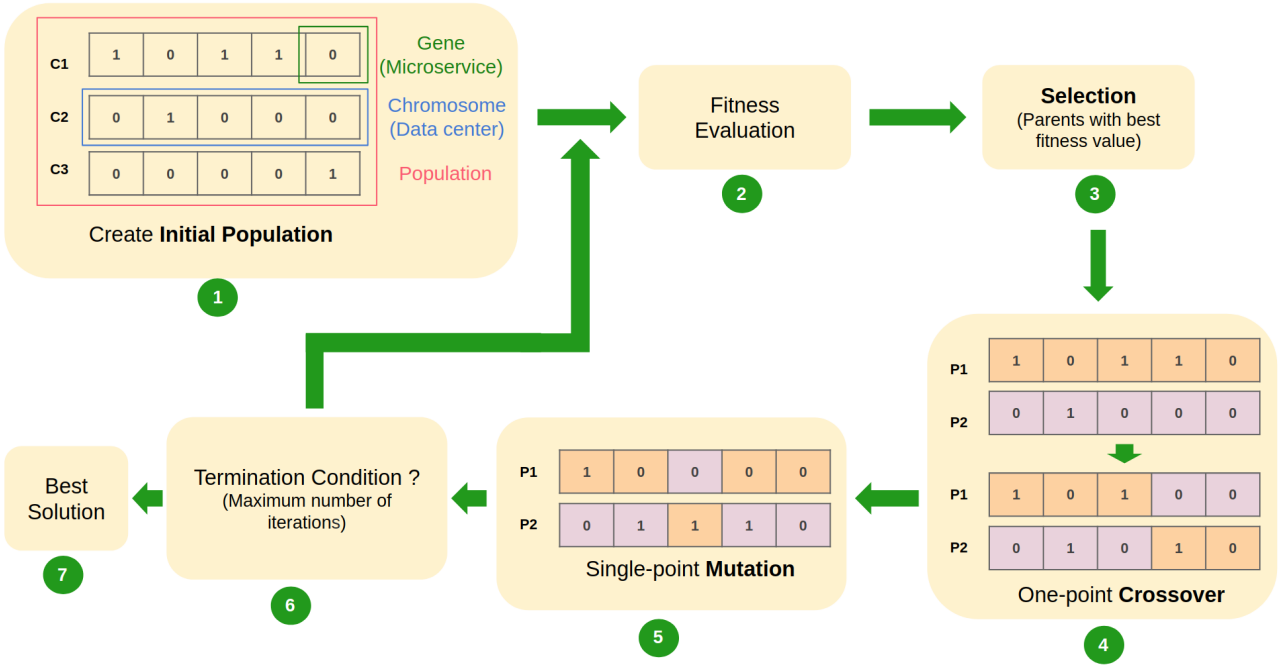


Figure 4.5: Flow chart of Genetic Algorithm

4.5 Experimental results

4.5.1 Experimental setting

In order to evaluate our approach to the placement of microservices, we consider the substrate network displayed in Figure 5.1, which reasonably represents a telco infrastructure including, cloud, fog and edge nodes. Considering the closeness of edge and fog nodes, we assume a smaller latency (one unit) between the edge and fog nodes than that between the fog nodes and the centralized cloud, which is equal to 3 units. In general, the central cloud is the largest and consists of nodes with a large capacity (1000 units) comparing to the fog nodes (capacity of 100 units) and edge nodes (capacity of 20 units). Following, we consider two types of services that differ in the number of microservices:

- The first service type (say, a lightweight application running at the edge such a firewall) involves a relatively small number of microservices: 3 microservices exchange messages with $\nu(\sigma_1^1, \sigma_2^1) = 2$ and $\nu(\sigma_2^1, \sigma_3^1) = 4$.

- The second service type (a heavyweight application) is composed of 10 microservices exchanging messages with $\nu(\sigma_1^2, \sigma_2^2) = \nu(\sigma_4^2, \sigma_5^2) = \nu(\sigma_6^2, \sigma_7^2) = \nu(\sigma_7^2, \sigma_8^2) = 3$, $\nu(\sigma_2^2, \sigma_3^2) = \nu(\sigma_3^2, \sigma_4^2) =$

$\nu(\sigma_5^2, \sigma_6^2) = 2$, $\nu(\sigma_8^2, \sigma_9^2) = 4$ and $\nu(\sigma_9^2, \sigma_{10}^2) = 1$.

We assume that for both service types, user exchanges only two messages with the first microservice and none with others ($\nu(\sigma_0^j, \sigma_1^j) = 2$ and $\nu(\sigma_0^j, \sigma_i^j) = 0$ for all j and $i > 1$); this reflects a single input and output message between the user and the service. In addition, in the computation of the quantities $\tilde{L}_j$, we have taken $\tilde{\nu}(\sigma_0^j, \sigma_1^j) = 100$ and $\tilde{\nu}(\sigma_0^j, \sigma_i^j) = 0$ for all j and $i > 1$. This is to force the algorithm to collocate the user and the first microservice.

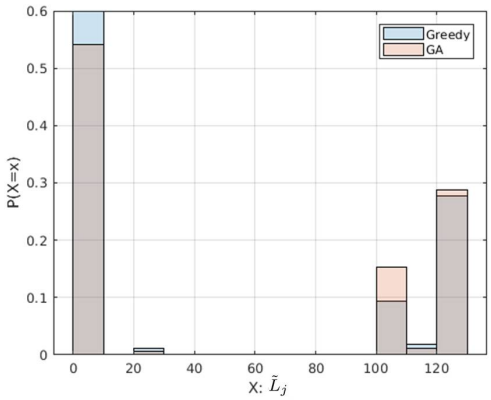
The capacity requirement for each microservice is set equal to 1. During our experiments, services of type 1 constitute $2/3$ of the services. Thus, we have a maximal population of $J_0 \approx 278$ services with $J_1 = 185$ services of type 1 and $J_2 = 93$ of type 2, which can be hosted by the system. The maximum order of magnitude of chromosome length is then $3J_1 + 10J_2 \approx 1480$ genes. The load ρ is approximately equal to $(2/3 \times 3 + 1/3 \times 10)J/C = J/J_0$. In the following, we consider two representative loads: $\rho = .7$ (light load) and $\rho = .9$ (heavy load).

4.5.2 Numerical results

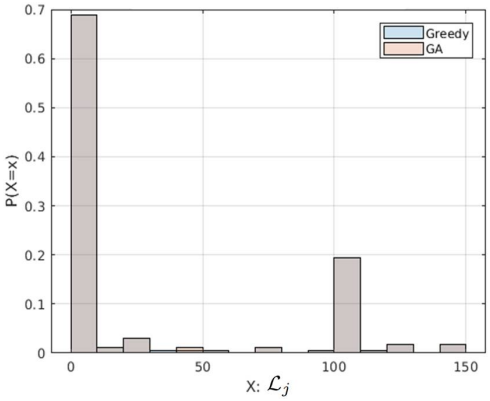
Our evaluation investigates to which extent maximizing the metric $\tilde{L}$ (defined in a network agnostic way) is relevant for controlling the global latency $\mathcal{L}$. For this purpose, we have first evaluated the network agnostic method (see Fig. 4.6) using the probability density function (pdf) of $(\tilde{L}_j)$ and that of $(\mathcal{L}_j)$ for load ρ equal to 0.7 and 0.9. We observe that the GA algorithm slightly improves $\tilde{L}$. As observed in Table 4.1, the mean value $\mathbb{E}(\tilde{L})$ of the series $(\tilde{L}_j)$ is better with the GA algorithm. However, the improvement is very marginal when looking at global latency, given by the series $(\mathcal{L}_j)$ with mean value $\mathbb{E}(\mathcal{L})$. This indicates that the network agnostic optimization is not sufficient to significantly improve the latency. Then, we focus on the network

Table 4.1: End to end latency of services.

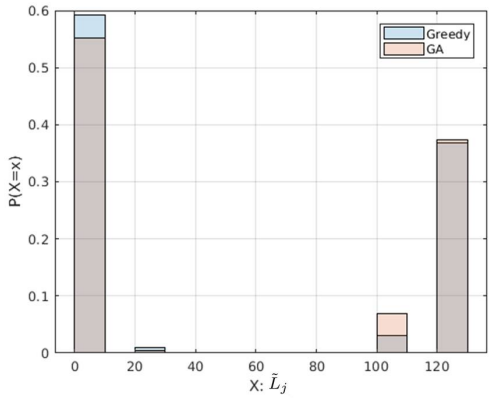
Placement	Load	J_1	J_2	$\mathbb{E}(\tilde{L})$	$\mathbb{E}(\mathcal{L})$
Greedy	0.71	118	52	49.78	31.39
GA network agnostic	0.71	118	52	56.20	31.28
Greedy	0.70	109	55	54.76	33.04
GA network aware	0.70	109	55	38.95	22.01
Greedy	0.90	125	76	52.06	40.21
GA network agnostic	0.90	125	76	56.51	40.06
Greedy	0.91	148	68	46.36	33.55
GA network aware	0.9	148	68	32.87	24.79



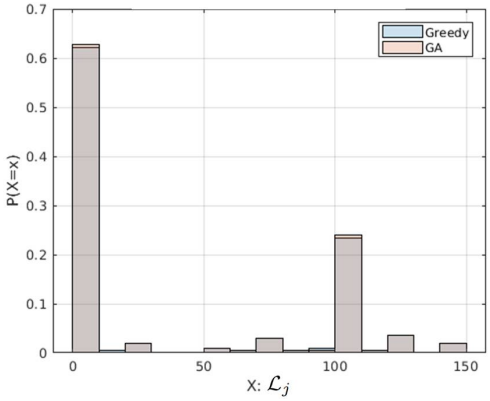
(a) Pdf of $(\tilde{L}_j)$ for $\rho = 0.7$



(b) Pdf of $(\mathcal{L}_j)$ for $\rho = 0.7$

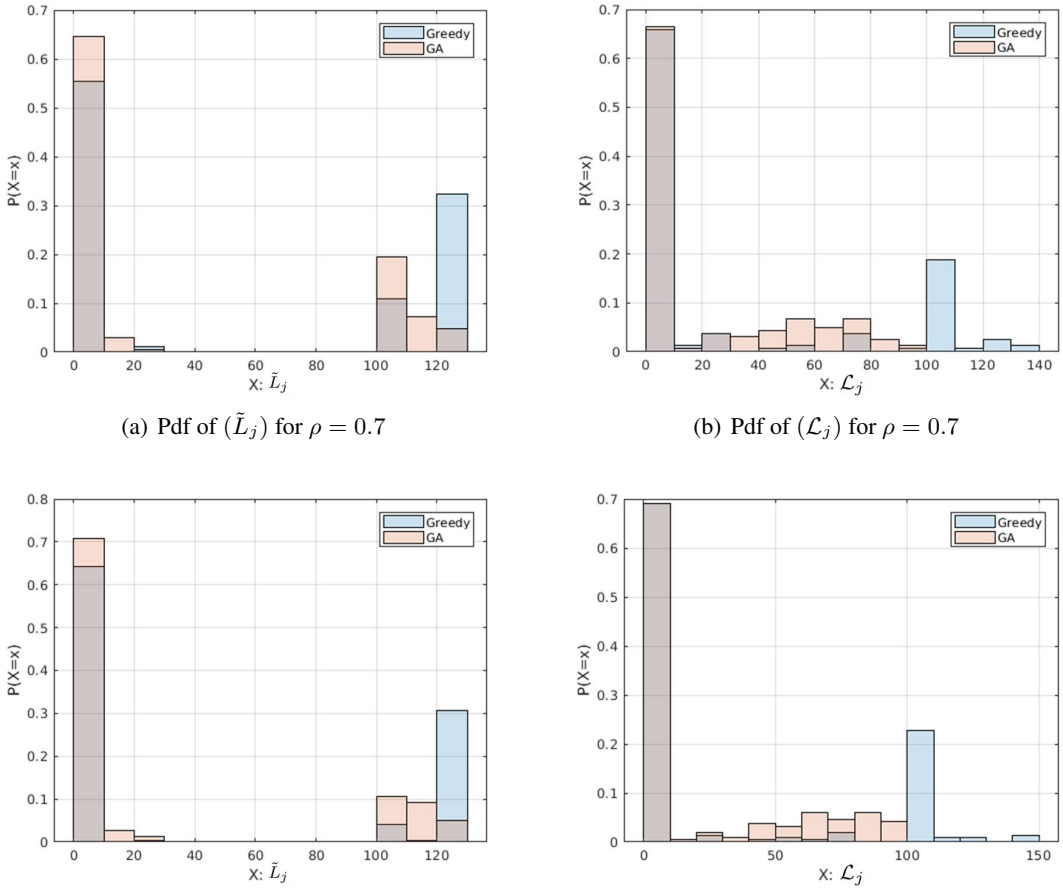


(c) Pdf of $(\tilde{L}_j)$ for $\rho = 0.9$



(d) Pdf of $(\mathcal{L}_j)$ for $\rho = 0.9$

Figure 4.6: Network agnostic optimization.



(a) Pdf of $(\tilde{L}_j)$ for $\rho = 0.7$

(b) Pdf of $(\mathcal{L}_j)$ for $\rho = 0.7$

(c) Pdf of $(\tilde{L}_j)$ for $\rho = 0.9$

(d) Pdf of $(\mathcal{L}_j)$ for $\rho = 0.9$

Figure 4.7: Network aware optimization.

aware metric in Figure 4.7 that provides the pdfs of the series $(\tilde{L}_j)$ and $(\mathcal{L}_j)$, for a load of $\rho = 0.7$ and $\rho = 0.9$. We see that the GA algorithm significantly improves the global latency; this can also be seen for mean values. It is worth noting that improving the latency does not increase the values of $(\tilde{L}_j)$. Thus, maximizing $\tilde{L}$ and minimizing $\mathcal{L}$ could go in opposite directions.

Results show that GA improves the placement performed by Greedy, however the main conclusion of this work is that placement algorithms either in the infrastructure layer (e.g., in Cloud OS environments as Kubernetes and Openstack) or upper in the Orchestration layer (e.g., ONAP) need to be network aware.

4.6 Conclusion

To meet ever more stringent requirements in terms of latency, 5G/6G networks are evolving from centralized to distributed architectures, for which the cloud-native paradigm with services decomposed into microservices is utmost relevant. This in turn raises the issue related to the distribution of network functions.

Also, provisioning a reliable and timely service delivery over a ISP network is a critical issue that we addressed through the introduction of a latency-effective microservice placement strategy that allocates the microservices on computing nodes, spanning from the edge to the cloud. We proposed two mathematical formulations of the notion of latency: the former favours the allocation of microservices in the same compute node and near to end users by promoting the presence of collocated services near to the end user, thereby lowering the cost of communications. The latter minimizes the communication delay between all the data centers and avoids as much as possible long distance communications. We further propose an ILP formulation of the placement problems, which are solved by a hybrid algorithm combining greedy and genetic methods. The key outcome of the work is that a network aware placement strategy is the most effective and is sometimes against a network agnostic optimization.

As part of our proceeding work (described in the coming chapter), we studied to continue this work by adding the dynamicity. Which proceeds to the online placement and migration of the containers or service instances as well as migration of these containerized applications or network functions while keeping the service active and the placement optimal.

Chapter 5

Dynamic migration of microservices for end-to-end latency control in 5G/6G networks

Content

In this chapter, we present the second contribution of this PhD thesis. This chapter is organized as follows: The model considered in this work (in particular the underlying cloud infrastructure) is detailed in Section 5.2. The dynamic system as well as the metrics considered as quality indicators are presented in Section 5.3. The placement and migration algorithms are described in Section 5.4. Simulation results are reported in Section 5.5. Concluding remarks are presented in Section 5.6.

5.1 Introduction

The decomposition of complex services into microservices, which can easily be instantiated, modified and deleted, is instrumental in the design of 5G/6G networks and widely adopted by the telecom industry to implement NFV. NFs decomposed into microservices are hosted in containers, giving rise to CNFs. Thanks to the flexibility of the cloud native approach, plethora of cloud providers, content providers and telecom network operators are today adopting the container-based microservices paradigm.

Numerous research works (refer to, e.g., [43] for a survey) have addressed the problem of placing VNFs represented as SFCs on data centers geographically distant. Several optimization criteria can be envisaged (e.g., load balancing, latency control). A big chunk of the studies however consider static situations, where a set of SFCs has to be placed on a distributed and virtualized architecture composed of data centers interconnected by transmission links. Optimization problems considering cloud and transport resources are then formulated based on objective criteria and solved via heuristics or Machine Learning techniques. Only a few works (see [12]) consider dynamic situations where SFCs join and leave the network. For instance in [12], the problem of placing SFCs (namely, network slices) has been addressed in a dynamic context, wherein SFC requests arrive following a non stationary Poisson process; Deep Reinforcement Learning techniques then prove very effective to cope with this kind of placement problem.

Beyond placement of SFCs, the decomposition of NFs into microservices, which can easily be migrated [43], introduces an additional degree of freedom in the placement of SFCs. While classical placement algorithms place the different components of a SFC on data centers for the whole lifetime of the SFC, migration makes it possible to continually modify the placement of microservices in order to improve some performance criteria or remedy an impairment.

Container migration is mostly addressed in the technical literature within the framework of service migration in connection with MEC, see notably [19]. In that context, the migration of containers is triggered by the move of users and is intended to guarantee some SLA that are expressed in terms of latency, bit rate, etc. Our motivation in the present paper is different as we consider the placement of VNFs (rather than user applications) decomposed into microservices embedded in containers. The corresponding SFC are placed on a hierarchy of clouds (edge, fog, and central clouds) and are then migrated in order to control the latency of all the placed VNFs and not only the latency of the individual service. Contrary to [19], which is relevant to Edge Multi Cloud Orchestrator (EMCO), the framework considered in this article addresses the network orchestration problem, in which the orchestration platform optimizes the placement of SFCs; container migration is an additional feature,

which so far only places SFCs (see for instance Open Network Automation Platform (ONAP)).
Compared to many works on SFC placement [43], we consider in this contribution VNFs, which comprise virtual Radio Access Network (RAN) functions, which shall be placed near a predefined geographical area. SFCs are hence rooted in the sense that microservices are placed near to the virtual end user, which is static, contrary to [19] that allows end user to move. Finally, SFCs are different from virtual network embedding as one objective of the placement and then of the migration is to collocate microservices in order to contain latency. The frameworks of virtual network reconfiguration or even circuit repacking in circuit switched networks present some similarities with the problem addressed in this work. However, the concept of latency, which is central in our analysis, cannot be easily handled as this metric depends on the number of messages exchanged between the microservices.
Through this work, we identify the factors that significantly influence the placement of SFCs composed of chained microservices. First, the proposed strategy is dynamic in nature, which means that after proposing the initial placement, the system continues to improve the placement by migrating or re-allocating the microservices. Second, our solution is user-centric as it aims at reducing the end-to-end latency while considering resource load balancing. Finally, the design rationale supports migration of microservices across geo-distributed cloud nodes by performing both vertical (moving microservices from the bottom edge to the top layer of the cloud and vice versa) and horizontal (moving microservices from one node to another in the same layer) migration.
We specifically answer to the following questions : 1) Which microservice requires to migrate and when? 2) Which factors have to be considered while choosing an optimal data center to place the migrated microservice? 3) In case of no available resources on the selected optimal data center, which microservices can be selected from the list of already placed ones to migrate on another node by avoiding the impact on its current communication delay? While taking in account all the above mentioned design criteria, the contribution of the present work can be summarized as follows:

- First, we formalize the model for the migration of microservices distributed across several data centers, considering a heterogeneous cloud architecture. In particular, the goal is to solve this ever-demanding migration problem by ensuring the lesser number of microservices are moved while keeping the placement optimal.

- Second, we introduce an approximate problem-solving solution with three heuristics that considerably reduce run time of the migration algorithm. The two heuristics emphasize on the placement of newly

arrived services while considering the current system's state and the third heuristic aims at enhancing the placement optimality in terms of end-to-end latency by performing the run-time migration upon service departures.

5.2 Model description

5.2.1 Cloud infrastructure

The model considered for simulation experiments and described in Figure 5.1 reasonably represents a national telecommunications cloud infrastructure with several interconnected data centers organized in a three-layer tree structure.

The lowest layer consists of edge nodes corresponding to the MEC level that have limited resource capacity and are geographically close to end-users, thereby ensuring low communication latency. The next layer is composed to regional nodes having intermediate capabilities in terms of resources. The top layer refers to a centralized cloud that acts as a national cloud with enormous capacity compared to the others, but operates at the expense of high latency.

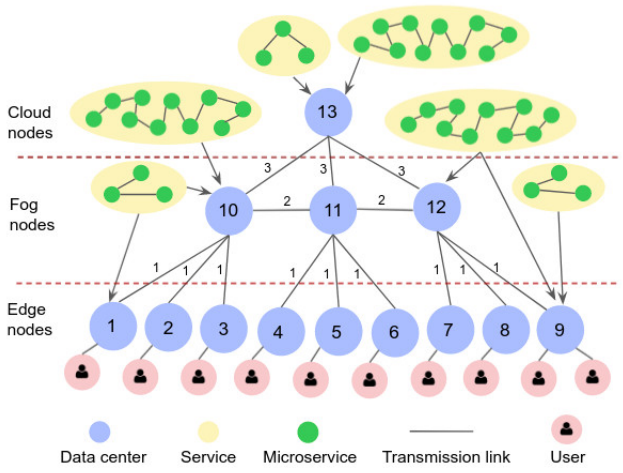


Figure 5.1: Cloud infrastructure of a network

Communications between the user and the microservices of an application (or NF in the context of this work) hosted on different the data centers induce latency. Obviously, the user experiences lowest latency if all the components of an application (e.g., cloud gaming, AR/VR) is hosted in the edge node. In turn, latency increases if the application is hosted in a regional or the centralized cloud. Data centers are geographically distributed. In most European countries, edge clouds are located within a distance of 50 to 100km. The distance between

edge clouds and core clouds is between 100 and 200 km and between 300 and 1000 km between core and centralized clouds. In the following, we shall take as time unit the propagation time between an edge and core clouds distant of 100 km. This time may slightly vary in practice, depending on the number of routers, switches and link capacities between the two clouds. But for thought experiments we assume that the transmission time between an edge and a regional data center is set to 1; transmission delay between fog nodes is equal to 2; the ones between fog nodes and the central cloud equal to 3. Within a node, bandwidth is assumed to be infinite because network operators typically over-dimension their transmission links to avoid bottlenecks.

The set of data centers $\mathcal{D} = \mathcal{D}_e \bigcup \mathcal{D}_f \bigcup \mathcal{D}_c$, where $\mathcal{D}_e$ is the set of data centers at the edge, $\mathcal{D}_f$ is that of fog data centers and $\mathcal{D}_c$ is the cloud data center. Given the hierarchical structure of the network topology considered, we associate with $D \in \mathcal{D}_e$ the data center $\mathrm{parent}(D) = D_f$, which is the fog data centers that D is connected to, and for $D \in \mathcal{D}_f$, we set $\mathrm{parent}(D) = D_c$.

5.2.2 Placement of services

In the following, we use the notation summarized in Table 5.1 for describing placement of services and the related metrics. We consider the problem of placing a set of services on a cloud infrastructure composed of the set of data centers $\mathcal{D} = \{D_1, \ldots, D_N\}$, where N is the total number of data centers. Each service S is composed of J_S microservices denoted by $\sigma_1, \cdots \sigma_{J_S}$; each microservice σ_j (with $j = 1, \ldots, J_s$) requires a certain amount of CPU, disk and RAM. In practice, RAM and CPU are both the most scarce resources of cloud infrastructures. We denote $c(\sigma)$ and $r(\sigma)$ by the resource requirements of an arbitrary microservice σ in terms of CPU and RAM, respectively.

The placement problem consists of finding a mapping function h from the set $\mathcal{S}$ of services to the set $\mathcal{D}$ of data centers. More precisely, we consider the mapping $h : S \in \mathcal{S} \rightarrow (h(\sigma_1), \ldots, h(\sigma_{J_S})) \in \mathcal{D}^{J_S}$,

$$\begin{aligned} h : \mathcal{S} &\longrightarrow \mathfrak{M}(\{0\} \cup \mathcal{D}) \\ S &\longrightarrow \{h(\sigma_1), \ldots, h(\sigma_{j_S})\} \end{aligned} \tag{5.1}$$

where $\mathfrak{M}(\{0\} \cup \mathcal{D})$ is the multiset with elements in $\{0\} \cup \mathcal{D}$ and $h(\sigma_j) = D_n$ if microservice σ_j is placed on data center D_n. If microservice σ_j cannot be placed because of resource exhaustion, then we set $h(\sigma_j) = 0$. In that case, no microservices of S are placed and $h(S) = \mathbf{0} \stackrel{def}{=} (0, \ldots, 0)$.

Let $\mathcal{M}_n^{(h)}$ denote the set of microservices placed on the data center D_n under placement policy h. Let C_n and R_n denote the CPU and RAM capacities of data center D_n, respectively. The following constraints shall apply:

$$\sum_{\sigma \in \mathcal{M}_n^{(h)}} c(\sigma) \leq C(D_n) \text{ and } \sum_{\sigma \in \mathcal{M}_n^{(h)}} r(\sigma) \leq R(D_n). \tag{5.2}$$

Table 5.1: Notation for the cloud infrastructure, the placement of services, and related metrics.

$\mathcal{D} = \{D_n, n = 1, \ldots, N\}$	set of data centers in the system
$C(D_n)$	CPU capacity of data center D_n
$\mathcal{D}_e$ (resp1., $\mathcal{D}_f$, resp2., $\mathcal{D}_c$)	set of edge (resp.1, fog, resp.2, centralized) data centers ($\mathcal{D} = \mathcal{D}_e \cup \mathcal{D}_f \cup \mathcal{D}_c$)
$R(D_n)$	RAM capacity of data center D_n
$S = \{\sigma_1, \ldots, \sigma_{j_S}\}$	service S composed of microservices σ_j for $j = 1, \ldots, j_S$
$\mathcal{C} = (C(D_n), n = 1, \ldots, N)$	capacity vector of the system
$c(\sigma)$	CPU requirement by microservice σ
$r(\sigma)$	RAM requirements by microservice σ
$\mathcal{S}$	set of services to be placed
$h : S \rightarrow (h(\sigma_1), \ldots, h(\sigma_{j_S}))$	placement of service S on the set of data centers ($h(\sigma_j) \in \mathcal{D}$)
$\mathcal{S}^{(h)}$	set of services placed under placement h
$\mathcal{S}_f^{(h)}$	set of fragmented placed services under placement h
$\mathcal{M}_n^{(h)}$	set of microservices placed under placement h on data center D_n
$\mathcal{M}^{(h)}$	set of microservices placed under placement h in the system
$\nu_S(\sigma, \sigma')$	number of messages exchanged between microservices σ and σ' of service S
$d_{n,m}$	delay between data centers D_n and D_m
Δ_S^h	the $(1+j_S) \times (1+j_S)$ delay matrix of service S, whose (i, j) entry is equal to $d_{h(\sigma_i),h(\sigma_j)}$
$\ell_S^{(h)}$	latency experience by service S under placement h

The set of services having a microservice hosted by data center D_n is denoted by $\mathcal{S}_n^{(h)}$ and is defined by

$$\mathcal{S}_n^{(h)} = \{S \in \mathcal{S} \mid \exists \sigma \in S \text{ and } h(\sigma) = D_n\}.$$

The set of services (resp. microservices) that can be placed is $\mathcal{S}^{(h)} = \bigcup_{n=1}^{N} \mathcal{S}_n^{(h)}$ (resp. $\mathcal{M}^{(h)} = \bigcup_{n=1}^{N} \mathcal{M}_n^{(h)}$). The mapping h has to satisfy constraints (5.2) while additional criteria can be considered, e.g., load balancing between data centers, maximization of the number of placed services, etc. For instance, the maximization of the utilization of the CPU of the cloud infrastructure reads

$$\max_h \sum_{S \in \mathcal{S}^{(h)}} \sum_{\sigma \in S} c(\sigma), \tag{5.3}$$

while the maximization of the fraction of services which can accepted in the system reads

$$\max_h \frac{|\mathcal{S}^{(h)}|}{|\mathcal{S}|}. \tag{5.4}$$

Finally, anti-affinity rules can be introduced to prevent two microservices from being placed on the same data center (for instance for security or resilience reasons).

5.2.3 Latency of services

In the following, we are interested in the latency experienced by a service S composed of microservices $\sigma_1 \cdots \sigma_{J_S}$. A dummy microservice σ_0 with no resource requirements is added to represent the location of the user of the service, which is attached to an edge node of the cloud infrastructure (see Figure 5.1). Microservices σ_j, $j = 0, \ldots, J_S$, exchange messages to execute the application they support. In the following, we define the message exchange matrix $\nu_S = (\nu_S(\sigma_i, \sigma_j))$ for service S, where $\nu_S(\sigma_i, \sigma_j)$ for $i, j = 0, \ldots, J_S$, is the number of messages exchanged between microservices σ_i and σ_j of service S. Even if the exchange of messages between two microservices is asymmetric, the latency only depends on the number of messages exchanged regardless of their direction. Hence, we can make the assumption that $\nu_S(\sigma_i, \sigma_j) = \nu_S(\sigma_j, \sigma_i)$ and in addition $\nu_S(\sigma_i, \sigma_i) = 0$. The $(j_S + 1) \times (j_S + 1)$ matrix ν_S is then symmetric with zeros on the diagonal.
If microservices σ_i and σ_j are not placed on the same data center, then the transmission across the links connecting the two data centers introduce latency in the execution of the service. Let $d_{n,m}$ denote the delay between data centers n and m. In the following, we neglect the delay inside a data center (i.e., $d_{n,n} = 0$) as this delay is low compared to transmission delays between remote data centers.
For a given placement h, let us define the $(j_S + 1) \times (j_S + 1)$ delay matrix $\Delta_S^{(h)} = (d_{h(\sigma_i),h(\sigma_j)})$ for service S

under placement h. Then, the latency affecting service S is

$$\ell_S^{(h)} = \sum_{0 \leq i < j \leq J_S} \nu_S(\sigma_i, \sigma_j) d_{h(\sigma_i), h(\sigma_j)} \tag{5.5}$$

Owing to the symmetry of matrices,

$$\ell_S^{(h)} = \frac{1}{2} \mathrm{Tr}(\nu_S \Delta_S^{(h)}), \tag{5.6}$$

where Tr is the trace operator.

The global latency of the system under placement h is defined as

$$\mathcal{L}^{(h)} = \sum_{S \in \mathcal{S}^{(h)}} \ell_S^{(h)} \tag{5.7}$$

and the average latency as

$$\overline{\mathcal{L}}^{(h)} = \frac{1}{|\mathcal{S}^{(h)}|} \sum_{S \in \mathcal{S}^h} \ell_S^{(h)}. \tag{5.8}$$

With regard to placement, we can introduce the following optimization problems: minimizing global (resp. average) latency $\min_h \mathcal{L}^{(h)}$ (resp., $\min_h \overline{\mathcal{L}}^{(h)}$) or minimizing the maximum latency of services $\min_h \max_{S \in \mathcal{S}^{(h)}} \ell_S^{(h)}$ with h achieving the maximum cloud occupancy (criterion (5.3)) or acceptance rate (criterion (5.4)).

5.3 Dynamical system and associated metrics

5.3.1 Dynamical setting

While many studies in service placement (VNFs or network slices) assume a static setting, where the global set of services to be placed is known in advance and fixed, we consider a dynamic system where services join and leave the system. In that case, the placement strategy should take account of the service dynamic in the sense that:

- Each arriving service has to be placed by taking into account the current state of the system, possibly by migrating some microservices in order to control the latency of the new service while also controlling that of services with migrated microservices;

- At each departure of a service, resources are released and can be used for microservices migration so as to reduce latency of services in the system, e.g., according to the optimization problems (see Section 5.2.3).

In addition, we assume that services are anchored in the sense that the service user is attached to an edge data center. In contrary to studies on MEC in which users are moving, we focus on network functions instantiated for fixed groups of users (e.g., a RAN area, a company, ephemeral groups of users willing to have connectivity to the network, etc.). For this purpose, a dummy microservice with zero capacity requirements is located at an edge data center. If services are accepted on a capacity basis only, then we have a blocking system. As long as the service can be placed, the service is accepted regardless of incurred latency.

In the following, we assume that there are K classes of services. Those services of class k ($k = 1, \ldots, K$) arrive according to Poisson processes with rate λ_k. A service of class k, if accepted, stays for a random amount of time with mean $1/\mu_k$. A service S_k of class k has a global resource requirement $A_k = \sum_{\sigma \in S_k} c(\sigma)$. The global capacity of the system is $C = \sum_{n=1}^{N} C(D_n)$.

If the global capacity C is finite then we have a multirate loss network (see the seminal paper [47]). This kind of model has been used to dimension multiservice circuit switched networks and the blocking probability can be derived in various load regimes (see for instance [26]).

Let n_k be the number of services of class k in the system. The probability $\mathbb{P}(n_1, \ldots, n_K)$ of having n_k services of class k, $k = 1, \ldots, K$, in the system is:

$$\mathbb{P}(n_1, \ldots, n_K) = \frac{1}{\mathcal{G}} \prod_{k=1}^{K} \frac{\rho_k^{n_k}}{n_k!},$$

where $\rho_k = \frac{\lambda_k}{\mu_k}$, the K-tuple $(n_1, \ldots, n_K)$ has to belong to the set of admissible states $\mathcal{A}$ defined by

$$\mathcal{A} = \{\mathbf{n} = (n_1, \ldots, n_K) \in \mathbb{N}^K : \sum_{k=1}^{K} n_k A_k \leq C\},$$

and the normalizing constant $\mathcal{G}$ is

$$\mathcal{G} = \sum_{\mathbf{n} \in \mathcal{A}} \prod_{k=1}^{K} \frac{\rho_k^{n_k}}{n_k!}.$$

The blocking probability of a service of class k is

$$\mathbf{P}_k = \frac{1}{\mathcal{G}} \sum_{\mathbf{n} \in \mathcal{A}_k^c} \prod_{k=1}^{K} \frac{\rho_k^{n_k}}{n_k!},$$

where

$$\mathcal{A}_k^c = \{(n_1, \ldots, n_K) \in \mathbb{N}^K : C - A_k < \sum_{k=1}^{K} n_k A_k \leq C\}.$$

It is worth noting that the above results are insensitive to the distribution of holding times of services. Loss systems have extensively been studied in the technical literature, notably in multi-rate circuit switched networks.

See for instance [26], which gives an estimation of loss probabilities in different regimes. When the capacity is infinite (as it is assumed in the following), the number N_k of services of class K is Poisson with mean λ_k/μ_k, that is,

$$\mathbb{P}(N_k = n_k) = \frac{\rho_k^{n_k}}{n_k!} e^{-\rho_k}. \tag{5.9}$$

While it is easy to determine the probability mass function of the number of services in the global system, which does not depend on the placement algorithm, the occupancy of individual data centers is much more complicated to derive and depends on the placement algorithm. In fact, the occupancy of one data center depends on that of other data centers. This correlation is very difficult to capture in a mathematical model. Even the overshoot process of edge data centers cannot be easily described. Simple approximations stating that the overshoot process of one edge data center is a Poisson process or an Interrupted Poisson Process [77], are in practice not accurate when the load is high, which is the case in our setting. This is why we shall rely on heuristics in the following.

5.3.2 Metrics

When dealing with a QoS requirement like latency, we could impose that when a service joins the system and the QoS objectives for this service cannot be met, then the service is rejected. This may however lead to under-utilization of the system. Instead, we propose to accept all services and we use the capability of migrating microservices to keep the latency under control. An issue is to determine control metrics.

So far, we have defined in Section 5.2.3 latency of services in a static situation. We can nevertheless define a random variable $\ell^{(h)}$ taking values in the set $\{\ell_S^{(h)}, S \in \mathcal{S}^{(h)}\}$. When dealing with a dynamic system, we compute the latency of those services that are in the system. Contrary to the static case, the service latency can vary in time due to migration. If a service S has a holding time τ_S, then we define the mean latency under a migration strategy m as

$$\bar{\ell}_S^{(m)} = \frac{1}{\tau_S} \int_{t_S}^{t_S+\tau_S} \ell_S^{(m)}(u) du,$$

where t_S is the arrival date of service S and $\ell_S^{(m)}(t)$ is the latency experienced by service S at time t.

When considering a population of services under service migration policy m and placement h, we define the mean latency as

$$\mathbb{E}(\ell^{(m)}) = \frac{1}{|\mathcal{S}^{(h)}|} \sum_{S \in \mathcal{S}^{(h)}} \bar{\ell}_S^{(m)}).$$

This is a global metric reflecting the efficiency of a migration policy m in terms of latency.

Latency is due to the placement of the microservices (including the dummy microservice) on distant data centers, which reflects the fragmentation of service. More precisely, for a given placement h, the fragmentation index of a service S is set equal to $\eta_S^{(h)} = |h(S)|$, thereby representing the number of data centers hosting service S. The set fragmented services is denoted by $\mathcal{S}_f^{(h)}$ under placement h.
With migration, the placement of a service may vary and impact its fragmentation. The fragmentation index of a placed service is denoted $\eta_S^{(m)}$ when the migration strategy m is applied. The objective of a migration strategy m is to decrease the initial fragmentation indices $\eta_S^{(h)}$ of services S for a placement h. The set of fragmented services after applying migration strategy m is denoted by $\mathcal{S}_f^{(m)}$.

5.4 Algorithms for placement and migration of services

The proposed heuristic algorithms place the newly arrived services (§5.4.1) and further reassign highly fragmented services (§ 5.4.2).

5.4.1 Placement of new services

For the placement of arriving services, we consider two greedy algorithms: The Greedy First Fit algorithm and the Greedy Best Fit algorithm.

5.4.1.1 Greedy First Fit algorithm (GFF)

This algorithm places the microservice chain on the first available data centre and is commonly used for bin-packing problems as it is very fast in searching for the first available block. In this way, the nodes closest to the end user are acquired first over the nodes located at a distance (the edge node, followed by the fog nodes, and then the cloud node in the final stage.
Our approach (Algorithm 2) involves the following steps:

1. Initialize by allocating the user of the service. For this purpose, a random location is selected at an edge node n (line 2). Note that the end user does not consume/occupy resource; this user is introduced for latency computation.

2. Further, place the chained microservices of the service on the selected edge node (closer to end-user location) until the resource capacity is met (lines 11-14).

3. Then, move to the nearest regional node (i.e., attached parent node - lines 16-17) to place the remaining microservices in case all the microservices are not placed.

4. Proceed to the cloud node until all the microservices are placed.

Algorithm 2: $GFF(S, \mathcal{D}, \mathcal{C})$ algorithm for the placement of a service S on a set of data centers $\mathcal{D}$ with capacity $\mathcal{C}$.

Input : Service $S = \{\sigma_1, \ldots, \sigma_{J_S}\}$;
$c(\sigma)$: required CPU capacity of microservice $\sigma \in S$;
Set of data centers $\mathcal{D} = \mathcal{D}_e \bigcup \mathcal{D}_f \bigcup \mathcal{D}_c$;
$\mathcal{M}^{(h)}$: set of placed microservices;
$\mathcal{M}_n^{(h)}$ set of placed microservices on data center D_n;
Output: $\mathcal{D}_S$: set of data centers hosting service S;
$\mathcal{M}_1^{(h)}, \cdots, \mathcal{M}_N^{(h)}$ set of placed microservices on the set of data centers;

$\mathcal{D}_S = \emptyset$;
$n = \text{random}(|\mathcal{D}_e|)$; //randomly pick up a edge data center
$D = D_n$;
$\text{Append}(\sigma_0, \mathcal{M}_n^{(h)})$;
$\text{Append}(D_n, \mathcal{D}_S)$;
$\hat{S} = S$;
while $\hat{S} != \emptyset$ **do**
    $\sigma = \text{ExtractFirstElement}(\hat{S})$;
    NotPlaced = True ;
    while *NotPlaced* **do**
        if $c(\sigma) \leq C(D)$ **then**
            $\text{Append}(\sigma, \mathcal{M}_n^{(h)})$;
            $\text{Append}(D, \mathcal{D}_S)$;
            NotPlaced=False;
            $C(D) = C(D) - c(\sigma)$; //Update residual capacity;
        else
            $D = \text{Parent}[D]$;
            $n = \text{index}[D]$;
        end
    end
    $\text{Remove}[\sigma, \hat{S}]$;
end

5.4.1.2 Greedy Best Fit algorithm (GBF)

This algorithm corresponds to a greedy method that aims at reducing the fragmentation of microservices that compose a given service by 1) keeping to a minimum the number of data centers occupied by microservices of a given service and 2) allocating all the co-joined microservices as much as possible on the same data center that has been selected in a greedy manner. In order to place the microservices in a best fit manner, the whole service must be placed on a single data center otherwise the whole service moves to the next available data center in a greedy manner. This strategy tends to reduce the latency caused by communications between microservices

(except the end user).

The two algorithms presented have small complexity. They can be adopted to place the microservices initially and on each service arrival while migration of microservices is executed only when the services leaves the system in order to fully optimize the released resources.

The computational complexity for GFF (in algorithm 2) for a total of S services to be placed on D data centers is $O(S)$. Likewise, the time complexity for GBF approach to map the whole service is $O(S)$. On the other hand, the migration algorithm 3 includes the sorting of fragmented services and then placing the highly fragmented service. Our implementation used the inbuilt $sort()$ function of Matlab which is based on Quick Sort (popularly known as fastest algorithms for sorting). This tends to provide the time complexity of $O(n * logn)$ for the set of fragmented services $S_f^* = n$.

5.4.2 Migration Strategy

Once placed, microservices could be migrated in order to improve the latency of services. The step by step executions (see Figure 5.2) proceeds as follows:

i The migration of microservices starts with the departure of service(s). Given that departed service(s) release(s) resources from their respective data center(s), it is pivotal to make use of these available resources to improve the latency of other services.

ii The migration is triggered if the number of departures is higher than a given threshold value to avoid the triggering of migration at each service departure.

iii Based on the ordered list of fragmented services (set $\mathcal{S}_f^{(h)}$), the most fragmented service (composed of chained microservices) is selected to proceed with the process.

iv The microservices which (i) experience high latency because they are located on distant data centers, and/or (ii) exchange many messages with end-users are chosen from the selected fragmented service list. As a consequence, only highly communicating and paired microservices are privileged for migration rather than all the microservices (even-though there is enough resources available) to minimize the global latency.

v As an optimal target data center to migrate the microservice, it is suitable to find the data center, where the end user is located that led to minimize the latency.

vi Note that the migration takes place in the case there are sufficient available resources to host the microservices that need to be migrated and if the migration results in a latency gain. Otherwise, it is necessary to

free some space by re-allocating some microservices composing service(s) experiencing no fragmentation and hence small latency: such microservices are typically placed on the same data center (or on a nearby data center) and exchange the least number of messages with the other microservices.

vii The candidate microservice is moved to the nearest data center to host as per greedy approach.

viii At the final stage, the highly active microservices are migrated after verifying that the *latency gain* corresponding to the difference between the latency reduction achieved by migrating the microservice of S and the latency increase due to the migration of the candidate microservices is positive.

Precisely, the migration strategy detailed Algorithm 3, involves the following steps:

1. Sort the set of placed services in decreasing order of fragmentation and create the ordered set $\mathcal{S}_f^*$ of fragmented services (line 3) .
2. If $\mathcal{S}_f^*$ is not empty, select the service S_i in $\mathcal{S}_f^*$ with the greatest fragmentation index (line 5).
3. Identify two microservices σ_i and σ_j (with $i \geq 0$, $j > 0$, $i \neq j$) among the services that are the most fragmented and induce the highest latency (line 6).
4. The migration takes place if (i) the required capacity $c(\sigma_j)$ is available on data center $D(\sigma_j)$ and the migration causes a latency gain (lines 13-16). If the necessary capacity is not available, the service S_j with least fragmentation index among the services hosted on data center $D(\sigma_i)$ is considered to free a capacity larger than or equal to $c(\sigma_j)$. If this is not possible, then migration cannot take place. Otherwise, the selected microservices are placed on other data centers by using the greedy algorithm and the migration of σ_j takes place.

5.5 Experimental results

5.5.1 Simulation setting

For the implementation of our proposed algorithms, we consider the cloud infrastructure depicted in Figure 5.1. For our experiments, we assume that centralized cloud has infinite capacity, fog nodes (from 10 to 12) have capacity 100, edge nodes (labeled from 1 to 9 in Figure 5.1) have capacity 20 (see Table 5.2).
Concerning latency between the nodes, the assumed latency between the edge and fog nodes is 1 unit, between the fog nodes is 2 units and for the centralized cloud is 3 units. Further, we have two types of services differing in the number of microservices:

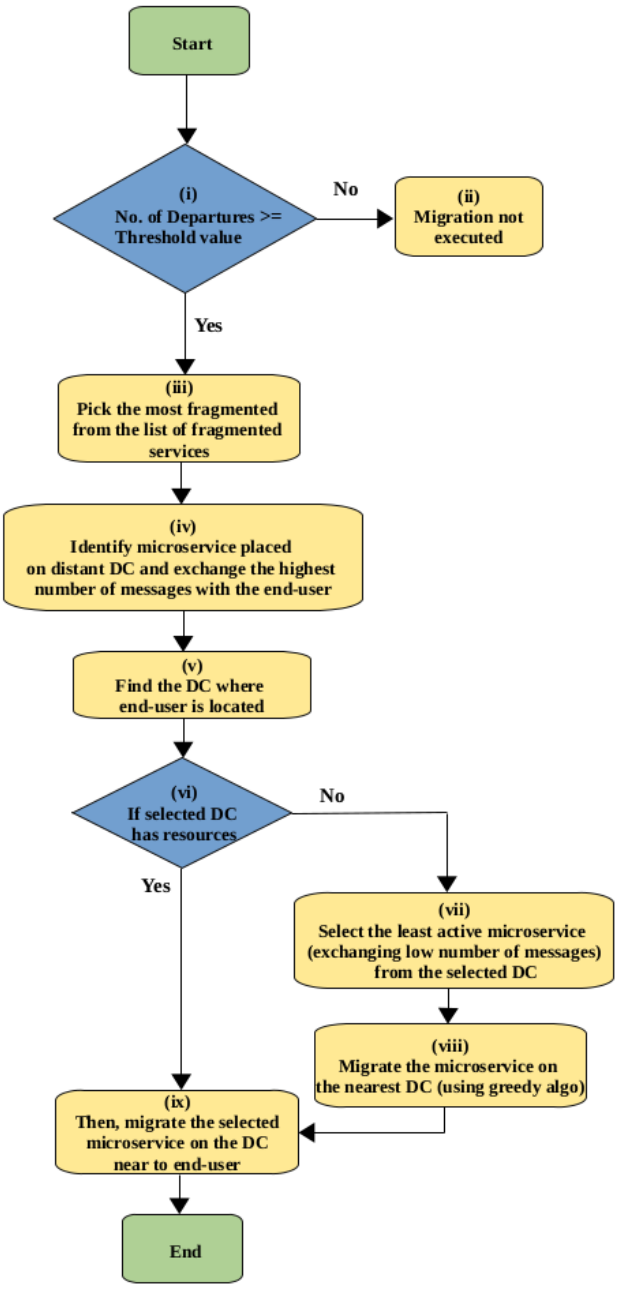


Figure 5.2: Flow chart of migration approach

Algorithm 3: $Migration(\mathcal{D}, \mathcal{C})$ algorithm for the migration of a service after the departure of a service on a set of data centers $\mathcal{D}$ with capacity $\mathcal{C}$

Input : Set of Data centers $\mathcal{D}$; Total available capacity $\mathcal{C}$;
Required CPU capacity $c(\sigma)$ for microservice $\sigma \in S$;
Output: Updated set $\mathcal{D}$ of data centers;

S_f^*: set of fragmented services;
$\mathcal{D}_S$= current placement in the data centers ;
$S_f^* = sortDescendOrder(S_f^*)$;
for *each iteration* $i = length(S_f^*)$ **do**
    $S_i = MaxFragmentedService(S_f^*)$;
    $find(\sigma_i, \sigma_j)$; // find 2 μservices placed on different DCs and inducing the maximum delay;
    $find(\mathcal{D}_{\sigma_i}, \mathcal{D}_{\sigma_j})$ // Get current location of μservice and end-user;
    if $c(\sigma_j) > C(D_{\sigma_i})$ **then**
        $S_j = MinFragmentedService(S_f^*, D(\sigma_i))$;
        $select(\sigma) == c(\sigma_j)$ // Select the set of μservices equals to capacity required to place the migrated μservice;
        $FirstFit(D_\sigma)$;
    end
    if $new_{L_{S_j}} + new_{L_{S_i}} < old_{L_{S_j}} + old_{L_{S_i}}$ **then**
        $Migrate(\sigma_j, D_{\sigma_i})$; $Update[D_S]$;
        remove S_i from S_f^*
    end
end

Table 5.2: Description of data centers

Data center type	**Number of data centers**	**Total capacity of each data center**
Edge DC ($\mathcal{D}_e$)	9	20
Fog DC ($\mathcal{D}_f$)	3	100
Cloud DC ($\mathcal{D}_c$)	1	∞

- **Small service** corresponds to lightweight applications (e.g., a firewall) and consists of a small number of microservices, namely 3 microservices exchanging messages with $\nu(\sigma_1^1, \sigma_2^1) = 2$ and $\nu(\sigma_2^1, \sigma_3^1) = 4$; the number $\nu(\sigma_0^1, \sigma_1^1)$ of messages between the end user and the first microservice may change.

- **Large service** corresponds to heavy-weight application; we specifically assume that each service comprises 10 microservices exchanging messages with $\nu(\sigma_1^2, \sigma_2^2) = \nu(\sigma_4^2, \sigma_5^2) = \nu(\sigma_6^2, \sigma_7^2) = \nu(\sigma_7^2, \sigma_8^2) = 3$, $\nu(\sigma_2^2, \sigma_3^2) = \nu(\sigma_3^2, \sigma_4^2) = \nu(\sigma_5^2, \sigma_6^2) = 2$, $\nu(\sigma_8^2, \sigma_9^2) = 4$ and $\nu(\sigma_9^2, \sigma_{10}^2) = 1$; as above, the number $\nu(\sigma_0^1, \sigma_1^1)$ may change.

The required capacity of each microservice is set equal to 1.

We assume that the two types of services (i.e., $K = 2$) arrive according to a Poisson process with rate λ_k; a service of type k (with $k \leq K$) stays in the system for an exponentially distributed period of time with mean $1/\mu_k$ equals to 1. Under the assumption that the resource requirement of microservices is equal to 1, the resource requirement of a service of type k is equal to A_k, where A_k is the number of microservices composing the service.

Since we assume that the capacity of the centralized cloud is infinite, we define the load of a system by considering edge and fog data centers only. The load offered by services of type k is

$$\mathfrak{r}_k = \frac{A_k \rho_k}{\mathcal{C}_0},$$

where $\rho_k = \frac{\lambda_k}{\mu_k}$ and the quantity $\mathcal{C}_0 = \sum_{D \in \mathcal{D}_e \cup \mathcal{D}_f} C(D)$ is the capacity of edge and fog data centers. The total load is equal to $\mathfrak{r}_1 + \mathfrak{r}_2$.

Since we deal with a system with no blocking, the number of services in the system has a Poisson probability mass function as stated in [47]. The mean and the variance of the number of microservices of type k in the system is then

$$\mathbb{E}(N_k) = A_k \rho_k = \mathfrak{r}_k \mathcal{C}_0 \text{ and } \mathrm{Var}(N_k) = A_k^2 \rho_k = A_k \mathfrak{r}_k \mathcal{C}_0, \tag{5.10}$$

respectively. While the total number of services of types 1 and 2 in the system have Poisson distributions, it is difficult to compute the number of microservices hosted by a data center for a given placement strategy (GFF, GBF, or migration).

5.5.2 Numerical results

For the simulation experiments, we have taken in a first step $\mu_1 = \mu_2 = 1$ and the load for type 1 (resp. 2) services $\mathfrak{r}_1 = 1.5$ (resp. $\mathfrak{r}_2 = 2.5$) with $A_1 = 3$ and $A_2 = 10$ as stated in the previous section. The fact that

$\mu_1 = \mu_2$ entails that large and small services stay in distribution for the same duration of time in the system. Since $\mathcal{C}_0 = 480$ (see Table 5.2), we can fix the arrival rate λ_1 and λ_2 of the Poisson processes describing the arrivals of services at the system if we assume that $\lambda_2 = \lambda_1/2$, indicating that there are less large services but they offer a larger load.

To compute the probability mass functions of the quantities of interest, we use the "Poisson Arrival See Time Averages" (PASTA) property: we record at each service arrival the values of a variable that we want to observe. Then, we compute the normalized histogram of the successive observations. Thanks to PASTA, this yields the stationary distribution of the random variable under consideration.

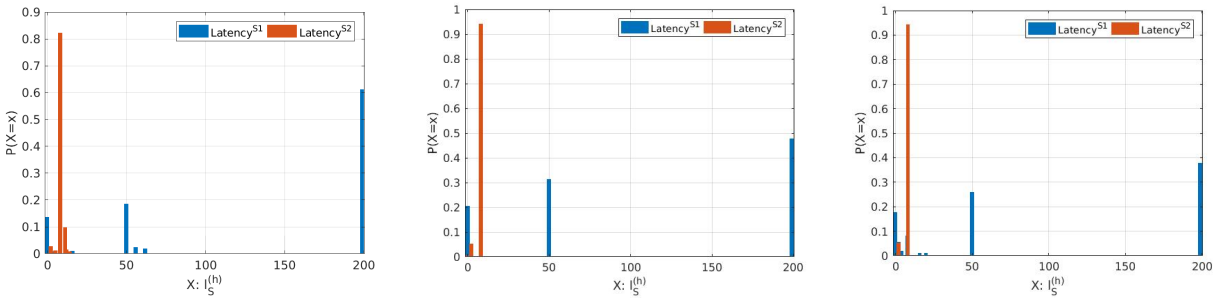
Majority of studied research works followed the greedy or first-fit algorithms while implementing or comparing their solution such as [9, 33, 80, 92]. Therefore, in order to demonstrate the potential of our proposed migration strategy, we compare it with GFF and GBF. We consider the migration strategy triggered by service departures and relying on GBF for placement. We analyze latency and fragmentation experienced by small and large services. Using the equation (5.5) for latency that accounts : (i) the number of exchange between the microservices itself and those with respective end-user of a service and (ii) the distance between the data centers for the microservices placed on distinct node or layer.

Table 5.3: Mean of global latency

Service Type	**Methods**	**Number of messages** $\nu(\sigma_0, \sigma_1)$ 2	5	10	50
Small	GFF	5.9	13.9	27.2	133.8
	GBF	4.4	11.1	22.2	111.9
	Migration	4.9	10.3	19.2	91.4
Large	GFF	8.1	8.1	8.2	8.2
	GBF	7.6	7.6	7.6	7.6
	Migration	7.6	7.6	7.6	7.6

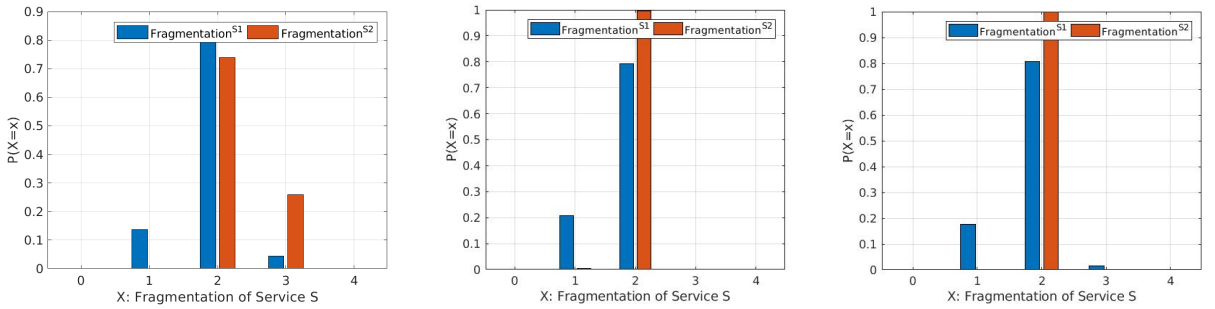
In a first step, to analyse the impact of placement and decision of moving the microservice near to the end-user through the number of messages, we let vary the number of the messages exchanged between the end user and the first microservice $\nu(\sigma_0, \sigma_1)$. In Table 5.3, our migration algorithm resulted in better performance by minimizing the average global latency for small and large services; the improvement is more significant for small services. As expected, the global latency increases with the number of messages exchanged between the microservices and the end-user since the first microservices may be placed in fog and cloud data centers. When $\nu(\sigma_0, \sigma_1)$ is equal to 2 or 5, we cannot observe much decrease in average latency after migration. If the number $\nu(\sigma_0, \sigma_1)$ increases, the global latency also improves after migration. Therefore, migration is relevant when

there is a high active exchange between microservices, notably between the user and the first microservice. Figure 5.3 compares the strategies in terms of the probability mass function (pmf) of latency $\ell_S^{(h)}$ for small and large services considering messages exchange between the user and the first microservice as $\nu(\sigma_0^1, \sigma_1^1) = 50$ and $\nu(\sigma_0^2, \sigma_1^2) = 2$ respectively. We observe in Figure 5.3(c) that the migration strategy globally minimizes the service latency in comparison to the GFF and GBF strategies (Figures 5.3(a) and 5.3(b)) even if the latency of large services is slightly increased compared to GBF.



(a) Greedy First Fit (GFF) (b) Greedy Best Fit (GBF) without migration (c) GBF with Migration

Figure 5.3: Symmetric Cloud topology - Latency of large (in red) versus small (in blue) services for $\nu(\sigma_0^1, \sigma_1^1) = 50$ and $\nu(\sigma_0^2, \sigma_1^2) = 2$.



(a) Greedy First Fit (GFF) (b) Greedy Best Fit (GBF) without migration (c) GBF with Migration

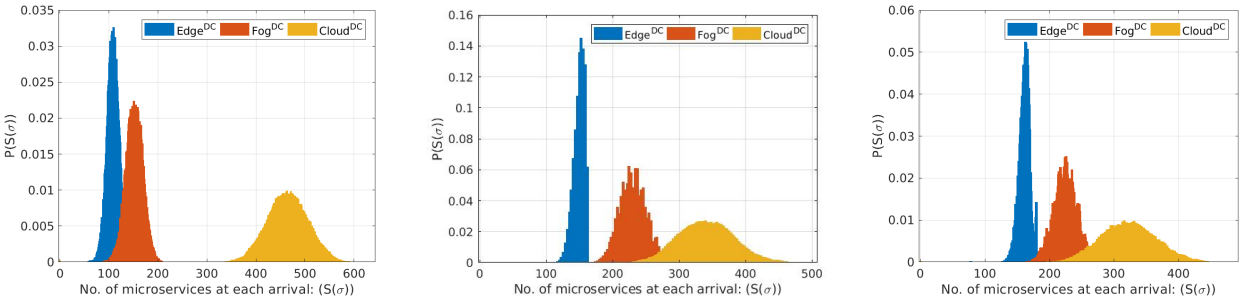
Figure 5.4: Symmetric Cloud topology - Fragmentation of large (in red) versus small (in blue) services

Likewise, we compare the strategies from the perspective of probability mass function (pmf) of fragmentation for small and large services in Figure 5.4. As expected, Figure 5.4(b) shows that for the GBF strategy, at most two data centers are used to place the service instead of the whole service on the same data center. This is due to the fact that (large-sized) services segregate or are allocated away from user's nodes when the resources at

edge are full. The fragmentation index is larger than 1 for the large-sized services that are far from the end-user and in search of a (fog/cloud) data-center with enough resource availability.

The migration strategy in Figure 5.4(c) still shows a far better outcome than GFF but remains little competitive against the GBF algorithm.

We finally study the resulting placement of the services at the different network layers (edge, fog and cloud) considering the GFF, GBF and "GBF along with migration" approaches, regarding small services (Figure 5.5) and large services (Figure 5.6). In particular, we consider the number of microservices placed on each layer which reflects the CPU resources that are consumed by the services. As expected, GFF (Figure 5.5(a)) first consumes the edge resources and then moves to the cloud layer. GBF consumes more of the edge and fog for small services (Figure 5.5(b)) compared to large services (Figure 5.6(b)) that are mostly placed on cloud data centers. Likewise, migration strategy (Figures 5.5(c) and 5.6(c)) shows a similar trend but performs better than the GBF strategy: more microservices migrate near the end user while less-active microservices moves on upper layer to make the space for actively communicating microservices.

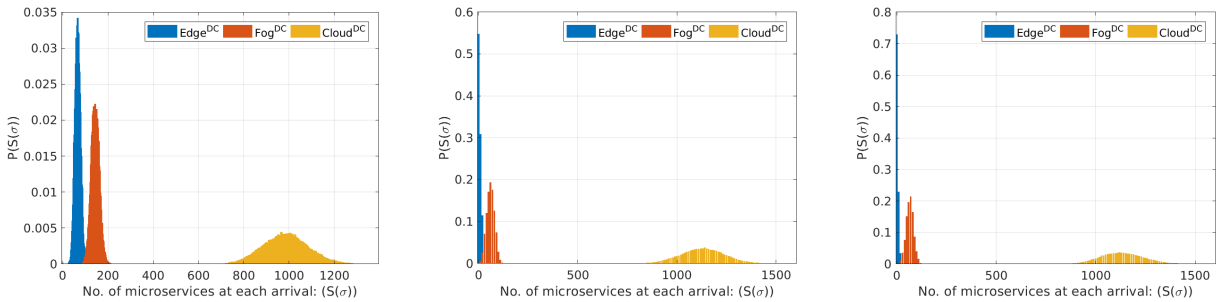


(a) First Fit (b) Best fit (c) Migration

Figure 5.5: Symmetric Cloud topology - Placement of Small Microservices on Different Layers.

Table 5.4: Occupancy Mean & Variance of Microservices at different layer (for $\mu_1 = \mu_2$).

Service	Algorithm	Symmteric					
		Mean			Variance		
		Edge	Fog	Cloud	Edge	Fog	Cloud
Small	GFF	108.41	150.54	459.30	375.42	602.19	2740
	GBF	149.80	228.91	341.60	367.74	857.31	2494
	Migration	161.45	226.19	325.04	396.10	821.65	2441
Large	GFF	66.51	145.56	981.35	291.30	601.40	1364
	GBF	6.0301	61.99	1130	71.29	505.41	1402
	Migration	3.1360	64.357	1134	41.97	487.98	1501



(a) First Fit (b) Best fit (c) Migration

Figure 5.6: Symmetric Cloud topology - Placement of Large Microservices on Different Layers

Table 5.5: Occupancy Mean & Variance of Microservices at different layer for $\mu_2 = \mu_1/10$.

Service Type	Algorithm	Symmteric					
		Mean			Variance		
		Edge	Fog	Cloud	Edge	Fog	Cloud
Small	GFF	110.98	161.34	445.81	397.49	653.85	3436
	GBF	150.12	226.90	343.40	362.73	877.20	2862
	Migration	160.51	224.52	328.624	388.82	783.73	2774
Large	GFF	64.01	135.08	995.39	303.12	601.11	1486
	GBF	5.64	63.95	1115	62.44	532.43	1535
	Migration	3.61	65.95	1126	38.43	452.33	1699

Table 5.4 reports the mean values and the variance of the occupancy at each layer for the various placement strategies. Note that the sum of the mean values on each line is roughly equal to $A_k \mathfrak{r}_k$ as stated in Equation (5.10). The small difference is due to the limited number of simulated events (service arrivals of 1 million). For the variance, the sum of each line is significantly different from the value in Equation (5.10). This is due to the fact that the number of microservices hosted by the various data centers are highly correlated. The correlation does not impact the mean but greatly affects the higher moments. This correlation seems to be impossible to model.

For the sake of completeness, we have carried out other experiments (Table 5.5) with large services lasting much longer than small services (with $\mu_2 = \mu_1/10$). We have kept the loads unchanged. The conclusion is roughly the same.

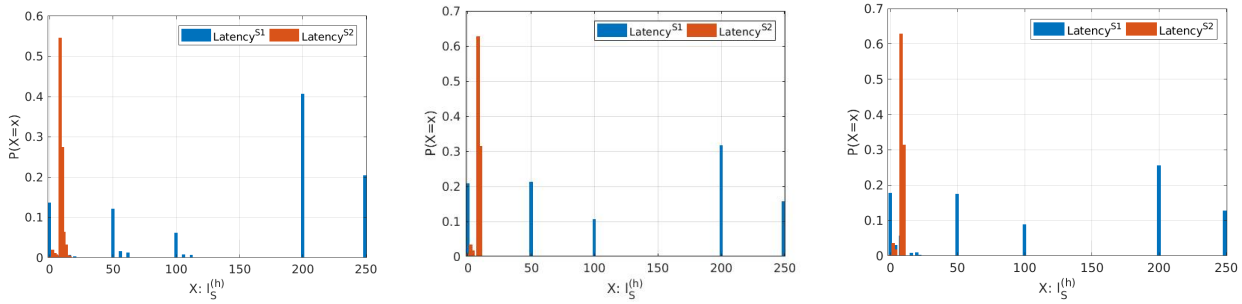
Finally, the results obtained so far are for a fully symmetric cloud topology in terms of delay along the links joining nodes at the different cloud layers. In order to study the impact of link transmission capacity, we have doubled the transmission delay between one fog node (node 11 in Figure 5.1) and the three attached edges nodes (nodes 4,5,6 in Figure 5.1). We compare in Table 5.6 the values of the mean global latency for the former and modified cloud topology. For both topologies, migration is efficient for small services (latency gain about 20 %) compared to large services when compared with GBF. The results for the latency, the fragmentation and the occupancy of nodes are roughly the same as in Figures 5.7, 5.8, 5.9 and 5.10. We have reported in Table 5.6 the values of the mean global latency for the symmetric and asymmetric cases. Likewise, Table 5.7 shows the mean and variance value of occupancy at each layer same as of symmetric topology. The conclusion is the same as in the symmetric case.

Table 5.6: Mean of global latency (Symmetric vs Asymmetric)

Service Type	**Methods**	**Symmetric**	**Asymmetric**
Small	GFF	133.759	148.278
	GBF	111.870	123.82
	Migration	91.411	102.49
Large	GFF	8.160	8.834
	GBF	7.617	8.314
	Migration	7.628	8.320

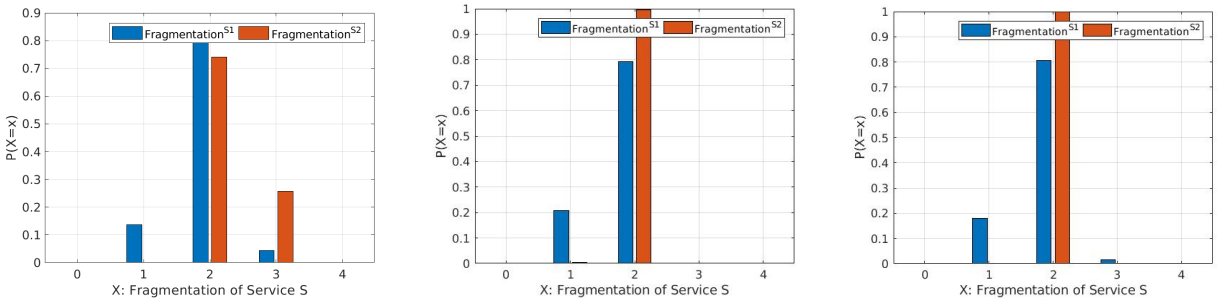
Table 5.7: Occupancy Mean & Variance of Microservices at different layer (for $\mu_1 = \mu_2$).

Service	Algorithm	Asymmteric					
		Mean			Variance		
		Edge	Fog	Cloud	Edge	Fog	Cloud
Small	GFF	107.46	150.60	461.26	369.14	583.56	2811
	GBF	149.56	230.34	340.92	375.87	873.48	2560
	Migration	160.73	226.82	323.95	398.68	826.95	2627
Large	GFF	67.43	145.40	990.67	290.24	581.61	1296
	GBF	6.304	60.72	1129	81.61	517.03	1537
	Migration	3.418	63.77	1131	42.76	487.90	1502



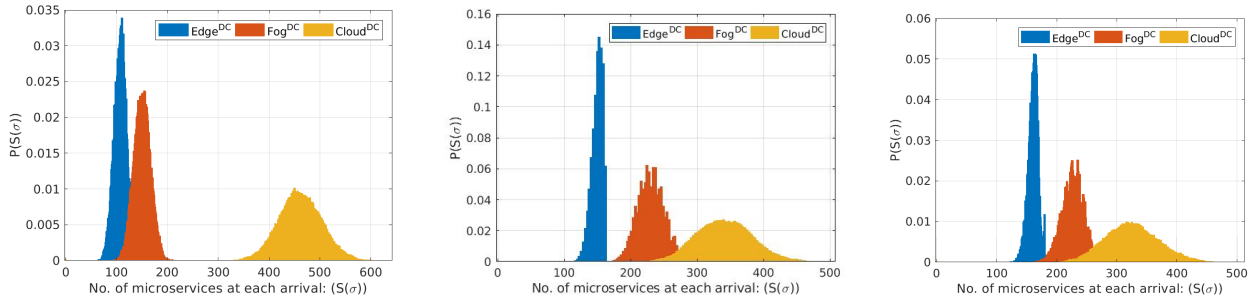
(a) Greedy First Fit (GFF) (b) Greedy Best Fit (GBF) without migration (c) GBF with Migration

Figure 5.7: Asymmetric Cloud topology - Latency of large (in red) versus small (in blue) services for $\nu(\sigma_0^1, \sigma_1^1) = 50$ and $\nu(\sigma_0^2, \sigma_1^2) = 2$.



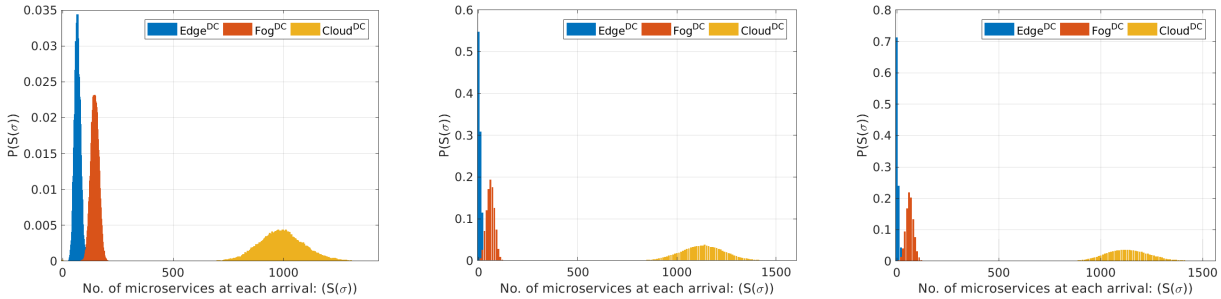
(a) Greedy First Fit (GFF) (b) Greedy Best Fit (GBF) without migration (c) GBF with Migration

Figure 5.8: Asymmetric Cloud topology - Fragmentation of large (in red) versus small (in blue) services



(a) First Fit (b) Best fit (c) Migration

Figure 5.9: Asymmetric Cloud topology - Placement of Small Microservices on Different Layers.



(a) First Fit (b) Best fit (c) Migration

Figure 5.10: Asymmetric Cloud topology - Placement of Large Microservices on Different Layers

5.6 Conclusion

Latency-sensitive applications require to carefully orchestrate the allocation and re-arrangement of (micro)services to prevent from a largely segmented placement of microservices. To address this issue in the context of network functions, we have introduced a migration approach that improves the placement of chains of the microservices in terms of latency. The proposed heuristic considers some data-centers distributed over a three-tier architecture along with the ephemeral nature of containerized services. The heuristic first chooses the highly-active microservices that are fragmented to dynamically place these latter near the end-user. At the same, the heuristic analyses the possible replacement of microservice that is already placed microservice in case of lower available resource occupancy at the desired data center. The simulation-based evaluation shows that migration performs better than static placement (e.g., GBF and GFF strategies considered in this work) and significantly reduces the latency.

Afterward, to demonstrate the real-time container/pod migration between different clusters, we devoted our proceeding chapter for the test-bed deployment which illustrates the complete chain of 4G/5G mobile networks using open-source technologies.

Chapter 6

PoC - Live migration of containerized microservices between remote Kubernetes Clusters

Content

In this chapter, we detailed our final contribution which is considered as a Proof-of-Concept of this PhD thesis work. The organization is as follows: in Section 6.2, we describe the cloud infrastructure along with the use case considered for illustrating the migration method. Section 6.3 gives the details and the motivation for implementing pod migration from one K8S cluster to another. Finally, concluding remarks (Section 6.4) are further presented.

6.1 Introduction

The microservices paradigm has recently gained popularity in the telecommunications industry with the emergence of NFV. Complex monolithic network functions, which were so far hosted on dedicated hardware, are

now preferably decomposed into ready-to-use and easy-to-manage microservices. This evolution also benefits from the emergence of container-based technologies, which have been popularised with the wide spread of cloud infrastructures, notably those based on Kubernetes (K8S) clusters. The main advantage of CNF lies in the greater flexibility offered, for example, to create, update, migrate or delete CNF.
However, the need for an appropriate mechanism to migrate a chain of CNFs (i.e., decomposed into microservices) across distributed cloud infrastructures (and thus across multiple Kubernetes clusters) requires a critical attention that is the main focus of this work. We introduce a prototype that supports migration of some CNFs of a private 5G core network instantiated into a three-level cloud infrastructure (Figure 6.1). Our prototype is based on Kubernetes (K8S) [52], which is a popular open source container orchestration and management engine for automating the deployment, scaling and managing of containerized applications. In practice, core network functions are instantiated on a data center. If the data center hosting all the CNFs gets overloaded, some CNFs are migrated. For this purpose, the prototype relies on existing K8S technologies and does not modify the K8S orchestration platform: a new pod[1] is created at the destination node to host the migrated components. Then, the requests received at the old pod are transferred to the new one when this latter gets fully active. To handle such a container migration, further development is needed, especially with distributed K8S clusters. From a practical point of view, this is quite challenging as many community groups are running in the race of achieving an efficient multi-cluster migration mechanism. Thus, the proposed solution contributes as a migration technique that can be followed to support the migration at any layer of the virtualization infrastructure along with unlocking the tangible opportunities for industry and managing the life cycle of cloud-native functions using Kubernetes.

6.2 Architecture of the testbed

In the following, we describe the cloud infrastructure (§ 6.2.1) that sustains the mobile core network (§ 6.2.2).

6.2.1 Network and cloud

We consider a three-tiered cloud infrastructure (Figure 6.1), which today reasonably represents traditional Internet Service Provider (ISP) networks involving 2 Tier/3 Tier networks. These networks, which have a national and regional footprints, interconnect end users to Tier 1 backbone networks used to exchange international traffic. The associated cloud infrastructure reflects the architecture of ISP networks with their delivered services.

[1] Several pods may be created to support the migration of large/distributed CNFs.

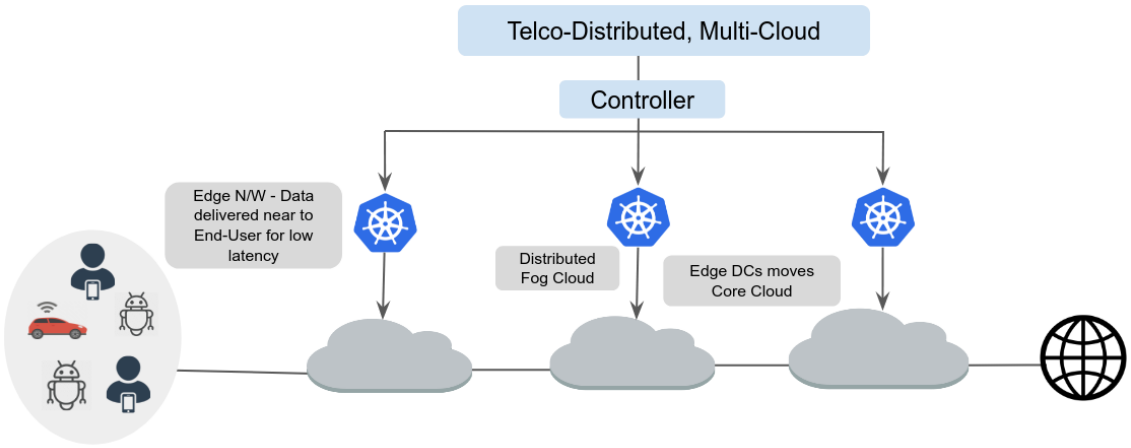


Figure 6.1: Extending Kubernetes paradigm to other domains

The cloud infrastructure includes a (national) *central cloud*, a regional cloud (attached to a Point of Presence) referred to as *fog cloud*, and clouds closer to end user (*edge cloud*). Edge data centers host operator services (e.g., cloudRAN, firewall), Business to Business (B2B) services (e.g., enterprise network functions) or applications requiring intensive computing (e.g., cloud gaming, AR/VR, etc.). We further assume that each data centre hosts a K8S cluster; the cloud infrastructure is therefore composed of a collection of distributed K8S clusters. Each cluster is characterised by specific resources capacity and bandwidth: by mean, the one farther to the end user is the cloud cluster, which has the largest centralised storage and compute resource, which offer high scalability and can be convincingly used on demand; followed by a fog cluster, which accumulates co-located nodes to reduce the distance between end-user devices and cloud data centres, and allows various functions to be easily moved to the end-user device for low-latency interactivity; finally, the edge cluster spreads accross edge nodes that are near the end-users and that provides comparatively lower latency at the cost of limited resource capacity compared to cloud and fog cloud.

6.2.2 5G Mobile Core Network

We consider an open-source core network, implemented using Magma [32], which supports diverse radio technologies, including LTE, 5G and WiFi. Magma was originally designed to extend the coverage of mobile networks but today Magma is seen as an effective solution for building private 5G networks. Thanks to the multi-operator capability offered by the Federation Gateway (FEG), Magma could also be advantageously used in the context of TowerCos [30]. As depicted in Figure 6.2, the main components of the Magma architecture include:

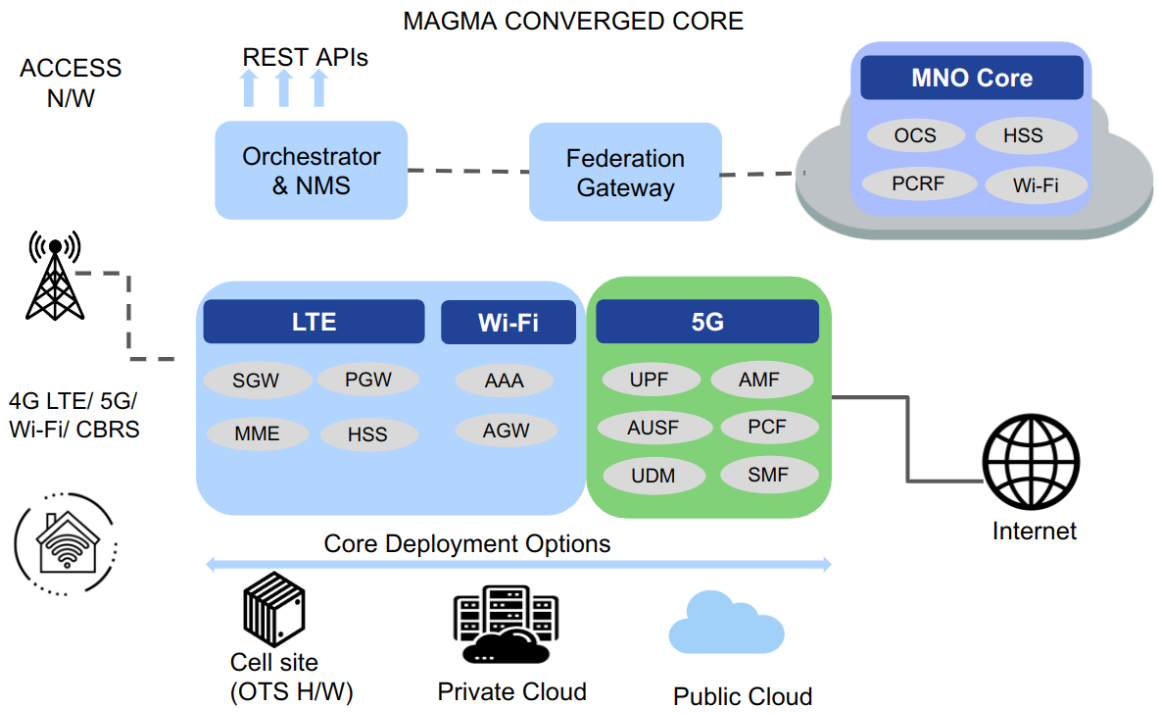


Figure 6.2: Magma Architecture and its components.

- **Orchestrator (Orc8r)**: Orchestrator is a cloud service that provides a simple and consistent way of securely configuring and monitoring the wireless network. The orchestrator has 3 main functions: a Network Management System (NMS) that supports, e.g., configuration and basic monitoring capabilities, KPIs exposed through a REST endpoint, and a secure communication channel for communication between the various gateways.

- **Access Gateway (AGW):** This functions provides mobile packet core for both 4G and 5G services. It follows a distributed architecture enabling horizontal scaling with a radio access network (RAN) including e.g., eNodeBs and gNodeBs. With 5G, AGW deals with the User Plane Function (UPF), Session Management Function (SMF), and the Access and Mobility Management Function (AMF). These three functions make up the so-called Minimal Viable Core (MVC), which is the minimal set of functions required to establish PDU sessions in 5G. In the case of 4G, the MVC comprises the MME and S/PGW functions. There is no authentication function (AUSF, UDM, UDR): authentication is mocked by provisioning via the NMS the IMSIs to UEs authorized to connect to the network.

- **Federation Gateway (FeG):** This function integrates the MNO core network within Magma by providing standard 3GPP interfaces to existing MNO components (notably the HSS in 4G and the AUSF in

5G). It acts as a proxy between the Magma AGW and the operator's network and facilitates the delivery of core functions, such as authentication, data plans, policy enforcement, and charging to be compliant with an existing MNO network and the expanded network using Magma core.

6.2.3 5G Mobile Core Network Setup

In practice, some networks functions of the Magma core network are containerized. In particular, the Magma Orchestrator (Orc8r) is deployed in Kubernetes and divided into various helm charts. The orchestrator henceforth contains various pods and services that are deployed using Minikube [69] as defined in [65]. Similarly, Magma access gateway is divided into sub-functions to support LTE, WiFi and 5G core. Overall, related sub-functions are deployed on the same K8s edge cluster and are instantiated on Bare metal at different edge nodes (as depicted in Figure 6.3).
As new CNFs require to be placed near to the user, orchestrator's pod(s) should be moved from edge cluster to a cloud cluster to free space on the edge cluster hosting the Magma core. For that purpose, Orc8r needs to be migrated to another K8s cluster without interrupting the current communication with AGW.

6.3 Migration of a 5G Core Mobile Network Component

6.3.1 Design Rational

In addition to migrating orc8r, a key requirement is to ensure that (i) there is no disconnection between AGW and the migrated orc8r, which implies that the network traffic gets properly routed and (ii) that any chained service (including AGW) that communicates with orc8r is not affected by the migration. For this purpose, we rely on Traefik [112], which is a load balancer that appropriately routes the network traffic to the desired destination (in our case, the migrated orc8r). In particular, Traefik provides a ingress controller for each migrated network function that accepts the traffic from outside the destination cluster and forwards the traffic to the migrated network function. In addition, an external DNS server (herein Cloud DNS) is used to provide hostname resolution and in particular, to handle the redirection process at the DNS level rather than by proxying.

6.3.2 Migration Strategy - Step by Step

The methodology adopted [105] to migrate orc8r is as follows:

1. The migration process starts by copying the orc8r service that needs to be migrated and by setting up a new Kubernetes cluster (in a new node located e.g. in the cloud) in which the ocr8r copy is instantiated.

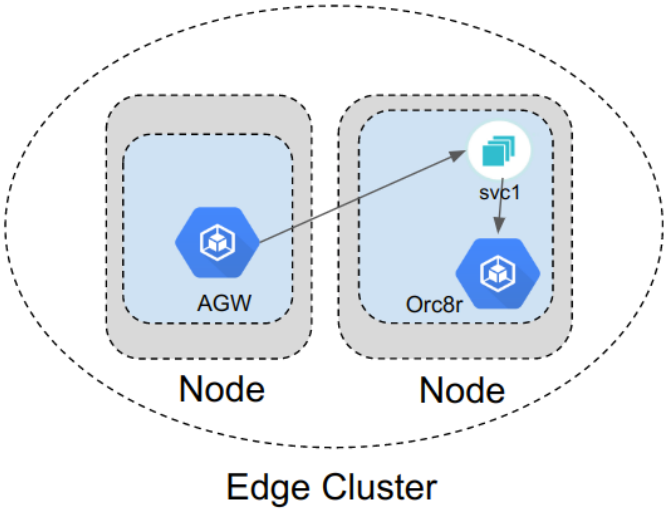


Figure 6.3: Deployment of AGW and Orc8r on K8s cluster (Edge)

For the sake of simplicity, the new instance of orc8r is named svc_2 while the original instance is named svc_1. In the configuration file, the namespace associated with svc_1 and svc_2 corresponds to *orc8r-ns*.

2. In the Traefik load balancer, a new IngressRoute is created to route the network traffic to the newly migrated service (svc_2 a.k.a. *orc8r-ns*) in the cloud cluster.

3. In Cloud DNS, a private DNS zone named $new.testnetwork.internal$ is created. In order to provide response to DNS clients that request name resolution for the newly migrated instance (svc_2 a.k.a. *orc8r-ns*), a record is added with the name $new.testnetwork.internal$ pointing to the load balancer IP address (located in a new cluster).

4. Finally, the old orc8r instance is deleted. During the migration, AGW continues communicating with the orc8r running in the cloud cluster, with essentially no downtime; the whole migration process remains completely transparent for AGW thanks to the DNS redirection and the routing performed by the load balancer.

6.3.3 Demonstration

We have used sample Kubernetes pods, namely AGW and Orc8r of Magma components and performed the manual migration steps at that time instead of executing it through a controller (as it was under development). Two different VMs have been created on a server and Minikube is used to create a Kubernetes cluster on top of each VM. To illustrate the deployment of pods, we have reported some screenshots:

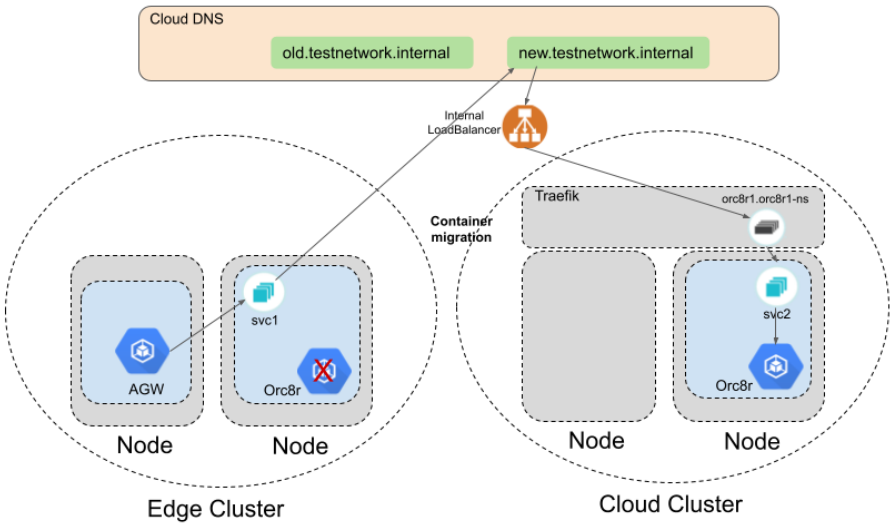


Figure 6.4: Layout of Orc8r migration in Cloud cluster

- Figure 6.5 shows the AGW and Orc8r pod deployment on the same cluster (namely, the edge cluster) through the controller VM.

```
ubuntu@vm-4:~$ kubectl get pods -n orc8r-ns
NAME    READY   STATUS    RESTARTS   AGE
orc8r   1/1     Running   0          92s
ubuntu@vm-4:~$
ubuntu@vm-4:~$ kubectl get pods -n agw-ns
NAME   READY   STATUS    RESTARTS   AGE
agw    1/1     Running   0          36s
```

Figure 6.5: Accessing the deployment of AGW and Orc8r on Edge cluster through Controller

- Further, Figure 6.6 shows the after migration view of the new running Orc8r pod on another minikube cluster (corresponding to cloud cluster) created on another VM.

```
ubuntu@vm-2:~$ kubectl get pod -n orc8r-ns
NAME    READY   STATUS    RESTARTS   AGE
orc8r   1/1     Running   0          14s
```

Figure 6.6: Deployment of Orc8r on Cloud cluster

- Likewise, an old orc8r pod has been deleted from the edge cluster after the activation of new orc8r pod (on cloud cluster) in Figure 6.7.

```
ubuntu@vm-1:~$ kubectl get pod -n orc8r-ns
NAME    READY   STATUS    RESTARTS   AGE
orc8r   1/1     Running   0          76m
ubuntu@vm-1:~$ kubectl delete pod orc8r -n orc8r-ns
pod "orc8r" deleted
ubuntu@vm-1:~$ kubectl get pod -n orc8r-ns
No resources found in orc8r-ns namespace.
```

Figure 6.7: Deletion of Orc8r on Edge cluster

- A Traefik IngressRoute routed the network traffic to service svc_2 if the destination hostname matches with $orc8r1.orc8r1-ns$ or $orc8r1.orc8r1-ns.svc.local$ as shown in Figure 6.8.

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: svc1
spec:
  entryPoints:
    - web
  routes:
  - kind: Rule
    match: Host(`orc8r1.orc8r1-ns`) || Host(`orc8r1.orc8r1-ns.svc.local`)
    services:
    - kind: Service
      name: svc2
      namespace: orc8r-ns
      port: 80
```

Figure 6.8: Demonstration of yaml file of Traefik IngressRoute

- Figure 6.9 represents the newly created ExternalName service pointing to the DNS name (new.testnetwork.internal) which resolves into load balancer IP.

```
kind: Service
apiVersion: v1
metadata:
  name: svc1
spec:
  externalName: new.testnetwork.internal
  type: ExternalName
```

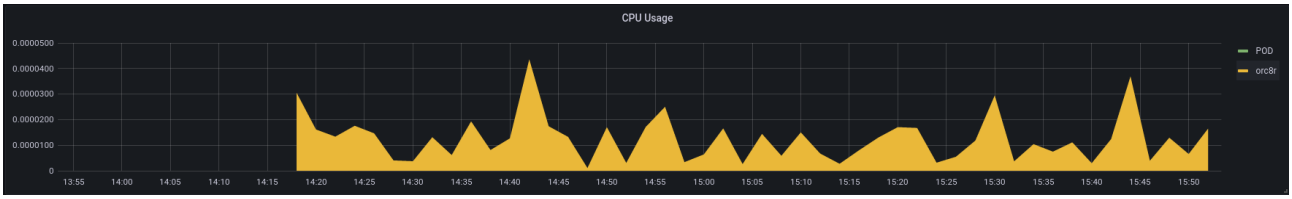
Figure 6.9: Demonstration of yaml file for Externalname service

The steps described above can be triggered by a controller (as elaborated below) for automating these steps.

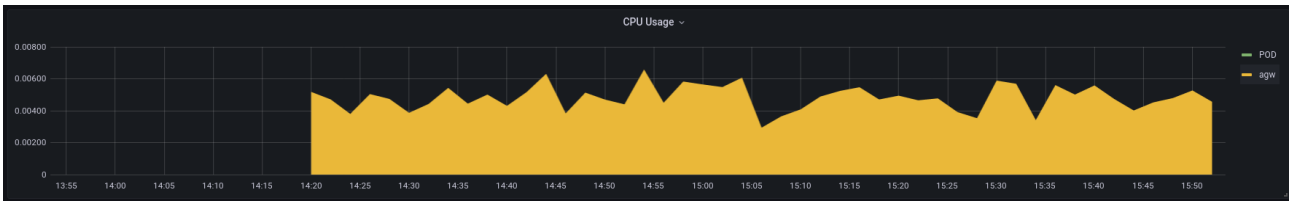
6.3.4 Development of the controller

In addition to a database storing information related to services, network and cloud topology, the controller contains four main components: Monitoring, Analysis, Discovery and Execute components. Based on the metrics collected from all the Kubernetes clusters, it performs the analysis and triggers the migration script based on a set of rules (e.g., rule defining that the resource limit exceeded).

For this purpose, we rely on the Prometheus [111] and Grafana [108] that are deployed as containers on controller VM (that hence acts as a Prometheus master). Prometheus is a monitoring and event alerting tool and Grafana enables the visualization & analysis of the data provided by Prometheus. In practice, a kube-prometheus-stack [110] helm chart is installed on VM for each cluster that includes Grafana and Prometheus operator. Notably, minikube cluster is deployed on a separate VM which blocks the access to Prometheus and Grafana services that are deployed in a cluster. Thus, the services Type from ClusterIP need to be upgraded to NodePort type that exposes the respective service via static port on node's/VM's IP. Also, a Nginx reverse proxy server is set up to publish ports using the Prometheus UI and Grafana service. This allows the Prometheus controller to receive the metric data from the respective cluster (see Figures 6.10 and 6.11 that depicts the CPU usage and storage IO distribution of pods created on edge cluster).



(a) CPU usage of Orc8r



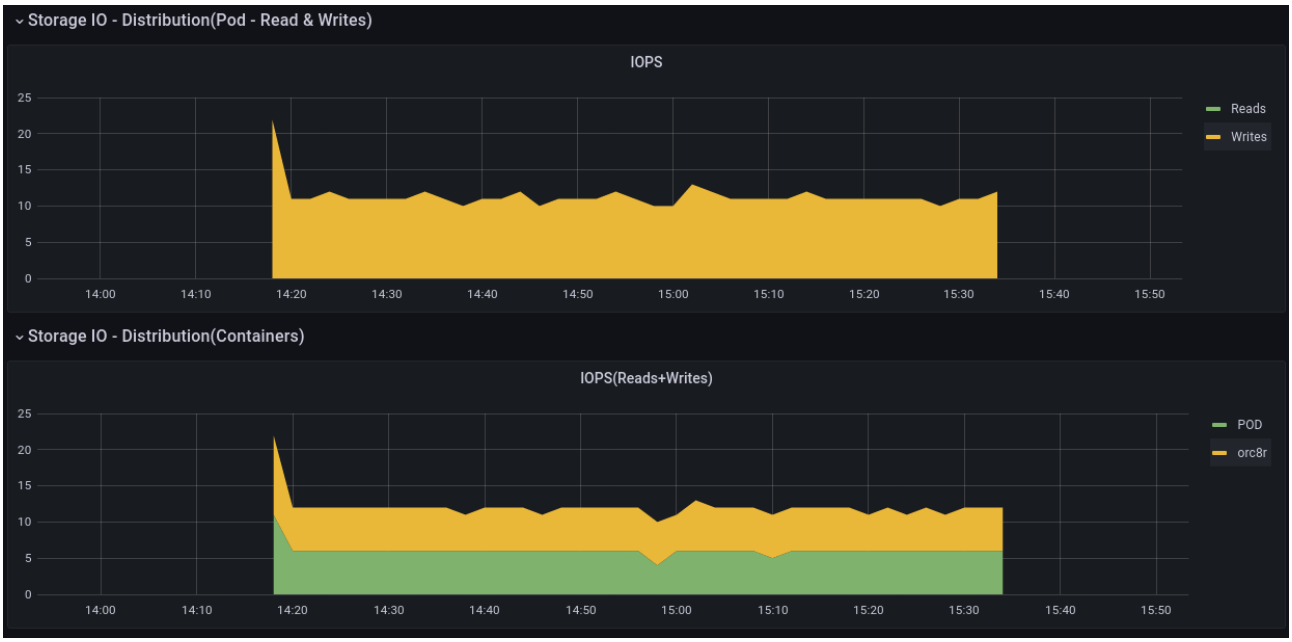
(b) CPU usage of AGW

Figure 6.10: Grafana visualization of CPU resources by the Orc8r and AGW pods.

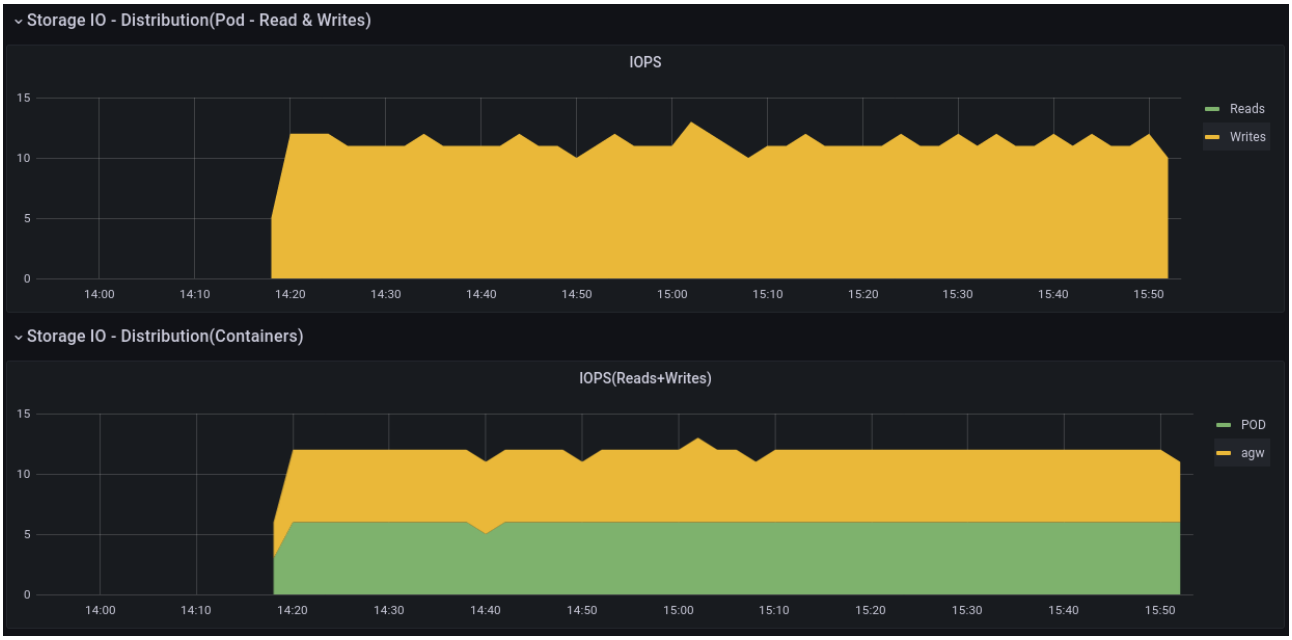
Controller also has a full access to remote clusters using the kubeconfig file. In minikube cluster, this necessitates to create a ssh proxy between the controller and cluster VMs using the following commands:

ssh -L6443:192.168.59.103:8443 10.4.11.92 $-forVM-1$

ssh -L6444:192.168.59.110:8443 10.4.11.213 $-forVM-2$



(a) Storage IO distribution of Orc8r



(b) Storage IO distribution of AGW

Figure 6.11: Grafana visualization of Storage IO distribution of the Orc8r and AGW pods.

Then, using the command *kubectl config use-context <context-name>*, we can switch between the clusters. Now, the controller has an access to the remote cluster in addition to metric data and can manage and handle the migration process based on the events or strategy introduced in [44].
This controller is also intended to continuously monitor the allocation of containerized applications so as to remap them based on the data provided by a monitoring system (e.g., Prometheus). Based on a set of requirements, a controller is able to manage and handle the migration process for all the clusters. In practice, for each cluster, a script should automate the creation of pods/services and re-routing of connection. A script can be bash or sh (distinct shells of Unix operating system) containing a sequence of commands in a file to automate the process. It eases the process as we do not require to execute multiple commands again and again for multiple migrations, a same script can be used and executed at the time of migration. In general, the script is executed by the controller to trigger the migration of pods from one cluster to another. In particular, the controller makes decisions and manages the load among different nodes of a cluster based on their available resources. Further, the proposed strategy follows a heuristic approach to choose an appropriate destination node. It also considers the ephemeral nature of containerized services and the distance between the data centers located on geo-distributed locations. Compared to the static placement approach, this migration testbed ensures an optimal placement of pods considering resource load and end-to-end latency.

6.4 Conclusion

Our objective is to perform a live migration of containerized network functions from one Kubernetes cluster to another to support the effective migration of 5G network functions. The need starts from the deployment of a private 5G core network using an open-source project (namely, Magma). Those sets of 5G components demand proper management in telecom networks. In order to increase the resource availability and prioritizing the latency-efficient service at edge-level clusters, there is a need to re-allocate the orc8r component of Magma. Bringing that into reality by proposing a proper mechanism is the principle goal of our work. Which will be further explored for more number of chained microservices accommodating by heuristic or meta-heuristic algorithms in our perspective future work. The key lesson learnt is that by implementing a relatively straightforward strategy involving DNS redirection and proper forwarding, a controller may trigger migration (on the basis of monitoring data provided by Prometheus) of pods or containers across distributed K8S clusters. In terms of performance, the migration time (in the testbed) is about a few tens of milliseconds, due to the creation time of pods. In real networks with large K8S clusters and significant propagation times, the migration time may be

increased upto a few tens of seconds but would remain within acceptable bounds when compared with manual configuration.

Chapter 7

Conclusion

Content

This chapter summarizes the PhD thesis work and key contributions in Section 7.1 along with possible perspectives for the future work in Section 7.2.

7.1 Thesis contributions : a summary

The recent adoption of cloud native technologies by the telecommunication industry is accompanied by the incoming development of Network Functions that are containerized and packaged as light-weighted microservices. In order to efficiently orchestrate Cloud-Native Network Functions (CNFs), thorough migration strategies should be supported to place and migrate the CNFs. Even so, numerous existing state-of-the-art strategies proposed to initially allocate VNFs and allow efficient usage of computing resources while inducing reduced load or latency. Still, the initial placement of services is sometimes not able to continuously meet the Service Level Agreement (SLA), resulting in degradation of communication network performance across time. Also, the real-time applications in the 5G/6G network require to be placed close to the end-user and migrated to follow the mobile user that moves from one position to another.

Aiming to that, in a first step, this thesis work conveyed the importance of network-aware and network-agnostic approaches while placing the chain of microservices in a distributed multi-cloud network aiming to minimize the end-to-end latency. We assumed that services are composed of a chain of microservices that has been de-

ployed in a container and need to be placed on a Cloud-Fog-Edge infrastructure. The chained microservices use required resource capacity and the microservice belonging to the respective service should be placed nearest to the same data center as per their communication affinity and user's location. The key challenge is to co-join the microservices belonging to the same service and ensure the lower communication delay between the user and respective service while guaranteeing the load of the system. To fulfill these demands, we proposed two ILP models - the former allocates the microservices on the same computing node and near to end users by promoting the presence of collocated services near to the end user, thereby lowering the cost of communications ; the latter mapped the microservices in a way to avoid as much as possible long distance communication that tends to minimize the communication delay between the data centers. This further has been solved using proposed heuristic approaches namely - Greedy and advanced Genetic algorithm. The simulation based executed performance tests shows that network-aware placement strategy is comparatively effective in comparison to network agnostic approach in terms of end-to-end latency.

We also introduce a dynamic microservice migration approach to carefully orchestrate the allocation and rearrangement of (micro)services to avoid a largely segmented solution space while services arrive and leave the network. For this purpose, we introduced a novel placement and migration strategy that chooses the microservice(s) to migrate and selects the optimal destination (data center) while considering the impact of the migration on other microservices. The formalized problem on microservices migration tries to solve this ever-demanding migration problem by ensuring the lesser number of microservices are moved while keeping the placement optimal. The proposed three heuristics considerably reduce run time of the migration algorithm. For placing the newly arrived services, two greedy algorithms namely - the Greedy First Fit algorithm and the Greedy Best Fit algorithm have been introduced that also take into account the latest state of the system and continuously analyse the optimality of proposed placement. Likewise, upon the departure of a set of services, a proposed migration algorithm has been executed to continuously improve the placement by migrating or re-allocating the microservices. The proposed solution is user-centric, aiming to reduce the end-to-end latency while considering the resource load balancing. The design rationale also supports the migration of microservices across geo-distributed cloud nodes by performing both vertical (moving microservices from the bottom edge to the top layer of the cloud and vice versa) and horizontal (moving microservices from one node to another in the same layer) migration. A simulation-based evaluation that shows that the migration of services is efficient using our proposed heuristic algorithm and minimizes the latency.

Moreover, as a Proof-Of-Concept (PoC), we implemented a Kubernetes-based solution to demonstrate the live

migration of pods between remote Kubernetes clusters. For this purpose, we deployed an open-source 5G core network including a chain of 5G microservices, then we supported the live migration of a containerized network functions for multi-cluster environment with a controller composed of Monitoring, Analysis, Discovery and Execute components able to trigger the migration based on an analysis of the collected metrics and set of defined rules (such as, rule defining that the resource limit exceeded).

7.2 Perspectives and Future Works

The set of solutions proposed in this thesis evidently enables an efficient placement of containerized microservices by automating the process of migration or re-allocation as per requirement near to the end-user's location to ensure the faster communication exchange. Also, the Kubernetes based cloud-native solution for the open-source 5G core network demonstrates an approachable mechanism to proceed with real-time scenario. Still, we believe that there would be an opportunity to extend this proposed work as discussed in the following section.

7.2.1 ML-based microservices migration

As stated in ML-based VNFs placement related research works (such as [11, 12, 82, 95, 114, 116, 117]), the DRL algorithm has been prominently used. In Reinforcement Learning (RL) algorithm, the Agent receives the Rewards or Penalties at each time interval based on its actions and continues to learn to make decisions based on it. Learning here is based on past experiences where the objective is to train the agent to fulfil a task within an uncertain environment. The rewards are used to quantify the quality of executed actions in respect to goal accomplishment. Moreover, the agent is composed of two components: Policy and Learning algorithm as shown in Figure 7.1. The policy is defined as a function that returns feasible action for a problem. It maps each action and state into probability of taking action in a particular state. The learning algorithm focuses on finding an optimal policy to maximize the expected cumulative long-term reward received during the task. Therefore, based on actions, observations, and rewards, the learning algorithm continues to update the policy parameters. Consequently, inspired from the work [12], where authors proposed heuristically assisted DRL based approach accelerates the learning proposed and improved the performance in comparison to state-of-the-art study; We also believe that, this combination of intelligent learning based approach with our proposed heuristic migration algorithm able to advances the optimality of VNFs/CNFs placement and its migration by calculating an optimal target node.

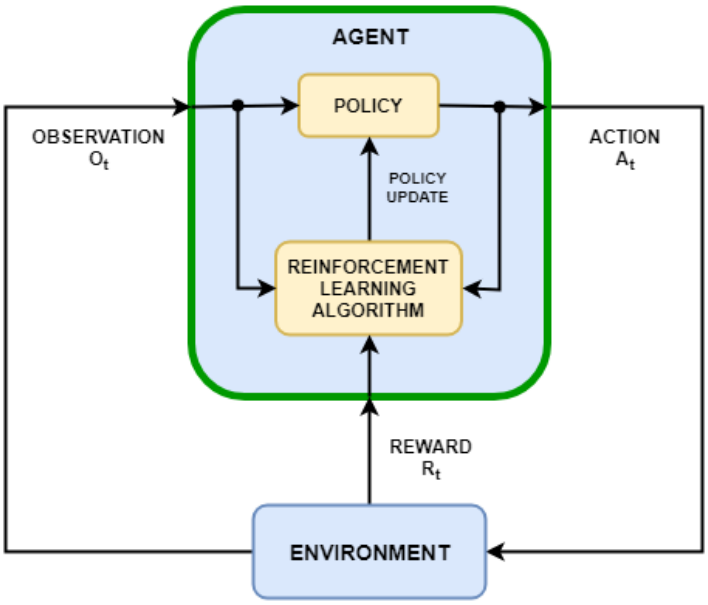


Figure 7.1: Standard DRL algorithm setup. Source [5]

7.2.2 Integration of proposed migration algorithm in implemented test-bed

The foremost interesting direction is to integrate the implemented test-bed with our proposed algorithms. Even though we have tested it for a real 5G core solution, still the step to evaluate and understand the way the algorithm performs when used in a real system is quite important. This allows us to evaluate the performance of an algorithm in a real mechanism.

Secondly, there is a possibility to evaluate the same prototype for other available open-source 5G core networks (such as, free5GC, open5gs and OAI-CN) as they are also based on cloud-native architecture. Currently, many of the Telecom enterprises are putting forth and investing in the production of their own core-network solution to fulfill the need of 5G/6G cellular network. Therefore, this comparison for various contributed solutions and executing simultaneous migration of multiple containers would help to provide an insight view and recognize the behavior of distinct components in real-time.

Bibliography

[1] Ansible. https://www.ansible.com/. Accessed: 06-Dec-2021.

[2] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/.

[3] Intel SGX: Enclave. https://www.intel.com/content/ dam/ develop/ external/ us/en/documents/ overview-of-intel-sgx-enclave-637284.pdf. Accessed: 17-Nov-2021.

[4] Open Container Initiative. https://opencontainers.org/. Accessed: 24-Oct-2021.

[5] Reinforcement Learning Agents, https://fr.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html. Accessed: 12-June-2023.

[6] Arif Ahmed, HamidReza Arkian, Davaadorj Battulga, Ali J Fahs, Mozhdeh Farhadi, Dimitrios Giouroukis, Adrien Gougeon, Felipe Oliveira Gutierrez, Guillaume Pierre, Paulo R Souza Jr, et al. Fog computing applications: Taxonomy and requirements. *arXiv preprint arXiv:1907.11621*, 2019.

[7] Ravindra K Ahuja, James B Orlin, and Ashish Tiwari. A greedy genetic algorithm for the quadratic assignment problem. *Computers & Operations Research*, 27(10):917–934, 2000.

[8] S. AJAGOPALAN, D. WILLIAMS, H. JAMJOOM, and al. A. split/merge: System support for elastic execution in virtual middleboxes. 2013.

[9] Sam Aleyadeh, Abdallah Moubayed, Parisa Heidari, and Abdallah Shami. Optimal container migration/re-instantiation in hybrid computing environments. *IEEE Open Journal of the Communications Society*, 3:15–30, 2022.

[10] NGMN Alliance. Cloud native enabling future telco platforms. 2021.

[11] Jose Jurandir Alves Esteves, Amina Boubendir, Fabice Guillemin, and Pierre Sens. DRL-based slice placement under realistic network load conditions. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 524–526. IEEE, 2021.

[12] Jose Jurandir Alves Esteves, Amina Boubendir, Fabrice Guillemin, and Pierre Sens. A heuristically assisted deep reinforcement learning approach for network slice placement. *arXiv preprint arXiv:2105.06741*, 2021.

[13] Mohamed Azab and Mohamed Eltoweissy. Migrate: Towards a lightweight moving-target defense against cloud side-channels. In *2016 IEEE security and privacy workshops (SPW)*, pages 96–103. IEEE, 2016.

[14] Mohamed Azab, Bassem Mokhtar, Amr S Abed, and Mohamed Eltoweissy. Toward smart moving target defense for linux container resiliency. In *IEEE 41st Conference on Local Computer Networks (LCN)*, pages 619–622. IEEE, 2016.

[15] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33(3):42–52, 2016.

[16] Thad Benjaponpitak, Meatasit Karakate, and Kunwadee Sripanidkulchai. Enabling live migration of containerized applications across clouds. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2529–2538. IEEE, 2020.

[17] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, Lav Gupta, and H Anthony Chan. Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications*, 102:1–16, 2017.

[18] El Houssine Bourhim, Halima Elbiaze, and Mouhamad Dieye. Inter-container communication aware container placement in fog computing. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–6. IEEE, 2019.

[19] Shalini Choudhury, Sumit Maheshwari, Ivan Seskar, and Dipankar Raychaudhuri. Shareon: Shared resource dynamic container migration framework for real-time support in mobile edge clouds. *IEEE Access*, 10:66045–66060, 2022.

[20] Josué Castañeda Cisneros, Sami Yangui, and al. Coordination algorithm for migration of shared vnfs in federated environments. In *IEEE NetSoft*, 2020.

[21] Hugo Gustavo Valin Oliveira da Cunha, Rodrigo Moreira, and Flávio de Oliveira Silva. A comparative study between containerization and full-virtualization of virtualized everything functions in edge computing. In *International Conference on Advanced Information Networking and Applications*, pages 771–782. Springer, 2021.

[22] Jad Darrous, Thomas Lambert, and Shadi Ibrahim. On the importance of container image placement for service provisioning in the edge. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2019.

[23] Nabila Djennane, Rachida Aoudjit, and Samia Bouzefrane. Energy-efficient algorithm for load balancing and vms reassignment in data centers. In *6th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 225–230. IEEE, 2018.

[24] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. Edge computing in iot context: Horizontal and vertical linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–4. IEEE, 2017.

[25] Ivan Farris, Tarik Taleb, Miloud Bagaa, and Hannu Flick. Optimizing service replication for mobile delay-sensitive applications in 5g edge network. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[26] Pawel Gazdzicki, Ioannis Lambadaris, and Ravi R. Mazumdar. Blocking probabilities for large multirate erlang loss systems. *Advances in Applied Probability*, 25(4):997–1009, 1993.

[27] R. Gember-Jacobson, C. Viswanathan, R. Prakash, and al. Opennf: Enabling innovation in network function control. 2014.

[28] Monika Gill and Dinesh Singh. Aco based container placement for caas in fog computing. *Procedia Computer Science*, 167:760–768, 2020.

[29] Keerthana Govindaraj and Alexander Artemenko. Container live migration for latency critical industrial applications on edge computing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 83–90. IEEE, 2018.

[30] Fabrice Guillemin and Luc Le Beller. Analysis of business opportunities for tower companies enabled by virtualization. In *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 163–168, 2022.

[31] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[32] Shaddi Hasan, Amar Padmanabhan, Bruce Davie, and al. Building flexible, low-cost wireless access networks with magma. *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023.

[33] Hassan Hawilo, Manar Jammal, and Abdallah Shami. Orchestrating network function virtualization platform: Migration or re-instantiation? In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pages 1–6. IEEE, 2017.

[34] Hassan Hawilo, Manar Jammal, and Abdallah Shami. Network function virtualization-aware orchestrator for service function chaining placement in the cloud. *IEEE Journal on Selected Areas in Communications*, 37(3):643–655, 2019.

[35] Ali Hmaity, Marco Savi, Francesco Musumeci, Massimo Tornatore, and Achille Pattavina. Virtual network function placement for resilient service chain provisioning. In *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 245–252. IEEE, 2016.

[36] Yang Hu, Mingcong Song, and Tao Li. Towards full containerization in containerized network function virtualization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[37] Rui Huang, Hongqi Zhang, Yi Liu, and Shie Zhou. Relocate: a container based moving target defense approach. In *7th International Conference on Computer Engineering and Networks*, page 8, 2017.

[38] Mohamed K Hussein, Mohamed H Mousa, and Mohamed A Alqarni. A placement architecture for a container as a service (caas) in a cloud environment. *Journal of Cloud Computing*, 8(1):1–15, 2019.

[39] Muthurajan Jayakumar (M Jay). Why use containers and cloud-native functions anyway? In *White Paper Communications Service Providers Cloud-Native Network Functions*. Intel.

[40] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. Stateful container migration in geo-distributed environments. In *CloudCom 2020-12th IEEE International Conference on Cloud Computing Technology and Science*, 2020.

[41] M. KABLAN, A. ALSUDAIS, E. KELLER, and al. Stateless network functions: Breaking the tight coupling of state and processing. 2017.

[42] Kiranpreet Kaur, Fabrice Guillemin, Veronica Quintuna Rodriguez, and Francoise Sailhan. Latency and network aware placement for cloud-native 5g/6g services. In *Consumer Communications & Networking Conference (CCNC)*, 2022.

[43] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Container placement and migration strategies for cloud, fog, and edge data centers: A survey. *International Journal of Network Management*, 32(6):e2212, 2022.

[44] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. A microservice migration approach to controlling latency in 5G/6G network, 2023. Accepted for publication in IEEE International Conference on Communications (ICC) 2023.

[45] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Dynamic migration of microservices for end-to-end latency control in 5g/6g networks, 04 2023. Submitted for publication in Journal of Network and Systems Management 2023.

[46] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Live migration of containerized microservices between remote Kubernetes Clusters. Accepted for publication in IEEE INFOCOM workshop 2023, March 2023.

[47] F. P. Kelly. Loss networks. *The Annals of Applied Probability*, 1(3):319–378, 1991.

[48] Hamed Rahmani Khezri, Puria Azadi Moghadam, Mohammad Karimzadeh Farshbafan, Vahid Shah-Mansouri, Hamed Kebriaei, and Dusit Niyato. Deep reinforcement learning for dynamic reliability aware nfv-based service provisioning. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[49] Mohammad Ali Khoshkholghi, Michel Gokan Khan, Kyoomars Alizadeh Noghani, and al. Service function chain placement for joint cost and latency optimization. *Mobile Networks and Applications*, pages 1–15, 2020.

[50] Petar Kochovski, Rizos Sakellariou, Marko Bajec, Pavel Drobintsev, and Vlado Stankovski. An architecture and stochastic method for database container placement in the edge-fog-cloud continuum. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 396–405. IEEE, 2019.

[51] Sai Venkat Naresh Kotikalapudi. Comparing live migration between linux containers and kernel virtual machine: investigation study in terms of parameters, 2017.

[52] Kubernetes. https://kubernetes.io. Accessed: 05-Dec-2022.

[53] Sameer G Kulkarni, Guyue Liu, KK Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. Reinforce: Achieving efficient failure resiliency for network function virtualization based services, 2018.

[54] Abdelquoddouss Laghrissi and Tarik Taleb. A survey on the placement of virtual resources and virtual network functions. *IEEE Communications Surveys & Tutorials*, 21(2):1409–1434, 2018.

[55] Aris Leivadeas, George Kesidis, Mohamed Ibnkahla, and al. Vnf placement optimization at the edge and cloud. *Future Internet*, 11(3), 2019.

[56] Defang Li, Peilin Hong, Kaiping Xue, et al. Virtual network function placement considering resource optimization and sfc requests in cloud datacenter. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1664–1677, 2018.

[57] Junling Li, Weisen Shi, Huaqing Wu, Shan Zhang, and Xuemin Shen. Cost-aware dynamic sfc mapping and scheduling in sdn/nfv-enabled space-air-ground integrated networks for internet of vehicles. *IEEE Internet of Things Journal*, 2021.

[58] Hongliang Liang, Qiong Zhang, Mingyu Li, and Jianqiang Li. Toward migration of sgx-enabled containers. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2019.

[59] Yeonjoo Lim and Jong-Hyouk Lee. Container-based service relocation for beyond 5g networks. *Journal of Internet Technology*, 23(4):911–918, 2022.

[60] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 98–106. IEEE, 2015.

[61] Liang Lv, Yuchao Zhang, Yusen Li, Ke Xu, Dan Wang, Wendong Wang, Minghui Li, Xuan Cao, and Qingqing Liang. Communication-aware container placement and reassignment in large-scale internet data centers. *IEEE Journal on Selected Areas in Communications*, 37(3):540–555, 2019.

[62] L. Ma, S. Yi, and Q. Li. In *Second ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.

[63] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing*, 18(9):2020–2033, 2018.

[64] Andrew Machen, Shiqiang Wang, Kin K Leung, Bong Jun Ko, and Theodoros Salonidis. Migrating running applications across mobile edge clouds: poster. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 435–436, 2016.

[65] Magma Orc8r deployment on Minikube, https://magma.github.io/magma/docs/orc8r/dev_minikube. Accessed: 09-Dec-2022.

[66] Sumit Maheshwari, Shalini Choudhury, Ivan Seskar, and Dipankar Raychaudhuri. Traffic-aware dynamic container migration for real-time support in mobile edge clouds. In *2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1–6. IEEE, 2018.

[67] Dimitrios Michael Manias, Manar Jammal, Hassan Hawilo, Abdallah Shami, Parisa Heidari, Adel Larabi, and Richard Brunner. Machine learning for performance-aware virtual network function placement. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[68] Zeynep Mavuş. Secure model for efficient live migration of containers. Master's thesis, 2019.

[69] Minikube. https://minikube.sigs.k8s.io/docs/. Accessed: 09-Dec-2022.

[70] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[71] Naoki Mizusawa, Kenji Nakazima, and Saneyasu Yamaguchi. Performance evaluation of file operations on overlayfs. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 597–599. IEEE, 2017.

[72] Amina Mseddi, Wael Jaafar, Halima Elbiaze, and Wessam Ajib. Joint container placement and task provisioning in dynamic fog computing. *IEEE Internet of Things Journal*, 6(6):10028–10040, 2019.

[73] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142. IEEE, 2017.

[74] Omogbai Oleghe. Container placement and migration in edge computing: concept and scheduling models. *IEEE Access*, 9:68028–68043, 2021.

[75] D. Ongaro, S. M. Rumble, and R. Stutsmanand al. Fast crash recovery in ramcloud. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[76] S. Palkar, C. LAN, S. Han, and al. E2: A framework for nfv applications. 2015.

[77] C.-G. Park and D.-H. Han. Comparisons of loss formulas for a circuit group with overflow traffic. *Journal of applied mathematics and informatics*, 30(1-2):135–145, 2012.

[78] Jianing Pei, Peilin Hong, Miao Pan, Jiangqing Liu, and Jingsong Zhou. Optimal vnf placement via deep reinforcement learning in sdn/nfv-enabled networks. *IEEE Journal on Selected Areas in Communications*, 38(2):263–278, 2019.

[79] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. Migros: Transparent operating systems live migration support for containerised rdma-applications. *arXiv preprint arXiv:2009.06988*, 2020.

[80] U Pongsakorn, Yasuhiro Watashiba, Kohei Ichikawa, Susumu Date, Hajimu Iida, and al. Container rebalancing: Towards proactive linux containers placement optimization in a data center. In *IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 788–795, 2017.

[81] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container migration in the fog: A performance evaluation. *Sensors*, 19(7):1488, 2019.

[82] P. T. A. Quang, A. Bradai, K. D. Singh, and Y. Hadjadj-Aoul. Multi-domain non-cooperative VNF-FG embedding: A deep reinforcement learning approach. In *Proc. IEEE INFOCOM 2019 - IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, pages 886–891, 2019.

[83] Pham Tran Anh Quang, Yassine Hadjadj-Aoul, and Abdelkader Outtagarts. A deep reinforcement learning approach for vnf forwarding graph embedding. *IEEE Transactions on Network and Service Management*, 16(4):1318–1331, 2019.

[84] Shunmugapriya Ramanathan, Koteswararao Kondepu, Tianliang Zhang, Behzad Mirkhanzadeh, Miguel Razo, Marco Tacca, Luca Valcarenghi, and Andrea Fumagalli. A comprehensive study of virtual machine and container based core network components migration in openroadm sdn-enabled network. *arXiv preprint arXiv:2108.12509*, 2021.

[85] A. Rkhami, Y. Hadjadj-Aoul, and A. Outtagarts. Learn to improve: A novel deep reinforcement learning approach for beyond 5G network slicing. In *Proc. 2021 IEEE 18th Annu. Consum. Commun. Netw. Conf. (CCNC)*, pages 1–6, 2021.

[86] Adalberto R Sampaio, Harshavardhan Kadiyala, Bo Hu, John Steinbacher, Tony Erwin, Nelson Rosa, Ivan Beschastnikh, and Julia Rubin. Supporting microservice evolution. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 539–543. IEEE, 2017.

[87] Ioannis Sarrigiannis, Elli Kartsakli, Kostas Ramantas, Angelos Antonopoulos, and Christos Verikoukis. Application and network vnf migration in a mec-enabled 5g architecture. In *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6. IEEE, 2018.

[88] Vincenzo Sciancalepore, Faqir Zarrar Yousaf, and Xavier Costa-Perez. z-torch: An automated nfv orchestration and monitoring solution. *IEEE Transactions on Network and Service Management*, 15(4):1292–1306, 2018.

[89] Farah Slim, Fabrice Guillemin, Annie Gravey, and Yassine Hadjadj-Aoul. Towards a dynamic adaptive placement of virtual network functions under onap. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 210–215, 2017.

[90] Quanying Sun, Ping Lu, Wei Lu, and Zuqing Zhu. Forecast-assisted nfv service chain deployment based on affiliation-aware vnf placement. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[91] Hong Tang, Danny Zhou, and Duan Chen. Dynamic network function instance scaling based on traffic forecasting and vnf placement in operator data centers. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):530–543, 2018.

[92] Zhiqing Tang, Xiaojie Zhou, Fuming Zhang, Weijia Jia, and Wei Zhao. Migration modeling and learning algorithms for containers in fog computing. *IEEE Transactions on Services Computing*, 12(5):712–725, 2018.

[93] Sanaz Tavakoli-Someh and Mohammad Hossein Rezvani. Utilization-aware virtual network function placement using nsga-ii evolutionary computing. In *IEEE Conference on Knowledge Based Engineering and Innovation*, 2019.

[94] Rob van der Meulen. What edge computing means for infrastructure and operations leaders. *Gartner, online, available, www. gartner. com*, 2017.

[95] Haozhe Wang, Yulei Wu, Geyong Min, Jie Xu, and Pengcheng Tang. Data-driven dynamic resource scheduling for network slicing: A deep reinforcement learning approach. *Inf. Sci.*, 498:106–116, Sep. 2019.

[96] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. A survey on service migration in mobile edge computing. *IEEE Access*, 6:23511–23528, 2018.

[97] Cloudify [Official site]. https://cloudify.co/. Accessed: 17-Nov-2021.

[98] KubeVirt [Official site]. https://kubevirt.io/. Accessed: 17-Nov-2021.

[99] KubeVirt GitHub. https://github.com/kubevirt/kubevirt. Accessed: 17-Nov-2021.

[100] Podmigration-operator https://github.com/schrej/podmigration-operator. Accessed: 17-Nov-2021.

[101] Podmigration-operator [Extended version]. https://github.com/ssu-dcn/podmigration-operator. Accessed: 17-Nov-2021.

[102] Velero. https://velero.io/. Accessed: 17-Nov-2021.

[103] Velero GitHub. https://github.com/vmware-tanzu/velero. Accessed: 17-Nov-2021.

[104] Restic. https://velero.io/docs/v1.7/restic/. Accessed: 17-Nov-2021.

[105] Migrating applications between Kubernetes clusters, https://medium.com/google-cloud/migrating-applications-between-kubernetes-clusters-8455cf1bfccd. Accessed: 05-Dec-2022.

[106] Edge Multi-Cluster Orchestrator (EMCO). https://smart-edge-open.github.io/ido-specs/doc/building-blocks/emco/smartedge-open-emco/.

[107] emco-base [Gitlab]. https://gitlab.com/project-emco/core/emco-base.

[108] Grafana,https://grafana.com/. Accessed: 13-Jan-2023.

[109] Kubernetes. https://kubernetes.io/.

[110] kube-prometheus-stack,https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack. Accessed: 13-Jan-2023.

[111] Prometheus, https://prometheus.io/. Accessed: 13-Jan-2023.

[112] Traefik, https://traefik.io/. Accessed: 05-Dec-2022.

[113] S. Woo, J. Sherry, S. Han, and al. Elastic scaling of stateful network functions. 2018.

[114] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang. NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning. In *Proc. 2019 IEEE/ACM 27th Int. Symp. Qual. Service (IWQoS)*, pages 1–10, 2019.

[115] Bo Xu, Song Wu, Jiang Xiao, Hai Jin, Yingxi Zhang, Guoqiang Shi, Tingyu Lin, Jia Rao, Li Yi, and Jizhong Jiang. Sledge: Towards efficient live migration of docker containers. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 321–328. IEEE, 2020.

[116] Zhongxia Yan, Jingguo Ge, Yulei Wu, Liangxiong Li, and Tong Li. Automatic virtual network embedding: A deep reinforcement learning approach with graph convolutional networks. *IEEE Journal on Selected Areas in Communications*, 38(6):1040–1057, 2020.

[117] Haipeng Yao, Xu Chen, Maozhen Li, Peiying Zhang, and Luyao Wang. A novel reinforcement learning algorithm for virtual network embedding. *Neurocomputing*, 284:1–9, Apr. 2018.

[118] Bowu Zhang, Jinho Hwang, and Timothy Wood. Toward online virtual network function placement in software defined networks. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2016.

[119] Rong Zhang, Yaxing Chen, Bo Dong, Feng Tian, and Qinghua Zheng. A genetic algorithm-based energy-efficient container placement strategy in caas. *IEEE Access*, 7:121360–121373, 2019.

[120] Rong Zhang, A-min Zhong, Bo Dong, Feng Tian, and Rui Li. Container-vm-pm architecture: A novel architecture for docker container placement. In *International Conference on Cloud Computing*, pages 128–140. Springer, 2018.

[121] Ruiting Zhou, Zongpeng Li, and Chuan Wu. An efficient online placement scheme for cloud container clusters. *IEEE Journal on Selected Areas in Communications*, 37(5):1046–1058, 2019.