Évaluation de déploiements de microservices conteneurisés pour la diffusion web de vidéos 360° dans une infrastructure périphérique-nuagique

par

Bogdan RUSU

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE DES TECHNOLOGIES DE L'INFORMATION
M. Sc. A.

MONTRÉAL, LE 7 NOVEMBRE 2025

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Bogdan RUSU, 2025

Cette licence Creative Commons signifie qu'il est permis de diffuser, d'imprimer ou de sauvegarder sur un autre support une partie ou la totalité de cette oeuvre à condition de mentionner l'auteur, que ces utilisations soient faites à des fins non commerciales et que le contenu de l'oeuvre n'ait pas été modifié.

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Aris Leivadeas, directeur de mémoire
Département de génie logiciel et des TI, École de technologie supérieure

M. Stéphane Coulombe, codirecteur
Département de génie logiciel et des TI, École de technologie supérieure

M. Carlos Vázquez, président du jury
Département de génie logiciel et des TI, École de technologie supérieure

M. Julien Gascon-Samson, membre du jury
Département de génie logiciel et des TI, École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 4 NOVEMBRE 2025

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur, Aris, et mon codirecteur, Stéphane, pour m'avoir offert la chance de réaliser une maîtrise avec mémoire. J'ai toujours voulu découvrir le monde de la recherche, et grâce à cette opportunité, j'ai pu apprendre, expérimenter et me dépasser, le tout sans pression ni peur de l'échec. Je leur en suis très reconnaissant.

J'aimerais aussi remercier ma famille pour leur soutien, leur compréhension et leur patience envers mes objectifs. Sans eux, je n'aurais pas écrit ce mémoire et, de ce fait, je n'aurais pas pu me dépasser, tant sur le plan académique que personnel.

En dernier lieu, mais non moins important, j'aimerais remercier le Conseil de recherches en sciences naturelles et en génie du Canada, Mitacs de même que Summit Tech Multimedia pour le financement dont j'ai bénéficié dans le cadre du programme Alliance-Mitacs Accélération, sans lequel la réalisation de ce projet n'aurait pas été possible.

Évaluation de déploiements de microservices conteneurisés pour la diffusion web de vidéos 360° dans une infrastructure périphérique-nuagique

Bogdan RUSU

RÉSUMÉ

Une vidéo 360° est un média qui offre une immersion panoramique, mais qui nécessite une haute résolution et une forte puissance de calcul, surtout en contexte multi-utilisateurs. Pour améliorer le traitement et la transmission en quasi temps réel, des techniques comme le découpage en tuiles sont utilisées. Ces tuiles, encodées à différentes qualités selon le regard de l'utilisateur, permettent une réduction de la bande passante en ne transmettant que les zones pertinentes.

Le traitement, souvent séquentiel, peut être accéléré en regroupant les étapes sur une seule machine puissante. Cela génère des architectures monolithiques qui causent un gaspillage de ressources. La conteneurisation peut aider à ajuster les ressources selon les besoins et permettre une meilleure scalabilité.

Chaque étape de traitement, souvent indépendante, peut être transformée en microservice, pour ensuite être déployée sous forme de conteneur. Ces conteneurs peuvent être répartis géographiquement pour réduire la latence en rapprochant le traitement auprès des utilisateurs, selon le principe de périphérie.

Ce mémoire propose une chaîne de traitement adaptée aux vidéos 360°, décomposée en microservices conteneurisés. Différents déploiements inspirés de la littérature sont évalués en termes de consommation de ressources et de performance en environnement multi-utilisateurs. Dans nos conditions de test, les résultats expérimentaux indiquent qu'en termes de délais de livraison, un déploiement monolithique semble offrir les délais les plus courts s'il se trouve proche du client. Cependant, cela se fait au prix d'une utilisation accrue des ressources, contrairement à des déploiements distribués. En effet, des déploiements de microservices distribués peuvent transmettre des vidéos 360° avec des délais très raisonnables, tout en utilisant moins de ressources et en étant répartis sur plusieurs machines géographiquement éloignées.

Mots-clés: conteneur, microservices, périphérie, virtualisation, vidéo 360°

Evaluation of Containerized Microservice Architectures for 360° Video Web Delivery Within an Edge-cloud Infrastructure

Bogdan RUSU

ABSTRACT

A 360° video is a medium that offers panoramic immersion but requires high resolution and significant computational power, especially in multi-user contexts. To improve processing and near real-time transmission, techniques such as tiling are used. These tiles, encoded at different qualities depending on the user's gaze, allow for bandwidth reduction by transmitting only the relevant areas.

The pipeline, often sequential, is usually accelerated by grouping steps on a single powerful machine. This generates monolithic architectures that lead to resource waste. Containerization can help adjust resources according to needs and enable better scalability.

Each processing step, often independent, can be transformed into a microservice, and then deployed as a container. These containers can be distributed geographically to reduce latency by bringing processing closer to users, following the edge computing principle.

This thesis proposes a processing pipeline adapted to 360° videos, decomposed into containerized microservices. Various deployment strategies inspired by the literature are evaluated in terms of resource consumption and performance in multi-user environments. Under our test conditions, experimental results indicate that, in terms of delivery time, a monolithic deployment appears to offer the shortest delays when located close to the client. However, this comes at the cost of increased resource usage, unlike more distributed deployments. Indeed, distributed microservice deployments can deliver 360° video within very reasonable delay while consuming fewer resources and being spread across geographically distant machines.

Keywords: container, microservices, edge, virtualisation, 360° video

TABLE DES MATIÈRES

Page

INTRODUCTION 1
0.1 Organisation du mémoire 3

CHAPITRE 1 REVUE DE LITTÉRATURE 5
1.1 Chaîne de traitement de la vidéo 360° 5
1.1.1 La vidéo 360° 5
1.1.2 Raccordement et projections 6
1.1.2.1 Projection équirectangulaire 8
1.1.3 Conversion de coordonnées 8
1.1.4 Codage vidéo et codecs pour de la vidéo 360° 10
1.2 Diffusion vidéo en continu 12
1.2.1 Fragment vidéo 13
1.2.2 Diffusion par tuiles adaptative 13
1.2.3 MPEG-DASH 16
1.2.4 MPEG-DASH SRD 18
1.2.5 GPAC 18
1.3 Organisation des tâches et des microservices pour la diffusion en continu de la vidéo 360° dans un réseau périphérique 18
1.3.1 Les entités impliquées dans un réseau en périphérie et leurs rôles 19
1.3.2 Diffusion et traitement à travers des microservices conteneurisés dans un réseau périphérique 21
1.4 Discussion 24
1.5 Conclusion 26

CHAPITRE 2 CHAÎNE DE TRAITEMENT VIDÉO 360° ET ARCHITECTURES PROPOSÉES 27
2.1 Chaîne de traitement vidéo 360° proposée 27
2.1.1 Sélection de la qualité des tuiles selon la région d'intérêt de l'utilisateur .. 28
2.1.2 Récupération des tuiles selon les qualités définies 29
2.1.3 Raccordement des tuiles pour la création du segment 29
2.1.4 Réécriture de segment dans un flux binaire compatible 30
2.1.5 Lecteur web interactif de vidéo 360° 31
2.1.6 Résumé et contributions 31
2.2 Déploiements de microservices proposés 32
2.2.1 Déploiement Monolithique 33
2.2.1.1 Tâches du microservice 33
2.2.1.2 Fonctionnement du déploiement 33
2.2.1.3 Avantages et inconvénients du déploiement 35
2.2.2 Déploiement Proximité 35
2.2.2.1 Tâches des microservices 35

2.2.2.2 Fonctionnement du déploiement 36
2.2.2.3 Avantages et inconvénients du déploiement 38
2.2.3 Déploiement Distribué 38
2.2.3.1 Tâches des microservices 39
2.2.3.2 Fonctionnement du déploiement 40
2.2.3.3 Avantages et inconvénients du déploiement 40
2.2.4 Déploiement Granularisé 41
2.2.4.1 Tâches des microservices 42
2.2.4.2 Fonctionnement du déploiement 43
2.2.4.3 Avantages et inconvénients du déploiement 44
2.2.5 Comparaison qualitative entre les déploiements 45
2.3 Gestion du nombre d'utilisateur 46
2.3.1 Types d'instance et leur relation 47
2.3.1.1 Conteneur 47
2.3.1.2 Service 47
2.3.1.3 Travailleur web 47
2.3.1.4 Relation entre les instance 48
2.3.1.5 Duplication des instances 48
2.3.2 Utilisation des instances 49

CHAPITRE 3 RÉSULTATS EXPÉRIMENTAUX 53
3.1 Configurations et caractéristiques vidéo des tuiles 53
3.1.1 Étapes pour la création des tuiles 54
3.2 Acquisition des données et méthodes de comparaison 56
3.2.1 Acquisition des données sur nos déploiements 56
3.2.1.1 cAdvisor 57
3.2.1.2 Jaeger 57
3.2.1.3 HTTP Archive Format 59
3.2.1.4 Sauvegarde des résultats 59
3.2.1.5 Extraction des résultats 60
3.2.2 Exécution des tests 62
3.3 Matériel et configuration 62
3.3.1 Configuration des machines et du réseau 62
3.3.1.1 L'impact du matériel sur FFmpeg 64
3.3.2 Orchestrations des répliques de microservices 66
3.4 Résultats et discussions 66
3.4.1 Impact du nombre d'instances 67
3.4.1.1 Impact du type d'instance 69
3.4.2 Comparaison quantitative entre les déploiements 72
3.4.2.1 Impact sur le matériel 76
3.4.2.2 Les délais de livraison plus en détail 77
3.4.2.3 Variabilité des valeurs recueillies 80

3.4.2.4 Retour sur l'analyse qualitative et les résultats des déploiements 84
3.5 Synthèse des résultats 85

CONCLUSION ET RECOMMANDATIONS 87
4.1 Limites de la recherche 88
4.2 Travaux futurs 88

ANNEXE I COMMANDE DE RACCORDEMENT DES TUILES FFMPEG 91

ANNEXE II COMMANDE FFMPEG POUR LA DIVISION EN TUILES 93

ANNEXE III COMMANDE DE CHANGEMENT DE DÉBIT AVEC FFMPEG 95

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES 97

LISTE DES TABLEAUX

Page

Tableau 1.1 Comparaison entre H.264 et H.265 11

Tableau 1.2 Comparaison des travaux de la littérature avec notre recherche 25

Tableau 2.1 Tableau récapitulatif des déploiements 45

Tableau 2.2 Nombre de conteneurs déployés pour chaque microservice sur chaque serveur 51

Tableau 3.1 Caractéristiques de la vidéo et des segments 54

Tableau 3.2 Caractéristiques des machines utilisées 63

Tableau 3.3 Utilisation des ressources et temps d'exécution pour un segment 2s (1 client - déploiement Granularisé) 64

LISTE DES FIGURES

Page

Figure 1.1 Caméras 360° Insta Pro 2, vue avant et arrière 6

Figure 1.2 Plan des tuiles d'une projection équirectangulaire Tirée de Wu (2021) ... 7

Figure 1.3 Plan des tuiles d'une projection cubique Tirée de Wu (2021) 7

Figure 1.4 Champ de vision de l'utilisateur dans un espace sphérique Tirée de Hosseini & Swaminathan (2016) 8

Figure 1.5 Coordonnées θ et ϕ dans un espace sphérique Tirée de Ye, Alshina & Boyce (2018) 9

Figure 1.6 Vue équirectangulaire tuilées 14

Figure 1.7 Techniques de diffusion adaptative par tuiles Tirée de Brochu (2024) 15

Figure 1.8 Architecture classique de MPEG-DASH Tirée de Sodagar (2011) 17

Figure 1.9 Structure d'un fichier MPD Tirée de Sodagar (2011) 17

Figure 1.10 Architecture générale d'un réseau en périphérie 19

Figure 1.11 Architecture MCF, MIF, MDF Tirée de Shen, Dai, Moustafa & Zhai (2021) 23

Figure 1.12 Tâches réparties en microservices Tirée de Santos *et al.* (2021) 23

Figure 1.13 Architecture de microservices dans une infrastructure de brouillard Tirée de Santos *et al.* (2021) 24

Figure 2.1 Interactions entre les tâches de la chaîne de traitement 28

Figure 2.2 Déploiement de l'architecture Monolithique 34

Figure 2.3 Déploiement de l'architecture Proximité 37

Figure 2.4 Déploiement de l'architecture Distribué 41

Figure 2.5 Déploiement de l'architecture Granularisé 43

Figure 2.6 Relation hiérarchique entre les instances 48

Figure 2.7 Duplication de conteneurs 49

Figure 2.8 Duplication de travailleurs web 49

Figure 2.9 Duplication de conteneurs et travailleurs web 50

Figure 3.1 Rendu final du raccordement de tuiles multi-qualité dans un scénario où l'utilisateur regarde au centre de la vidéo 55

Figure 3.2 Exemple d'un événement avec des sous-événements dans Jaeger Tirée de Shkuro (2021) 58

Figure 3.3 Temps d'exécution pour le raccordement des tuiles avec FFmpeg selon différents matériels 65

Figure 3.4 Passerelle API pour les conteneurs d'un microservice 67

Figure 3.5 Temps moyen de livraison d'un segment selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité . 69

Figure 3.6 Nombre moyen de CPUs utilisés pendant la diffusion selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité 70

Figure 3.7 Mémoire RAM moyenne utilisée pendant la diffusion selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité 71

Figure 3.8 Temps moyen de livraison d'un segment en fonction du nombre de clients et du déploiement 73

Figure 3.9 Nombre moyen de CPUs utilisés pendant le visionnement en fonction du nombre de clients et du déploiement 74

Figure 3.10 Mémoire RAM moyenne utilisée pendant le visionnement en fonction du nombre de clients et du déploiement 75

Figure 3.11 Temps d'attente et réception moyen par nombre de clients - déploiement Proximité 78

Figure 3.12 Temps d'attente et réception moyen par nombre de clients - déploiement Distribué 79

Figure 3.13 Temps d'attente et réception moyen par nombre de clients - déploiement Granularisé 79

Figure 3.14 Temps d'attente et réception moyen par nombre de clients - déploiement Monolithique 80

Figure 3.15 Variabilité du nombre de cœurs CPU utilisés par les déploiements (10 essais par nombre de clients) 81

Figure 3.16 Variabilité de la mémoire RAM utilisée par les déploiements (10 essais par nombre de clients) 82

Figure 3.17 Variabilité des délais de livraison des déploiements (10 essais par nombre de clients) 83

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Application Programming Interface
AVC	Advanced Video Coding
CPU	Central Processing Unit
CFS	Completely Fair Scheduler
DASH	Dynamic Adaptive Streaming over HTTP
DDR4	Double Data Rate 4
DIMM	Dual Inline Memory Module
GPU	Graphics Processing Unit
HAR	HTTP Archive Format
HEVC	High Efficiency Video Coding
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
kbps	kilobits per second
LPDDR3	Low Power Double Data Rate 3
MCF	Media Control Function
MCTS	Motion-Constrained Tile Sets
MDF	Media Distribution Function
MIF	Media Ingestion Function
MIME	Multipurpose Internet Mail Extension
MP4	MPEG-4 Part 14
MPD	Media Presentation Description
MPEG	Moving Picture Experts Group
RAM	Random-access Memory
SRD	Spatial Representation Description
vCPU	Virtual Central Processing Unit
VR	Virtual Reality

LISTE DES SYMBOLES ET UNITÉS DE MESURE

°	Degré
s	Secondes
ϕ	Angle de projection sur les axes X et Y - Colatitude
θ	Angle de projection sur les axes Z et Y - Longitude
W	Largeur en pixel d'une image
H	Hauteur en pixel d'une image
m	Position horizontale d'un point
n	Position verticale d'un point
u	Coordonné horizontale normalisé
v	Coordonné verticale normalisé
MHz	Megahertz
GHz	Gigahertz
GB	Gigabyte

INTRODUCTION

Contexte

Les vidéos 360° deviennent de plus en plus accessibles et utilisées. Ce type de média permet à l'utilisateur de s'immerger dans le contenu, rendant ainsi l'expérience plus agréable et mémorable. Cependant, pour offrir une telle expérience, la diffusion en continu doit s'effectuer dans des délais restreints. Plusieurs méthodes sont donc employées pour réduire le temps de traitement et la taille des données diffusées.

Le traitement typique consiste à découper la vidéo en segments de courte durée, eux-mêmes divisés en tuiles. Une tuile représente une section spatiale de la vidéo, mais avec une résolution réduite. Chaque tuile est ensuite encodée à différents débits. En fonction du point de vue de l'utilisateur, la qualité des tuiles est sélectionnée, puis celles-ci sont diffusées et affichées dans sa fenêtre de visualisation. Plusieurs approches existent pour optimiser l'utilisation des tuiles, permettant ainsi d'économiser de la bande passante en adaptant la taille des segments transmis.

Problématique

L'ensemble des étapes décrites, y compris la transmission du segment, doit être exécuté dans un temps restreint. Ce délai est généralement inférieur à la durée du segment lui-même. Pour y parvenir rapidement, des machines dotées de ressources importantes sont nécessaires, surtout lorsque les requêtes de plusieurs utilisateurs sont traitées simultanément. Le traitement associé à de multiples clients requiert donc un déploiement sur plusieurs machines afin de disposer de suffisamment de ressources. Ce besoin provient toutefois d'une mauvaise optimisation de leur utilisation.

Pour réduire le nombre de machines nécessaires, une meilleure répartition des tâches de traitement peut aider. La virtualisation, notamment la conteneurisation, est une bonne solution

à ce problème. Elle permet d'isoler les étapes de traitement pour un segment et d'allouer des ressources spécifiques à chacune, tout en garantissant une bonne expérience utilisateur. Cette cohésion des tâches peut être sous forme de microservices. À leur tour, les microservices peuvent être déployés sur différentes machines adaptées à leurs objectifs, permettant ainsi d'optimiser l'utilisation des ressources.

Cependant, il est difficile d'estimer les ressources requises, leur quantité, ainsi que l'impact qu'un tel déploiement peut engendrer sur celles-ci. Cette difficulté vient de l'absence d'une méthodologie standardisée pour l'architecture en microservices conteneurisés, du nombre de répliques, ainsi que de l'hétérogénéité des infrastructures matérielles utilisées.

Objectifs et contributions

L'objectif principal de ce mémoire est d'évaluer l'impact de déploiements en microservices conteneurisés pour la diffusion de vidéos 360° dans une infrastructure périphérique-nuagique. Plus précisément, les objectifs sont :

- Proposer des architectures de microservices pour le traitement de vidéos 360°, déployables dans une infrastructure périphérique-nuagique ;
- Comparer ces architectures en termes d'utilisation des ressources et des délais de livraisons.

Les contributions de ce travail sont :

- La définition d'une chaîne de traitement pour les vidéos 360° ;
- Création de déploiements de microservices pour la diffusion de vidéos 360° dans un environnement web ;
- Une analyse de l'utilisation des ressources et des délais induits par ces déploiements ;
- La mise en place d'une solution ouverte pour la lecture et le traitement de vidéos 360° pour le web.

Les contributions de ce travail de recherche ont fait l'objet d'un article soumis à la conférence *IEEE Conference on Network Function Virtualization and Software Defined Networks*, intitulé : « Evaluation of Containerized Microservice Architectures for 360° Video Web Delivery Within an Edge-Cloud Infrastructure », rédigé par Aris, Stéphane et moi-même.

0.1 Organisation du mémoire

Ce mémoire est structuré comme suit. Le premier chapitre présente une revue de la littérature en lien avec le projet de recherche. Nous détaillons les méthodes de diffusion adaptative et examinons les travaux similaires à notre travail de recherche. Le deuxième chapitre décrit la chaîne de traitement utilisée et son fonctionnement, ainsi que les déploiements évalués. Enfin, le troisième chapitre présente les résultats expérimentaux. Il décrit la méthodologie d'acquisition des données, détaille la vidéo 360° utilisée avec ses caractéristiques, et analyse les résultats obtenus. Finalement, nous présentons nos conclusions.

CHAPITRE 1

REVUE DE LITTÉRATURE

Ce chapitre a pour but de présenter les technologies de base et l'état de l'art de la littérature sur trois aspects en lien avec la vidéo 360° : chaîne de traitement d'une vidéo 360°, la diffusion vidéo en continu, et la diffusion en continu de la vidéo dans un réseau en périphérie. Ensuite, nous terminerons par une courte discussion afin d'identifier les points manquants dans les travaux de la littérature.

1.1 Chaîne de traitement de la vidéo 360°

Pour obtenir une vidéo 360° interactive, le média doit passer par plusieurs étapes de traitement. Cela est dû au fait que le contenu d'une vidéo 360° provient de plusieurs caméras, générant ainsi plusieurs petites vidéos qu'il faut raccorder. Plusieurs projections peuvent être utilisées lors de l'assemblage, ce qui implique qu'il existe différentes manières de calculer le point de vue de l'utilisateur.

Dans cette section, nous présenterons plus en détail ce qu'est une vidéo 360°, nous expliquerons les différentes tâches de cette chaîne de traitement, puis nous décrirons les principales projections issues de l'état de l'art.

1.1.1 La vidéo 360°

Une vidéo 360°, également appelée vidéo immersive (Quest, 2022), est une vidéo de haute résolution qui permet à l'utilisateur de se sentir entouré par l'image. Contrairement à une vidéo classique, elle offre du contenu dans toutes les directions, peu importe où l'utilisateur tourne la tête.

Cette vidéo est réalisée à l'aide de caméras spéciales, appelées caméras 360°. Il existe deux types de caméras 360° : convergentes et divergentes. Les caméras convergentes sont orientées vers un sujet, c'est-à-dire que toutes les lentilles de la caméra pointent vers un même objet,

mais sous des angles différents. En revanche, dans le cas d'une caméra divergente, les lentilles sont disposées autour d'un même centre et filment dans des directions opposées. La figure 1.1 présente une caméras 360° Insta Pro 2 [1] qui est une caméra divergente à plusieurs lentilles. Chaque lentille possède un champ de vision limité et couvre un champ spécifique. Ces champs de vision finissent par se chevaucher et il faut alors traiter ces zones pour éliminer les éléments redondants. Ainsi, nous obtenons une vidéo 360° sans artéfacts et de bonne qualité visuelle pour l'expérience de l'utilisateur.

Figure 1.1 Caméras 360° Insta Pro 2, vue avant et arrière

1.1.2 Raccordement et projections

Comme mentionné, une vidéo 360° est composée de plusieurs vidéos de résolution inférieure provenant de différentes caméras. Ces nombreuses vidéos ont des parties dupliquées qui doivent

[1] https://www.insta360.com/product/insta360-pro2

être fusionnées pour obtenir une projection utilisable pour la diffusion de vidéos 360°. Il existe plusieurs types de projections, par exemple : la projection équirectangulaire (figure 1.2), cubique (figure 1.3), octaédrique, etc. (Ye *et al.*, 2018).

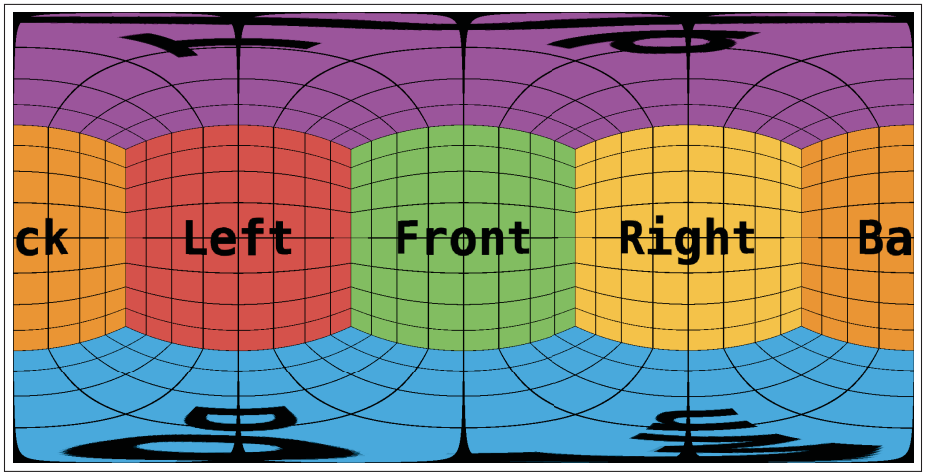

Figure 1.2 Plan des tuiles d'une projection équirectangulaire
Tirée de Wu (2021)

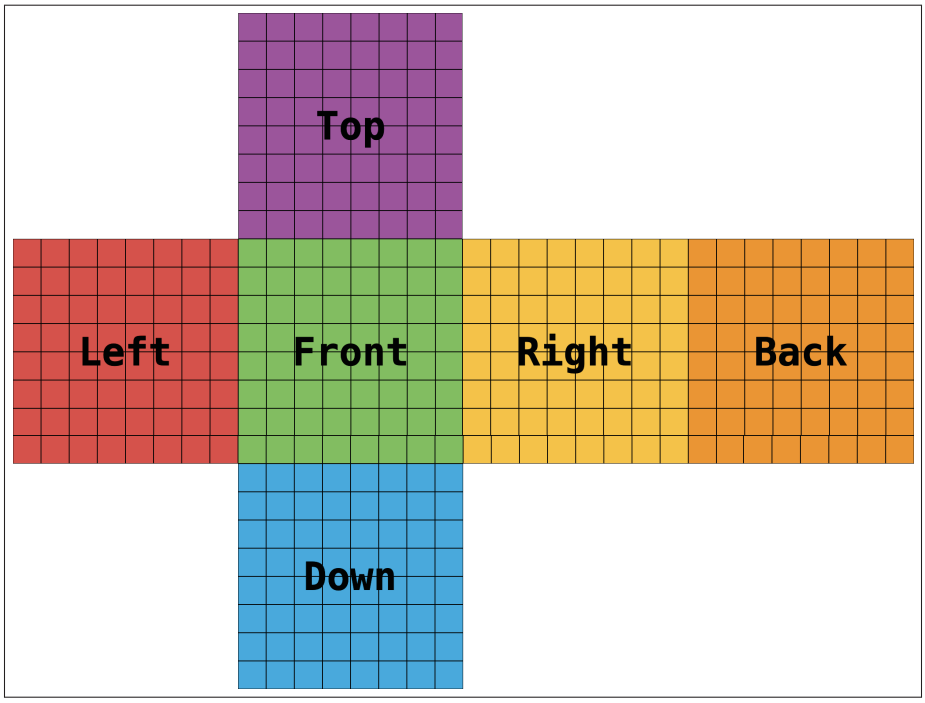

Figure 1.3 Plan des tuiles d'une projection cubique
Tirée de Wu (2021)

1.1.2.1 Projection équirectangulaire

La projection équirectangulaire est l'une des plus simples et couramment utilisée comme projection pour la vidéo à 360°. Elle projette une vue sphérique en un simple plan 2D. Son principal désavantage est la distorsion au niveau des pôles. Étant donné qu'ils sont étirés horizontalement, cela augmente le nombre de pixels, ce qui impacte les performances en compression et le temps de codage. De plus, comme il s'agit d'extrémités peu observées — le contenu principal se situant plutôt au centre — la quantité de pixels ajoutés est généralement redondantes (Chiariotti, 2021). Cependant, en raison de sa simplicité, elle est facile à manipuler et se code plus rapidement (Jamali, Golaghazadeh, Coulombe, Vakili & Vazquez, 2019).

1.1.3 Conversion de coordonnées

La projection d'une vidéo 360° est réalisée dans un espace sphérique complet. Toutefois, le champ de vision de l'utilisateur est limité. Ainsi, comme le montre la figure 1.4, l'utilisateur ne perçoit qu'une faible portion de la sphère à tout moment. L'espace sphérique comporte plusieurs coordonnées, mais celles qui nous intéressent pour la conversion entre espaces sont les deux angles θ et ϕ, qui définissent respectivement la colatitude et la longitude, comme illustré sur la figure 1.5 (Weisstein, 2024). Afin d'obtenir des coordonnées en deux dimensions et de pouvoir travailler dans d'autres types de projections, tel que la projection équirectangulaire, une conversion de ces coordonnées est nécessaire.

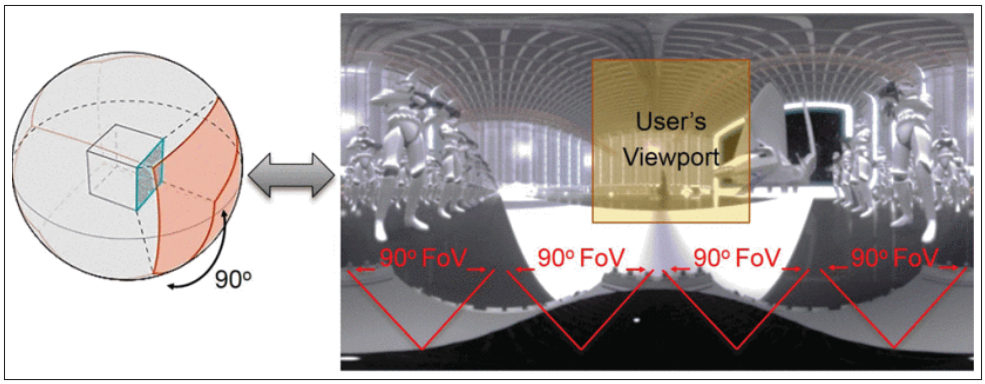

Figure 1.4 Champ de vision de l'utilisateur dans un espace sphérique
Tirée de Hosseini & Swaminathan (2016)

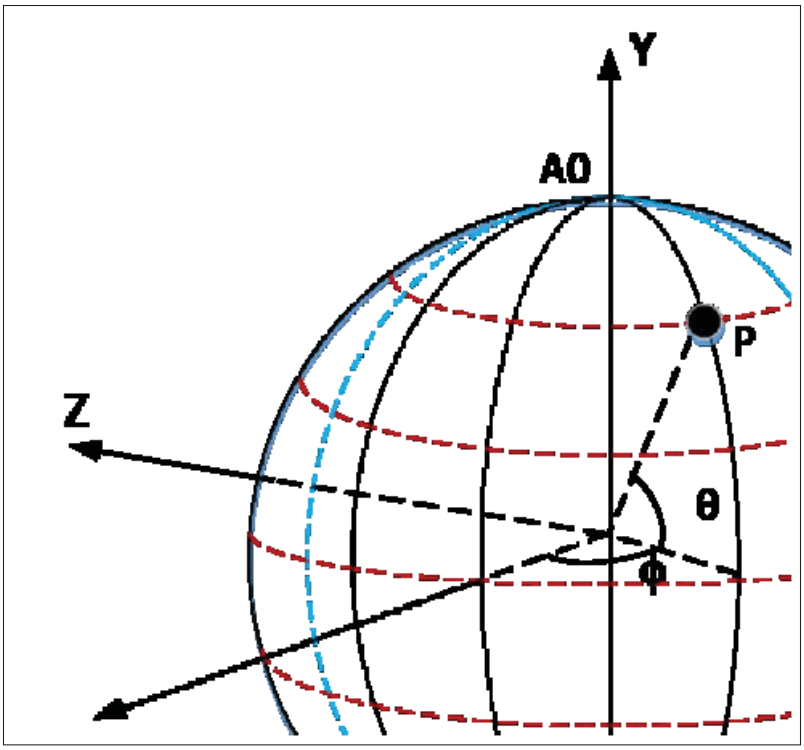

Figure 1.5 Coordonnées θ et ϕ dans un espace sphérique
Tirée de Ye *et al.* (2018)

Selon Ye *et al.* (2018), on peut convertir des coordonnées 2D en coordonnées sphériques de la manière suivante :

$$
\begin{aligned}
u &= \frac{m+0.5}{W}, \quad 0 \leq m < W \\
v &= \frac{n+0.5}{H}, \quad 0 \leq n < H \\
\phi &= (u-0.5) \times 2\pi \\
\theta &= (0.5-v) \times \pi
\end{aligned}
\tag{1.1}
$$

où m et n représentent un point donné en pixels dans le plan 2D, W et H désignent respectivement la largeur et la hauteur en pixels de l'image (sa résolution), u et v sont les coordonnées normalisées dans l'intervalle $[0, 1]$, et ϕ, θ sont les coordonnées sphériques correspondantes.

Cependant, comme nous cherchons à effectuer la conversion inverse (d'un point sphérique vers un point 2D), nous utilisons les formules inverses suivantes pour déterminer (m, n) :

$$\begin{aligned} u &= \frac{\phi}{2\pi} + 0.5 \\ v &= 0.5 - \frac{\theta}{\pi} \\ m &= u \times W - 0.5 \\ n &= v \times H - 0.5 \end{aligned} \tag{1.2}$$

1.1.4 Codage vidéo et codecs pour de la vidéo 360°

Une vidéo n'est en fait qu'une succession d'images, appelées des trames. En raison de plusieurs facteurs, tels que la taille, la quantité et la qualité des trames qui composent la vidéo, celle-ci nécessite rapidement beaucoup de bande passante pour pouvoir être diffusée. Dans ces conditions, pour jouer une simple vidéo, il faudrait une bande passante irréalistement grande pour fournir une expérience utilisateur moyenne. Pour permettre la diffusion de la vidéo, nous utilisons des codecs afin de faire de la compression vidéo et ainsi grandement réduire les exigences en bande passante.

Il y a deux aspects à la compression vidéo : l'encodage et le décodage. L'encodage convertit les trames de la vidéo en un flux de bits, tout en supprimant les redondances entre les trames, s'il y en a. Les encodeurs actuels utilisent divers outils, dont le codage par transformée pour ne garder que les différences entre les trames, ce qui permet de réduire considérablement la quantité de données à transmettre. Une fois la vidéo encodée, transmise et reçue par le client, ce dernier décode le flux de données pour afficher successivement les trames, générant ainsi la vidéo.

Ce processus de codage est réalisé à travers des codecs. Étant donné que nous travaillons avec des vidéos 360° qui requièrent au minimum une résolution de 4K (Budagavi *et al.*, 2015), celles-ci demandent donc davantage de bande passante pour être transmises. Les codecs que

nous devons utiliser doivent donc être suffisamment puissants pour être rapidement encodés au niveau du serveur et facilement décodés par l'équipement de l'utilisateur. Il existe plusieurs codecs. Cependant, les deux codecs que nous aborderons sont H.264 (Advanced Video Coding, AVC) et H.265 (High Efficiency Video Coding, HEVC).

Cloudinary (2025a) compare dans le tableau 1.1 les deux codecs :

Tableau 1.1 Comparaison entre H.264 et H.265

Fonctionnalité	**H.264 (AVC)**	**H.265 (HEVC)**
Efficacité de compression	Standard	Environ 50% meilleure que H.264
Résolution supportée	Jusqu'à 4K	Jusqu'à 8K
Compatibilité matérielle	Universelle	Matériel plus récent
Utilisation CPU/GPU	Faible	Élevée
Cas d'utilisation	Large compatibilité d'appareils, anciens systèmes, diffusion en temps réel	Grande efficacité, haute résolution, VR

Comme le tableau le résume, H.265 est beaucoup mieux adapté à la réalité virtuelle. Cependant, il est compatible avec moins d'appareils et nécessite beaucoup plus de puissance pour encoder la vidéo. À l'inverse, H.264 est plus interopérable avec différents appareils et requiert moins de ressources. Mais il est moins efficace à compresser que H.265 (Cloudinary, 2025a).

Il faut aussi savoir que H.265 est bien plus adapté pour la réalité virtuelle, car il intègre une fonctionnalité qui peut être utilisée pour de la diffusion adaptative en tuiles (qui sera décrit dans la prochaine section) en fonction de la fenêtre de vue de l'utilisateur. Cela peut être réalisé par le "motion-constrained tile sets" (MCTS) (ITU-T, 2024). Cela permettrait de réduire le flux de bits

à envoyer pour les zones non visibles par l'usager à un instant donné et donc de sauver de la bande passante (Ghaznavi-Youvalari *et al.*, 2017).

Cependant, il existe une question de droits et de redevances pour H.265 et H.264. Sans entrer trop dans les détails, si nous voulons utiliser l'un de ces deux codecs, nous devons payer des redevances pour la propriété intellectuelle liée aux technologies de compression. Toutefois, les règles diffèrent entre H.264 et H.265. De manière générale, pour H.265, les 100 000 premiers appareils sont exemptés de payer les royalties, mais après, une redevance se doit d'être payée par appareil (VIA-LA, 2023). Des redevances similaires en lien avec le nombre d'appareils existent également pour H.264, mais il n'y a pas de redevance sur du contenu diffusé via le web, contrairement à H.265 (Ozer, 2009, 2017).

1.2 Diffusion vidéo en continu

La diffusion en continu désigne la transmission de contenu multimédia sans que celui-ci soit envoyé en totalité avant de pouvoir commencer à le consommer. Dans le cas de la diffusion vidéo, la vidéo n'est pas transmise en une seule fois, mais plutôt en segments de N secondes au fil du temps, où N est configurable. Nous diffusons par segments, car il est plus simple d'adapter de petits extraits selon les conditions réseau du client. Nous pouvons ajuster la qualité et le débit à chaque segment selon les conditions fluctuante du réseau. Plusieurs traitements existent, tels que la diffusion par tuiles adaptative et la fragmentation vidéo. De manière générale, la diffusion vidéo nécessite de la communication entre le client et le serveur, suivant une architecture web classique. Ces échanges se font via le protocole HTTP (Hypertext Transfer Protocol) (Fielding, Nottingham & Reschke, 2022) et l'on utilise généralement le protocole MPEG-DASH (Dynamic Adaptive Streaming over HTTP) (Sodagar, 2011) pour faire de la diffusion en continu. Cette section définit davantage la fragmentation, la diffusion par tuiles adaptative et les protocoles MPEG-DASH et DASH SRD.

1.2.1 Fragment vidéo

Un fragment vidéo n'est en fait qu'une partie d'une vidéo, un segment, qui peut se décoder indépendamment des autres. Ceci permet de faire du visionnement vidéo continu et adaptatif, étant donné qu'on peut modifier la qualité de la vidéo à coder durant le visionnement (Cloudinary, 2025b). Le conteneur MPEG-4 (MP4) (Moving Picture Experts Group (MPEG), n.d.) peut contenir une vidéo fragmentée. La différence principale entre un fichier MP4 non fragmenté et un fichier fragmenté réside dans l'arrangement des métadonnées, appelées boîtes ou atomes.

Un fichier MP4 non fragmenté contient, en général, quatre boîtes principales : 'ftype', 'free', 'moov' et 'mdat'. 'ftype' définit le type de fichier et sa compatibilité, 'free' définit l'espace libre, 'moov' détient toutes les métadonnées en lien avec le média, et 'mdat' contient toutes les données liées au média (Authority, 2025).

Un fichier fragmenté contient aussi 'ftype', 'moov' et 'mdat', ainsi que deux nouvelles boîtes : 'moof' et 'sidx'. 'moof' contient un fragment vidéo, et 'sidx' l'index des boîtes (Authority, 2025).

Une vidéo fragmentée est composée des boîtes mentionnées, ainsi que de plusieurs répétitions de 'moof' et 'mdat'. Le nombre de répétitions dépend de la durée souhaitée pour un fragment (n) par rapport à la durée totale de la vidéo (N), soit environ N/n instances de 'moof' et 'mdat' (Bulley, 2016).

1.2.2 Diffusion par tuiles adaptative

La diffusion vidéo peut nécessiter une grande bande passante, ce qui n'est pas toujours disponible ou économique. Ce besoin est encore plus vrai dans le cas d'une vidéo à 360°, dont la résolution est généralement plus élevée. C'est un aspect important, car si la transmission du contenu n'est pas suffisamment rapide, l'expérience utilisateur en souffre. Le visionnage peut alors devenir désagréable et l'immersion, moins réussie. Étant donné que la bande passante ne peut pas toujours être augmentée, il faut donc s'y adapter. Pour cela, on utilise la méthode de diffusion

par tuiles souvent combinées avec d'autres techniques. Plusieurs travaux ont déjà démontré qu'il est possible d'économiser de la bande passante tout en maintenant une expérience visuelle satisfaisante pour l'utilisateur (Son, Jang & Ryu, 2018; Graf, Timmerer & Mueller, 2017).

La diffusion par tuiles consiste à diviser la vidéo originale en plusieurs petites vidéos de résolution inférieure, qui sont spatialement dépendantes — autrement dit, des tuiles. Chaque tuile représente une portion spatiale de la vidéo d'origine. Par exemple, comme illustré à la figure 1.6, la première tuile peut représenter le coin supérieur gauche, la deuxième celle située juste à sa droite, et ainsi de suite. Cette dépendance spatiale est importante, car un mauvais raccordement entre les tuiles produira vraisemblablement une image finale incohérente, en particulier dans le cas d'une projection équirectangulaire.

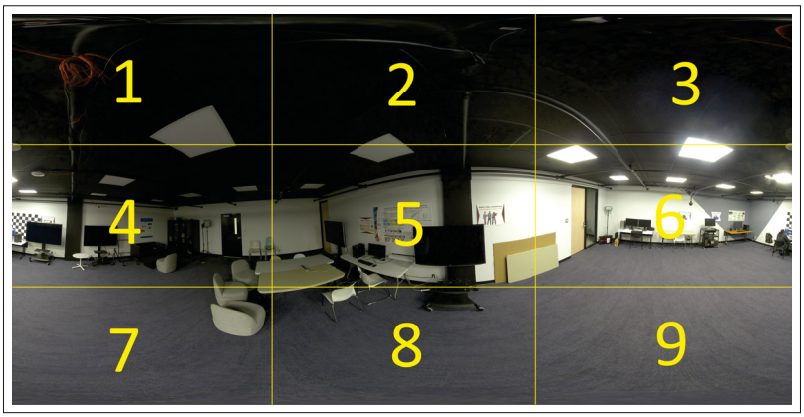

Figure 1.6 Vue équirectangulaire tuilées

La diffusion par tuiles est un bon début pour économiser de la bande passante. Cependant, se contenter de découper et de raccorder les tuiles n'apporte aucun bénéfice supplémentaire. Au contraire, cela augmente le temps de traitement et ajoute une latence supplémentaire. Pour exploiter le plein potentielle de la division par tuiles, plusieurs méthodes de traitement et de sélection adaptative ont été proposées. Elles reposent principalement sur le point de vue de l'utilisateur et sa fenêtre de vue. Graf *et al.* (2017) présente principalement trois techniques de diffusion adaptative par tuiles : une diffusion complète simple, une diffusion complète avancée et une diffusion partielle, comme illustré à la figure 1.7.

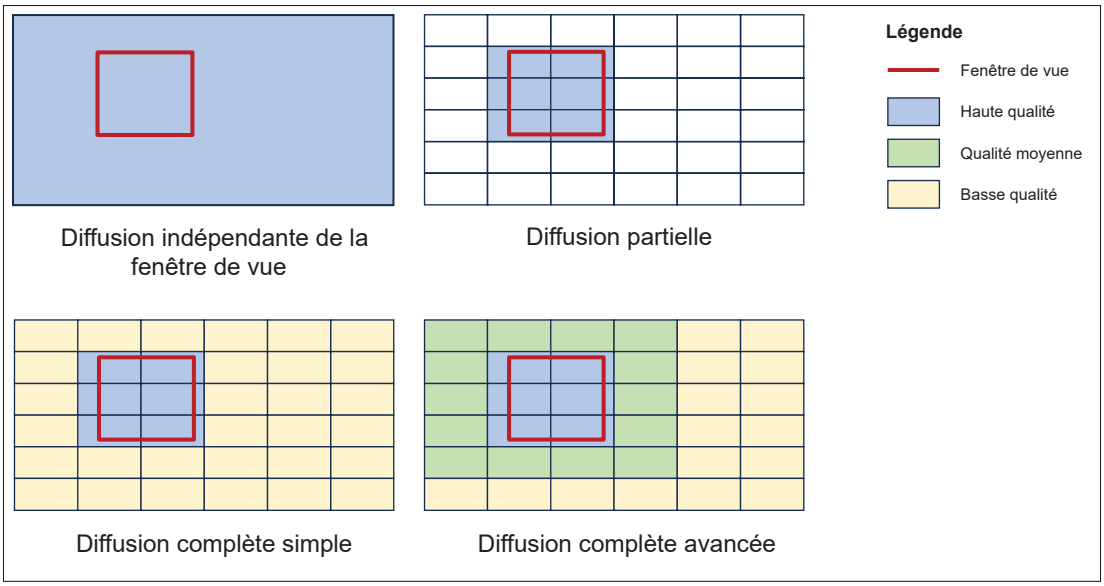

Figure 1.7 Techniques de diffusion adaptative par tuiles
Tirée de Brochu (2024)

La diffusion indépendante de la fenêtre de vue correspond au cas de base, dans lequel toutes les tuiles sont transmises avec le même niveau de qualité, même si l'utilisateur ne les consomme pas toutes. Une telle méthode n'entraîne pas de réelles économies de bande passante. Elle ajoute même une charge de traitement pour le raccordement des tuiles. Dans ce contexte, il pourrait être plus rapide de ne pas utiliser de tuiles et de directement transmettre le segment original.

La diffusion partielle, quant à elle, consiste à transmettre uniquement les tuiles nécessaires pour remplir la fenêtre de visualisation de l'utilisateur. Les autres tuiles ne sont pas envoyées. Étant donné qu'il y a moins de tuiles à traiter, le raccordement et la transmission sont plus rapides, car la quantité de données est réduite. Toutefois, si l'utilisateur modifie son point de vue trop rapidement, il risque de se retrouver sans contenu à consommer, ce qui peut considérablement nuire à l'expérience utilisateur.

La diffusion complète simple transmet toutes les tuiles, mais en adaptant leur qualité en fonction du point de vue de l'utilisateur. Les tuiles visibles dans la fenêtre de vue sont en plus haute qualité, tandis que les autres sont envoyées en basse qualité. Cette méthode garantit que l'utilisateur dispose toujours d'un contenu visible, même s'il change subitement d'orientation. Toutefois,

elle implique également la transmission constante de données qui ne seront probablement pas consommées, entraînant ainsi un gaspillage de bande passante.

La diffusion complète avancée est similaire à la diffusion complète simple, à la différence que les tuiles proches de la fenêtre de vue sont transmises en qualité moyenne, plutôt qu'en basse qualité. Elle présente donc les mêmes avantages et inconvénients, avec un compromis légèrement amélioré en termes de qualité visuelle périphérique (Graf *et al.*, 2017).

Il est possible d'envisager d'autres combinaisons, mais l'essentiel est de comprendre qu'il existe un compromis entre le débit utilisé, la qualité perçue par l'utilisateur et l'expérience de visionnement. Ces méthodologies ont pour objectif de s'adapter aux conditions spécifiques de l'utilisateur afin d'optimiser l'immersion auprès du client.

1.2.3 MPEG-DASH

MPEG-DASH est un protocole de diffusion vidéo qui repose sur une architecture client-serveur. Comme illustré à la figure 1.8, chaque entité a ses propres responsabilités. Le serveur HTTP détient et transmet le contenu multimédia, tandis que le client DASH effectue les requêtes pour obtenir le manifeste et les segments nécessaires.

Le contenu multimédia est structuré selon un manifeste appelé Media Presentation Description (MPD). Comme illustré à la figure 1.9, ce manifeste est composé de périodes, elles-mêmes constituées d'ensembles d'adaptation, qui contiennent des représentations accompagnées de leurs informations.

Les périodes définissent des intervalles de temps dans la vidéo. Il est possible de représenter l'ensemble de la vidéo à l'aide d'une seule période afin de réduire la complexité. Chaque période contient plusieurs ensembles d'adaptation. Ces ensembles définissent les métadonnées vidéo (résolution maximale, taux de trames, etc.) des différentes représentations. Les représentations contiennent les métadonnées d'un segment (débit, codec, résolution, taux de trame, etc.). Il existe une représentation pour chaque variation du contenu multimédia. Par exemple, un même

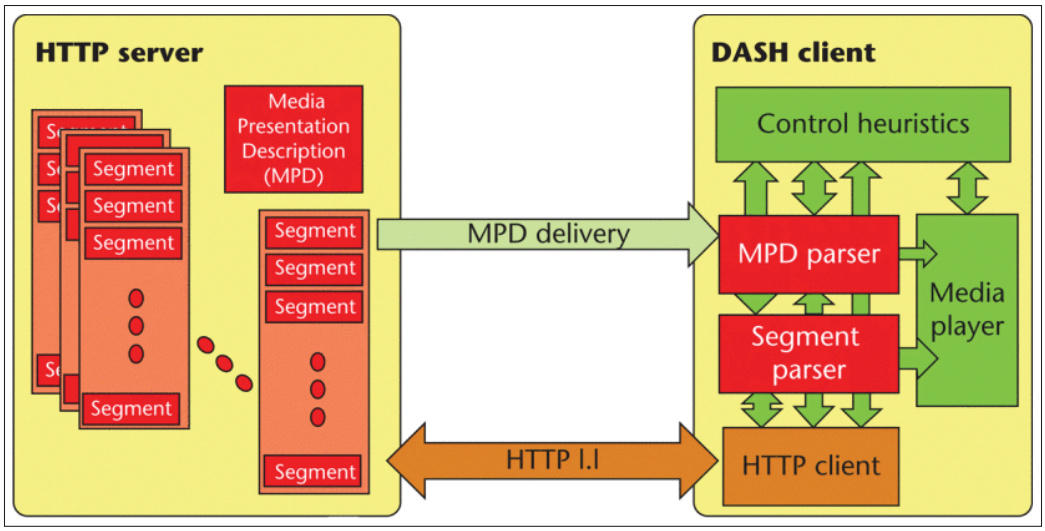

Figure 1.8 Architecture classique de MPEG-DASH
Tirée de Sodagar (2011)

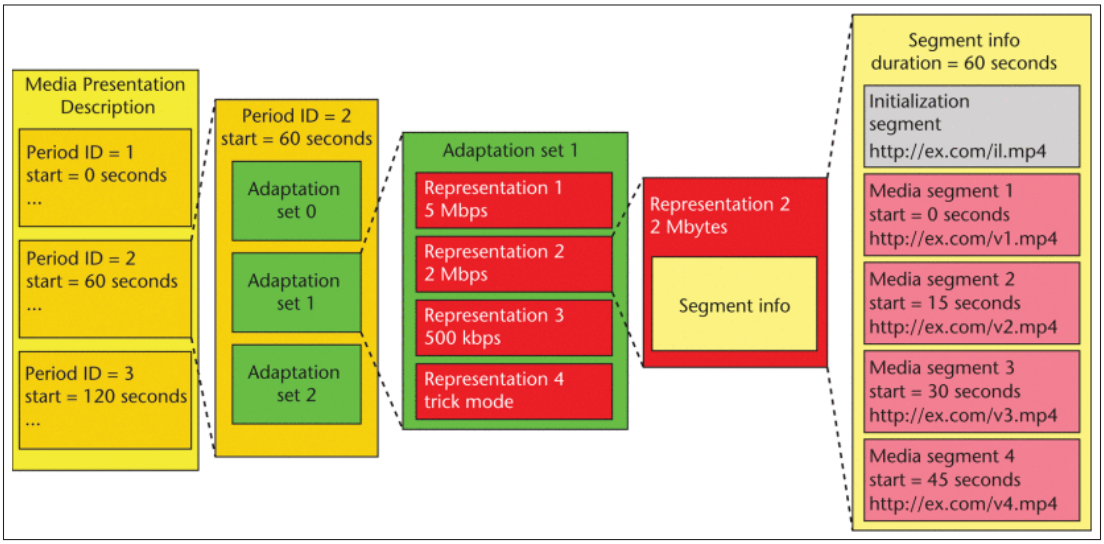

Figure 1.9 Structure d'un fichier MPD
Tirée de Sodagar (2011)

segment peut être encodé avec différents codecs, à des taux de trame différents, ou à des débits variés. C'est dans les représentations que l'on spécifie l'emplacement du fichier multimédia (Sodagar, 2011). La représentation indique également le débit minimum requis pour pouvoir lire un segment. Le client DASH sélectionne, à chaque instant, la qualité du segment vidéo à

télécharger en fonction du débit disponible à ce moment-là, permettant ainsi la diffusion en continu et de manière adaptative.

1.2.4 MPEG-DASH SRD

MPEG-DASH Spatial Representation Description (SRD) est une extension de MPEG-DASH qui décrit les relations spatiales entre les vidéos. La partie SRD s'ajoute au manifeste MPD en décrivant leur position 2D. On définit la coordonnée du coin haut gauche (x, y) et la résolution des tuiles (taille et largeur) Niamut *et al.* (2016).

1.2.5 GPAC

GPAC est un logiciel libre qui, entre autres, permet la lecture de vidéos à 360° (Le Feuvre, 2020). On peut voir GPAC comme un outil facilitant la connexion entre les différents aspects d'optimisation pour la diffusion de vidéos 360°. Il prend en charge la diffusion par tuiles adaptatives grâce à MPEG-DASH SRD, sur des vidéos encodées et segmentées avec H.265, tout en permettant l'interaction avec le contenu à 360°. C'est un outil capable de gérer à lui seul toutes les étapes de la chaîne de diffusion.

1.3 Organisation des tâches et des microservices pour la diffusion en continu de la vidéo 360° dans un réseau périphérique

Dans un réseau en périphérie, comme illustré à la figure 1.10, on retrouve trois entités principales : les serveurs nuagiques (ou centres de données), les serveurs en périphérie, et les clients. Il est important de noter qu'il peut exister plusieurs couches de serveurs en périphérie. Lorsque plusieurs couches sont utilisées, on crée une nouvelle entité : les serveurs de brouillard. Ces deux types de serveurs se distinguent par leur proximité avec le client. Les serveurs situés plus près du client, en communication directe avec celui-ci sont appelés serveurs en périphérie. Les serveurs situés entre les serveurs en périphérie et les centres de données sont appelés serveurs de brouillard. Chaque entité de cette chaîne joue un rôle spécifique et remplit au moins une tâche déterminée.

Dans cette section, nous présenterons deux aspects de la diffusion en continu de vidéos 360° dans un réseau en périphérie : les entités impliquées dans le réseau et leurs rôles respectifs, puis l'organisation des microservices conteneurisés dans un tel réseau pour la diffusion en continu.

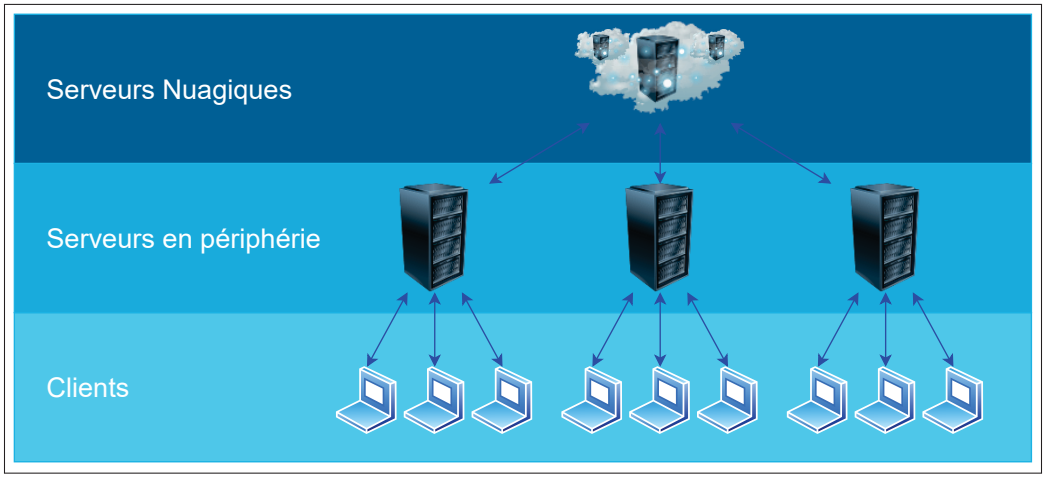

Figure 1.10 Architecture générale d'un réseau en périphérie

1.3.1 Les entités impliquées dans un réseau en périphérie et leurs rôles

Comme mentionné précédemment, un réseau en périphérie comprend principalement quatre types d'entités : le centre de données, les serveurs de brouillard, les serveurs en périphérie et les clients. Chaque entité joue un rôle spécifique dans la chaîne de diffusion. Une entité sans fonction claire ne ferait qu'ajouter de la latence inutile, ralentissant ainsi le système. Les travaux de Rigazzi, Kainulainen, Turyagyenda, Mourad & Ahn (2019); Lo, Huang & Hsu (2018); Santos *et al.* (2021) définissent les tâches associées à ces différentes entités. Nous présentons ici la manière dont ces études attribuent les rôles aux entités, en mettant en évidence à la fois leurs similitudes et leurs différences.

Nous présentons premièrement, le serveur nuagique. Selon Lo *et al.* (2018); Rigazzi *et al.* (2019), ce serveur est chargé de traiter les segments vidéo capturés en temps réel par la caméra, de les encoder avec H.265, puis de diffuser le contenu en utilisant MPEG-DASH. Il agit comme le serveur HTTP de l'architecture de MPEG-DASH. En revanche, le travail de Santos *et al.* (2021)

attribue au serveur nuagique un rôle plus limité : la simple capture de vidéo 360°, sans traitement supplémentaire.

Deuxièmement, nous présentons les serveurs de brouillard. Seuls Rigazzi *et al.* (2019) et Santos *et al.* (2021) intègrent cette entité. Dans Rigazzi *et al.* (2019), les serveurs de brouillard récoltent principalement l'orientation de l'utilisateur, qu'ils transmettent ensuite au serveur en périphérie. Dans Santos *et al.* (2021), l'architecture est plus complexe : deux couches de serveurs de brouillard sont utilisées. La première, située sous le serveur nuagique, est responsable de l'encodage et de la fusion des vidéos (probablement en référence à l'assemblage de tuiles). La seconde, située encore plus près des clients, se charge principalement du transport des flux vidéo, mais peut également participer au processus de fusion.

Troisièmement, nous présentons les serveurs en périphérie. Santos *et al.* (2021) les utilise principalement pour effectuer le rendu de la vidéo. Dans Lo *et al.* (2018), le serveur en périphérie agit comme un client DASH : il traite l'orientation de l'utilisateur à l'aide des informations SRD, télécharge uniquement les tuiles correspondant à la fenêtre de vue, les assemble en une vidéo unique, et transmet celle-ci au client. Rigazzi *et al.* (2019) adopte une approche très similaire à celle Lo *et al.* (2018).

Enfin, nous présentons les clients. Dans les trois travaux, autre que de recevoir, décoder et afficher le contenu final, les clients se doivent de transmettre leur orientation. Qu'il s'agisse de casques de réalité virtuelle ou de lecteurs vidéo classiques, leur rôle dans la création de contenu est simplement de fournir leurs point de vue au serveur en périphérie.

On constate qu'il n'existe pas de répartition fixe des rôles entre les entités. Seul le serveur nuagique semble assumer un rôle relativement unique — celui de traiter le flux vidéo capturé. En revanche, les responsabilités des serveurs de brouillard et en périphérie varient fortement d'un travail à l'autre. Cette diversité s'explique en grande partie par le nombre de machines utilisées. Les travaux comme Rigazzi *et al.* (2019) et Santos *et al.* (2021) disposent d'une infrastructure avec plus de machine et peuvent donc répartir les rôles de manière plus fine, contrairement à

Lo *et al.* (2018), qui s'appuie sur une architecture plus classique. Soit un serveur nuagique, un serveur périphérique et un casque de réalité virtuelle.

Ce que nous cherchons à démontrer ici, c'est qu'il n'existe pas de rôles préétablis pour chaque entité. Les responsabilités sont attribuées de manière à optimiser les performances et à réduire la latence au maximum selon l'architecture que nous voulons utiliser ou disposons.

1.3.2 Diffusion et traitement à travers des microservices conteneurisés dans un réseau périphérique

Définissons d'abord deux termes : microservices et conteneurs. Un microservice est un terme désignant un logiciel qui ne réalise qu'une seule tâche de manière indépendante. Ceux-ci communiquent à travers des appels API (Services, 2025). Plusieurs microservices travaillent ensemble pour réaliser un objectif global plus complexe. Étant simples et indépendants, ceux-ci peuvent être organisés en différentes architectures adaptées à nos besoins.

Un conteneur est un ensemble de logiciels et de leurs dépendances, emballés ensemble afin de standardiser leur déploiement (Docker Inc., 2025a). L'objectif principal d'un conteneur est d'assurer que le logiciel emballé puisse être exécuté dans n'importe quel environnement (Docker Inc., 2025a). Cela vise à résoudre les problèmes d'hétérogénéité et de compatibilité, afin rendre un système et son déploiement plus interopérable.

Maintenant, pour définir des microservices conteneurisés, ceux-ci ne sont que des microservices — c'est-à-dire de simples tâches logicielles — qui sont déployés via des conteneurs. Il existe plusieurs moyens de déployer des conteneurs selon les situations. Un exemple simple serait Docker, qui est une plateforme de déploiement de conteneurs (Docker Inc., 2025b).

Dans la section 1.3.1, nous avons présenté les rôles des entités présentes dans un réseau périphérique, mais pas comment répartir ces rôles. Dans notre travail de recherche, nous allons répartir ces rôles en microservices qui seront conteneurisés. Nous allons nous appuyer principalement sur deux travaux de la littérature, Shen *et al.* (2021) et Santos *et al.* (2021), pour

illustrer des architectures de microservices conteneurisés diffusant de la vidéo 360° dans un réseau périphérique.

Le travail de Shen *et al.* (2021), comme le démontre la figure 1.11, utilise trois microservices : un premier pour la fonction de contrôle du média (MCF pour *Media Control Function*), un second pour la fonction d'ingestion du média (MIF pour *Media Ingestion Function*), puis un troisième pour la fonction de distribution du média (MDF pour *Media Distribution Function*).

Le premier microservice, MCF, a trois tâches importantes : authentifier l'utilisateur, gérer les sessions de visionnement et orchestrer la grappe de microservices. La gestion des sessions est liée aux informations de la vidéo. L'orchestration vise à optimiser l'utilisation des ressources des autres microservices MIF et MDF. Par exemple, elle surveille l'utilisation des ressources des MDF et ajuste leur dimensionnement selon la consommation. Ce microservice peut être placé dans un réseau public tel un serveur nuagique (Shen *et al.*, 2021).

Le second microservice, MIF, pré-traite le contenu vidéo pour sa diffusion. Principalement, il raccorde les vues et encode le contenu. Ce microservice se situe sur des serveurs en périphérie, idéalement proches des caméras ou de la source de contenu, afin de réduire les délais de transmission. Cela implique que les serveurs en périphérie doivent être suffisamment puissants pour effectuer le traitement vidéo (Shen *et al.*, 2021).

Le troisième microservice, MDF, a pour objectif de diffuser des vidéos 360°, via une approche adaptative par tuiles, auprès de plusieurs utilisateurs simultanément. Ce microservice se situerait sur un serveur en périphérie, proche du microservice MIF, afin de réduire la latence de communication (Shen *et al.*, 2021).

Le second travail, celui de Santos *et al.* (2021), nous présente une architecture de microservices plus granulaire et simpliste. Il utilise six microservices, chacun ayant une seule tâche : la capture vidéo, l'encodage, la fusion, le transport de la vidéo, la prédiction de vue et le rendu (figure 1.12). Comme décrit dans la section 1.3.1 et le démontre la figure 1.13, la capture vidéo se fait sur le serveur nuagique. L'encodage, la fusion et le transport vidéo se font sur des serveurs de brouillard.

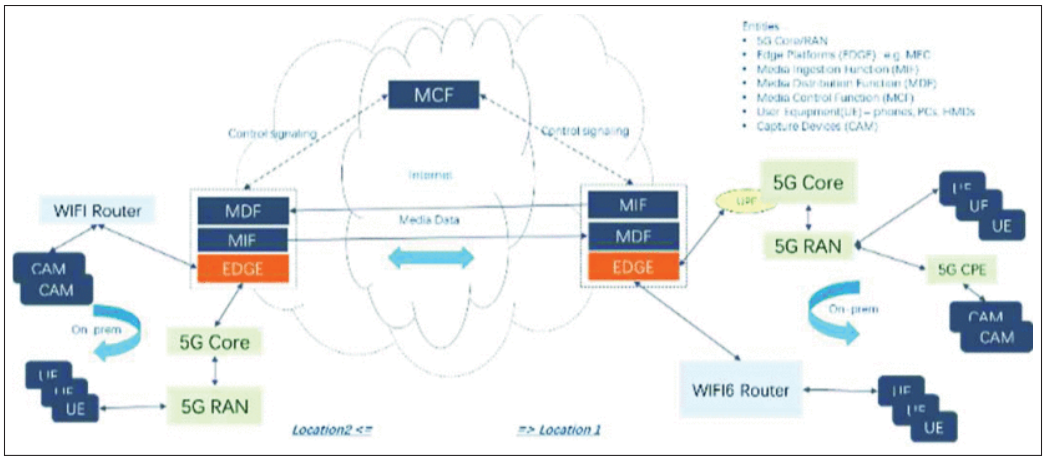

Figure 1.11 Architecture MCF, MIF, MDF
Tirée de Shen *et al.* (2021)

Puis la prédiction et le rendu vidéo sont réalisés sur des serveurs en périphérie. L'opération de la prédiction de vue ne fait que de la diffusion par tuile adaptative. Les deux travaux présentent des architectures de microservices bien distinctes, mais qui répondent à la même problématique.

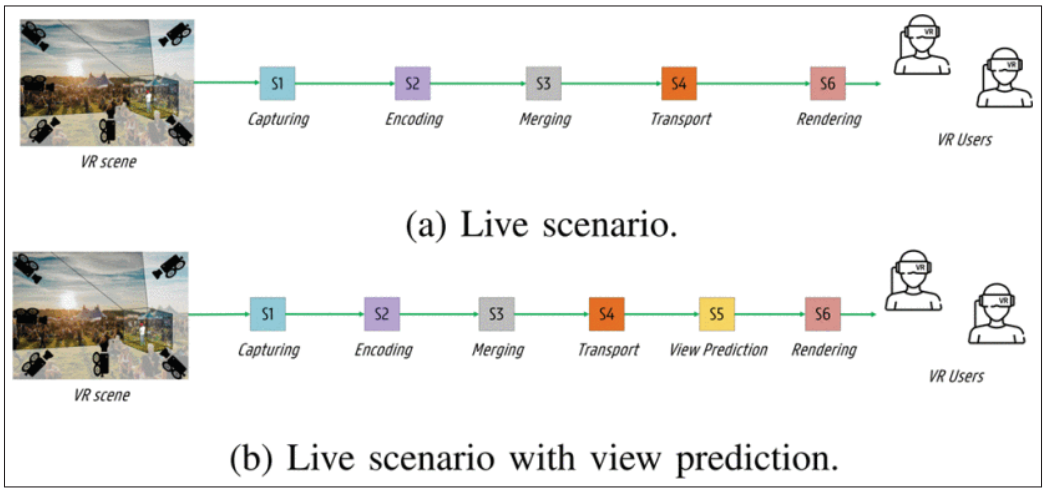

Figure 1.12 Tâches réparties en microservices
Tirée de Santos *et al.* (2021)

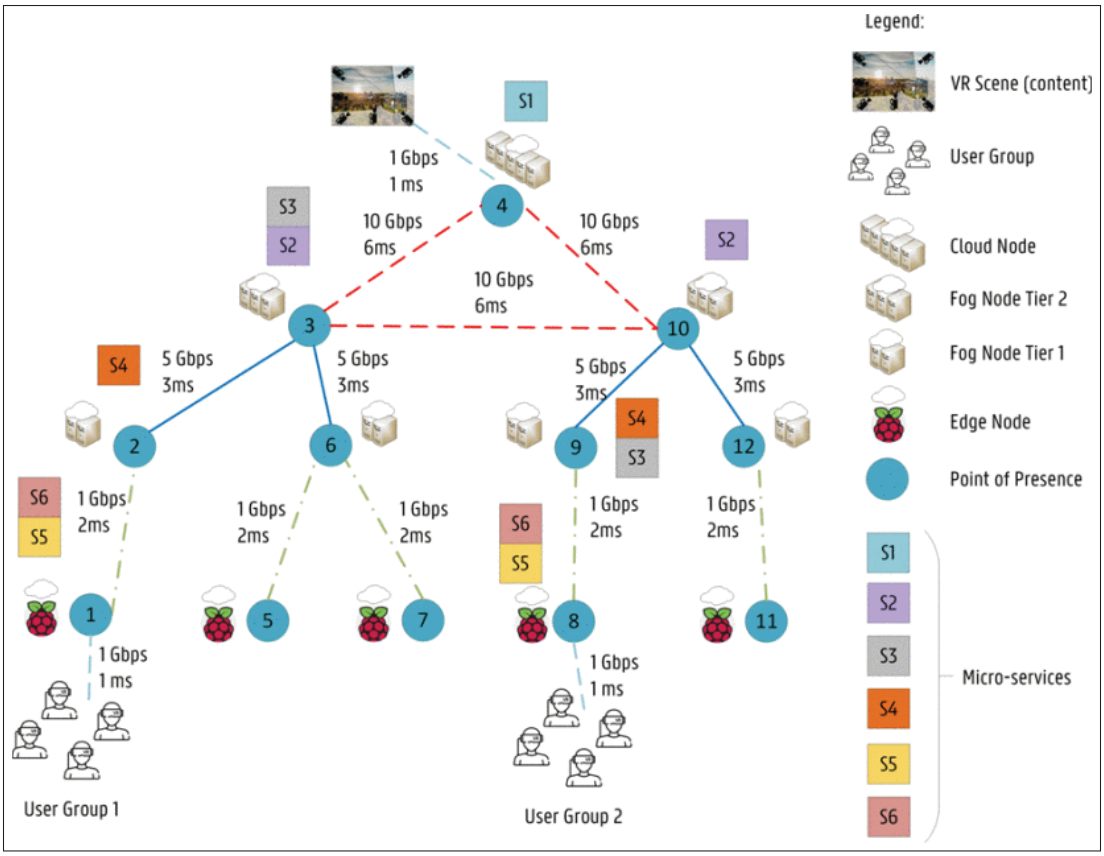

Figure 1.13 Architecture de microservices dans une infrastructure de brouillard
Tirée de Santos *et al.* (2021)

1.4 Discussion

Comme on peut le prévoir, le travail de ce mémoire touchera a cinq thématiques principales : la vidéo 360°, la diffusion en continu sur le web, les déploiements en microservices, la conteneurisation et l'infrastructure périphérie-nuagique. L'objectif est d'évaluer le comportement d'un système de diffusion continue de vidéos à 360° déployé via des microservices conteneurisés dans une infrastructure périphérique-nuagique.

Revenons sur les travaux présentés. Comme le montre le tableau 1.2, les travaux pertinents de la littérature sont complémentaires. Tous les travaux étudiés réalisent de la diffusion de vidéo 360° et la plupart dans un réseau périphérique. Cependant pas tous utilisent des microservices, des conteneurs ou font de la diffusion en continue.

Tableau 1.2 Comparaison des travaux de la littérature avec notre recherche

Caractéristiques	**Rigazzi *et al.* (2019)**	**Santos *et al.* (2021)**	**Lo *et al.* (2018)**	**Shen *et al.* (2021)**	**Notre recherche**
Réseau périphérique	✓	✓	✓	✗	✓
Microservices	✗	✓	✗	✓	✓
Conteneurs	✗	✓	✗	✓	✓
Diffusion en continu	✓	✗	✓	✗	✓

Par exemple, le travail de Rigazzi *et al.* (2019) et de Lo *et al.* (2018) utilisent MPEG-DASH pour faire de la diffusion de vidéo 360° en continu dans un réseau périphérique, mais n'utilisent pas de microservices conteneurisés. En revanche, les travaux de Santos *et al.* (2021) et de Shen *et al.* (2021) font de la diffusion de vidéo 360° dans un réseau périphérique grâce à des microservice conteneurisés, mais sans utiliser MPEG-DASH, donc sans diffusion en continu.

Par ailleurs, les travaux de Shen *et al.* (2021); Rigazzi *et al.* (2019); Lo *et al.* (2018) exploitent le codec H.265 pour la diffusion par tuiles adaptatives. Cependant, H.265 souffre d'une adoption limitée dans les navigateurs web, notamment en raison des problématiques de licences et des contraintes légales (Deveria & Schoors, 2025b). De plus, ces quatre études se focalisent principalement sur l'optimisation de la bande passante afin de réduire les délais de transmission. Elles n'évaluent pas l'impact de leurs architectures sur les ressources matérielles et suppose que l'infrastructure disponible est suffisante pour leur déploiement.

Nos contributions portent précisément sur ces deux aspects : la conception d'une chaîne de traitement pour des segments de vidéos à 360° adaptable à plusieurs codec, et l'analyse de l'impact des différentes architectures sur le matériel. Nous proposons d'adapter et d'évaluer l'impact des architectures de microservices conteneurisés de la littérature sur la consommation des ressources et le temps de livraison, tout en adaptant le flux vidéo pour utiliser H.264 plutôt

que H.265. En interchangeant H.265 par H.264, nous offrons une solution plus accessible pour la diffusion web (Deveria & Schoors, 2025a).

1.5 Conclusion

Ce chapitre a présenté l'état de l'art de la diffusion en continu de vidéo 360° dans un réseau en périphérie.

L'élément clé à retenir est qu'une vidéo 360° est volumineuse, et que son traitement est coûteux en temps et en ressources. Cela rend sa diffusion difficile à réaliser dans un réseau traditionnel. Pour y remédier, plusieurs approches sont adoptées afin de ne transmettre au client que l'essentiel : diffusion par tuiles adaptatives, fragmentation de la vidéo ou utilisation de différents codecs.

Il est également crucial de savoir comment diviser ces tâches pour les déployer efficacement, en fonction des machines disponibles sur le réseau. Les réseaux en périphérie sont bien adaptés à l'utilisation d'architectures à base de microservices, qui peuvent traiter ce type de média de manière efficace.

La combinaison de ces approches, si elle est bien réalisée, permet de fournir un contenu immersif à l'utilisateur tout en n'utilisant pas une quantité extrême de ressources.

CHAPITRE 2

CHAÎNE DE TRAITEMENT VIDÉO 360° ET ARCHITECTURES PROPOSÉES

Plusieurs étapes sont nécessaires afin de pouvoir générer un segment de vidéo 360°. Ces étapes font partie d'une chaîne qui les interconnecte pour produire le segment. Les tâches de cette chaîne sont ensuite encapsulées dans des microservices déployables sur une infrastructure hybride combinant un serveur périphérique et un serveur nuagique. Chaque microservice peut également être dupliqué afin d'offrir une réponse plus rapide à plusieurs utilisateurs simultanés.

Dans ce chapitre, nous décrirons notre chaîne de traitement pour la génération d'un segment de vidéo 360°, ainsi que le fonctionnement de chacune des tâches qui la compose. Nous présenterons ensuite quatre déploiements de microservices inspirés de la littérature. Enfin, nous expliquerons comment ces déploiements permettent de répondre à plusieurs requêtes simultanées.

2.1 Chaîne de traitement vidéo 360° proposée

Dans cette section, nous présenterons notre chaîne de traitement pour la diffusion en continu par tuiles adaptatives pour la vidéo 360°. Notre chaîne est inspirée des travaux de Rigazzi *et al.* (2019); Lo *et al.* (2018); Santos *et al.* (2021); Shen *et al.* (2021). À travers ces travaux, nous avons observé certaines étapes similaires que nous avons catégorisées en quatre tâches principales :

1. La sélection de la qualité des tuiles selon la région d'intérêt de l'utilisateur.
2. La récupération des tuiles selon les qualités définies.
3. Le raccordement des tuiles pour la création du segment.
4. La réécriture du segment dans un flux binaire compatible.

La figure 2.1 montre les interactions entre les tâches de notre chaîne de traitement. L'interaction de cette chaîne nous permet de générer un segment de vidéo 360°, que l'on diffuse dans un lecteur web.

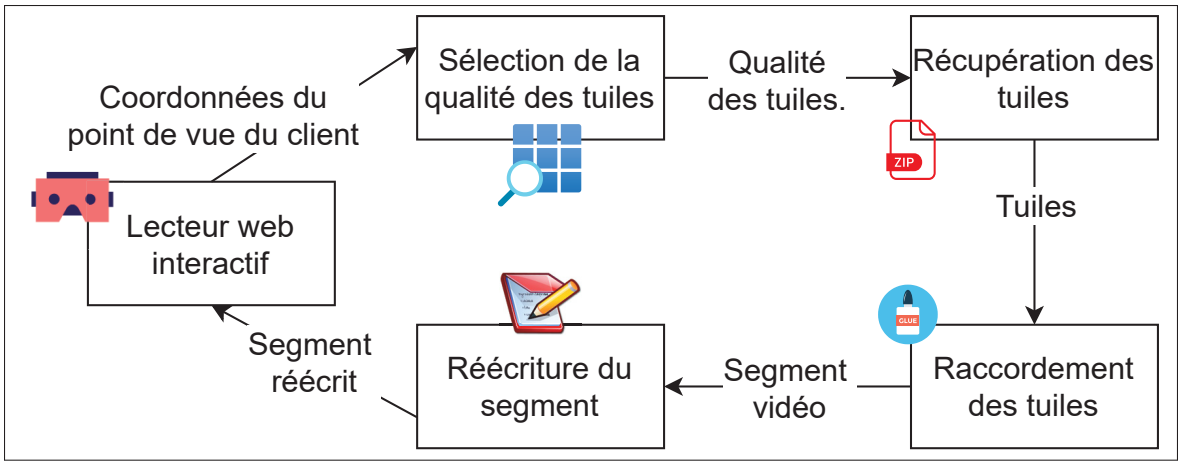

Figure 2.1 Interactions entre les tâches de la chaîne de traitement

Il est a noter que notre approche diffère de la littérature, car nous avons choisi de ne pas utiliser le codec H.265. Cette décision a conduit à concevoir une nouvelle chaîne de traitement, nécessitant des méthodes alternatives pour gérer les tuiles vidéo tout en assurant une couverture visuelle complète à 360°.

2.1.1 Sélection de la qualité des tuiles selon la région d'intérêt de l'utilisateur

Graf *et al.* (2017) présente plusieurs méthodologies de diffusion par tuiles permettant d'optimiser l'utilisation de la bande passante. En adaptant la qualité des tuiles en fonction du point de vue du client, il est possible de générer un flux média dont la taille totale est réduite, permettant ainsi un transfert plus rapide et une économie de bande passante. Pour cela, le client doit transmettre les coordonnées de son point de vue au serveur pour pouvoir permettre la sélection.

Cependant, deux problèmes majeurs subsistent. Premièrement, la taille d'une tuile n'est souvent pas équivalente à celle de la région d'intérêt de l'utilisateur. Deuxièmement, l'utilisateur ne regarde pas uniquement le centre d'une tuile. Cela signifie qu'il faut plusieurs tuiles pour couvrir la région d'intérêt de l'utilisateur et offrir un effet d'immersion. La raison de pourquoi nous devons effectuer une sélection de la qualité.

Notre approche consiste à nous baser sur le positionnement spatial des coins de la fenêtre de visualisation de l'utilisateur. À partir d'un point m et n, représentant le centre du point de vue de l'utilisateur, et en tenant compte de la résolution de la fenêtre de visualisation ainsi que de celle du segment, nous projetons les quatre coins de la fenêtre sur les pixels correspondant à la résolution du segment.

Chaque tuile contenant au moins un pixel situé à l'intérieur de cette zone projetée est considérée comme une tuile en haute qualité. Les autres sont affichées en basse qualité. Ainsi, notre méthodologie s'adapte dynamiquement à la résolution du segment et à celle de la fenêtre de visualisation.

2.1.2 Récupération des tuiles selon les qualités définies

Cette tâche consiste à récupérer les tuiles avec la qualité sélectionnée lors de l'étape précédente. Il s'agit d'une interaction entre le microservice qui effectue la requête et celui qui détient les tuiles. Les déploiements seront présentés plus loin, mais à titre d'exemple, la figure 2.5 illustre cette tâche à travers l'interaction entre le microservice 3 et le microservice 4.

Étant dans un environnement web, la réception d'une réponse nécessite l'envoi d'une requête. Pour optimiser l'efficacité et accélérer la transmission des tuiles, celles-ci sont regroupées dans un fichier compressé au format ZIP. Une fois ce fichier reçu, le microservice demandeur décompresse le fichier afin de récupérer les tuiles.

2.1.3 Raccordement des tuiles pour la création du segment

Graf *et al.* (2017) présente la technique de diffusion complète simple, qui diffuse les tuiles en haute qualité dans la fenêtre de visualisation de l'utilisateur et en basse qualité pour le reste. Cette approche permet une meilleure interaction avec le contenu tout en économisant de la bande passante. Peu importe la direction dans laquelle l'utilisateur tourne brusquement la tête, du contenu sera toujours disponible.

Comme mentionné précédemment, H.265 dispose de la fonctionnalité MCTS, qui facilite la diffusion par tuiles adaptatives et répond au cas d'utilisation où l'utilisateur changerait son point de vue rapidement. Cependant, H.265 présente une accessibilité limitée pour le web (Deveria & Schoors, 2025b) et nécessite du matériel plus performant pour l'encodage (Cloudinary, 2025a).

Nous avons donc opté pour l'utilisation de H.264 en raison de sa meilleure compatibilité web (Deveria & Schoors, 2025b) et de ses exigences matérielles réduites (Cloudinary, 2025a). Toutefois, ce choix introduit de nouveaux défis techniques puisque H.264 ne possède pas de fonctionnalité native pour la diffusion par tuiles adaptatives. En d'autres termes, H.264 ne peut pas travailler directement avec des tuiles vidéo et il faut les raccorder pour générer l'image complète avant l'encodage et la diffusion.

Pour réaliser le raccordement des tuiles, nous avons utilisé FFmpeg avec son filtre "xstack" (FFmpeg, 2019). Ce filtre permet de créer une mosaïque vidéo. Dans notre cas, il sert à assembler les tuiles dans un ordre spécifique pour reconstituer un segment à sa résolution initiale, mais composé de plusieurs tuiles de différentes qualités visuelles. La qualité d'une tuile est en fait le débit auquel nous avons encodé la tuile. Un haut débit représente une tuile de haute qualité et vice-versa. La commande et une explication de son fonctionnement se retrouve dans l'annexe I.

2.1.4 Réécriture de segment dans un flux binaire compatible

Les travaux de la littérature réécrivent le segment en un nouveau flux de données pour être transmis au client. Dans notre cas, nous ne réécrivons pas, car nous avons déjà un segment encodé et diffusable. Cependant, en raison du fonctionnement du lecteur (section 2.1.5), nous devons fragmenter notre vidéo. C'est-à-dire réarranger la structure des boîtes du conteneur MP4.

Pour ce faire, nous utilisons l'outil "mp4fragment" de Bento4 [1]. Il prend simplement la vidéo raccordée en entrée et nous renvoie un vidéo fragmenté, qui est ensuite envoyé au lecteur web. Il faut savoir que nous avons déjà des segments d'une durée inférieure à la durée totale. Dans notre

[1] https://www.bento4.com/documentation/mp4fragment

cas, nous ne fragmentons pas le segment en plus petite durée. Nous recréons un segment de la même durée, mais avec des métadonnées différentes dans le conteneur MP4.

2.1.5 Lecteur web interactif de vidéo 360°

Le lecteur interactif n'est en fait qu'une simple page HTML et JavaScript qui utilise la librairie A-Frame [2] et un objet MediaSource [3].

A-Frame est un framework web qui utilise Three.js pour permettre la création de plusieurs applications VR dans un navigateur web. Il est possible de créer une gamme d'applications, mais dans notre cas, nous l'utilisons pour le visionnement et l'interaction avec une vidéo 360°. Lorsque l'utilisateur change de point de vue, nous récoltons les rotations de la caméra (ϕ, θ) que nous convertissons en coordonnées pixels (m, n). Ces valeurs sont ensuite transmises pour décider de la qualité des tuiles.

L'objet MediaSource est l'outil qui nous permet de faire de la diffusion en continu. L'objet détient un tampon auquel nous pouvons ajouter des segments vidéo. Cependant, les segments se doivent d'être fragmentés, justifiant l'étape de la réécriture de segment.

L'ensemble des deux technologies nous permet de créer un lecteur web interactif pour vidéo 360°.

2.1.6 Résumé et contributions

En résumé, notre chaîne de traitement se distingue des approches typiques de la littérature, principalement par sa capacité à prendre en charge différents codecs, offrant ainsi la liberté de ne pas dépendre d'un codec spécifique. La littérature utilise généralement le codec H.265. Cependant, en raison de restrictions de licence, l'utilisation de H.265 sur le web est moins accessible.

---

[2] https://aframe.io/

[3] https://developer.mozilla.org/en-US/docs/Web/API/MediaSource

De ce fait, nous avons décidé d'utiliser le H.264. Ce choix nous a privés des fonctionnalités liées à la diffusion adaptative par tuiles pour la vidéo 360°. Nous devons donc raccorder les tuiles nous-mêmes, ce que nous réalisons à l'aide de FFmpeg et du filtre `xstack`.

De plus, étant donné que nous n'utilisons pas GPAC, puisque nous faisons de la diffusion sur le web, nous avons dû créer notre propre lecteur. Nous avons utilisé A-Frame et un objet MediaSource afin de concevoir un lecteur interactif simple. Cependant, contrairement à la littérature, nous avons également dû fragmenter nos segments, car nous utilisons un tampon MediaSource.

Nos contributions, à ce stade, sont donc la proposition d'une chaîne alternative pour le traitement de vidéos 360°, accompagnée d'un lecteur interactif adapté à celle-ci.

2.2 Déploiements de microservices proposés

Dans la section 2.1, nous avons établi une chaîne de traitement et décrit quatre tâches principales communes aux travaux de l'état de l'art mentionnés. Ces tâches sont indépendantes les unes des autres, ce qui permet de les diviser ou de les regrouper. On peut ainsi soit distribuer chaque tâche dans un microservice distinct, soit regrouper plusieurs tâches dans un seul microservice. Ces microservices peuvent ensuite être déployés sur différentes machines selon diverses infrastructures. Dans notre cas, nous nous basons sur une infrastructure composée d'un serveur périphérique et d'un serveur nuagique, sur lesquels nous déployons nos microservices.

La distribution des microservices dans une infrastructure périphérique-nuagique a pour but d'optimiser l'utilisation des ressources. En répartissant la charge sur plusieurs machines, nous pouvons utiliser du matériel plus adapté pour certaines tâches tout en allégeant la charge de travail des serveurs principaux. Ce partage de charge permet de libérer des ressources et d'ajuster dynamiquement le nombre de microservices en fonction de la demande. De plus, l'utilisation de serveurs périphériques géographiquement proches des clients permet une livraison plus rapide des segments vidéo.

Dans cette section, nous présentons quatre déploiements de microservices, constitués des tâches décrites précédemment, au sein d'une infrastructure périphérique. L'objectif de ces déploiements est d'analyser leurs forces et faiblesses pour le traitement vidéo 360° à travers un serveur périphérique et un serveur nuagique. Ces quatre déploiement s'inspirent des travaux de Le Feuvre (2020); Shen *et al.* (2021); Rigazzi *et al.* (2019); Santos *et al.* (2021).

2.2.1 Déploiement Monolithique

Ce déploiement est inspiré de l'outil GPAC (Le Feuvre, 2020). Il consiste à réaliser toutes les tâches dans un seul microservice situé proche du client. Ce microservice se charge à la fois du traitement vidéo pour les segments et de la gestion des fichiers multimédias.

2.2.1.1 Tâches du microservice

Le déploiement est très simple. Comme son nom l'indique, il s'agit d'un déploiement monolithique, c'est-à-dire que toutes les tâches sont regroupées au sein d'un seul microservice.

Microservice 1 :

- Génère la page HTML du lecteur web qui interagira avec le microservice 1 et l'envoie au client.
- Reçoit les requêtes pour les segments du lecteur web.
- Sélectionne la qualité des tuiles selon la région d'intérêt.
- Raccorde les tuiles à l'aide de FFmpeg.
- Fragmente le segment raccordé.
- Envoie en réponse la vidéo fragmentée au lecteur web.

2.2.1.2 Fonctionnement du déploiement

Comme illustré par la figure 2.2, tout s'exécute dans un seul microservice. Tous les fichiers nécessaires (MPD, tuiles, etc.) sont stockés localement dans le microservice. Cela permet

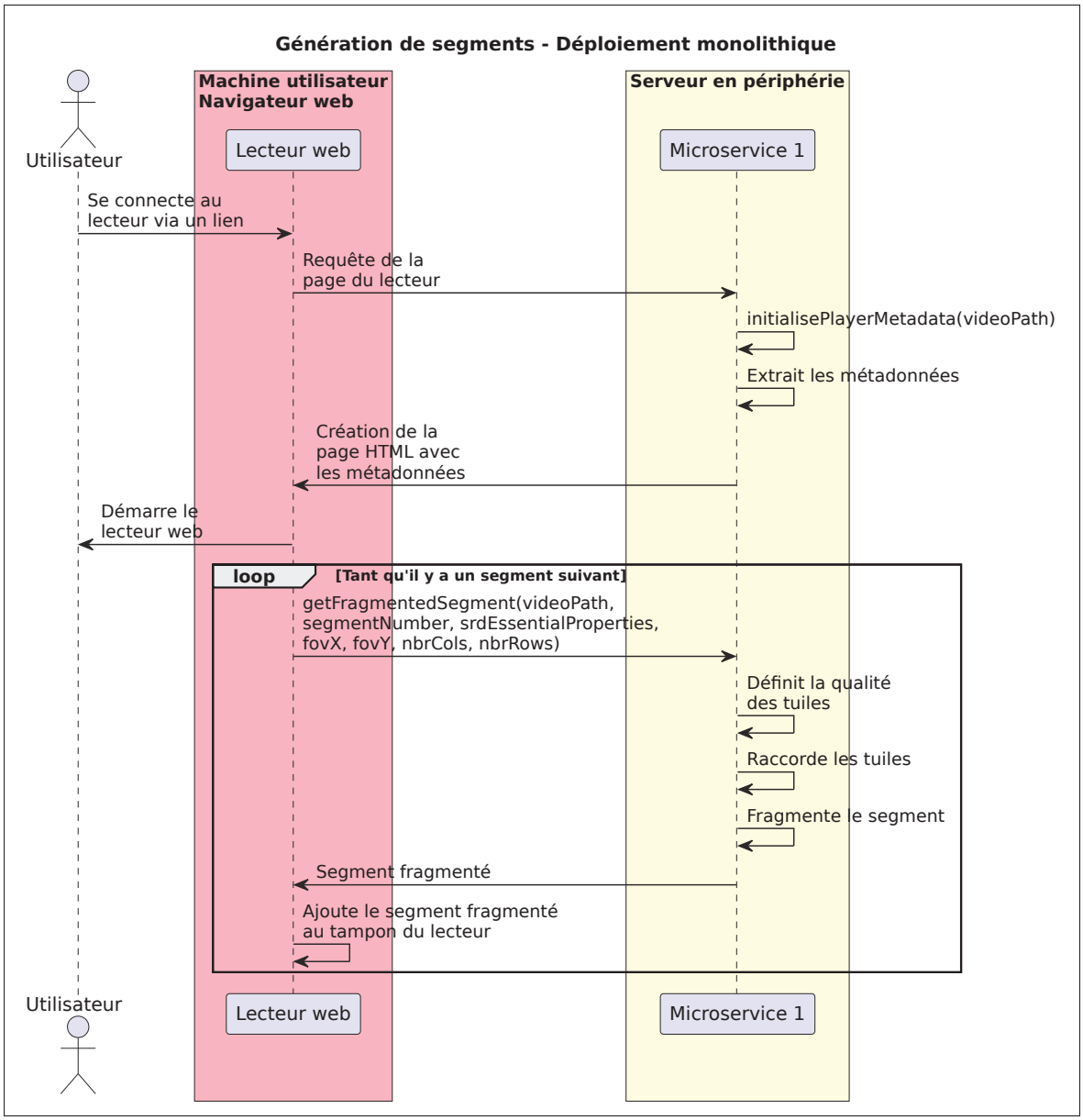

Figure 2.2 Déploiement de l'architecture Monolithique

d'effectuer toutes les opérations de traitement localement, sans nécessiter de communications inter-microservices. Il faut aussi noter que le microservice s'exécute dans le serveur périphérique.

2.2.1.3 Avantages et inconvénients du déploiement

Ce déploiement présente un avantage important : toutes les données nécessaires sont disponibles localement. L'absence de communication enlève des délais potentiels. De plus, le traitement des segments peut s'exécuter séquentiellement, sans arrêts intermédiaires.

Cependant, ce modèle présente également un inconvénient majeur. Sa structure monolithique rend les tâches fortement cohésives, ce qui limite la possibilité d'exécuter le traitement vidéo en parallèle. Les étapes étant séquentielles, il est difficile de paralléliser ou de répartir la charge entre plusieurs instances. L'absence de modularité limite la capacité du système à gérer plusieurs utilisateurs simultanément.

2.2.2 Déploiement Proximité

Cette architecture est inspirée de celle de Shen *et al.* (2021). Elle a pour objectif d'effectuer toutes les opérations exigeantes sur un seul microservice proche du client. Les médias ne sont pas stockés sur le microservice où les traitements s'effectuent, mais sur un autre microservice. Il est donc nécessaire d'y faire une requête pour les récupérer.

2.2.2.1 Tâches des microservices

Dans ce déploiement, trois microservices sont utilisés, avec les tâches réparties de la manière suivante.

Microservice 1 :

- Envoie en réponse le fichier MPD de la vidéo demandée au microservice 2.
- Envoie en réponse un fichier ZIP des tuiles du segment demandé au microservice 2.

Microservice 2 :

- Reçoit une requête pour le prochain segment du lecteur web.
- Envoie une requête au microservice 1 pour obtenir le fichier MPD de la vidéo.
- Sélectionne la qualité des tuiles selon la région d'intérêt.

- Envoie une requête au microservice 1 pour récupérer les tuiles.
- Raccorde les tuiles à l'aide de FFmpeg.
- Fragmente le segment raccordé.
- Envoie en réponse la vidéo fragmentée au lecteur web.

Microservice 3 :

- Génère la page HTML du lecteur web qui interagira avec le microservice 2 et l'envoie au client.

En résumé, le microservice 1 agit comme un MCF contenant toutes les tuiles et fichiers multimédias. Lorsqu'il est interrogé, il retourne les fichiers demandés. Il est situé sur le serveur nuagique. Le microservice 2 peut être assimilé à un MIF, car il est chargé de toutes les opérations de traitement pour générer un nouveau segment (détermination de la qualité des tuiles, téléchargement, raccordement et fragmentation). Il est déployé sur un serveur périphérique géographiquement proche du client. Le microservice 3 sert uniquement à générer la page du lecteur. Une fois cette tâche accomplie, il n'intervient plus dans les chaînes d'opérations.

2.2.2.2 Fonctionnement du déploiement

Comme illustré plus en détail à la figure 2.3, l'utilisateur se connecte à la page via un lien menant à la vidéo. Une fois qu'il déclenche la lecture d'une vidéo 360°, le navigateur envoie une requête au microservice 3 pour obtenir la page HTML du lecteur.

Avant d'envoyer cette page, le microservice 3 doit la générer à l'aide de certaines métadonnées liées à la vidéo. Il communique donc avec le microservice 2, responsable des tâches vidéo, notamment de l'extraction des métadonnées du fichier MPD. Le microservice 2 envoie une requête au microservice 1 pour obtenir ce fichier, en extrait les métadonnées nécessaires, puis les transmet au microservice 3 pour qu'il puisse générer la page du lecteur. Ces métadonnées incluent, par exemple, le nombre de segments, le codec et le type MIME du fichier.

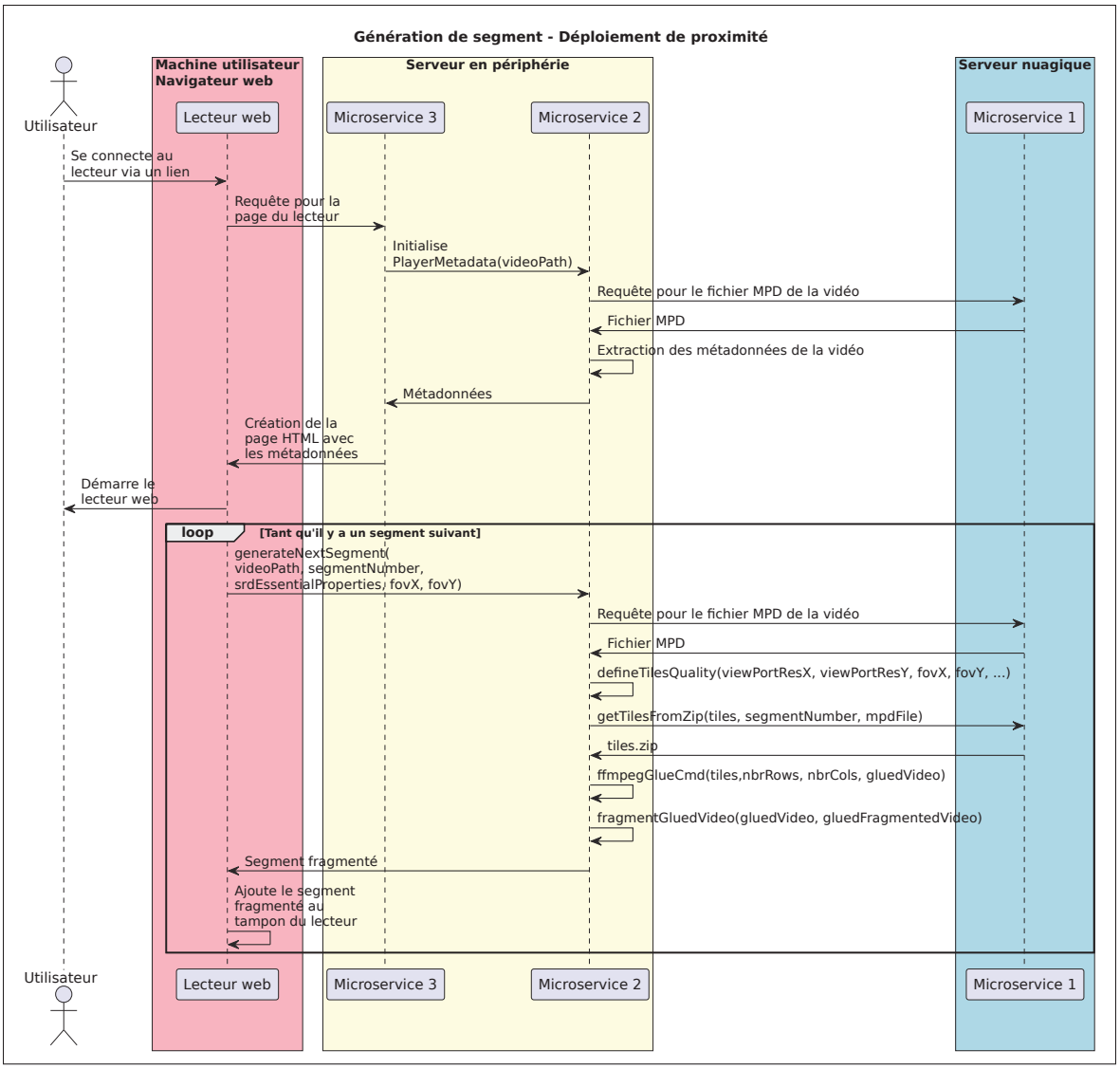

Figure 2.3 Déploiement de l'architecture Proximité

Une fois la page générée, les requêtes pour les segments vidéo sont lancées. Tant qu'il reste des segments à lire, une requête est envoyée pour le suivant.

Il est important de noter que la page HTML du lecteur communique directement avec le microservice 2, sans repasser par le microservice 3. Cela permet de réduire les délais liés à l'intercommunication.

La génération d'un segment se fait en plusieurs étapes. Lors de la requête, certaines informations sont envoyées, telles que le segment demandé ainsi que les coordonnées (m, n) du point de regard de l'utilisateur. Le microservice 2 sélectionne ensuite la qualité des tuiles, effectue une requête auprès du microservice 1 pour récupérer les tuiles, les raccorde, fragmente, puis envoie le résultat au lecteur web.

2.2.2.3 Avantages et inconvénients du déploiement

Ce déploiement présente un avantage principal : tous les calculs et traitements sont réalisés localement par le microservice 2, réduisant ainsi la latence entre les étapes, à l'exception de la récupération des tuiles.

Cependant, la division des tâches est peu équilibrée, car les tâches sont majoritairement centralisées dans le microservice 2. Dans un contexte périphérique, cela signifie qu'il faudrait des serveurs périphériques plus puissants que dans des serveurs nuage. Il serait probablement plus judicieux de tout regrouper dans le microservice 1, sur un serveur nuagique plus performant, plutôt que d'exiger d'une machine périphérique l'exécution de plusieurs opérations complexes.

En outre, sur le plan matériel, le fait que le microservice 2 réalise des tâches très variées (calculs, encodage, écriture de fichiers, etc.) rend difficile le choix d'un matériel optimisé. Il est impossible de choisir une configuration dédiée, car le service n'est pas spécialisé. Certains matériels seraient plus efficaces pour le traitement vidéo, d'autres pour la lecture ou l'écriture de fichiers.

2.2.3 Déploiement Distribué

Cette architecture est inspirée de Rigazzi *et al.* (2019). C'est une version similaire à l'architecture Proximité. Elle a pour objectif de mieux répartir les tâches a travers les microservices. L'idée est d'observer l'impact d'une meilleure répartition des tâches.

2.2.3.1 Tâches des microservices

Ce déploiement réutilise le déploiement Proximité, mais redistribue les tâches de manière à n'effectuer que du traitement vidéo dans le serveur en périphérie :

Microservice 1 :

- Envoie en réponse le fichier MPD au microservice 2.
- Reçoit une requête du microservice 2 pour les tuiles du segment.
- Sélectionne la qualité des tuiles selon la région d'intérêt.
- Envoie en réponse un fichier ZIP des tuiles du segment demandé au microservice 2.

Microservice 2 :

- Reçoit une requête du lecteur web pour le prochain segment.
- Envoie une requête au microservice 1 pour obtenir les tuiles du segment demandé par le lecteur web.
- Raccorde les tuiles en utilisant FFmpeg.
- Fragmente le segment raccordé.
- Envoie en réponse la vidéo fragmentée au lecteur web.

Microservice 3 :

- Génère la page HTML du lecteur web qui interagira avec le microservice 2 et l'envoie au client.

Comme on peut le constater, par rapport à l'architecture Proximité, le nombre et la nature des microservices restent inchangés. Il y a juste une autre répartition des tâches.

Comme dans le déploiement Proximité, tous les fichiers liés à la vidéo (tuiles et fichier MPD) sont stockés dans le microservice 1, situé sur un serveur nuagique. Nous avons donc regroupé toutes les fonctions liées à ces fichiers dans ce même microservice. Cela permet de supprimer le délai de transmission du fichier MPD, étant donné qu'il peut être lu localement, contrairement au déploiement Proximité. La fonction de détermination de la qualité des tuiles est également

intégrée au microservice 1. Puisque celui-ci gère les tuiles, il est plus efficace qu'il envoie directement les bonnes tuiles, selon la qualité demandée.

Le microservice 2 est uniquement chargé du traitement vidéo : raccordement et fragmentation. Une fois les tuiles reçues, il assemble le segment et l'envoie au lecteur. Il suit une approche similaire au microservice MIF de Shen *et al.* (2021), car nous ne faisons que du traitement vidéo sur le serveur en périphérie.

Enfin, comme dans la version précédente, le microservice 3 se limite à générer la page HTML du lecteur web.

2.2.3.2 Fonctionnement du déploiement

Comme l'illustre la figure 2.4, le fonctionnement du déploiement distribué est très similaire à celui du déploiement de Proximité. Les tâches principales sont conservées, à l'exception du fait que c'est désormais le microservice 1 qui détermine la qualité des tuiles.

2.2.3.3 Avantages et inconvénients du déploiement

L'avantage principal de ce déploiement réside dans une meilleure répartition des tâches. Une distribution plus cohérente permet de supprimer certaines étapes intermédiaires, ce qui réduit les redondances et optimise la communication. De plus, cette séparation permet de mieux tirer avantage du matériel.

L'inconvénient principal reste que le microservice 2 est chargé de la tâche la plus exigeante : le raccordement des tuiles. Dans un scénario avec de nombreux utilisateurs, ce microservice devient un goulot d'étranglement, n'ayant pas la capacité de traiter toutes les requêtes de manière efficace. Dans un tel cas, il serait potentiellement plus judicieux de déplacer le raccordement vers le microservice 1, afin de tirer avantage des ressources plus importantes du serveur nuagique.

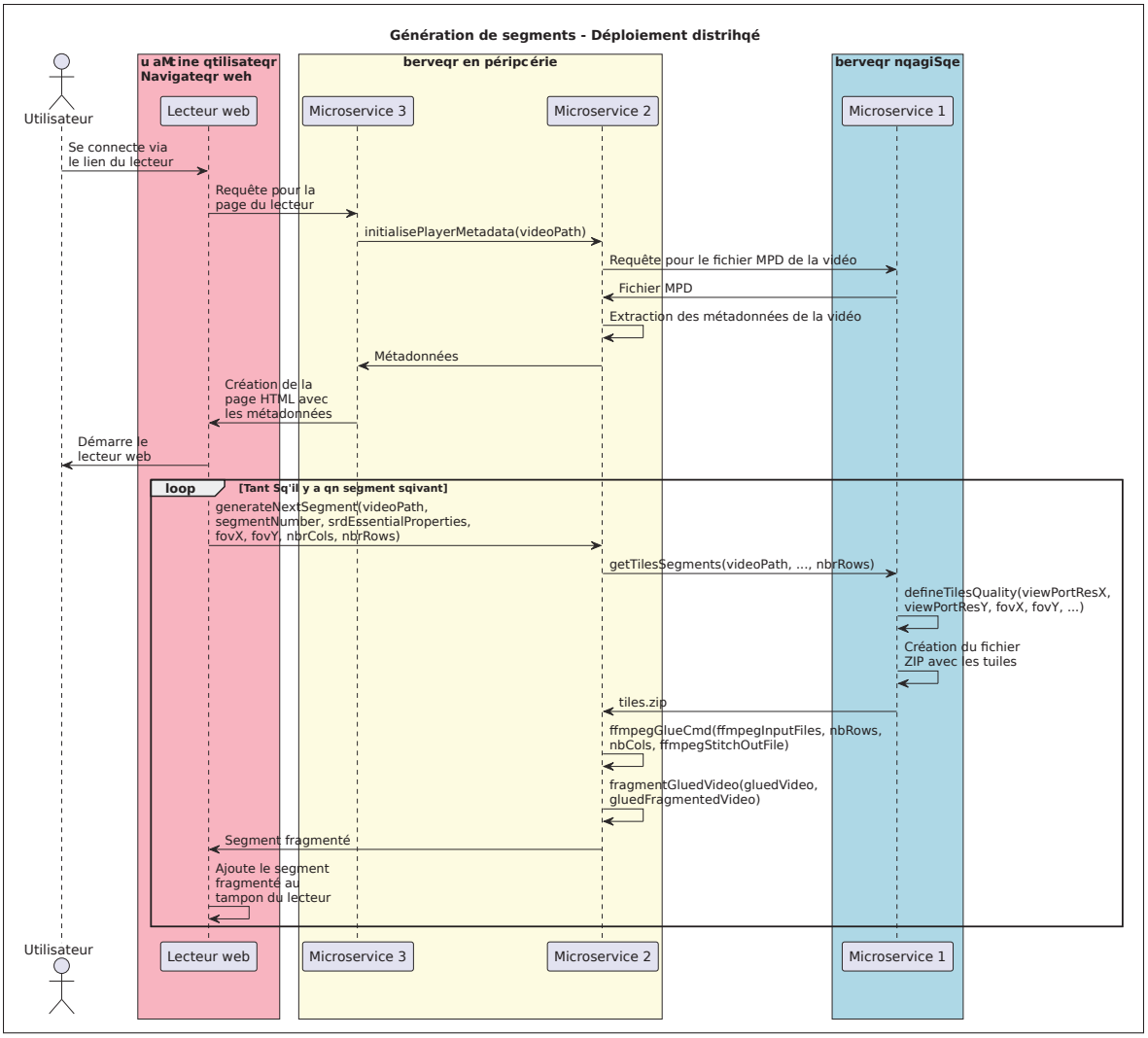

Figure 2.4 Déploiement de l'architecture Distribué

2.2.4 Déploiement Granularisé

Cette architecture ressemble celle de Santos *et al.* (2021). Elle vise à isoler chaque tâche de traitement dans un microservice distinct. L'objectif est de déterminer si une division plus fine des services peut améliorer les performances du système.

2.2.4.1 Tâches des microservices

Le déploiement Granularisé, contrairement au déploiement Monolithique, assigne une seule tâche par microservice. Il n'y a aucun regroupement de fonctionnalités, contrairement aux autres déploiements.

Microservice 1 :

- Reçoit la région d'intérêt et les métadonnées du segment.
- Sélectionne la qualité des tuiles en fonction des coordonnées reçues.
- Transmet la qualité des tuiles au microservice 2.

Microservice 2 :

- Gère les tuiles et le fichier MPD.
- Reçoit la demande de segment du microservice 3, accompagnée des métadonnées de la région d'intérêt transmises par le microservice 1.
- Compresse les tuiles du segment selon les qualités calculées, puis les transmet au microservice 3.

Microservice 3 :

- Reçoit la demande de segment du microservice 4 et la transmet au microservice 2.
- Décompresse le fichier ZIP contenant les tuiles.
- Raccorde les tuiles pour reconstruire le segment vidéo.
- Transmet le segment raccordé au microservice 4.

Microservice 4 :

- Reçoit la demande de segment depuis la page HTML et la transmet au microservice 3.
- Fragmente le segment vidéo reçu.
- Envoie le segment fragmenté à la page HTML.

Microservice 5 :

- Génère la page HTML du lecteur web qui interagira avec le microservice 4 et l'envoie au client.

Les étapes de traitement restent les mêmes que dans les versions précédentes, mais elles sont ici réparties en plusieurs microservices, chacun exécutant une tâche spécifique.

Le microservice 1 calcule exclusivement la qualité des tuiles en fonction de la région d'intérêt et retourne une liste des qualités. Le microservice 2 gère les fichiers multimédias (tuiles et MPD). Le microservice 3 assemble les tuiles en un segment vidéo complet. Le microservice 4 fragmente ce segment. Enfin, le microservice 5, comme dans les autres déploiements, se contente de générer la page HTML du lecteur.

2.2.4.2 Fonctionnement du déploiement

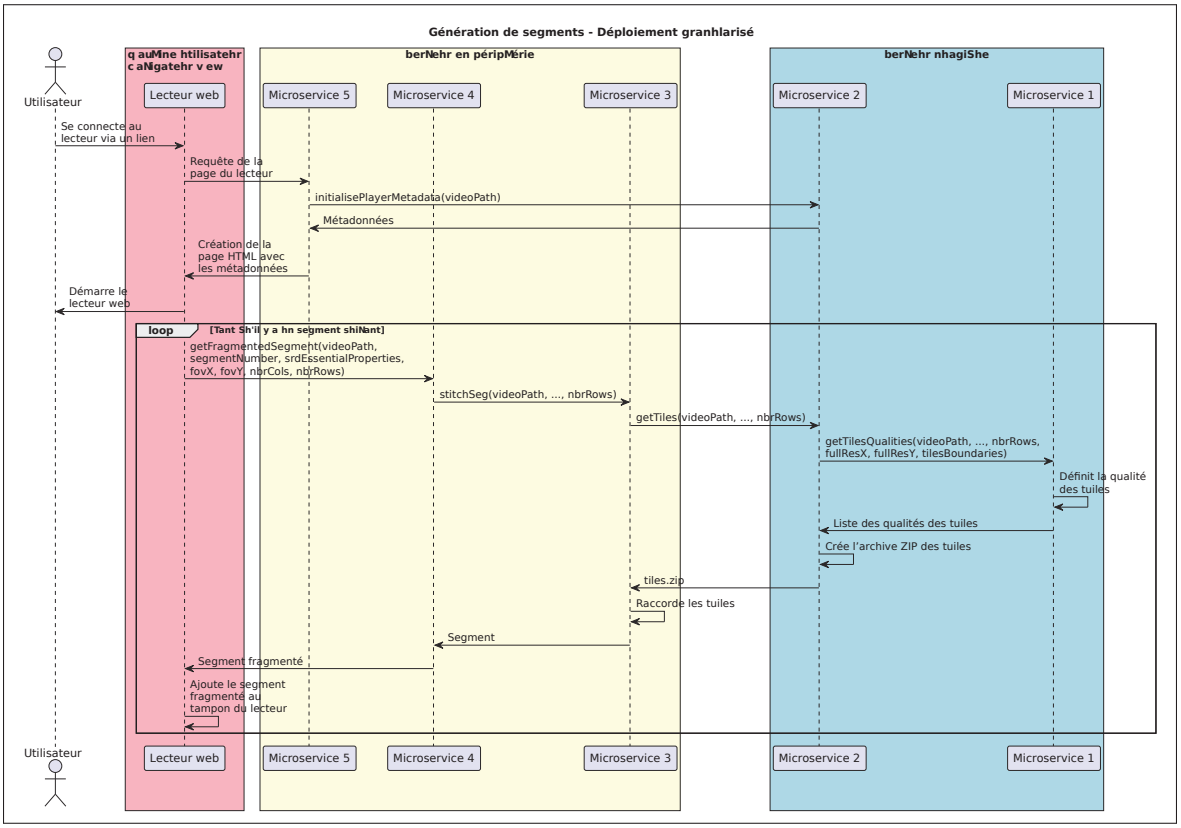

Figure 2.5 Déploiement de l'architecture Granularisé

Comme illustré à la figure 2.5, le positionnement des tâches du déploiement Granularisé est similaire à celui du déploiement Distribué. On y retrouve la fragmentation et le raccordement sur

le serveur périphérique, ainsi que la gestion des médias et la sélection des tuiles sur le serveur nuagique.

L'initialisation est plus directe, car elle n'implique plus de microservice intermédiaire pour collecter les métadonnées nécessaires.

Concernant la génération des segments, les étapes sont identiques, mais la requête pour un segment est transmise à chaque microservice, chacun dépendant de la réponse du précédent dans la chaîne. Plus concrètement, une fois le calcul de qualité effectué, le microservice 1 transmet la liste des qualités au microservice 2, qui utilise ces informations pour récupérer et compresser les tuiles dans un fichier ZIP. Celui-ci est ensuite transmis au microservice 3, qui raccorde les tuiles en un segment vidéo, puis envoie le segment au microservice 4. Ce dernier fragmente le segment vidéo et l'envoie au lecteur web, qui l'ajoute à son tampon de diffusion.

2.2.4.3 Avantages et inconvénients du déploiement

Comme ce déploiement reprend en grande partie la logique du déploiement Distribué, les avantages et inconvénients sont similaires.

Le premier avantage est une initialisation plus directe du lecteur web, grâce à une réduction des communications intermédiaires entre ce dernier et le microservice chargé des fichiers multimédias.

Ensuite, cette architecture est hautement modulaire. Cette modularité permet de déployer les microservices sur une ou plusieurs machines selon les besoins, ce qui facilite l'adaptation du matériel à la tâche : calculs, traitement vidéo, compression, etc.

L'inconvénient principal réside dans l'augmentation du volume de communications entre microservices. Cela peut introduire des délais supplémentaires, surtout si les microservices sont répartis sur des machines situées à différents emplacements géographiques.

2.2.5 Comparaison qualitative entre les déploiements

Résumons les caractéristiques principales de chaque déploiement à travers le tableau 2.1.

Tableau 2.1 Tableau récapitulatif des déploiements

	Répartition des tâches	**Adaptabilité matérielle**	**Volume de communication entre microservices**
Monolithique	Aucune	Très faible	Aucun
Proximité	Faible	Faible	Faible
Distribué	Forte	Forte	Faible
Granularisé	Très forte	Très forte	Très grand

Le premier point de comparaison porte sur les deux premiers critères : la répartition des tâches et l'adaptabilité au matériel. Comme on peut le voir dans le tableau récapitulatif, ces deux critères sont fortement corrélés. Plus les tâches sont réparties finement, plus l'adaptabilité au matériel augmente. En d'autres termes, plus le travail est divisé, plus on peut déployer les microservices sur des machines adaptées à leurs fonctions spécifiques. Par exemple, les machines optimisées pour le traitement vidéo n'ont pas besoin de réaliser de la détection de qualité ou de la lecture et écriture de fichiers. Les déploiements distribués et granularisés sont donc les plus adaptés en termes d'adaptabilité matérielle, car leur tâches sont bien réparties.

Il existe également une seconde corrélation, cette fois entre la répartition des tâches et le volume de communication entre les microservices. Plus on utilise de microservices pour exécuter les tâches, plus le volume de communication augmente. Cela est évident si l'on compare les déploiements monolithique et granularisé. Un déploiement constitué d'un seul microservice centralisé, qui exécute toutes les fonctions localement, n'a pas besoin de communiquer avec d'autres services, contrairement à un système hautement distribué. Ce volume de communication peut avoir un impact négatif, notamment en raison des délais ajoutés par la dispersion des

hôtes, surtout dans un contexte de déploiement en périphérie où les machines ne sont pas nécessairement proches géographiquement ou étant reliés pas des liens lents ou partagés et fortement sollicités.

Ainsi, nous pouvons identifier un triangle de compromis entre la répartition des tâches, l'adaptabilité au matériel et le volume de communication nécessaire. Plus on souhaite augmenter l'adaptabilité matérielle, plus il faut répartir les tâches, ce qui entraîne une augmentation des communications, et vice versa.

Grâce à cette analyse qualitative, nous pouvons émettre l'hypothèse que le déploiement Distribué serait le plus performant. En effet, il permet une forte répartition des tâches, une bonne adaptabilité au matériel, tout en maintenant un faible volume de communication. La principale différence entre le déploiement de Proximité et le déploiement Distribué est une meilleure répartition des tâches, sans pour autant augmenter le nombre de microservices.

2.3 Gestion du nombre d'utilisateur

Les microservices des déploiements présentés communiquent via des API basées sur le protocole HTTP. Cette architecture a été conçue pour simplifier le développement et standardiser les échanges entre microservices. Le modèle requête-réponse typique des communications web s'avère particulièrement pratique pour orchestrer des tâches distribuées sur plusieurs machines.

Cependant, dans un contexte multi-utilisateurs simultanés, l'incapacité à répondre rapidement aux requêtes entraîne la formation de files d'attente. Pour prévenir ces goulots d'étranglement, une solution consiste à augmenter le nombre d'instances capables de traiter les requêtes. Dans un environnement web conteneurisé, il existe plusieurs types d'instances que nous pouvons utiliser pour dupliquer le nombre de répondant d'un microservice et ainsi réduire le goulot d'étranglement.

Dans cette section, nous décrirons trois différentes instances qui peuvent être utilisées, les relations entre elles et la manière dont nous les utiliserons avec nos microservices.

2.3.1 Types d'instance et leur relation

Cette sous-section présente les définitions des trois instances (conteneur, service, travailleur) et leur relation. Ces trois instances peuvent être dupliquées en un certain nombre de répliques afin de pouvoir répondre à plusieurs requêtes simultanément.

2.3.1.1 Conteneur

Le conteneur est l'entité qui regroupe le code ainsi que toutes ses dépendances nécessaires à son fonctionnement Docker Inc. (2025a). Il constitue une unité logicielle autonome et exécutable. Chaque conteneur correspond à une instance unique d'exécution. Il peut être vu comme un processus.

2.3.1.2 Service

Un service représente une abstraction au sein d'un environnement Docker Swarm [4] (Docker, Inc., 2024). Il n'est pas une entité physique, mais plutôt une interface permettant d'orchestrer et de gérer une ou plusieurs réplique d'un même conteneur. Le rôle d'un service est de simplifier le déploiement en permettant la création de ces répliques et en assurant leur distribution de manière automatique sur les différentes machines disponibles.

Il existe une correspondance directe entre un service et ses conteneurs : demander N répliques d'un service crée N conteneurs. Cependant, la façon dont la charge de travail est répartie entre les répliques n'est pas en notre pouvoir. C'est Docker Swarm qui gère la répartition de la charge de travail.

2.3.1.3 Travailleur web

Le travailleur web est une entité interne à l'application web. Il représente un processus séparé chargé de traiter une requête HTTP à la fois (MDN Web Docs, n.d.). Dans notre contexte, cela

[4] https://docs.docker.com/engine/swarm/

signifie exécuter une ou plusieurs tâches définies dans un microservice, comme le traitement d'un segment vidéo.

Pour gérer plusieurs requêtes simultanées, il est nécessaire d'avoir plusieurs travailleurs. Un seul travailleur qui tente de gérer toutes les requêtes devient un goulot d'étranglement.

Même s'il n'est pas explicitement lié aux services ou aux conteneurs, le travailleur web vit à l'intérieur de l'application. Et comme nos conteneurs exécutent des applications web, ils peuvent contenir un ou plusieurs travailleurs web.

2.3.1.4 Relation entre les instance

Les trois instance sont liées de manière hiérarchique comme illustré à la figure 2.6. Le service orchestre le déploiement des conteneurs, qui à leur tour exécutent les travailleurs web pour répondre aux requêtes HTTP.

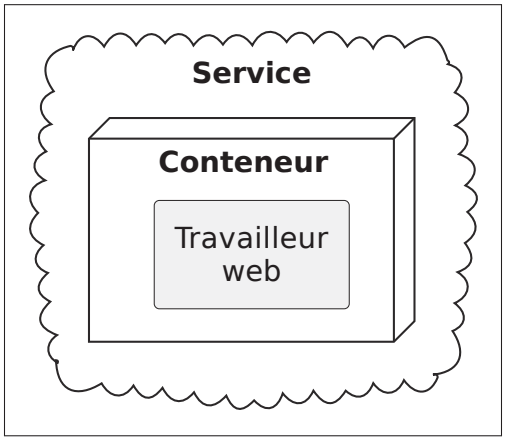

Figure 2.6 Relation hiérarchique entre les instances

2.3.1.5 Duplication des instances

Il est possible de dupliquer les instance pour répondre à plusieurs requêtes en simultané et donc d'éviter un goulot d'étranglement. Nous pouvons soit augmenter le nombre de conteneurs (figure 2.7), soit le nombre de travailleurs web (figure 2.8), ou les deux simultanément (figure 2.9).

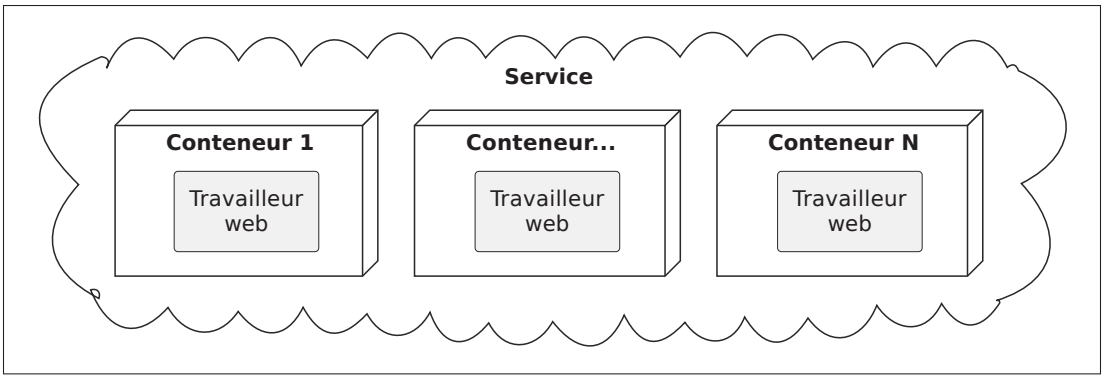

Figure 2.7 Duplication de conteneurs

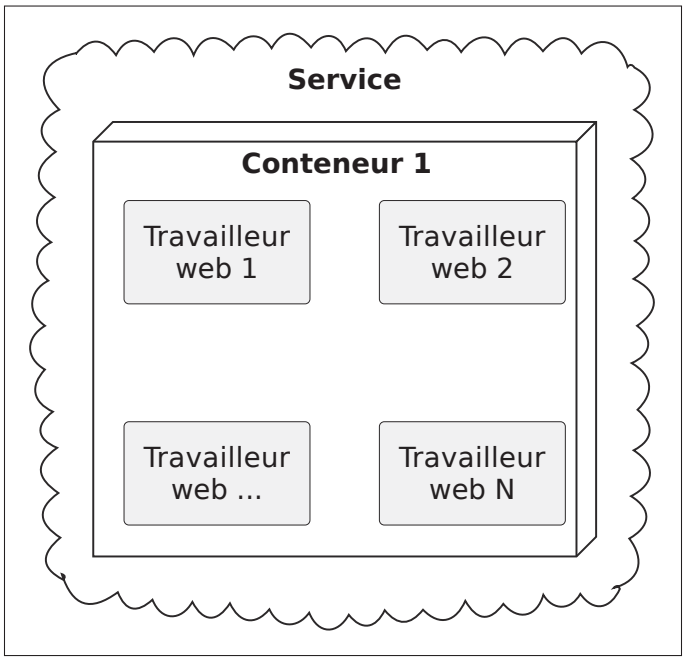

Figure 2.8 Duplication de travailleurs web

2.3.2 Utilisation des instances

Dans ce mémoire, nous testerons l'impact du nombre de conteneurs et de travailleurs web, mais pas des deux simultanément. Nous déploierons un certain nombre de répliques en fonction des tests et des architectures.

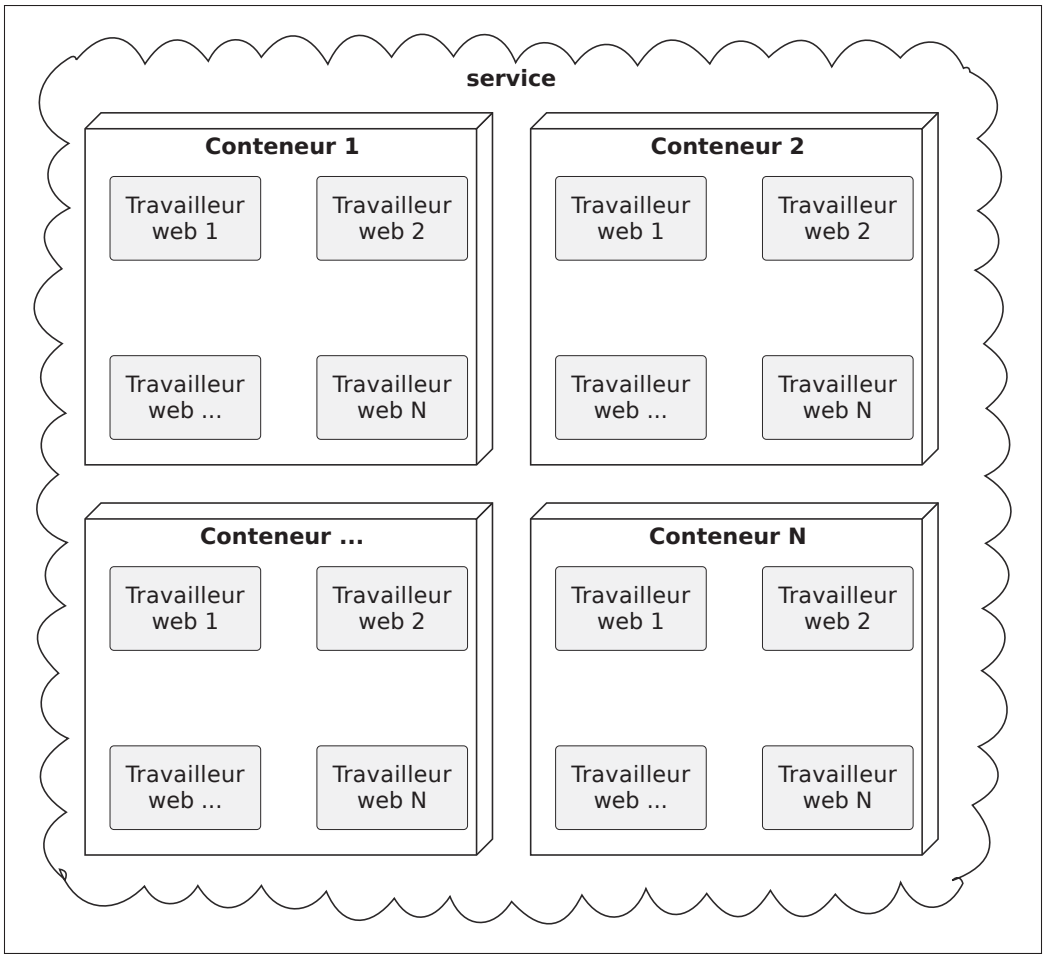

Figure 2.9 Duplication de conteneurs et travailleurs web

Nous déploierons un nombre N de conteneurs, ou de travailleurs web, pour les microservices exécutant une tâche de la chaîne de traitement. Le microservice qui génère la page HTML n'est pas constamment en service. Une fois la page du lecteur web générée, il n'est plus sollicité. Il n'a donc pas d'impact majeur sur les tests. Dans une optique d'économie de ressources, nous ne créerons qu'une seule réplique. Le tableau 2.2 présente les déploiements ainsi que le nombre de conteneurs répliqués pour chaque microservice sur chaque serveur.

Pour le déploiement monolithique, nous avons déployé trois répliques du seul microservice défini afin d'éviter des goulots d'étranglement du au nombre de requêtes.

Tableau 2.2 Nombre de conteneurs déployés pour chaque microservice sur chaque serveur

Déploiement		Nombre de répliques	
		Serveur périphérique	Serveur Nuage
Monolithique	Microservice 1	3	0
Proximité	Microservice 1	0	1
	Microservice 2	3	0
	Microservice 3	1	0
Distribuée	Microservice 1	0	1
	Microservice 2	3	0
	Microservice 3	1	0
Granularisée	Microservice 1	0	3
	Microservice 2	0	3
	Microservice 3	3	0
	Microservice 4	3	0
	Microservice 5	1	0

Pour les déploiements Distribué et Proximité, nous avons choisi de mettre trois répliques uniquement pour le microservice 2, car c'est celui qui exécute l'opération de raccordement des tuiles. Les opérations du microservice 1 dans ces deux déploiements ne requièrent pas énormément de ressources et de temps. Dans un futur chapitre nous présenterons de manière quantitative les requis de chaque tache principale.

Pour le déploiement Granularisé, nous avons décidé de déployer trois répliques de chaque microservice afin d'éviter les potentiels goulots d'étranglement générés par un autre microservice. Par exemple, même si le microservice 4 s'exécute très rapidement, tant qu'il n'aura pas reçu la réponse du microservice 3, il mettra les autres requêtes en attente. Ce phénomène ne serait pas dû à une incapacité de répondre rapidement, mais au fait que les autres ne lui répondent pas assez vite. Contrairement aux trois autres déploiements, chaque tâche est répartie dans un microservice distinct, augmentant ainsi le nombre de communications et la dépendance entre chacun d'eux.

CHAPITRE 3

RÉSULTATS EXPÉRIMENTAUX

Cette recherche vise à étudier les effets de la conteneurisation et de la décomposition en microservices des différentes étapes d'une chaîne de traitement de vidéo 360°, en particulier sur l'utilisation des ressources et le délai de livraison d'un segment. Pour ce faire, nous avons mis en œuvre les différents déploiements présentés à la section 2.2. L'analyse porte sur l'utilisation des ressources (nombre de CPUs et quantité de mémoire RAM), le temps d'exécution des tâches de la chaîne de traitement, ainsi que le délai de livraison des segments vidéo.

Dans ce chapitre, nous décrivons en détail l'ensemble vidéo 360° utilisé, le matériel sur lequel les expérimentations ont été menées, ainsi que les outils et les métriques employés pour l'acquisition des données recherchées. Ensuite, nous présentons tous les résultats obtenus. Finalement, nous présentons une analyse et une discussion de ces résultats.

En particulier, nous étudions l'influence du nombre et du type d'instance sur les performances d'un déploiement. Puis, nous analyserons et comparerons de manière quantitative les différents déploiements.

3.1 Configurations et caractéristiques vidéo des tuiles

Nos déploiements ont pour objectif de permettre le traitement nécessaire à la diffusion interactive de vidéo 360°. Pour ce faire, il nous faut du contenu 360°. Nous avons utilisé une vidéo issue de notre inventaire, filmée avec notre caméra 360°, une Insta360 Pro 2, comme illustré à la figure 1.1.

Les déploiements ne réalisent pas de diffusion en direct ; les segments et les tuiles sont préparés à l'avance. Nous utilisons la même vidéo et les mêmes médias pour chaque test, ce qui nous permet de disposer d'une base identique pour chaque test sur chaque déploiement. Le tableau 3.1 présente les caractéristiques vidéos.

Tableau 3.1 Caractéristiques de la vidéo et des segments

Résolution vidéo entière	4K (3840×1920)
Résolution des tuiles	768×384
Nombre de tuiles	25 (5×5)
Codec	H.264 (AVC1)
Profil du codec	Baseline
Durée de la vidéo	2 min 21 s
Durée d'un segment	2 secondes
Nombre de segments	71 segments
Projection	Équirectangulaire

En résumé, nous utilisons une vidéo avec une résolution initiale de 4K. La vidéo est divisée en 25 tuiles (5 rangées et 5 colonnes) de résolutions inférieures à l'originale. Pour conserver la relation spatiale entre les tuiles, nous utilisons un manifeste MPEG-DASH (MPD) avec l'extension SRD. Les informations spatiales sont utilisées lors de la génération d'un segment pour ajuster la qualité des tuiles selon la région d'intérêt. La durée d'un segment est de 2 secondes et elle est basée sur les paramètres de test de Immich, Villas, Bittencourt & Madeira (2019).

3.1.1 Étapes pour la création des tuiles

Comme mentionné précédemment, nous ne faisons pas de diffusion en direct, mais traitons du contenu déjà préparé. Nous préparons notre contenu en trois étapes.

Premièrement, nous divisons notre vidéo en tuiles. Pour cela, nous utilisons FFmpeg et le filtre `crop` [1]. Un exemple de commande avec son fonctionnement en détail se trouve dans l'annexe II. Cette annexe présente un cas de découpage en 4 tuiles ; toutefois, nous adaptons cette méthode pour générer 25 tuiles, chacune ayant une position spatiale assignée. Cela aboutit à 25 vidéos indépendantes.

Deuxièmement, chaque tuile vidéo est découpée en segments de 2 secondes. Nous obtenons ainsi 71 segments vidéo par tuile, soit un total de 1775 segments vidéo (25 tuiles × 71 segments).

[1] https://ffmpeg.org/ffmpeg-filters.html#crop

Troisièmement, nous ré-encodons chaque segment à un débit de 20 kbps. Nous avons encodé à ce très bas débit afin de générer des tuiles segmentées de très basse qualité visuelle. La combinaison des segments originaux et de ceux ré-encodés à 20 kbps sera utilisée lors du raccordement des tuiles. Les tuiles de haute qualité, c'est-à-dire celles qui se trouvent dans la fenêtre de visualisation de l'utilisateur, seront les segments originaux ; les autres, qui ne sont pas dans cette fenêtre, seront ceux ré-encodés à 20 kbps. Nous avons choisi un débit très faible afin de réduire la taille du média autant que possible. La valeur du débit est arbitraire, nous voulions quelque chose de vraiment basse qualité afin d'aisément voir le raccordement. La figure 3.1 présente un résultat de raccordement des tuiles. Au total, nous obtenons 3550 segments vidéo (1775 segments $\times$ 2 niveaux de qualité). Pour cette transformation de débit, nous utilisons la commande FFmpeg présentée dans l'annexe III.

Enfin, nous avons créé le manifeste MPD avec l'extension SRD que nous allons utiliser pour nos tests.

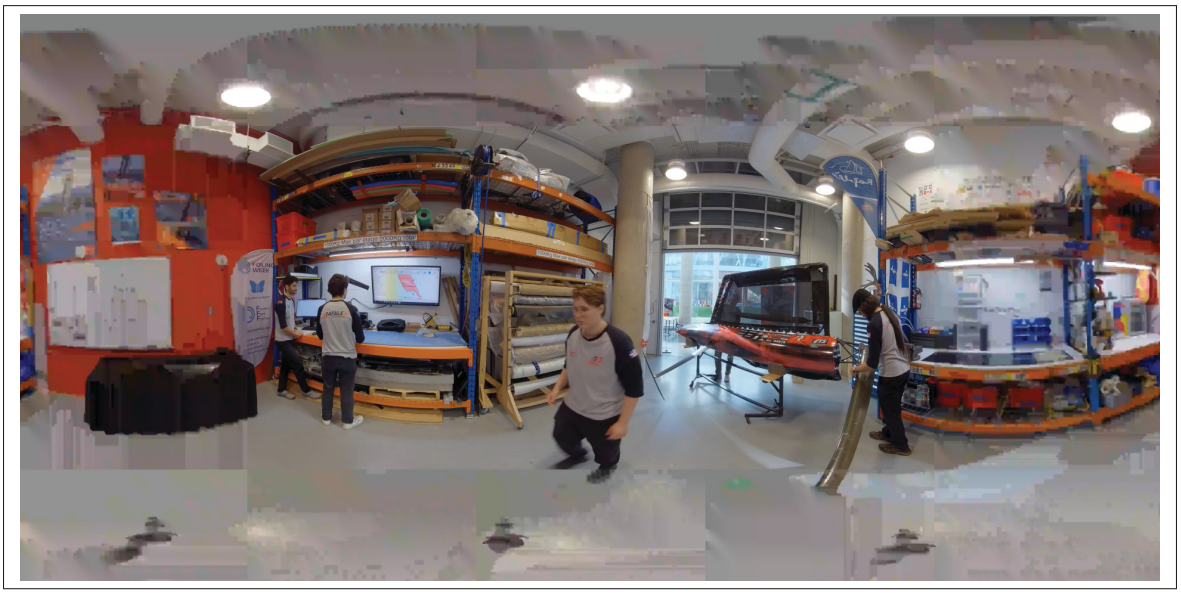

Figure 3.1 Rendu final du raccordement de tuiles multi-qualité dans un scénario où l'utilisateur regarde au centre de la vidéo

3.2 Acquisition des données et méthodes de comparaison

Afin de pouvoir quantifier la performance de nos déploiements, ou tout simplement mesurer les répercussions de chaque tâche de la chaîne de traitement, de manière individuelle, nous devons acquérir des données. Nous avons choisi quatre données qui serviront de points de comparaison : le nombre de cœurs CPU utilisés, la quantité de mémoire RAM, le délai de livraison d'un segment et le temps d'exécution d'une tâche.

L'acquisition du nombre de cœurs CPU et de la quantité de mémoire RAM nous permettra d'observer l'utilisation des ressources d'un déploiement. Le délai de livraison d'un segment nous permettra de déterminer quel déploiement exécute la chaîne de traitement le plus rapidement possible. La combinaison de ces trois données nous permettra de voir s'il existe un compromis entre le nombre de ressources utilisées et le délai de livraison d'un segment. C'est-à-dire, voir si un déploiement qui utilise plus de ressources donnera de plus bas délais, ou si d'autres facteurs sont en jeu, tels que la disposition des microservices, le partage des tâches, le matériel, etc.

Le temps d'exécution d'une tâche permettra d'identifier celles qui demandent le plus de temps et de déterminer si ce cela constitue un facteur de goulot d'étranglement. Il sera analysé en combinaison avec l'utilisation des ressources afin de mieux comprendre le comportement de chaque tâche au sein de la chaîne de traitement.

Cette section décrit plus en détail les données que nous acquérons, les outils qui nous permettent de les acquérir, ainsi que la méthode d'exécution de nos tests.

3.2.1 Acquisition des données sur nos déploiements

Nos déploiements étant entièrement constitués de conteneurs et de services, nous avons dû utiliser divers outils pour collecter les données recherchées. Ces outils sont présentés dans cette sous section.

3.2.1.1 cAdvisor

cAdvisor [2] (Container Advisor) est l'un des outils les plus connus de la communauté pour la collecte de données sur les ressources utilisées par les conteneurs. cAdvisor est capable de récupérer le nombre de cœurs CPU, la quantité de mémoire RAM et plusieurs autres informations des conteneurs en fonction.

cAdvisor est en réalité un conteneur qui s'ajoute au déploiement pour superviser les autres conteneurs. Dans notre déploiement multi-serveur, nous avons déployé un conteneur cAdvisor sur chaque serveur.

3.2.1.2 Jaeger

Jaeger [3] est un outil qui permet du traçage distribué. Le traçage distribué permet de diagnostiquer des architectures en microservices. Lorsqu'un événement survient (clic sur un bouton, sélection d'un lien, etc.), on mesure le temps d'exécution de celui-ci (Reinholds, 2024).

Par exemple, dès qu'on fait une requête pour un segment vidéo, on commence à chronométrer. Cet événement appelle ses sous-événements en chemin (sélection des tuiles, raccordement, etc.), et ceux-ci démarrent également un chronomètre pour leur propre exécution. Une fois que tous les sous-événements sont terminés, on note les délais de chaque événement ainsi que le délai total avec les dépendances. Ensuite, on envoie les résultats à Jaeger, qui les compile et les affiche.

Comme montré dans la figure 3.2, on peut voir un événement principal avec plusieurs sous-événements, qui, eux aussi, ont leurs sous-événements. Chacun a une durée propre.

Pour déclencher un événement auprès de Jaeger, nous utilisons OpenTelemetry [4]. OpenTelemetry est le cadriciel qui permet réellement le traçage (chronométrer, noter les sous-événements et l'arbre de ceux-ci, etc.). Le traçage distribué est une technique largement utilisée pour mesurer

[2] https://github.com/google/cadvisor

[3] https://www.jaegertracing.io/

[4] https://opentelemetry.io/

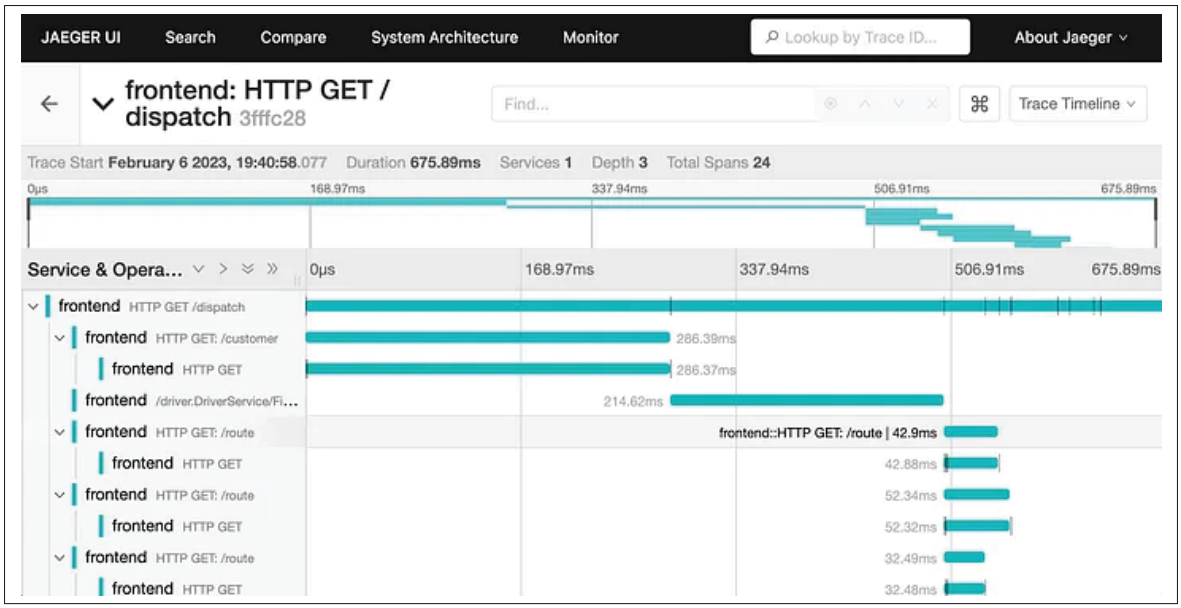

Figure 3.2 Exemple d'un événement avec des sous-événements dans Jaeger
Tirée de Shkuro (2021)

les performances dans une infrastructure nuagique qui utilise plusieurs machines à différents emplacements géographiques (OpenTelemetry Authors, 2025).

Pour utiliser OpenTelemetry avec Jaeger, nous utilisons le module "opentelemetry" fourni par PyPI. Nous établissons ensuite la connexion entre le conteneur Jaeger et les microservices.

OpenTelemetry et Jaeger sont capables de faire le traçage de manière automatique pour les structures API, telles que nos déploiements. Dès qu'on appelle l'URL d'une API, le processus de traçage démarre automatiquement et retourne le temps d'exécution pris pour la requête. Cependant, dans certains cas, nous avons besoin de sous-événements pour mieux inspecter et déboguer.

Nous utilisons Jaeger pour mesurer les temps d'exécution de nos principales étapes : sélection de la qualité des tuiles selon la région d'intérêt, la récupération des tuiles nécessaires, le raccordement des tuiles et la réécriture du segment. Ainsi, nous pouvons extraire des temps d'exécution précis, comme ceux présentés dans le tableau 3.3.

Il est important de noter que Jaeger nous donne uniquement les temps d'exécution des traitements, et non les temps de transmission réseau. Avec Jaeger, nous mesurons les délais d'exécution, mais rien au niveau de la communication réseau.

3.2.1.3 HTTP Archive Format

Le format de fichier HTTP Archive Format (HAR) [5] permet d'enregistrer les interactions entre le navigateur web et le site auquel il s'est connecté. Plus précisément, il permet de visualiser les requêtes réalisées par le navigateur, les objets concernés et les temps associés à ces requêtes. Ce format est généralement extrait depuis l'onglet réseau des outils de développement lorsque l'on inspecte une page.

Il existe énormément d'informations que l'on peut extraire d'un fichier HAR. Cependant, nous nous concentrons principalement sur deux mesures : le temps d'attente et le temps de réception. Le temps d'attente correspond au temps que le navigateur a attendu avant de recevoir une réponse. Dans notre cas, cela correspond au temps nécessaire pour que le déploiement génère un segment, en passant par les principales étapes définies précédemment. Le temps de réception correspond au temps requis pour lire la réponse reçue. Il existe d'autres délais enregistrés par le fichier, mais leur impact est minime dans le cadre de nos tests. La somme de ces deux temps constituera notre délai de livraison, que nous utiliserons pour observer la performance de nos déploiements.

Comme nous effectuons une requête par segment, nous disposons d'une granularité suffisante pour collecter plusieurs échantillons (71 segments, donc 71 échantillons).

3.2.1.4 Sauvegarde des résultats

Grâce à cAdvisor, Jaeger et le fichier HAR, nous pouvons extraire toutes les informations nécessaires pour quantifier la performance de nos déploiements. Cependant, ces outils ne conservent pas les résultats de manière permanente. La plupart sont volatiles ou temporaires.

[5] https://en.wikipedia.org/wiki/HAR_(file_format)

Pour sauvegarder les résultats de manière permanente, ou du moins suffisamment longtemps pour pouvoir les acquérir, nous ajoutons deux autres conteneurs à chaque déploiement : Prometheus [6] et Grafana [7]. Ces conteneurs servent principalement à compléter les lacunes fonctionnelles de cAdvisor. En effet, cAdvisor ne réalise aucune sauvegarde ni mise en mémoire des informations qu'il collecte. C'est Prometheus qui permet de stocker de manière permanente les résultats collectés par cAdvisor. Grafana, quant à lui, sert uniquement d'interface graphique pour visualiser les résultats sauvegardés par Prometheus. Nous avons utilisé le panneau "Swarm Stack Monitoring" fourni sur le site de Grafana.

Contrairement à cAdvisor, Jaeger conserve ses mesures assez longtemps pour que nous puissions extraire les temps nécessaires. En ce qui concerne le fichier HAR, il est possible de le récupérer de l'onglet réseau lorsque l'on inspecte la page du lecteur web. Les résultats sont sauvegardés dans le navigateur jusqu'au prochain rafraîchissement. Une fois la page rafraîchie, les données sont perdues et il faut relancer l'expérience.

À ce stade, les déploiements ne contiennent pas seulement des conteneurs liés à la génération de segments, mais aussi des conteneurs dédiés à la collecte des mesures. Jaeger, cAdvisor, Prometheus et Grafana effectuent une surveillance active, ce qui signifie que des ressources sont utilisées pour leur fonctionnement, entraînant ainsi une augmentation de la consommation de CPU et de RAM. Cependant, pour nos expériences, nous ne prenons en compte que les mesures liées aux conteneurs des déploiements. Nous n'incluons pas les ressources utilisées par les outils de collecte de mesures.

3.2.1.5 Extraction des résultats

L'extraction des résultats se fait à l'aide d'un script Python que nous avons développé. Nous notons le temps de début et de fin de l'expérience.

---

[6] https://prometheus.io/

[7] https://grafana.com/

Pour les temps d'exécution provenant de Jaeger, nous effectuons des appels à l'API pour chaque événement tracé entre les temps du test. Nous extrayons leur durée ainsi que le temps de début de l'événement. Il en va de même pour le fichier HAR ; le script Python le lit et extrait les informations pertinentes, avec le temps auquel la requête a été lancée.

Pour les résultats de cAdvisor, nous utilisons l'API web de Prometheus. Nous extrayons les valeurs de deux métriques principales : `container_cpu_usage_seconds_total` et `container_memory_working_set_bytes`.

La métrique `container_memory_working_set_bytes` indique la quantité de mémoire RAM (en bytes) utilisée par le conteneur. Cette métrique n'inclut pas les données mises en cache par le conteneur.

La métrique `container_cpu_usage_seconds_total` représente le temps cumulatif consommé par le CPU. Nous convertissons ce temps de manière à estimer le nombre de cœurs CPU utilisés. La formule de conversion que nous utilisons est inspirée d'un tableau de Grafana provenant de solution (2017), et se résume comme suit :

$$\text{irate}\left(\text{container_cpu_usage_seconds_total}\{\text{label} = \text{"container_name"}\}[T\ \text{m}]\right)$$

où `irate`, définissant le *instantaneous rate*, représente l'augmentation d'un compteur sur une courte fenêtre de temps, dans notre cas, 1 seconde. `label` et `container_name` servent à identifier le conteneur pour lequel on souhaite récupérer la métrique. Le segment $[T\ \text{m}]$ définit la durée de l'intervalle de temps T, en minutes (m), sur lequel les données sont récupérées.

Nous ne comprenons pas entièrement le fonctionnement de cette conversion, qui transforme le temps en nombre de cœurs, c'est pourquoi nous nous basons sur les expressions fournies par Grafana. Cependant, certaines explications plus détaillées existent, comme dans Gokan (2021). Dans notre cas, nous nous sommes limités à recueillir l'information telle qu'expliquée par la

communauté, car le travail sur les métriques relevait d'un domaine hors de notre champ de recherche.

Il existe plusieurs autres métriques fournies par Prometheus et cAdvisor, mais ces deux métriques sont suffisantes pour nos expériences. Il faut noter que les mesures sont extraites à chaque seconde pour ces métriques.

Après avoir extrait ces résultats, nous les sauvegardons dans une petite base de données SQLite. Chaque test dispose de sa propre base de données. La sauvegarde de ces résultats servira à la génération des graphiques et à l'interprétation des résultats.

3.2.2 Exécution des tests

Pour l'exécution des tests, nous utilisons un script Python qui lance un nombre déterminé de fureteurs simultanément. Nous avons utilisé un navigateur basé sur Firefox [8].

Pour nos déploiements, nous exécutons les tests dix fois par déploiement, par nombre de clients. Avec 71 segments par client, nous considérons disposer d'un nombre suffisant d'échantillons pour extraire l'information nécessaire.

3.3 Matériel et configuration

Cette section décrit les machines utilisées pour les tests que nous avons effectués, met en évidence l'impact du matériel sur nos expérimentations, et explique la manière dont nous gérons les déploiements composés de plusieurs microservices.

3.3.1 Configuration des machines et du réseau

Le tableau 3.2 présente les caractéristiques des machines utilisées pour nos divers tests. Les détails de l'utilisation des machines seront précisés dans la section 3.4.

[8] https://www.mozilla.org/fr/firefox/new/

Tableau 3.2 Caractéristiques des machines utilisées

Entité	**Ressource**	**Nom**	**Ressources disponibles**
Serveur nuagique	CPU	Standard E2ads v6	2 vCPU
	RAM	N/A	16 GB
	Système d'exploitation	Ubuntu Server 24.04	
	Réseau	Public – Est des États-Unis	
Serveur périphérique	CPU	AMD Ryzen 9 5900X	12 cœurs – 24 threads
	RAM	DIMM DDR4 Synchronous 2400 MHz	32 GB
	Système d'exploitation	Ubuntu Server 24.04	
	Réseau	Local	
Machine utilisateur	CPU	Intel(R) CORE(TM) i7-8565U CPU @ 1.80GHz	4 cœurs – 8 threads
	RAM	LPDDR3 Synchronous 2133 MHz	16 GB
	Système d'exploitation	Fedora 41 Workstation	
	Réseau	Local	

Le serveur périphérique et la machine utilisateur sont des machines auxquelles nous avions un accès physique. Le serveur nuagique est une machine virtuelle que nous avons déployée sur Azure [9]. Nous avons utilisé Azure, car nous avions besoin d'une machine située dans un réseau externe et géographiquement éloigné. En plaçant le serveur périphérique et la machine utilisateur dans un réseau local, avec le serveur nuagique dans un réseau distant, nous pouvons imiter une infrastructure périphérique-nuagique avec une machine proche de l'utilisateur et une autre plus éloignée.

On peut également constater que le serveur périphérique est plus puissant que le serveur nuagique. Cela s'explique pour deux raisons. Premièrement, il y a des restrictions sur les

[9] https://azure.microsoft.com/en-ca

Tableau 3.3 Utilisation des ressources et temps d'exécution pour un segment 2s (1 client - déploiement Granularisé)

Tâche	Nombre de CPUs	RAM (GB)	Temps d'exécution (s)
Sélection de la qualité des tuiles	0.002	0.17	7.269E-4
Récupération des tuiles	0.028	0.029	0.048
Raccordement des tuiles	3.43	0.77	1.313
Réécriture de segment	0.019	0.21	0.024

ressources disponibles via Azure. Deuxièmement, comme le montre le tableau 3.3 et les figures des déploiements de la section 2.2, les tâches des microservices déployés sur le serveur nuagique (identification de la qualité des tuiles et récupération des tuiles) sont peu exigeantes en termes de ressources, tandis que les tâches liées au traitement vidéo (raccordement des tuiles et réécriture de segments), exécutées sur le serveur périphérique, le sont davantage. Pour tenir compte de ces deux aspects, nous avons donc choisi d'utiliser un serveur périphérique plus puissant que le serveur nuagique. Cette configuration est également observée dans le travail de Shen *et al.* (2021).

3.3.1.1 L'impact du matériel sur FFmpeg

Comme mentionné à la section 2.1.3, nous utilisons FFmpeg avec le filtre `xstack` pour faire le raccordement de nos tuiles en différentes qualités. Comme le montre le tableau 3.3, le raccordement des tuiles est l'opération la plus longue. Elle prend près des trois quarts de la durée d'un segment lorsqu'on l'exécute sur notre serveur périphérique.

Nous étions curieux de voir si le matériel avait un impact sur le temps d'exécution de cette commande, c'est-à-dire, si un plus grand nombre de cœurs permettrait de raccorder plus rapidement, et si oui, dans quelle mesure. Nous avons donc réalisé un test qui consistait simplement à raccorder 25 tuiles dans un environnement non virtualisé, soit directement sur le matériel. Les médias utilisés dans l'expérience sont définis à la section 3.1 et au tableau 3.1. Certaines tuiles étaient codées en haute qualité, tandis que d'autres étaient en basse qualité. Nous avons effectué 10 tests de raccordement identiques sur différents matériels et avons pris

la moyenne de ces résultats. La figure 3.3 montre le temps d'exécution de la commande de raccordement avec 25 tuiles sur différents matériels.

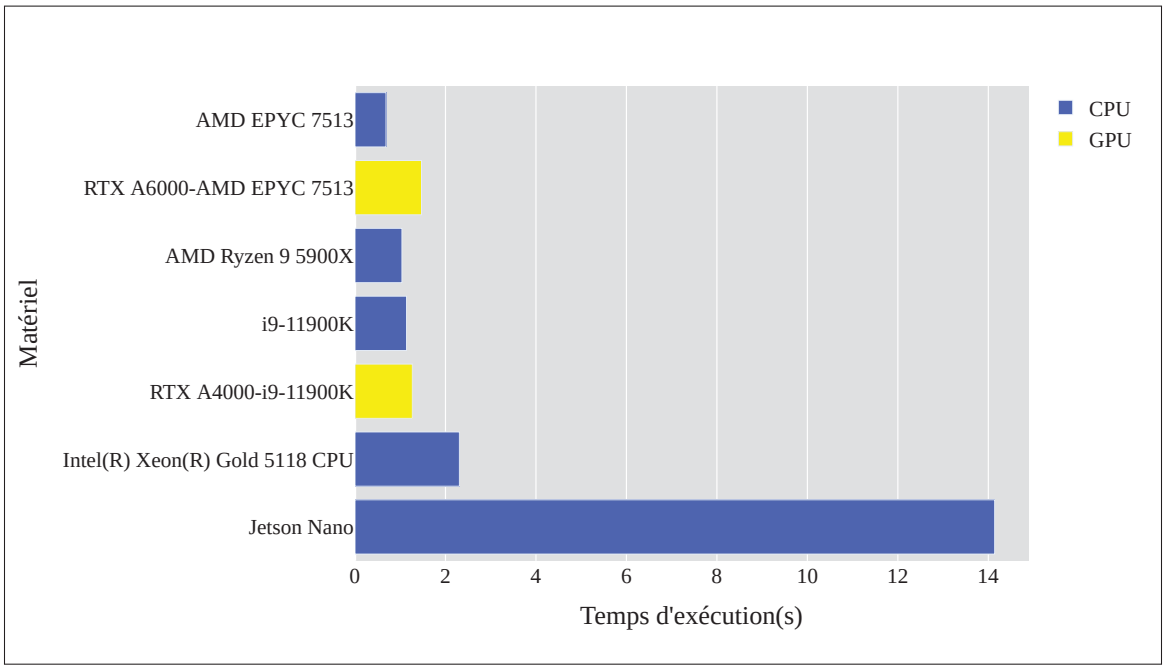

Figure 3.3 Temps d'exécution pour le raccordement des tuiles avec FFmpeg selon différents matériels

Un constat intéressant est que l'utilisation du GPU nuit au temps d'exécution (RTX A4000 et A6000). Cela s'explique par le fait que certains filtres ne s'exécutent pas sur le GPU (NVIDIA Developer Blog, 2019), comme le filtre `xstack`, semblerait-il. La latence observée sur le GPU par rapport au CPU s'explique par le besoin de faire des transferts de données entre la mémoire du système et la mémoire du GPU (NVIDIA Developer Blog, 2019). Afin que le GPU puisse effectuer l'encodage, deux transferts sont nécessaires : un premier transfert des médias décodés vers le CPU pour l'application du filtre, puis un second transfert des tuiles raccordées vers le GPU pour l'encodage vidéo.

Sinon, il est à constater que notre serveur périphérique utilise le processeur `AMD Ryzen 9 5900X` au lieu du `AMD EPYC 7513`, en raison de restrictions de disponibilité de ce dernier.

3.3.2 Orchestrations des répliques de microservices

Lors de notre recherche, nous avons utilisé deux méthodes pour permettre l'intercommunication entre les conteneurs. Nous avons utilisé Docker Swarm et des conteneurs Docker avec une passerelle d'API.

Au début, nous avions l'intention de n'utiliser que Docker Swarm comme orchestrateur. Grâce à sa fonctionnalité de service qui gère la communication et le balancement de charge entre les différents conteneurs, cela nous aurait enlevé une responsabilité. Cependant, en raison de contraintes réseau et de configurations, nous avons utilisé Docker Swarm seulement pour les tests des instances sur du matériel (section 3.4.1).

Pour les tests de comparaison entre les déploiements (section 3.4.2), nous voulions tester dans une infrastructure plus fidèle, c'est-à-dire un serveur périphérique et un serveur nuagique sur des réseaux différents. Étant donné que le serveur nuagique n'était pas dans un réseau que nous contrôlions, nous ne pouvions pas utiliser Docker Swarm. Donc, pour imiter le comportement de Docker Swarm, nous avons utilisé des conteneurs avec une passerelle API. Comme le montre la figure 3.4, la passerelle API est un conteneur Nginx [10] qui assigne les requêtes aux répliques des microservices selon la charge ou l'ordre. C'est ainsi que nous gérions la communication et le balancement de charge entre les répliques de microservices.

3.4 Résultats et discussions

Nous allons à présent aborder en détail les résultats expérimentaux obtenus au cours de nos différentes évaluations. Cette section vise à montrer l'impact et les comportements globaux des déploiements et de leur configuration.

Dans un premier temps, nous analyserons l'influence du nombre d'instances déployées sur la performance. Nous observerons aussi les différences entre les types d'instance définis dans la section 2.3.1.

[10] https://nginx.org/

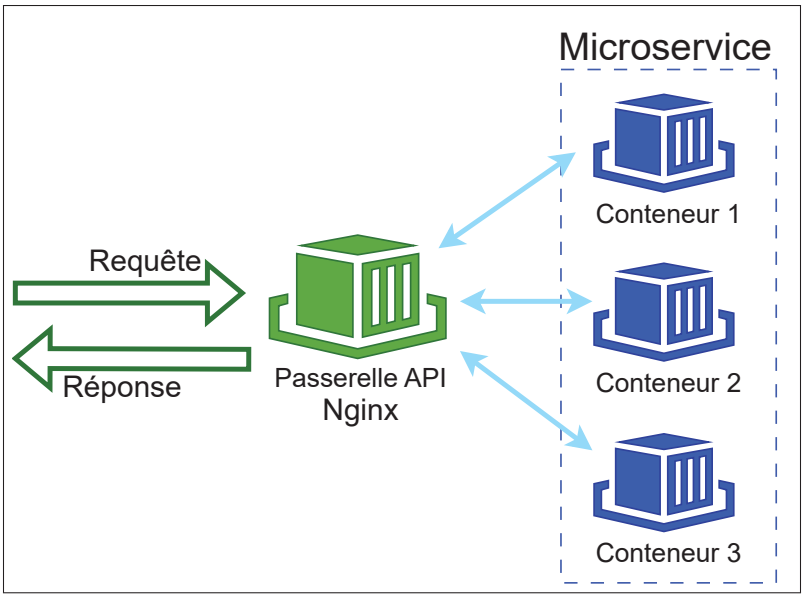

Figure 3.4 Passerelle API pour les conteneurs d'un microservice

Ensuite, nous procéderons à une comparaison quantitative des différents déploiements présentés. Nous nous appuierons sur les métriques de performance recueillies par les outils présentés. Ces comparaisons permettront de distinguer l'efficacité des configurations, mais aussi d'identifier les compromis entre le délai de livraison et le coût en ressources.

3.4.1 Impact du nombre d'instances

Lors des premiers tests avec plusieurs clients en simultané sur nos déploiements, nous avons observé des délais de livraison étrangement élevés dès que nous traitions plus d'un client à la fois. En analysant de plus près ces délais, nous avons constaté que le temps de livraison d'un segment était proportionnel au nombre de clients, lorsqu'un seul conteneur répondait aux requêtes. Par exemple, lorsque nous testions avec cinq clients en simultané, le délai de livraison d'un même segment était au minimum cinq fois plus long que pour un seul client. Nous étions donc tombés sur un goulot d'étranglement provoquant des délais excessifs. Cela nous a amené à nous questionner sur l'impact du nombre d'instances sur le fonctionnement global de nos déploiements.

Cette expérience vise à évaluer l'impact du type et du nombre d'instances, principalement sur les délais de livraison, et secondairement sur l'utilisation des ressources. L'expérience consiste à faire varier à la fois le type d'instance, le nombre d'instances et le nombre de clients demandant des segments simultanément. Le nombre d'instances est arbitraire. Nous cherchions tout simplement à faire ressortir les impacts de manière générale. C'est pourquoi nous avons fait un choix abrupt de 3 et 10 répliques, au lieu d'adopter une approche plus graduelle (1, 2, 3...).

Les tests ont été réalisés sur deux machines virtuelles : l'une disposant de 16 cœurs et de 12 GB de RAM agissant comme le serveur périphérique, et l'autre de 4 cœurs et 4 GB de RAM qui agit comme le serveur nuagique. Les répliques sont déployées à l'aide de services Docker Swarm, permettant de générer plusieurs conteneurs ou travailleurs web. Le déploiement utilisé pour ces tests est le déploiement Proximité. Nous avons choisi ce déploiement, car il exécute les tâches liées au traitement des segments sur un seul serveur tout en nécessitant des requêtes vers une autre machine pour accéder aux médias. Il ne s'agit ni d'un déploiement entièrement monolithique, ni d'un système trop granulé — un bon compromis pour évaluer des tâches exigeantes en ressources.

Les figures 3.5, 3.6 et 3.7 illustrent respectivement l'impact des instances et du nombre des instances sur le délai de livraison, l'utilisation du CPU et l'utilisation de la RAM en situation multi-clients.

Sur le plan des délais de livraison (figure 3.5, les résultats montrent qu'une augmentation du nombre d'instances permet de réduire les temps de transmission en situation de multi-clients. En revanche, l'utilisation moyenne du CPU et de la mémoire RAM augmente avec le nombre d'instances. Plus le nombre d'instances est élevé, plus la consommation des ressources est importante.

Nous pouvons ainsi conclure que, pour assurer un service efficace à plusieurs clients simultanément tout en maintenant des délais de livraison raisonnables, il est nécessaire de déployer plusieurs instances. Il est cependant difficile de fixer une relation optimale entre le nombre de clients et le nombre de répliques par microservice, puisque les délais de livraison avec 3 répliques sont

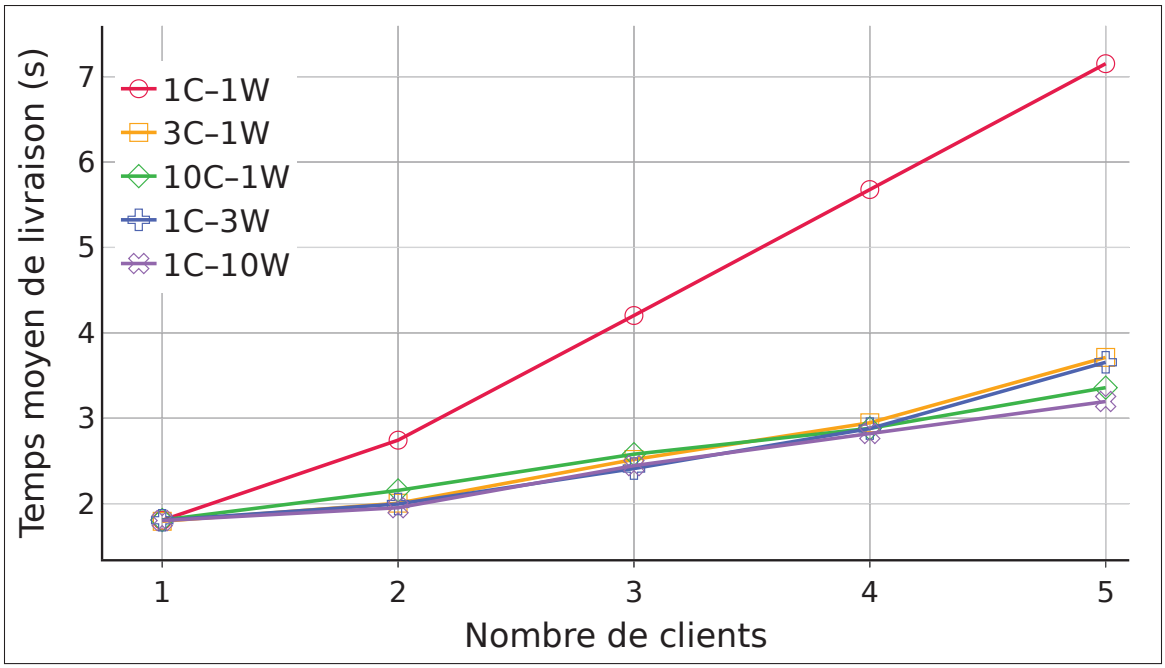

Figure 3.5 Temps moyen de livraison d'un segment selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité

très proches de ceux obtenus avec 10. Cela s'explique par le manque de cœurs CPU disponibles sur la machine virtuelle périphérique. Néanmoins, une approche logique serait de prévoir au minimum une réplique de chaque microservice par client. Cette stratégie permet de limiter les risques de goulots d'étranglement dus à l'accumulation des requêtes dans les files d'attente.

3.4.1.1 Impact du type d'instance

Tel que mentionné, les figures 3.5, 3.6 et 3.7 comparent également les performances de deux types d'instances : les conteneurs et les travailleurs web.

En termes de délais de livraison (figure 3.5), nous n'observons pas de différence significative entre l'utilisation de conteneurs et celle de travailleurs web. Les points représentés sur les courbes sont trop proches pour mettre en évidence un avantage clair de l'un ou l'autre.

Cependant, concernant l'utilisation des cœurs CPU et de la mémoire RAM, certains comportements particuliers émergent. La figure 3.6 montre clairement que 10 conteneurs consomment moins de

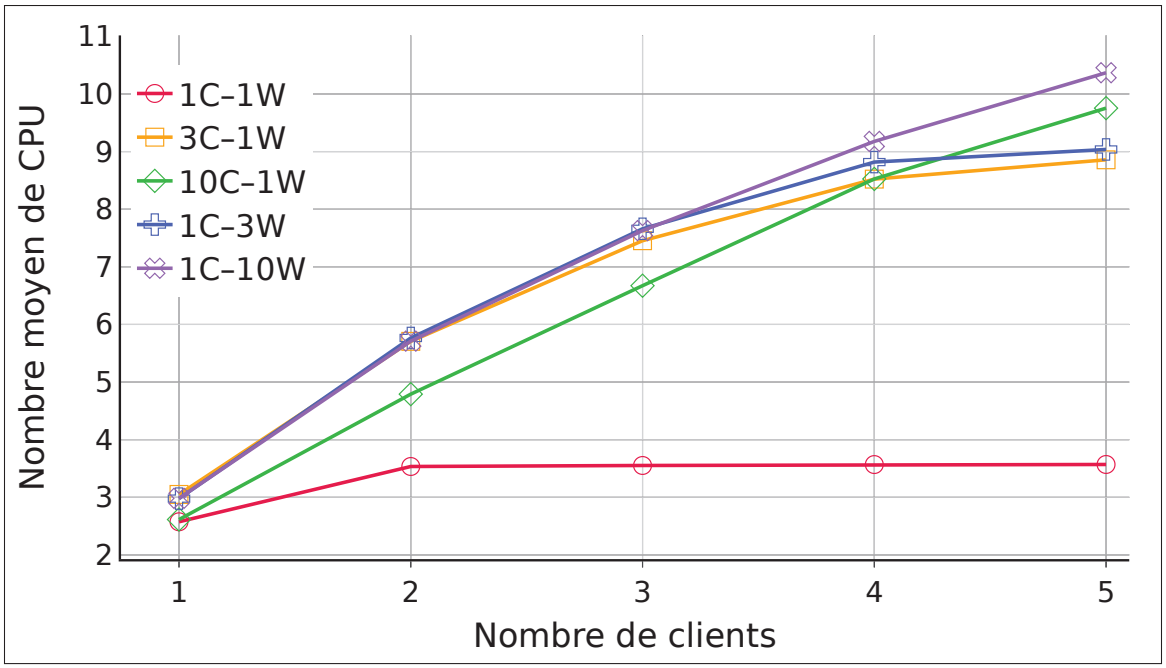

Figure 3.6 Nombre moyen de CPUs utilisés pendant la diffusion selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité

cœurs que 10 travailleurs web. Une tendance similaire, bien que beaucoup plus subtile, peut également être observée entre seulement 3 conteneurs et 3 travailleurs web, plus précisément lorsque nous sommes à 4 ou 5 clients.

Il est difficile d'affirmer avec certitude la raison de ces comportements. Cependant, notre hypothèse est qu'ils sont liés au changement de contexte des `threads` nécessaire entre les travailleurs web.

Il est également à noter que les travailleurs web sont des processus qui s'exécutent dans l'environnement du conteneur, donc limités aux ressources disponibles de ce dernier. Comme l'indique la légende, les travailleurs web sont sur un seul conteneur, et les ressources de celui-ci ne sont pas définies. Par défaut, elles sont laissées à la discrétion de Docker, qui n'applique pas de limite si aucune n'est indiquée. Le conteneur utilise autant que le système d'exploitation lui en fournit (Documentation, 2025). De plus, comme nous n'avons pas configuré explicitement le nombre de threads par travailleur web, nous nous retrouvons par défaut avec un seul thread

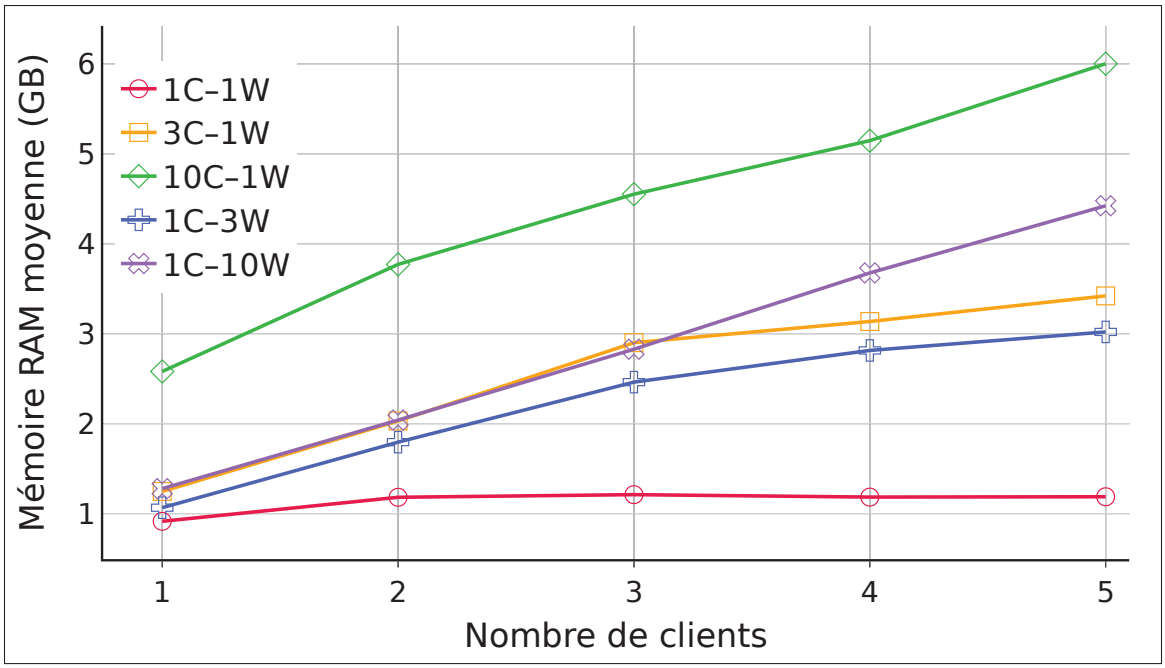

Figure 3.7 Mémoire RAM moyenne utilisée pendant la diffusion selon le nombre de clients, par conteneur (C) ou travailleur Web (W), avec déploiement Proximité

par travailleur (uWSGI Project, 2025). Cela nécessite de nombreux changements de contexte, générant ainsi davantage de *overhead* et une consommation CPU plus élevée (squashlabs, 2023).

Les conteneurs, contrairement aux travailleurs web, sont des entités indépendantes qui réservent leurs propres ressources. Ainsi, un conteneur exécutant un travailleur web avec un seul thread n'a pas de ressources à partager. Elles sont attribuées par le système d'exploitation, ce qui évite les changements de contexte et, par conséquent, réduit le *overhead*.

De plus, sur la figure 3.6, nous pouvons constater que, pour 5 clients, 10 conteneurs dépassent de presque 1 CPU les courbes de 3 conteneurs ou 3 travailleurs web. Cela s'explique par le nombre d'instances. Plus il y a de répliques, plus on utilise de ressources . Alors, à moins de répliques, nous consommons moins de ressources, et même que nous tendons vers une stabilisation de celle-ci. On pourrait décrire cela comme une limitation de ressources, où le système se refuse le droit d'utiliser davantage de ressources, même s'il y en a de disponibles. Ceci est encore plus vrai lorsqu'on observe la mémoire à la figure 3.7, où nous observons clairement une sous-utilisation

de ressources. Malgré les 16 GB disponibles, le système se limite à 3,5 GB pour 3 conteneurs, alors que pour 10 conteneurs, il utilise jusqu'à 6 GB. Cela montre clairement l'impact du nombre de répliques sur la consommation des ressources et explique pourquoi, à 5 clients, 10 conteneurs utilisent plus de ressources que 3 conteneurs, malgré une charge identique.

Davantage sur l'utilisation de la mémoire, les différences entre conteneurs et travailleurs web sont beaucoup plus évidentes sur la figure 3.7, on peut voir que les travailleurs web tendent à consommer moins de mémoire que les conteneurs. Par exemple, pour 10 instances, les travailleurs web utilisent en moyenne 37, 80% de mémoire en moins que les conteneurs.

Deux facteurs principaux expliquent cet écart. Premièrement, les travailleurs web partagent une partie de leur mémoire (uWSGI Documentation, n.d.), en particulier dans le cas des travailleurs uWSGI [11] que nous utilisons. Deuxièmement, les conteneurs nécessitent une consommation supplémentaire pour assurer leur propre fonctionnement, c'est-à-dire la gestion et l'exécution de l'environnement du conteneur, en plus de la charge de la tâche associée au microservice (Server Fault, 2014). Donc, plus de conteneurs signifie plus de mémoire à réserver.

3.4.2 Comparaison quantitative entre les déploiements

Comme le démontre l'expérience précédente, il est nécessaire de disposer de plusieurs répliques d'une instance afin de pouvoir répondre efficacement à plusieurs clients simultanément. Sinon, une file d'attente se forme, créant un goulot d'étranglement et engendrant ainsi de grands délais.

Sur les machines définies dans le tableau 3.2, nous avons déployé des répliques de conteneurs selon les détails fournis dans le tableau 2.2. Il est à noter que la machine utilisateur ne dispose d'aucune réplique assignée, car elle ne participe pas à la génération de segments. Son rôle se limite à lancer et gérer les clients. Nous nous sommes limités à 3 répliques pour la plupart des services, ce choix ayant été fait de manière arbitraire, selon les résultats de l'expérience précédente.

[11] https://uwsgi-docs.readthedocs.io/en/latest/

En raison de restrictions réseau, nous avons décidé d'utiliser des conteneurs avec un travailleur web comme méthodologie de réplication.

Les figures 3.8, 3.9 et 3.10 comparent les performances des quatre déploiements.

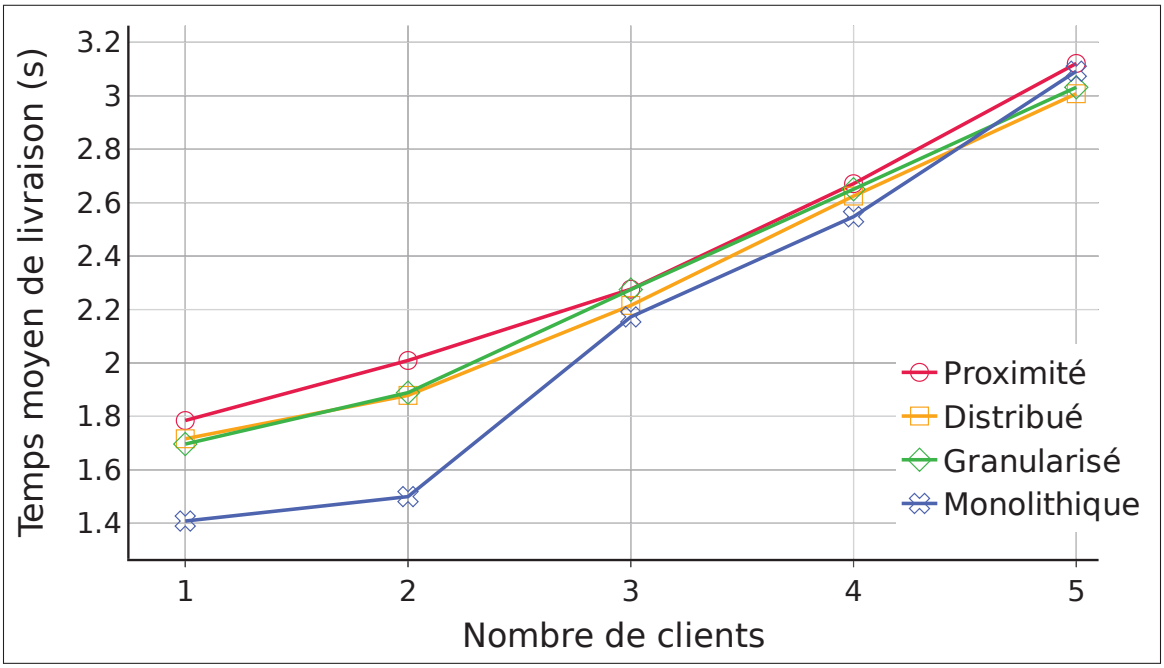

Figure 3.8 Temps moyen de livraison d'un segment en fonction du nombre de clients et du déploiement

D'après la figure 3.8, le déploiement Monolithique présente les délais de livraison les plus faibles jusqu'à 4 clients. Le déploiement Proximité affiche les délais les plus élevés pour tous les clients. Les déploiements Granularisé et Distribué montrent des performances similaires pour les deux premiers clients, mais à partir de 3 clients, le déploiement Distribué devient légèrement plus rapide et se révèle même être le plus performant à 5 clients.

Des écarts notables apparaissent jusqu'à 4 clients entre les déploiements Monolithique et Proximité. Ces différences s'expliquent principalement par le mode d'accès aux tuiles : dans le déploiement Monolithique, les tuiles sont récupérées localement dans le conteneur, tandis que dans le déploiement Proximité, elles doivent être sollicitées auprès d'un conteneur situé sur un réseau distant. Les communications réseau introduisent des délais supplémentaires. Un

Figure 3.9 Nombre moyen de CPUs utilisés pendant le visionnement en fonction du nombre de clients et du déploiement

phénomène similaire est observé dans le déploiement Granularisé. Contrairement au déploiement Monolithique, où les trois conteneurs sont co-localisés, les microservices du déploiement Granularisé sont répartis sur deux machines (chacune exécutant six conteneurs). Cette répartition entraîne davantage de communications inter-machines et contribue à l'augmentation des temps de livraison.

De manière générale, aucun des déploiements ne parvient à respecter l'objectif de livraison de 2 secondes par segment dès que l'on dépasse 2 clients. Le déploiement Proximité franchit même légèrement ce seuil pour 2 clients. Comme le processus de génération des segments ne parvient pas à suivre leur durée réelle, l'utilisateur est exposé à des pauses perceptibles dégradant ainsi l'expérience de diffusion.

Concernant l'utilisation des CPU (figure 3.9), les différents déploiements présentent une consommation globalement similaire. Jusqu'à trois clients, le déploiement Monolithique utilise légèrement moins de cœurs CPU, tandis que les trois autres déploiements affichent des valeurs

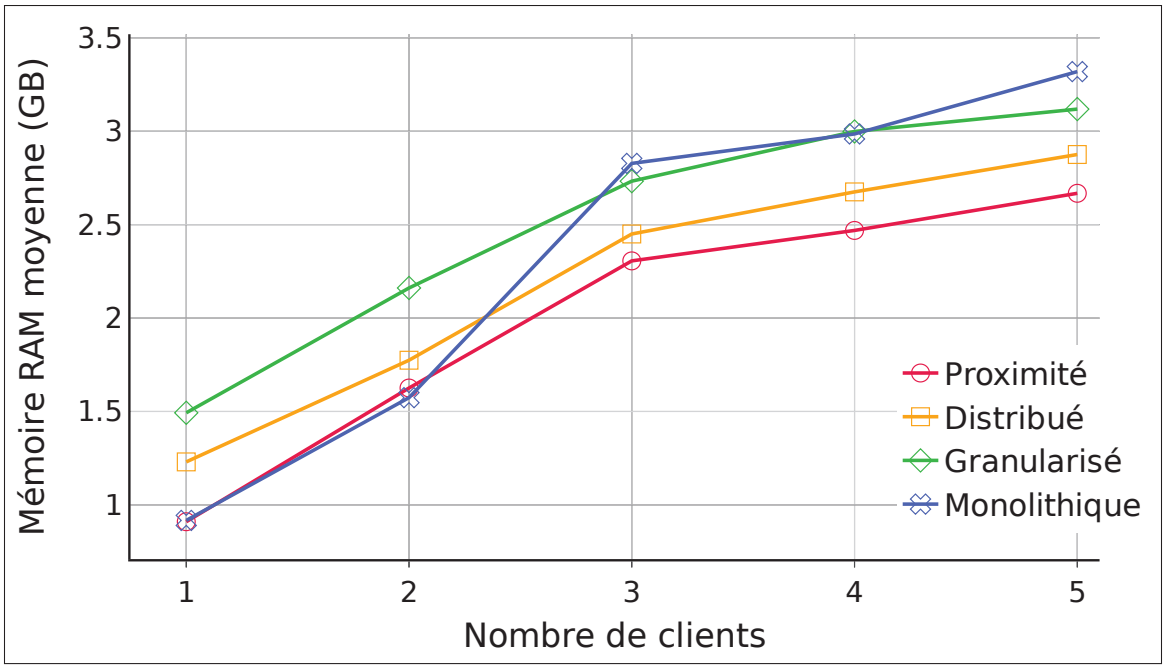

Figure 3.10 Mémoire RAM moyenne utilisée pendant le visionnement en fonction du nombre de clients et du déploiement

supérieures et très proches les unes des autres. Au-delà de trois clients, la tendance s'inverse et le déploiement Monolithique devient celui qui consomme le plus de cœurs CPU. Ce comportement s'explique par le nombre total de cœurs disponibles à travers tous les serveurs. Le déploiement Monolithique ne travaille que sur le serveur périphérique, alors que les trois autres déploiements utilisent également un serveur nuagique, donc 2 cœurs supplémentaires dans le total disponible. Moins de cœurs disponibles signifie moins de consommation possible pour le déploiement Monolithique au début. Après trois clients, les ressources limitées entraînent davantage de contention, ce qui augmente l'utilisation CPU pour le déploiement Monolithique, et inversement pour les trois autres déploiements.

En ce qui concerne l'utilisation de la mémoire (figure 3.10), les quatre déploiements présentent un comportement globalement similaire, bien que leurs courbes restent distinctes. Le déploiement Proximité apparaît comme le plus efficace en termes de consommation mémoire, tandis que les déploiements Monolithique et Granularisé en consomment davantage. Nos résultats suggèrent que les architectures comportant soit un nombre très élevé (Granularisé), soit un nombre très

faible (Monolithique) de microservices tendent à induire une consommation mémoire plus importante dans des scénarios multi-clients, contrairement à une approche plus distribuée qui parvient à mieux équilibrer cette charge. Ce comportement est également faiblement observable pour l'utilisation du CPU. À 4 et 5 clients, le déploiement Proximité et le déploiement Distribué utilisent légèrement moins de cœurs.

3.4.2.1 Impact sur le matériel

Les figures 3.9 et 3.10 mettent également en évidence les limites de la configuration actuelle de nos machines. On peut observer une certaine contention des ressources, ainsi qu'une sous-utilisation.

D'après le tableau 3.3, le raccordement des tuiles se distingue comme la tâche la plus intensive en ressources et la plus longue à exécuter. Pour atténuer les risques de goulots d'étranglement liés à ces délais d'exécution élevés, nous avons choisi de déployer trois instances de chaque microservice sur le serveur en périphérie. Le tableau 3.3 indique qu'un conteneur chargé du raccordement utilise un peu moins de 3.5 cœurs CPU en moyenne. Par conséquent, trois conteneurs peuvent consommer ensemble, en moyenne jusqu'à environ 10.5 cœurs, comme le montrent clairement les courbes de la figure 3.9. Nous atteignons ainsi une utilisation quasi maximale du processeur 12 cœurs du serveur périphérique, le reste étant probablement réservé au système d'exploitation et aux autres conteneurs qui travaillent sur l'acquisition des données.

Comme mentionné précédemment, cette disponibilité limitée en ressources CPU entraîne des temps de livraison plus élevés. Nous pensons que cette contention explique pourquoi nous n'avons pas pu maintenir un temps de livraison par segment inférieur à 2 secondes au-delà de deux clients. Ces résultats révèlent une contention clairement liée au nombre de CPUs, désignant ce dernier comme la ressource la plus sollicitée.

Pour y remédier, deux solutions peuvent être mises en place. En premier lieu, une augmentation du nombre de cœurs devrait permettre de réduire le temps de livraison et d'assurer une diffusion en temps réel pour plus de deux clients. Du moins, c'est ce que l'on peut constater dans

la figure 3.3, où le processeur AMD EPYC 7513, doté de 32 cœurs et 64 threads, semble encoder bien plus rapidement que le CPU du serveur périphérique. En deuxième lieu, pour forcer davantage l'utilisation du CPU, plus de répliques pourraient être déployées. La figure 3.6 montre que, pour trois répliques, après quatre clients, des plateaux commencent à se former. Un comportement semblable s'observe sur la figure 3.9, où l'on voit chuter brusquement la croissance du nombre de cœurs utilisés. Cela signifie que les déploiements tendent à ne pas utiliser plus de ressources que nécessaire, même si celles-ci sont disponibles. Ainsi, pour forcer davantage l'utilisation du CPU, il faudrait augmenter le nombre de répliques.

Il en est de même pour l'utilisation de la mémoire. Selon la figure 3.10, l'utilisation de la mémoire ne présente aucun signe de contention, les conteneurs se limitent au strict nécessaire. Bien que 32 GB de mémoire soient disponibles, les déploiements n'en utilisent au maximum qu'environ 3,25 GB.

Ces comportements apparaissent également dans les figures 3.6 et 3.7. On y constate que l'utilisation du CPU et de la mémoire varie selon le nombre de répliques déployées.

3.4.2.2 Les délais de livraison plus en détail

Les résultats des délais de livraison, illustrés à la figure 3.8, représentent la somme du temps moyen d'attente et du temps moyen de réception. Tel que mentionné dans la section 3.2.1.3, le temps d'attente correspond au délai nécessaire pour que le premier octet de la réponse arrive, autrement dit, le temps que les serveurs prennent pour générer le segment. Le temps de réception, quant à lui, correspond au délai nécessaire pour recevoir la réponse côté navigateur, c'est-à-dire le temps requis pour télécharger le segment localement.

Le temps de réception dépend de plusieurs facteurs, notamment des conditions du réseau, telles que la vitesse de connexion, le fournisseur, le nombre d'appareils connectés simultanément, etc. Or, tel que défini dans la table 3.2, la machine client et le serveur périphérique sont situés sur un réseau domestique, sur lequel nous n'avons pas un contrôle total. Il est donc pertinent d'analyser

plus en profondeur les délais de livraison et d'évaluer l'impact relatif des temps d'attente et de réception.

Les figures 3.11, 3.12, 3.13 et 3.14 présentent les délais de livraison, décomposés en temps d'attente et temps de réception, pour chaque déploiement.

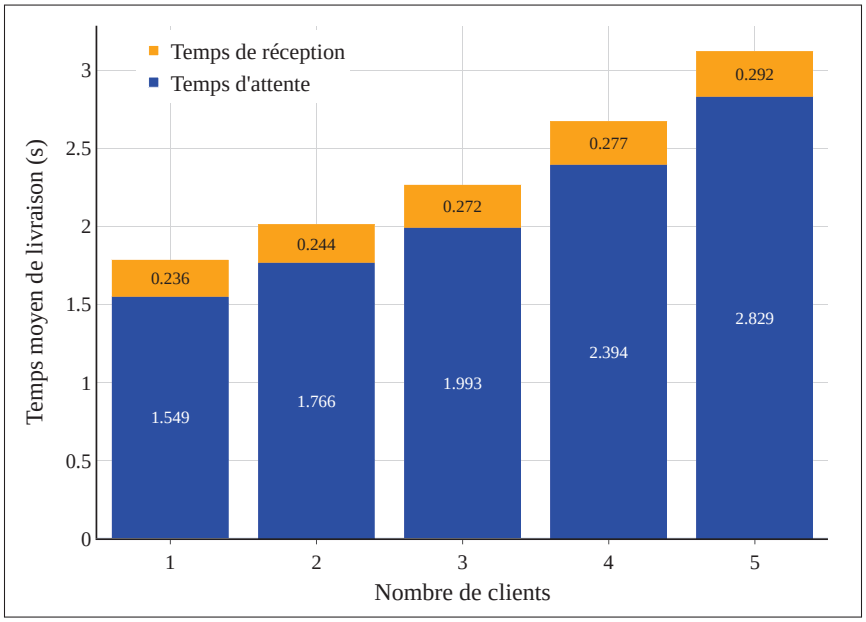

Figure 3.11 Temps d'attente et réception moyen par nombre de clients - déploiement Proximité

Trois constats intéressants peuvent être observés. Premièrement, si les clients étaient dans un réseau plus contrôlé, avec un temps de réception beaucoup plus faible, il serait probablement possible de servir jusqu'à trois clients simultanément.

Deuxièmement, le temps de réception semble augmenter légèrement avec le nombre de clients pour les déploiements Proximité, Distribué et Granularisé. En revanche, pour le déploiement Monolithique, l'augmentation est beaucoup plus marquée, passant d'environ 0,235 seconde à au moins 0,3 seconde. Un constat similaire peut être fait pour les temps d'attente : dans les déploiements distribués, les temps d'attente par client restent relativement similaires et augmentent de manière progressive, tandis que, pour le déploiement Monolithique, on observe des augmentations beaucoup plus brusques entre 2 et 3 clients.

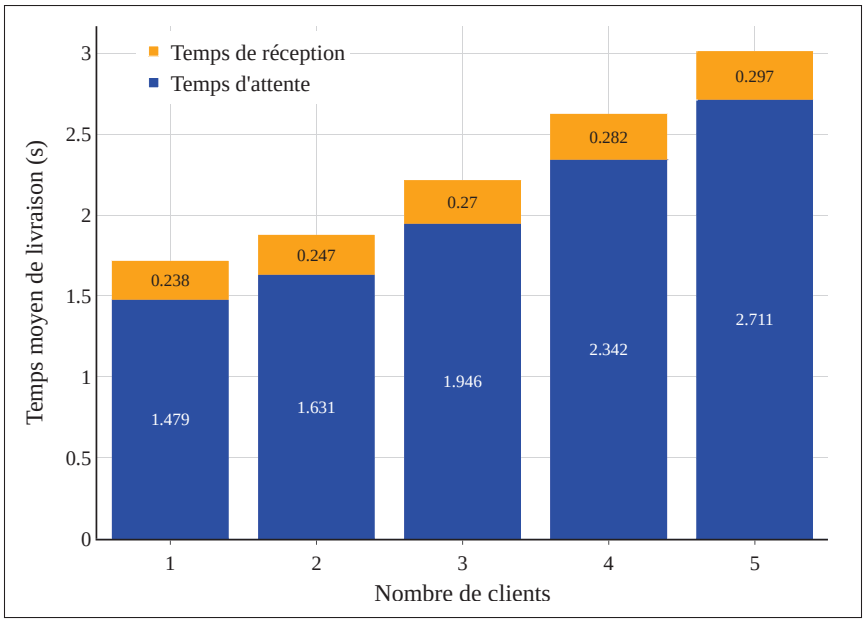

Figure 3.12 Temps d'attente et réception moyen par nombre de clients - déploiement Distribué

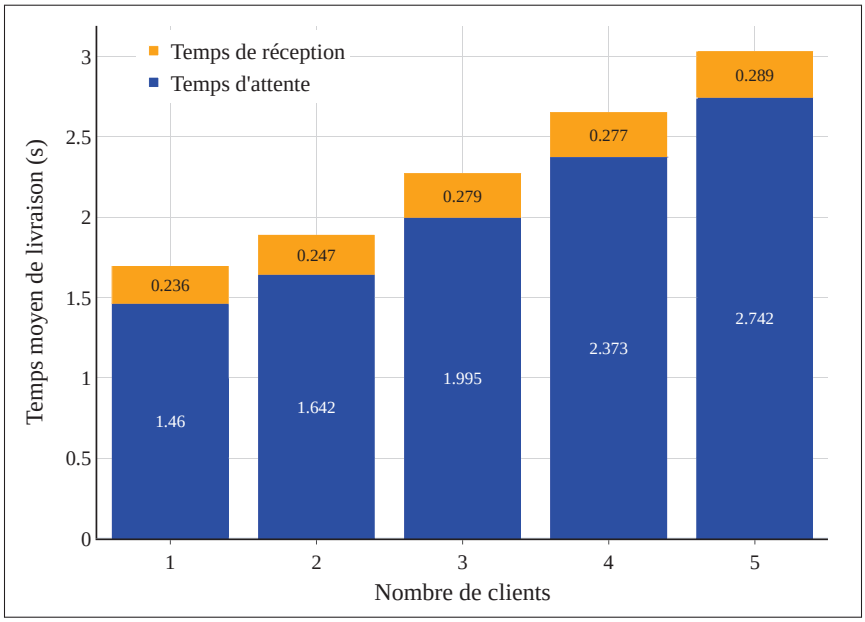

Figure 3.13 Temps d'attente et réception moyen par nombre de clients - déploiement Granularisé

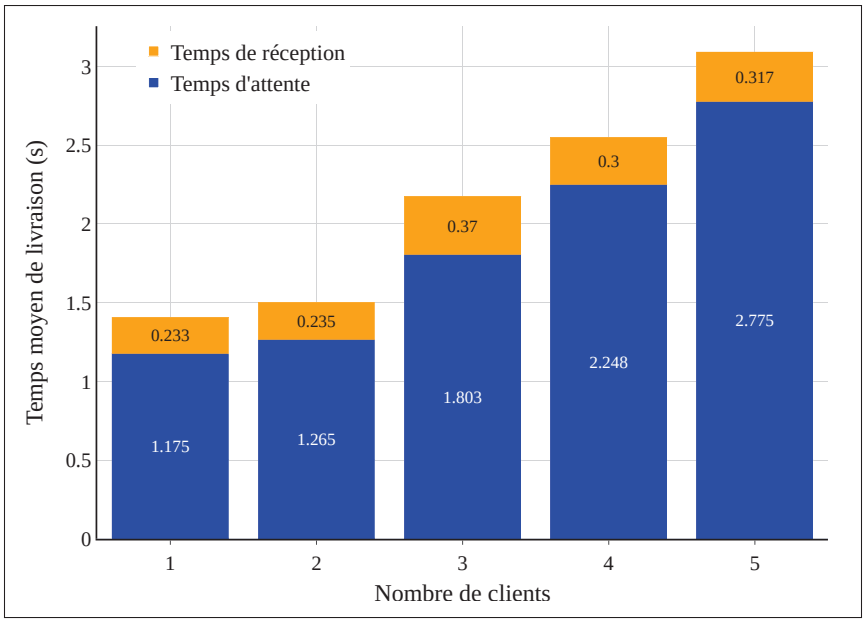

Figure 3.14 Temps d'attente et réception moyen par nombre de clients - déploiement Monolithique

Troisièmement, il est intéressant de constater que les temps d'attente ne diffèrent presque pas entre les quatre déploiements pour 4 et 5 clients. Ce seuil de 3 correspond au nombre de répliques de chaque microservice déployé sur la machine périphérique, démontrant ainsi une fois de plus l'impact direct du nombre de répliques sur les délais de livraison.

3.4.2.3 Variabilité des valeurs recueillies

Étant donné que nous avons réalisé 10 essais par client, nous avons décidé de générer des diagrammes à moustaches afin d'observer les fluctuations des valeurs recueillies. Ces fluctuations nous permettent de mieux déterminer si les déploiements sont prévisibles et stables.

Les figures 3.17, 3.15 et 3.16 présentent des diagrammes à moustache pour les délais de livraison, le nombre de cœurs CPU et la quantité de mémoire RAM utilisée pour chaque déploiement lors des 10 essais par client.

Il est à noter qu'il n'y a pas seulement 10 valeurs par nombre de clients pour les graphiques des délais de livraison, contrairement aux graphiques du nombre de cœurs CPU et de mémoire RAM. On affiche la variabilité entre chaque client. Par exemple, pour un seul client, nous avons 10 valeurs (1 client par essai, multiplié par 10 essais, soit 10 valeurs). En revanche, pour cinq clients, nous avons 50 valeurs (5 clients par essai, multiplié par 10 essais, soit 50 valeurs).

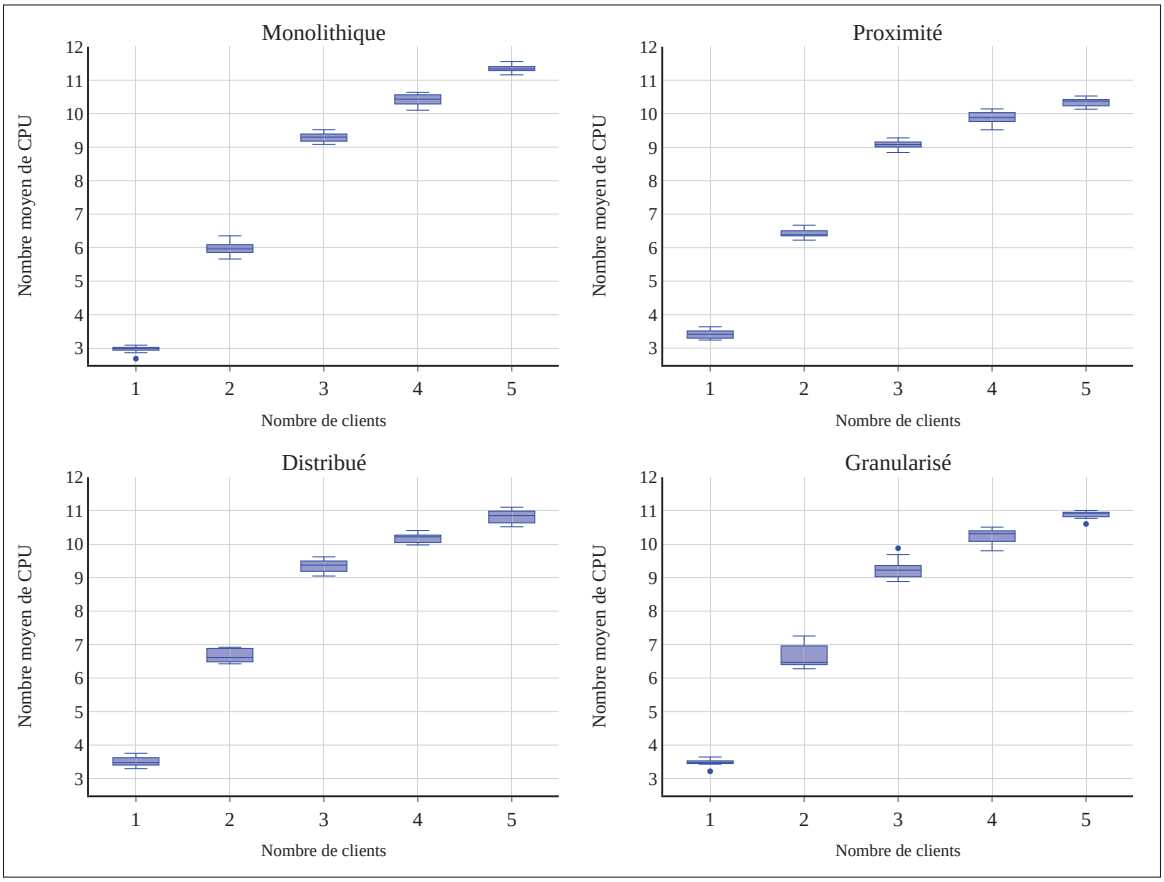

Figure 3.15 Variabilité du nombre de cœurs CPU utilisés par les déploiements (10 essais par nombre de clients)

En commençant par le CPU, présenté dans la figure 3.15, nous pouvons observer une grande stabilité des valeurs pour les déploiements Proximité et Monolithique, deux approches relativement similaires. En revanche, pour les déploiements Granularisé et Distribué, nous constatons davantage d'instabilité entre les valeurs recueillies, particulièrement pour le cas de 2 clients dans le déploiement Granularisé.

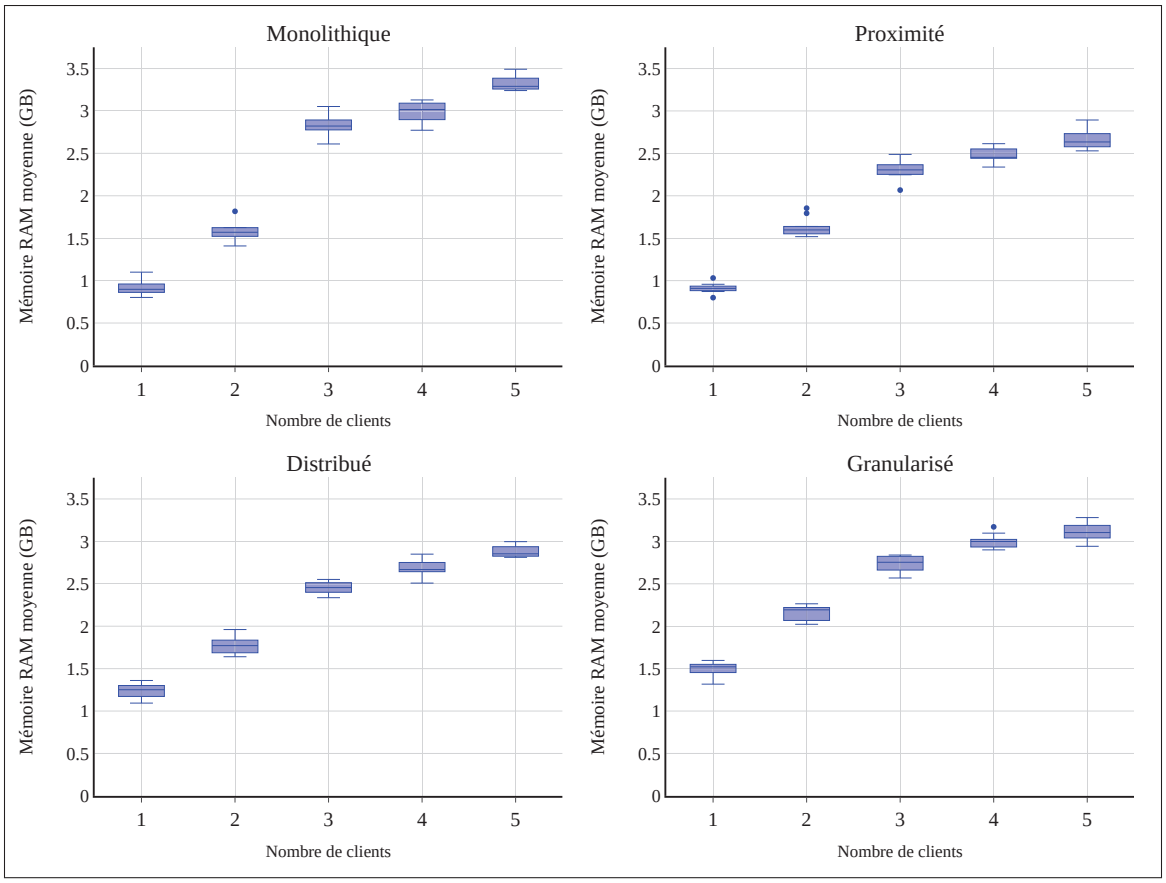

Figure 3.16 Variabilité de la mémoire RAM utilisée par les déploiements (10 essais par nombre de clients)

Pour la mémoire RAM utilisée, illustrée dans la figure 3.16, il est difficile de tirer des conclusions claires. Les quatre déploiements présentent des comportements différents et leurs variations changent également selon le nombre de clients. Même l'étendue des données n'est pas stable, ce qui complique la prévision de la quantité de mémoire nécessaire pour chaque scénario.

En ce qui concerne les délais de livraison, selon la figure 3.17, la variabilité des valeurs semble globalement similaire entre les différents déploiements. Cependant, à travers les quatre graphiques, on remarque que l'étendue des données augmente au fur et à mesure que le nombre de clients croît. Cette augmentation peut s'expliquer par deux raisons. Premièrement, comme précisé dans l'annexe, les valeurs des graphiques représentent les délais de tous les clients. Plus qu'il y a de clients, plus le nombre total de valeurs s'accroît, ce qui accroît mécaniquement la

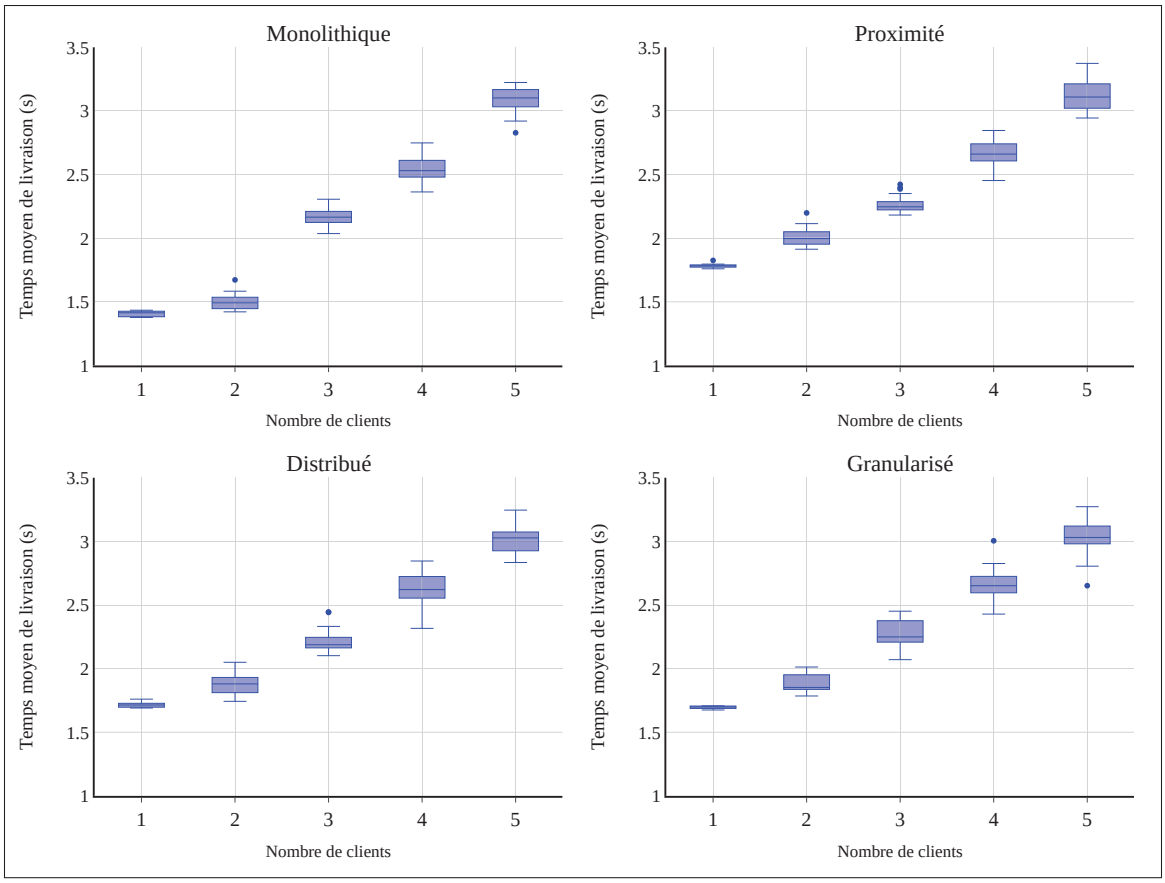

Figure 3.17 Variabilité des délais de livraison des déploiements (10 essais par nombre de clients)

variabilité. Deuxièmement, tel que prévu, plus il y a de clients simultanés, plus les délais de livraison s'allongent, ce qui élargit davantage l'éventail des valeurs possibles. Cela expliquerait pourquoi tous les déploiements apparaissent très stables pour un seul client, puis deviennent progressivement plus instables à partir de quatre et cinq clients, où l'étendue des valeurs devient nettement plus importante. Cependant, une fois de plus, il ne semble pas y avoir un déploiement plus stable et prévisible qu'un autre.

Pour conclure, malgré les variations observées et la discussion présentée, nous considérons que les valeurs recueillies restent globalement fiables. Bien que certaines boîtes soient plus larges que d'autres, cela reste cohérent, car nous travaillons avec des aspects qui ne fonctionnent pas de manière entièrement déterministe. La collecte des ressources et des délais peut être influencée

par des conditions externes, telles que le système d'exploitation, la conteneurisation ou encore l'état du réseau. Malgré ces incertitudes, les résultats obtenus demeurent cohérents.

3.4.2.4 Retour sur l'analyse qualitative et les résultats des déploiements

Dans la section 2.2.5, nous avions émis l'hypothèse selon laquelle le déploiement Distribué serait le plus performant. À ce moment-là, nous ne pouvions pas être plus précis, car les critères de performance n'avaient pas encore été définis. Ceux-ci ont été établis au cours du présent chapitre. Cependant, même après nos tests, il est difficile de désigner un déploiement comme étant objectivement le plus performant, étant donné que notre analyse repose sur trois axes de comparaison : le nombre de cœurs utilisés, la quantité de mémoire consommée et les délais de livraison.

Pour qu'un déploiement soit considéré comme le plus performant, il faudrait qu'il présente simultanément la plus faible consommation de ressources et les délais de livraison les plus courts. Or, les résultats observés démontrent que les performances dépendent notamment du nombre de répliques et de la distribution des tâches.

Si l'on se base uniquement sur les délais de livraison, dans nos conditions de tests, le déploiement Monolithique se distingue par les délais les plus faibles en contexte multi-utilisateur. Cette efficacité est principalement attribuable à son faible niveau de distribution ainsi qu'à sa proximité géographique auprès des clients. Toutefois, il convient de souligner que le déploiement Distribué et Granularisé sont aussi capables de répondre à deux clients en moins de deux secondes. Cela signifie qu'ils sont, en pratique, également viables.

Cependant, si l'on observe uniquement les déploiements distribués — à savoir Proximité, Distribué et Granularisé — des différences, bien que mineures, peuvent être constatées. Ces écarts proviennent de la manière dont les tâches sont réparties entre les microservices. Plus la granularité est élevée, plus le nombre de microservices augmente, ce qui réduit la quantité de CPU disponible pour les calculs et entraîne des délais légèrement plus élevés, comme le montrent les courbes du déploiement Granularisé. À l'inverse, une bonne répartition des tâches,

comme dans le déploiement Distribué, permet une meilleure allocation de la mémoire et des délais légèrement plus faibles en contexte multi-clients.

De plus, les délais de livraison du déploiement Monolithique semblent augmenter plus brusquement à mesure que le nombre de clients croît, contrairement aux trois autres déploiements, dont la montée en latence semble plus graduelle. Il serait donc pertinent d'émettre l'hypothèse qu'au-delà d'un certain nombre de clients, le déploiement Monolithique deviendrait le moins efficace, comme on le constate déjà auprès de 5 clients. Ainsi, chaque déploiement pourrait, à tour de rôle, se révéler le plus performant selon les conditions du contexte. Dans un scénario à forte charge, le déploiement Granularisé pourrait même s'imposer comme le plus rapide, grâce à sa plus grande quantité de répliques.

En résumé, il est difficile de désigner un déploiement comme étant universellement plus performant qu'un autre, car la performance dépend de nombreuses variables. Il faut notamment prendre en compte le nombre de clients à servir, les délais de livraison souhaités, le nombre de répliques envisagées, ainsi que les ressources disponibles et leurs caractéristiques. Si l'on se base uniquement sur les délais de livraison, le déploiement Monolithique se révèle le plus performant jusqu'à 4 clients. En revanche, en matière d'optimisation des ressources, le déploiement Proximité produit les résultats les plus intéressants. Si l'on désire un juste milieu entre les deux, le déploiement Distribué serait la réponse. Il serait donc plus pertinent d'envisager une approche morphique, dans laquelle le système adapte dynamiquement sa stratégie de déploiement en fonction de l'état des variables à un instant donné.

3.5 Synthèse des résultats

Dans ce chapitre, nous avons présenté l'ensemble des données vidéo et leurs caractéristiques, ainsi que nos outils et méthodologies pour l'acquisition des données, suivis du matériel utilisé pour mener nos expériences. L'ensemble de ces installations et configurations nous a permis de réaliser nos expérimentations, à partir desquelles nous avons recueilli nos résultats.

Nous avons étudié l'impact du nombre et du type d'instance. Nous avons constaté que ces critères influencent effectivement l'utilisation des ressources et les délais de livraison. Un nombre élevé d'instances permet de servir plusieurs clients simultanément plus rapidement. Concernant le type d'instance, nous avons observé qu'un travailleur web utilise moins de mémoire et de CPU qu'un conteneur.

Grâce à ces informations, nous avons décidé de mettre en place nos différents déploiements sous forme de conteneurs afin de tester les limites de ceux-ci. La comparaison quantitative de ces déploiements nous a permis de découvrir que le déploiement Monolithique présente un délai de livraison plus court jusqu'à 4 clients, mais au prix d'une utilisation plus élevée de cœurs CPU et de mémoire RAM. Alors que le déploiement Distribué présente des délais raisonnables tout en entraînant une consommation de ressources plus faible. Cela conduit à la conclusion qu'une bonne répartition des tâches permet de réduire l'usage des ressources, tout en assurant un service efficace à plusieurs clients dans un temps restreint.

La comparaison entre les déploiements a également révélé une forte dépendance au nombre de répliques. Plus il y a de répliques, plus nous utilisons de ressources et donc plus nous pouvons livrer les segments dans un temps limité.

CONCLUSION ET RECOMMANDATIONS

Ce mémoire présente nos travaux de recherche sur l'intégration des vidéos 360° avec une architecture microservices déployée dans une infrastructure périphérique-nuagique. Notre étude s'est concentrée sur la conception et l'évaluation de différents modèles de déploiement pour la génération et la diffusion par tuiles adaptatives de segments vidéo 360° via un lecteur web interactif. Les objectifs principaux de cette recherche étaient de concevoir des architectures microservices adaptées à ce contexte spécifique, puis d'analyser leurs performances et leur comportement sur le matériel disponible.

Pour ce faire, nous avons d'abord adapté la chaîne de génération d'un segment inspirée des travaux de la littérature. Notre chaîne de traitement est composée de quatre étapes importantes : la sélection de la qualité des tuiles selon la région d'intérêt de l'utilisateur ; la récupération des tuiles selon les qualités définies ; le raccordement des tuiles pour la création du segment ; la réécriture du segment dans un flux binaire compatible.

Cette chaîne, très proche de celle définit dans la littérature, se distingue par le choix du codec. L'état de l'art préconise principalement l'utilisation du codec H.265 pour gérer l'assemblage des tuiles et la réécriture des segments. Toutefois, en raison de sa faible accessibilité sur le web, nous avons opté pour le codec H.264, ce qui a nécessité une adaptation de la chaîne. Contrairement à H.265 et aux approches présentées dans la littérature, H.264 ne prend pas en charge la diffusion adaptative par tuiles. Nous avons donc dû adapté l'étape d'assemblement en utilisant FFmpeg. Le segment résultant a ensuite été fragmenté afin d'être diffusé via un lecteur web.

Nous avons ensuite regroupé ou divisé ces tâches en microservices, ce qui nous a permis de créer quatre déploiements : Monolithique, Proximité, Distribué et Granularisé. Ces déploiements sont inspirés des travaux de la littérature. Nous avons testé ces déploiements et fait plusieurs découvertes. Entre autres que la répartition des taches impacte la consommation des ressources, ce qui influence les délais de livraison. Cela dépend notamment du nombre et du type d'instances

que nous choisissons d'utiliser. Il en résulte un compromis triangulaire entre le nombre de clients que nous souhaitons servir, la quantité de ressources disponibles, et le nombre de microservices que nous voulons déployer.

Un article scientifique de cette recherche, intitulé «Evaluation of Containerized Microservice Architectures for 360° Video Web Delivery Within an Edge-cloud Infrastructure», a été soumis à *IEEE Conference on Network Function Virtualization and Software Defined Networks* en juin, en collaboration avec mes directeurs. Cette soumission résume l'impact des déploiements en microservices conteneurisés pour le traitement et la diffusion de vidéos 360°.

4.1 Limites de la recherche

Notre principale limite fut celle du matériel disponible. Plus précisément, nous aurions voulu disposer d'un CPU ayant plus de cœurs. Comme nous l'avons mentionné, le CPU fut la ressource la plus critique dans notre cas et nous aurions souhaité un processeur plus puissants. Ainsi, nous aurions peut-être pu davantage pousser les limites des déploiements et obtenir des comportements plus représentatifs, non impactés par les contraintes matérielles.

Nous pouvons aussi noter une limitation réseau. N'ayant pas accès à des machines sur des réseaux que nous contrôlions, nous n'avons pas pu utiliser Docker Swarm pour les tests finaux des déploiements. Cela a engendré des ajustements et une réflexion sur la distribution des conteneurs pour la collecte des données. Dans certains cas, nous étions limités à des communications unidirectionnelles. L'infrastructure réseau fut aussi la raison de l'utilisation d'une passerelle API auprès des conteneurs, car nous ne pouvions pas utiliser Docker Swarm sans adresse publique pour les machines locales.

4.2 Travaux futurs

Ce projet de recherche pourrait être poursuivi et bonifié à travers des travaux futurs.

Premièrement, nous jugeons qu'il faudrait trouver une alternative au raccordement. Le filtre "xstack" de FFmpeg n'est pas optimal pour ce cas d'utilisation. Il serait intéressant d'explorer davantage le raccordement des tuiles et de proposer des alternatives. Par exemple, on pourrait essayer d'adapter le filtre pour l'exécuter sur le GPU, afin d'éviter un transfert de données. Ainsi, on raccorderait les tuiles plus rapidement et pourrait se rapprocher du temps réel.

Deuxièmement, pour se rapprocher davantage des cas d'utilisation réels, il serait pertinent d'ajouter une tâche de capture en direct dans notre chaîne. C'est-à-dire de prendre les segments provenant directement d'une caméra 360° et d'ajouter cette étape et ses sous-étapes à notre chaîne de traitement. Ensuite, observer leur impact sur l'utilisation des ressources et les délais engendrés.

Enfin, il serait pertinent d'ajouter un casque de réalité virtuelle, car cela rapprocherait l'expérience d'un cas réel. Dans notre projet, nous avons utilisé un lecteur web avec des librairies, mais un casque permettrait de mieux ressentir l'effet d'immersion, tout en mettant l'accent sur l'expérience utilisateur.

ANNEXE I

COMMANDE DE RACCORDEMENT DES TUILES FFMPEG

Cette annexe presente un exemple de commande FFmpeg avec le filtre xstack. Cette commande permet de créer une mosaïque de 3×3 avec neuf vidéos :

```
ffmpeg -i tuile_1.mp4 <...> -i tuile_9.mp4 -filter_complex "[0:v][1:
   v] <...> [8:v] xstack=inputs=9:layout=0_0|w0_0|w0+w1_0|0_h0|w0_h0
   |w0+w1_h0|0_h0+h1|w0_h0+h1|w0+w1_h0+h" resultat.mp4
```

La partie qui demande un peu de réflexion est celle concernant la disposition, ou « layout », de la commande. Cet aspect du filtre définit où chaque tuile se positionne spatialement (par exemple, la tuile 1 dans le coin supérieur droit, la tuile 2 à droite de la tuile 1, etc.), comme illustré sur la figure I-1.

Le point (0, 0) correspond à l'origine de la première tuile et définit le coin supérieur gauche de la vidéo. Pour la position horizontale, les valeurs $w0$ et $w0 + w1$ indiquent la position horizontale du coin supérieur gauche d'une tuile, où la somme $w0 + w1$ représente le nombre de tuiles à parcourir horizontalement avant de définir la position horizontale du point d'origine de la tuile. De même, pour la position verticale, les valeurs $h0$ et $h0 + h1$ déterminent la position verticale, avec $h0 + h1$ correspondant au nombre de tuiles à parcourir verticalement. Le _ sert à séparer la position horizontale et verticale.

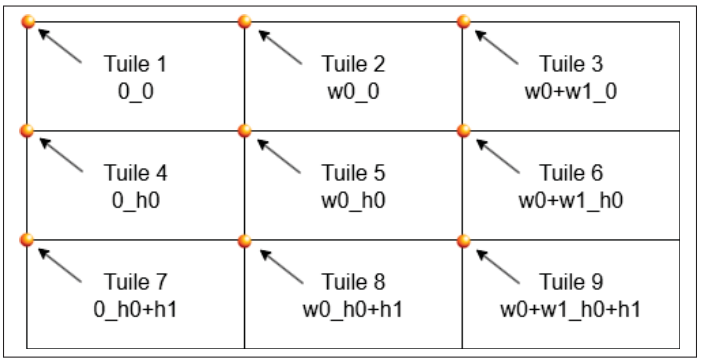

Figure-A I-1 Layout de 9 tuiles

ANNEXE II

COMMANDE FFMPEG POUR LA DIVISION EN TUILES

Pour la création des tuiles, nous utilisons FFmpeg. La commande ci-dessous permet de diviser spatialement une vidéo en plusieurs vidéos indépendantes, comme illustré à la figure 1.6.

```
ffmpeg -i input.mp4 \
-filter_complex "[0:v]crop=in_w/2:in_h/2:0:0[out1]; \
                 [0:v]crop=in_w/2:in_h/2:in_w/2:0[out2]; \
                 [0:v]crop=in_w/2:in_h/2:0:in_h/2[out3]; \
                 [0:v]crop=in_w/2:in_h/2:in_w/2:in_h/2[out4]" \
-map "[out1]" tuile1.mp4 \
-map "[out2]" tuile2.mp4 \
-map "[out3]" tuile3.mp4 \
-map "[out4]" tuile4.mp4
```

Cette commande utilise le filtre `crop` pour effectuer la division spatiale. L'exemple présenté ici divise la vidéo en 4 tuiles de résolution égale, soit une grille de 2×2.

Concrètement, la commande prend en entrée le fichier `input.mp4`, qu'elle passe à travers le filtre `crop`. Chaque ligne du filtre suit la forme `[0:v]crop=in_w/2:in_h/2:0:0[out1];`, où chaque instruction `crop=` définit une tuile.

Dans cet exemple, nous avons 4 instructions `crop`, donc 4 tuiles. L'élément `[0:v]` spécifie que l'on travaille sur le premier fichier d'entrée (ici `input.mp4`), et uniquement sur sa composante vidéo (`:v`).

Les arguments `in_w/2` et `in_h/2` définissent respectivement la largeur et la hauteur de chaque tuile. Ici, nous utilisons la résolution de la vidéo d'entrée, d'où l'emploi de `in_w` et `in_h`, mais ces valeurs pourraient aussi être absolues.

Les deux derniers arguments (`0:0`, `in_w/2:0`, etc.) définissent les coordonnées X et Y du coin supérieur gauche de la région à extraire. Par exemple, `0:0` correspond au pixel situé dans le coin supérieur gauche de la vidéo originale, tandis que `0:in_h/2` correspond au pixel à la première colonne et à la moitié de la hauteur de la vidéo.

Enfin, `[out1]` attribue un nom à la tuile générée, qui est ensuite utilisée avec l'option `-map` pour écrire la tuile dans un fichier, par exemple : `-map "[out1]" tuile1.mp4`.

ANNEXE III

COMMANDE DE CHANGEMENT DE DÉBIT AVEC FFMPEG

Afin de réduire la qualité d'un segment vidéo, nous avons utilisé la commande FFmpeg ci-dessous.

```
ffmpeg -i tuile_n.mp4 -b:v 20k tuile_n_20k.mp4
```

Cette commande prend une vidéo en entrée (*-i tuile_n.mp4*) sur laquelle elle modifie le débit vidéo (`b:v`), spécifié par l'option `b:v` — où `b` désigne le bitrate et `v` indique qu'il s'applique à la composante vidéo. Le fichier généré en sortie est *tuile_n_20k.mp4*.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Authority, M. R. (2025). Boxes (Atoms). Repéré à https://mp4ra.org/registered-types/boxes.

Brochu, O. (2024). *Optimisation débit-distorsion pour la diffusion par tuiles de vidéos 360° en temps réel*. (Mémoire de maîtrise, École de technologie supérieure, Université du Québec).

Budagavi, M., Furton, J., Jin, G., Saxena, A., Wilkinson, J. & Dickerson, A. (2015). 360 degrees video coding using region adaptive smoothing. *2015 IEEE International Conference on Image Processing (ICIP)*, pp. 750-754. doi : 10.1109/ICIP.2015.7350899.

Bulley, P. (2016). What is Fragmented MP4 (fMP4) and how is it different than normal MP4 ? Repéré à https://stackoverflow.com/questions/35177797/what-is-fragmented-mp4-fmp4-and-how-is-it-different-than-normal-mp4.

Chiariotti, F. (2021). A survey on 360-degree video : Coding, quality of experience and streaming. Repéré à https://www.sciencedirect.com/science/article/pii/S014036642100253X.

Cloudinary. (2025a). Pixel Perfect : H.264 VS H.265 Explained. Repéré le 2025-05-09 à https://cloudinary.com/guides/video-formats/pixel-perfect-h-264-vs-h-265-explained.

Cloudinary. (2025b). Fragmented MP4. Repéré le 2025-05-07 à https://cloudinary.com/glossary/fragmented-mp4.

Deveria, A. & Schoors, L. (2025a). MPEG-4/H.264 video format. Repéré à https://caniuse.com/mpeg4.

Deveria, A. & Schoors, L. (2025b). HEVC/H.265 video format. Repéré à https://caniuse.com/hevc.

Docker, Inc. [Accessed : 2025-06-20]. (2024). How services work in swarm mode. Repéré à https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/.

Docker Inc. (2025a). Use containers to Build, Share and Run your applications. Repéré à https://www.docker.com/resources/what-container/.

Docker Inc. [Accessed : 2025-06-16]. (2025b). Docker Official Website. Repéré à https://www.docker.com.

Documentation, D. (2025). Resource constraints. Repéré le 2025-09-30 à https://docs.docker.com/engine/containers/resource_constraints/.

FFmpeg. (2019). Create a mosaic out of several input videos using xstack. Repéré le 2025-05-30 à https://trac.ffmpeg.org/wiki/Createamosaicoutofseveralinputvideosusingxstack.

Fielding, R. T., Nottingham, M. & Reschke, J. [Accessed : 2025-06-16]. (2022). HTTP Semantics (RFC 9110). Repéré à https://httpwg.org/specs/rfc9110.html.

Ghaznavi-Youvalari, R., Zare, A., Fang, H., Aminlou, A., Xie, Q., Hannuksela, M. M. & Gabbouj, M. (2017). Comparison of HEVC coding schemes for tile-based viewport-adaptive streaming of omnidirectional video. *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, pp. 1-6. doi : 10.1109/MMSP.2017.8122227.

Gokan, M. [Accessed : 2025-09-04]. (2021). Re : Average CPU Repéré à https://github.com/google/cadvisor/issues/2026#issuecomment-1003120833.

Graf, M., Timmerer, C. & Mueller, C. (2017). Towards Bandwidth Efficient Adaptive Streaming of Omnidirectional Video over HTTP : Design, Implementation, and Evaluation. *Proceedings of the 8th ACM on Multimedia Systems Conference*, (MMSys'17), 261–271. doi : 10.1145/3083187.3084016.

Hosseini, M. & Swaminathan, V. (2016). Adaptive 360 VR Video Streaming : Divide and Conquer. *2016 IEEE International Symposium on Multimedia (ISM)*, pp. 107-110. doi : 10.1109/ISM.2016.0028.

Immich, R., Villas, L., Bittencourt, L. & Madeira, E. (2019). Multi-tier Edge-to-Cloud Architecture for Adaptive Video Delivery. *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 23-30. doi : 10.1109/FiCloud.2019.00012.

ITU-T. (2024). *Recommendation ITU-T H.265 (V10) (07/2024) - High efficiency video coding* (Rapport n°H.265). Repéré à https://www.itu.int/rec/T-REC-H.265-202407-I.

Jamali, M., Golaghazadeh, F., Coulombe, S., Vakili, A. & Vazquez, C. (2019). Comparison of 3D 360-Degree Video Compression Performance Using Different Projections. *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, pp. 1-6. doi : 10.1109/CCECE.2019.8861868.

Le Feuvre, J. (2020). GPAC filters. *Proceedings of the 11th ACM Multimedia Systems Conference*, (MMSys '20), 249–254. doi : 10.1145/3339825.3394929.

Lo, W.-C., Huang, C.-Y. & Hsu, C.-H. (2018). Edge-Assisted Rendering of 360° Videos Streamed to Head-Mounted Virtual Reality. *2018 IEEE International Symposium on Multimedia (ISM)*, pp. 44-51. doi : 10.1109/ISM.2018.00016.

MDN Web Docs. [Accessed : 2025-06-20]. (n.d.). Using Web Workers - Web APIs | MDN. Repéré à https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

Moving Picture Experts Group (MPEG). [Accessed : 2025-06-16]. (n.d.). MPEG-4 Standard Overview. Repéré à https://www.mpeg.org/standards/MPEG-4/.

Niamut, O. A., Thomas, E., D'Acunto, L., Concolato, C., Denoual, F. & Lim, S. Y. (2016). MPEG DASH SRD : spatial relationship description. *Proceedings of the 7th International Conference on Multimedia Systems*, (MMSys '16), 1-8. doi : 10.1145/2910017.2910606.

NVIDIA Developer Blog. [Published July 24, 2019]. (2019, juillet, 24). NVIDIA FFmpeg Transcoding Guide. Repéré à https://developer.nvidia.com/blog/nvidia-ffmpeg-transcoding-guide/.

OpenTelemetry Authors. [Last modified June 24, 2025]. (2025, juin, 24). What is OpenTelemetry ? Repéré à https://opentelemetry.io/docs/what-is-opentelemetry/.

Ozer, J. (2009). What Is H.264 ? Repéré à https://www.streamingmedia.com/Articles/Editorial/What-Is-.../What-Is-H.264-74735.aspx.

Ozer, J. [Accessed : 2025-06-02]. (2017). HEVC, H.264, and MPEG-DASH Royalty Update. Streaming Learning Center. Repéré à https://streaminglearningcenter.com/wp-content/uploads/2017/03/HEVC_H264_DASH_Royalties-1.pdf.

Quest, M. (2022). Immersive Media Formats. Repéré à https://creator.oculus.com/getting-started/immersive-media-formats/?locale=fr_FR.

Reinholds, A. [Last updated January 12, 2024]. (2024, janvier, 12). A Complete Guide to Distributed Tracing. Repéré à https://newrelic.com/fr/blog/best-practices/distributed-tracing-guide.

Rigazzi, G., Kainulainen, J.-P., Turyagyenda, C., Mourad, A. & Ahn, J. (2019). An Edge and Fog Computing Platform for Effective Deployment of 360 Video Applications. *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*, pp. 1-6. doi : 10.1109/WCNCW.2019.8902860.

Santos, J., van der Hooft, J., Vega, M. T., Wauters, T., Volckaert, B. & De Turck, F. (2021). Efficient Orchestration of Service Chains in Fog Computing for Immersive Media. *2021 17th International Conference on Network and Service Management (CNSM)*, pp. 139-145. doi : 10.23919/CNSM52442.2021.9615539.

Server Fault. [Accessed : 2025-06-26]. (2014). Can a Docker Container Use the Full CPU Power and Memory of the Host OS ? Repéré à https://serverfault.com/questions/615758/can-a-docker-container-uses-the-full-cpu-power-and-memory-of-the-host-os.

Services, A. W. (2025). Microservices. Repéré à https://aws.amazon.com/microservices/.

Shen, G., Dai, J., Moustafa, H. & Zhai, L. (2021). 5G and Edge Computing Enabling Experience Delivery Network (XDN) for Immersive Media. *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pp. 1-7. doi : 10.1109/HPSR52026.2021.9481809.

Shkuro, Y. [Accessed : 2025-07-19]. (2021). Experiment : Migrating OpenTracing-Based Application in Go to Use the OpenTelemetry SDK. Repéré à https://medium.com/jaegertracing/experiment-migrating-opentracing-based-application-in-go-to-use-the-opentelemetry-sdk-29b09fe2fbc4.

Sodagar, I. (2011). The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia*, 18(4), 62-67. doi : 10.1109/MMUL.2011.71.

solution, G. L. [Dashboard #609 on Grafana.com]. (2017). Docker Swarm & Container Overview. Repéré à https://grafana.com/grafana/dashboards/609-docker-swarm-container-overview/.

Son, J., Jang, D. & Ryu, E.-S. (2018). Implementing 360 video tiled streaming system. *Proceedings of the 9th ACM Multimedia Systems Conference*, (MMSys '18), 521–524. doi : 10.1145/3204949.3208119.

squashlabs. (2023). Deploying Flask Web Apps : From WSGI to Kubernetes. Repéré le 2025-09-30 à https://www.squash.io/optimizing-flask-apps-from-wsgi-server-configurations-to-kubernetes/.

uWSGI Documentation. [Accessed : 2025-06-26]. (n.d.). Things To Know. Repéré à https://uwsgi-docs.readthedocs.io/en/latest/ThingsToKnow.html.

uWSGI Project. (2025). Quickstart for Python/WSGI applications. Repéré le 2025-09-30 à https://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html.

VIA-LA. (2023). HEVC LICENSE FEES. Repéré le 2025-06-02 à https://www.via-la.com/licensing-2/hevc/hevc-license-fees/.

Weisstein, E. W. [Accessed : 2025-06-19]. (2024). Spherical Coordinates. Repéré à https://mathworld.wolfram.com/SphericalCoordinates.html.

Wu, C.-H. (2021). 360-converter. Repéré à https://github.com/chinhsuanwu/360-converter.

Ye, Y., Alshina, E. & Boyce, J. (2018). *JVET-F1003 : Algorithm descriptions of projection format conversion and video quality metrics in 360Lib*.