# LuffyMQ : Système de messagerie en files d'attente, géodistribué et adapté aux environnements périphériques

par

Amna SNENE

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

MONTRÉAL, LE "4 NOVEMBRE 2025"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

**PRÉSENTATION DU JURY**

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Julien Gascon-Samson, directeur de mémoire
Département de génie logiciel et des TI, École de technologie supérieure

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

M. Kaiwen Zhang, 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 "27 OCTOBRE 2025"

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

## REMERCIEMENTS

Je souhaite exprimer ma gratitude à mon directeur de recherche, Julien Gascon-Samson, pour m'avoir offert l'opportunité de réaliser mes études de maîtrise au sein de son laboratoire. Je lui suis reconnaissante pour son soutien, ses précieux conseils et son accompagnement.

Je remercie également l'organisation MITACS, grâce à laquelle j'ai pu effectuer un stage et entreprendre cette maîtrise au Canada, ainsi que Julien pour son appui financier.

Je remercie mes collègues et badminton team qui ont rendu mon passage à l'ÉTS plus agréable.

J'exprime également ma profonde gratitude à mes proches et à mes ami·e·s, en Tunisie comme au Canada (nombreux·ses mais tous·tes précieux·ses), pour leur soutien et leurs encouragements constants.

Je tiens à remercier tout particulièrement ma mère, Khadija, qui a développé un intérêt pour les outils technologiques afin de communiquer avec moi pendant ces années de maîtrise. Merci Khdayej !

À mon père, Hedi, j'ai toujours cru que ce n'est pas une simple coïncidence que tu sois né le 10 décembre, la Journée des droits de l'homme. Tes valeurs sont les étoiles qui m'orientent dans les moments les plus sombres.

À ma sœur Selma et à mon frère Mehdi, vous êtes pour moi une immense source de fierté. Merci d'avoir toujours été là.

Enfin, une pensée émue à ma grand-mère qui nous a quittés durant cette aventure : tu resteras à jamais présente dans nos cœurs et nos esprits.

**LuffyMQ : Système de messagerie en files d'attente, géodistribué et adapté aux environnements périphériques**

Amna SNENE

## RÉSUMÉ

L'essor de l'Internet des objets (IoT) entraîne une hausse significative du volume de données générées en périphérie du réseau, où les exigences de faible latence et de gestion efficace des ressources deviennent critiques. L'informatique en périphérie s'impose ainsi comme une réponse aux limites du modèle centralisé infonuagique, en rapprochant les ressources de calcul des sources de données. Dans ce contexte, les courtiers de messages jouent un rôle clé, mais les solutions existantes sont rarement adaptées aux environnements edge et se concentrent principalement sur le modèle publication/abonnement, sans proposer de solution unifiée. Ce travail introduit LuffyMQ, un système de messagerie géodistribué, construit à partir de RabbitMQ et enrichi de modules spécifiques pour répondre aux contraintes de l'edge. Notre approche prend en charge simultanément les deux modèles asynchrones principaux : publication/abonnement et point à point avec files d'attente. Des stratégies intelligentes de répartition de charge et de placement de nœuds de synchronisation ont été proposées. Les expérimentations menées sur un environnement géodistribué démontrent l'efficacité de LuffyMQ. Dans le modèle point à point, le système réduit la latence moyenne jusqu'à 75% par rapport à la stratégie native de RabbitMQ et aux configurations centralisées. Dans le modèle publication/abonnement, il maintient une latence proche de la borne inférieure tout en améliorant la mise en échelle, atteignant jusqu'à 200% de messages supplémentaires traités sous forte charge.

**Mots-clés:** Internet des objets, système de messagerie, file d'attente, publication/abonnement, RabbitMQ

**LuffyMQ : A Geo-Distributed Queue Messaging System Adapted to Edge Environments**

Amna SNENE

## ABSTRACT

The rise of the Internet of Things (IoT) is driving a significant increase in the volume of data generated at the network edge, where requirements for low latency and efficient resource management are critical. Edge computing emerges as a promising response to the limitations of the centralized cloud model, by bringing computing resources closer to the data sources. In this context, message brokers play a key role, but existing solutions are rarely adapted to edge environments and mostly focus on the publish/subscribe model, without providing a unified solution. This work introduces *LuffyMQ*, a geo-distributed messaging system built on RabbitMQ and extended with specific modules to address edge constraints. Our approach supports both major asynchronous models : publish/subscribe and point-to-point with message queues (work queue). Intelligent strategies for load balancing and synchronization node placement are proposed. Experiments conducted in a geo-distributed environment demonstrate the effectiveness of *LuffyMQ*. In the point-to-point model, the system reduces average latency by up to 75% compared to RabbitMQ's native strategy and centralized configurations. In the publish/subscribe model, it maintains latency close to the lower bound while improving scalability, achieving up to 200% more messages processed under heavy load.

**Keywords:** Internet of Things, messaging system, queue, publish/subscribe, RabbitMQ

## TABLE DES MATIÈRES

Page

## LISTE DES TABLEAUX

Page

## LISTE DES FIGURES

Page

## LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

| | |
|---|---|
| AI | Artificial Intelligence |
| AMQP | Advanced Message Queuing Protocol |
| CaaS | Containers as a Service |
| CoAP | Constrained Application Protocol |
| DBaaS | Database as a Service |
| DDS | Data Distribution Service |
| DLQ | Dead Letter Queue |
| DLX | Dead Letter Exchange |
| DPWS | Devices Profile for Web Services |
| DRaaS | Disaster Recovery |
| HTTP | Hypertext Transfer Protocol |
| IaaS | Infrastructure as a Service |
| ICMP | Internet Control Message Protocol |
| ILP | Integer Linear Programming |
| IoT | Internet of Things |
| JMS | Java Message Service |
| LTE | Long Term Evolution |
| MQTT | Message Queuing Telemetry Transport |
| MQTT-SN | MQTT for Sensor Networks |
| OASIS | Organization for the Advancement of Structured Information Standards |
| OPC UA | Open Platform Communications Unified Architecture |
| PaaS | Platform as a Service |
| Pub/Sub | Publish/Subscribe |
| QoE | Quality of Experience |
| QoS | Quality of Service |

| | |
|---|---|
| RPC | Remote Procedure Call |
| SaaS | Software as a Service |
| SDN | Software-defined networking |
| STOMP | Simple Text Oriented Messaging Protocol |
| TTL | Time To Live |
| XaaS | Everything as a Service |
| XML | Extensible Markup Language |
| XMPP | Extensible Messaging and Presence Protocol |

# INTRODUCTION

L'internet des objets (IoT) s'est fortement développé, avec une multitude d'appareils connectés en périphérie qui interagissent entre eux et avec le monde extérieur. Cette croissance devrait se poursuivre dans les années à venir. Toutefois, pour les scénarios IoT dans lesquels de faibles latences de bout en bout entre les producteurs et les consommateurs de données sont une exigence essentielle, les latences liées à l'acheminement des messages de la périphérie du réseau vers le nuage (*cloud*) peuvent s'avérer inacceptables (Satyanarayanan, 2017). L'informatique en périphérie (*edge computing*) se présente comme une solution clé à ces scénarios en profitant des ressources de calcul placées à la périphérie du réseau, à proximité de la source de génération de données (Shi & Dustdar, 2016). Cela se réfère à décentraliser une partie des ressources informatiques disponibles dans les grands centres de données en les distribuant à la périphérie du réseau.

L'adoption de l'informatique en périphérie nécessite la réadaptation des technologies existantes, originellement conçues pour les environnements infonuagiques, notamment les courtiers de messages (*message brokers*). Cette réadaptation est cruciale pour améliorer la gestion de la latence et le traitement des données dans les systèmes IoT distribués.

Les courtiers de messages jouent un rôle essentiel dans la transmission des données dans les systèmes distribués, simplifiant la conception d'applications à grande échelle et facilitant la communication entre des environnements hétérogènes (Subramoni, Marsh, Narravula, Lai & Panda, 2008).

Leur mise en œuvre efficace est donc critique : elle influence directement les performances globales du système, où une bonne conception assure un déploiement rapide et efficient, tandis qu'une mise en œuvre inadéquate peut compromettre l'ensemble du système.

Les systèmes de messagerie prennent généralement en charge deux principaux styles de messagerie asynchrone : la messagerie en file d'attente, également appelée messagerie point à point avec file d'attente et la messagerie de type publication/abonnement (*publish/subscribe*).

Toutefois, la plupart des travaux existants de messagerie en périphérie se focalisent sur le modèle *publish/subscribe*, sans proposer de solution unifiée capable de combiner efficacement les deux paradigmes au sein d'un même système caractérisé par des ressources distribuées et limitées.

Dans ce contexte, nous proposons un système de messagerie géodistribué, spécifiquement conçu pour les environnements edge, reposant sur une grappe (*cluster*) de courtiers déployés sur des nœuds périphériques. Ce *cluster* agit comme une entité logique unifiée, prenant en charge simultanément les modèles publication/abonnement et point à point avec file d'attente.

Concernant la communication publication/abonnement, notre architecture garantit une diffusion cohérente des messages à l'ensemble des clients partageant les mêmes abonnements peu importe leur localisation ou leur point d'attachement au réseau. Pour le modèle point à point avec file d'attente, nous avons élaboré des stratégies intelligentes de répartition de charge. Le système privilégie les consommateurs locaux pour le traitement des messages, tout en maintenant une vue globale sur les consommateurs répartis dans les autres nœuds du réseau. Lorsque les consommateurs locaux ne peuvent plus absorber la charge, les stratégies procèdent à une évaluation dynamique des capacités disponibles sur les nœuds périphériques, permettant ainsi une redistribution en temps réel des messages vers les nœuds les plus aptes à les traiter efficacement.

Ces stratégies sont formalisées sous la forme d'un problème d'optimisation visant à déterminer le placement optimal du nœud de synchronisation. L'objectif est de minimiser la latence globale du système, tout en respectant les contraintes de bande passante propres à chaque site. Afin de limiter les reconfigurations fréquentes sans bénéfice significatif, un système de pénalisation est intégré. Celui-ci vise à garantir une plus grande stabilité de l'architecture.

Enfin, nous introduisons LuffyMQ, une extension du courtier RabbitMQ, qui implémente l'ensemble des comportements décrits précédemment. LuffyMQ est couplé à un contrôleur centralisé chargé d'orchestrer dynamiquement la configuration de la couche de liaison entre les courtiers. La solution reste pleinement compatible avec les clients RabbitMQ existants, assurant ainsi une intégration transparente.

**Organisation du mémoire**

Ce mémoire est structuré en cinq chapitres. Le premier chapitre, intitulé notions de base et revue de littérature, expose les concepts fondamentaux liés aux thématiques abordées ainsi qu'un état de l'art des travaux existants. Le deuxième chapitre présente la formulation du problème étudié et décrit en détail l'architecture du système proposé. Le troisième chapitre est consacré à l'implémentation de la solution ainsi qu'aux évaluations expérimentales, avec une analyse des résultats obtenus et une discussion de leurs limites. Enfin, la conclusion synthétise les principaux apports de ce travail et propose des perspectives pour de futures recherches.

# CHAPITRE 1

# NOTION DE BASE ET REVUE DE LITTÉRATURE

## 1.1 Internet des objets (IoT)

L'Internet des objets (IoT), tel que défini par l'Union internationale des télécommunications (Telecommunication Standardization Sector (ITU-T), 2012), représente une infrastructure mondiale interconnectant des objets physiques (capteurs, appareils intelligents, etc.) et des entités virtuelles (données, services numériques). Cette interconnexion ouvre la voie à de nombreuses applications dans divers secteurs tels que la santé (montres connectées mesurant le rythme cardiaque), la gestion de l'énergie (compteurs électriques intelligents) ou encore l'automobile (véhicules équipés de capteurs et d'actionneurs). L'IoT se distingue par la grande hétérogénéité de ses infrastructures. En effet, les dispositifs connectés varient considérablement en termes de capacités de calcul, de protocoles de communication, de sources d'énergie et de modes d'interaction avec les réseaux. Certains objets, dotés de capteurs légers et fonctionnant sur batterie, sont conçus pour une faible consommation énergétique et une autonomie de plusieurs années, tandis que d'autres, plus puissants, peuvent exécuter des traitements avancés localement. Cette diversité rend l'intégration et la gestion des données issues de ces systèmes hétérogènes complexes.

## 1.2 Infonuagique et IoT

L'infonuagique, ou cloud computing, désigne un modèle informatique moderne permettant un accès omniprésent, pratique et à la demande à un ensemble partagé de ressources configurables. Ces ressources sont détenues et gérées par des fournisseurs tiers, et peuvent être rapidement mises à disposition ou libérées grâce à un haut niveau d'automatisation, réduisant ainsi la nécessité d'intervention humaine (Mell & Grance, 2011).

À l'origine, trois modèles de services fondamentaux ont structuré l'offre cloud :

- Le *Software as a Service (SaaS)*, qui permet l'accès à des applications hébergées sans gestion de l'infrastructure sous-jacente.
- Le *Platform as a Service (PaaS)*, qui propose un environnement complet pour le développement et le déploiement d'applications.
- L'*Infrastructure as a Service (IaaS)*, qui offre des ressources matérielles virtualisées avec un contrôle étendu sur les systèmes déployés.

Au fil du temps, ce modèle s'est élargi sous la forme de XaaS *(Everything as a Service)*, englobant une variété croissante de services spécialisés. Parmi eux figurent notamment le DRaaS *(Disaster Recovery as a Service)*, le DBaaS *(Database as a Service)*, ou encore le CaaS *(Containers as a Service)*, illustrant la capacité du cloud à répondre à des besoins métiers diversifiés (Duan *et al.*, 2015).

Le *cloud computing* séduit par sa souplesse d'utilisation, sa réduction des coûts d'infrastructure, et la simplicité de déploiement qu'il offre aux organisations de toutes tailles. Il permet aux entreprises de bénéficier d'une puissance de calcul importante sans investissement matériel initial ni charge liée à la maintenance. Toutefois, cette externalisation soulève des défis critiques, notamment en matière de sécurité, de confidentialité, de souveraineté des données et de latence réseau (Jaeger, ‚Jimmy, & Grimes, 2008).

Dans un contexte marqué par l'explosion des besoins en traitement de données alimentée par l'Internet des Objets (IoT), la 5G et l'intelligence artificielle (AI), le cloud ne peut plus être envisagé de manière isolée. Traditionnellement, les systèmes IoT envoyaient toutes leurs données vers le cloud pour traitement. Mais face aux contraintes de bande passante, aux exigences de confidentialité, et surtout aux besoins de latence faible, des solutions complémentaires ont émergé (Escaleira, Cunha, Gomes, Barraca & Aguiar, 2023).

Ces solutions consistent à rapprocher la puissance de traitement des lieux de production des données. Cela permet de réduire les délais de traitement et d'améliorer significativement la qualité de service (QoS) et la qualité d'expérience (QoE).

## 1.3 Ressources informatiques en périphérie

Le modèle classique de l'infonuagique centralisée, fondé sur des centres de données éloignés, atteint ses limites. Pour surmonter ces contraintes, d'autres modèles architecturaux ont vu le jour combinant le nuage, l'informatique en brouillard *(fog)* et l'informatique en périphérie *(edge)* (Mortazavi, Salehe, Gomes, Phillips & de Lara, 2017). Cette approche permet une répartition plus adaptée des ressources de calcul et de stockage selon la localisation, la criticité et le volume des données.

Bien que les termes brouillard et périphérie (*fog* et *edge*) soient parfois utilisés de manière interchangeable, une distinction s'impose. Tous deux visent à rapprocher les capacités de traitement des sources de données, mais ils diffèrent par leur position dans l'architecture et par la nature des services qu'ils fournissent.

### 1.3.1 Informatique en brouillard (*Fog Computing*) et IoT

Le concept de informatique en brouillard ou *fog computing* a été introduit en 2012 par Cisco (Bonomi, Milito, Zhu & Addepalli, 2012) afin de désigner une couche intermédiaire située entre les objets connectés et les serveurs cloud. Cette approche vise à rapprocher les capacités de calcul, de stockage et de réseau des utilisateurs finaux, en plaçant des nœuds de traitement appelés dispositifs Fog à proximité des sources de données. Ces dispositifs peuvent être déployés dans divers environnements : sur un poteau électrique, dans une usine ou même sur une station de base. (Naha *et al.*, 2018)

Contrairement au nuage, qui repose sur une architecture centralisée, le fog adopte un modèle décentralisé. Il exploite les ressources disponibles sur des équipements proches des utilisateurs pour exécuter des traitements intermédiaires et stocker temporairement des données. Cela permet de réduire la latence, de limiter la bande passante utilisée et d'améliorer la réactivité des services.

Dans les environnements mobiles, cette approche est parfois appelée micro-nuage (cloudlet) : une petite unité de traitement située à proximité de l'utilisateur, offrant des services personnalisés avec une faible latence (Yi, Li & Li, 2015).

### 1.3.2 Informatique en périphérie (*edge computing*) et IoT

L'informatique en périphérie ou *edge computing* est une forme d'architecture distribuée dans laquelle les traitements de données sont réalisés directement au niveau des équipements terminaux, ou à proximité immédiate, tels que les capteurs, caméras, passerelles ou équipements de bord (Chen, Qin & Wang, 2022). Contrairement au fog, qui sert d'intermédiaire entre les objets et le nuage, l'edge s'effectue à l'extrémité du réseau, là où les données sont générées.

Il est particulièrement adapté aux applications critiques nécessitant des réponses rapides, telles que la santé connectée, la surveillance intelligente ou la maintenance industrielle prédictive. Il repose sur des dispositifs embarqués ou de petits serveurs capables de traiter, filtrer ou analyser les données localement, réduisant ainsi les délais de traitement et assurant une plus grande autonomie des systèmes, même en cas de perte de connexion avec les couches supérieures.

Complémentaire au nuage et au fog, la couche edge s'intègre dans une vision plus large des infrastructures hiérarchisées où chaque niveau joue un rôle spécifique, selon la proximité des données, la criticité des traitements et les exigences de performance. La figure 1.1 illustre cette répartition. Dans la suite, le terme périphérie ou edge sera utilisé pour désigner de manière générale les environnements fog et edge.

## 1.4 Courtiers de messages

Comme indiqué précédemment, les courtiers de messages occupent une place essentielle dans la transmission des données au sein des systèmes distribués. Ils simplifient la conception d'applications à grande échelle et favorisent la communication entre des environnements hétérogènes. En assurant un découplage entre les producteurs et les consommateurs de messages, ils permettent à ces derniers d'être totalement indépendants et de fonctionner sans aucune

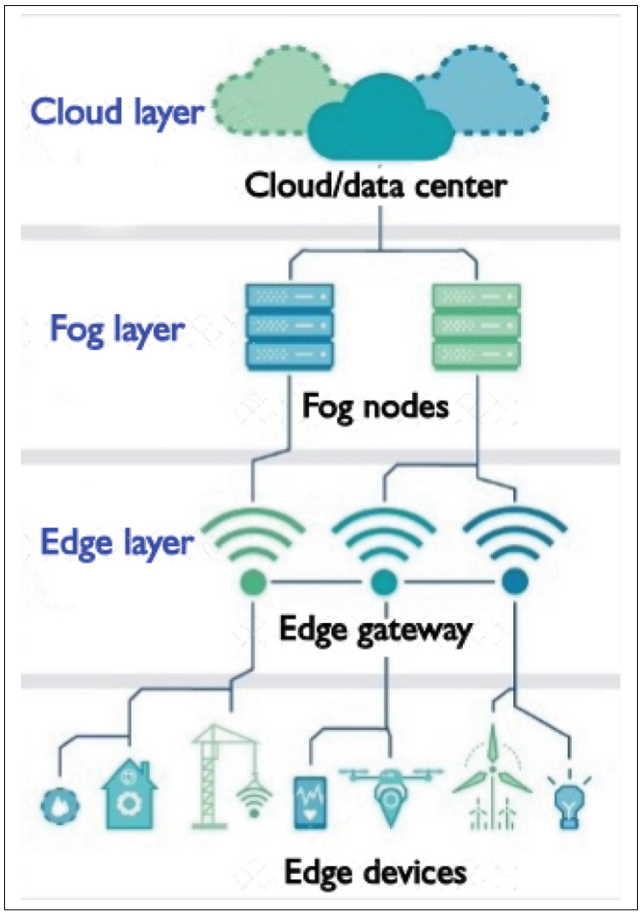



Figure 1.1 Architecture hiérarchisée edge-Fog-Nuage
Tirée de Karamimirazizi et al. (2024)

connaissance mutuelle. Cela permet de concevoir des systèmes flexibles et faiblement couplés. En effet, contrairement aux systèmes basés uniquement sur un modèle d'appel de procédure distante (RPC), les courtiers de messages reposent principalement sur un modèle de transmission de messages asynchrone, sans relation stricte entre les requêtes et les réponses. La plupart des systèmes de messagerie prennent également en charge un mode requête-réponse, mais ce n'est pas leur fonctionnalité principale (ActiveMQ, 2025).

### 1.4.1 Études comparatives des courtiers de messages

De nombreux travaux ont porté sur l'évaluation des courtiers de messages, tant dans le nuage que dans les environnements périphériques. Ces études comparent les performances des intergiciels de messagerie selon divers critères, tels que le protocole utilisé, la popularité de la solution, ou encore le modèle de communication pris en charge.

Dans ce cadre, l'étude de Koziolek et al. (Koziolek, Grüner & Rückert, 2020) propose une comparaison de trois courtiers MQTT déployés en périphérie : EMQX, HiveMQ et VerneMQ. L'évaluation porte sur plusieurs dimensions, incluant des indicateurs quantitatifs (débit maximal, latence moyenne) ainsi que des aspects qualitatifs (sécurité, extensibilité, facilité de déploiement et d'utilisation).

Dans la même perspective, l'étude de Nast et al. (Nast, Raddatz, Rother, Golatowski & Timmermann, 2023) analyse différents protocoles utilisés dans l'Internet des objets, qu'ils soient fondés sur le modèle publication/abonnement ou compatibles avec celui-ci. L'évaluation repose sur des critères industriels tels que l'interopérabilité, la sécurité, la capacité d'évolution, la compatibilité avec les contraintes de temps réel et la fiabilité. Les protocoles examinés incluent notamment MQTT, CoAP, AMQP, XMPP, MQTT-SN, DDS, OPC UA et DPWS.

L'étude (Dobbelaere & Sheykh Esmaili, 2017) s'intéresse à la comparaison entre Kafka et RabbitMQ, deux courtiers largement utilisés dans l'industrie. L'analyse comprend une évaluation qualitative ainsi qu'une évaluation quantitative dans un environnement centralisé représentatif d'une architecture en nuage. L'objectif est d'identifier les cas d'usage les plus appropriés pour chacun des courtiers.

Dans la même optique, Bagaskara et al. (Bagaskara, Setyorini & Wardana, 2020) proposent une comparaison similaire entre Kafka et RabbitMQ, mais cette fois dans un environnement de type brouillard informatique, plus représentatif du traitement en périphérie.

Ces travaux mettent en évidence que Kafka et RabbitMQ incarnent deux approches distinctes en matière de courtage de messages : les courtiers en file d'attente (Queue-based), comme

RabbitMQ, visent la distribution ciblée et flexible des messages, tandis que les courtiers fondés sur un modèle de journal structuré (Log-based), comme Kafka, sont davantage adaptés au traitement de flux, à la persistance des messages et à l'analyse en continu.

Une comparaison conceptuelle entre ces deux paradigmes, à savoir la messagerie traditionnelle et le traitement de flux, est résumée dans le tableau suivant (Dobbelaere & Sheykh Esmaili, 2017) :

Tableau 1.1 Comparaison entre messagerie et traitement de flux

| **Caractéristique** | **Messagerie (ex. RabbitMQ)** | **Traitement de flux (ex. Kafka)** |
|---|---|---|
| Style de communication | • Publication/abonnement<br>• Point à point avec file d'attente | publication/abonnement |
| Structure de base | File d'attente | Journal structuré / partitions |
| Mode de diffusion d'un message | • Un-à-plusieurs<br>• Un-à-un | Un-à-plusieurs |
| Mode de transmission | Envoi direct aux consommateurs *(push)* | Lecture déclenchée par les consommateurs *(pull)* |
| Durée de conservation des messages | Temporaire (jusqu'à consommation ou expiration) | Long terme (conservation selon une politique définie) |
| Consommation des messages | Message supprimé après traitement par un consommateur | Lecture multiple possible : un même message peut être relu par un ou plusieurs consommateurs |
| Gestion de l'état de lecture | • Centralisée au niveau du courtier<br>• L'état de lecture n'est pas maintenu par le client | • Répartie entre les composants<br>• Chaque consommateur gère sa position de lecture *(offset)* |
| Cas d'usage typiques | • Systèmes transactionnels<br>• Communication asynchrone | • Traitement continu de données<br>• Analyse en temps réel |

Un cas d'usage typique de Kafka concerne la collecte et l'analyse en temps réel des événements utilisateur, comme les clics, recherches ou ajouts au panier dans une application e-commerce. Chaque événement est envoyé immédiatement à un sujet Kafka. Le log immuable de Kafka conserve ces événements de manière ordonnée et durable, permettant à plusieurs applications

de les consommer simultanément pour des usages différents : mise à jour en temps réel des recommandations personnalisées ou stockage dans un entrepôt de données pour une analyse ultérieure.

RabbitMQ, quant à lui, est particulièrement adapté aux scénarios transactionnels nécessitant un routage complexe et des traitements asynchrones. Par exemple, lors du traitement d'une commande : après confirmation du paiement, un message est publié sur RabbitMQ. L'utilisateur reçoit immédiatement une confirmation, tandis que plusieurs services consommateurs exécutent en arrière-plan des tâches distinctes, telles que : l'envoi d'e-mails de confirmation, la mise à jour du stock, la génération de factures et la notification de partenaires logistiques. Grâce aux règles de routage avancées de RabbitMQ, les messages peuvent être distribués vers différentes files. Une fois la tâche exécutée, le message est supprimé de la file.

En conclusion, Kafka et RabbitMQ répondent à des besoins différents et peuvent être utilisés de manière complémentaire selon les cas d'usage : Kafka excelle dans le traitement de flux et l'analyse en continu, tandis que RabbitMQ est privilégié pour la messagerie transactionnelle, le routage complexe et la gestion asynchrone de tâches.

Dans ce travail, nous introduisons un courtier de messages orienté file d'attente, conçu pour étendre les mécanismes classiques de messagerie vers une infrastructure distribuée. Il adopte le fonctionnement d'un cluster géo-distribué, avec pour objectif de réduire la latence et de prendre en compte les contraintes matérielles propres aux environnements edge.

### 1.4.2 Courtiers distribués

Dans un système distribué basé sur un courtier, ce dernier joue un rôle central dans l'architecture : il assure entre autres la gestion des connexions clients, le traitement des abonnements et le routage des messages. Bien que cette approche centralisée soit simple à implémenter, elle présente des limitations majeures (Rahmani, 2023).

En effet, le courtier central constitue un point de défaillance unique : en cas d'indisponibilité, toute communication est interrompue. Aussi, cette architecture montre des limites en matière de montée en charge, notamment face à un énorme volume de données qui est souvent le cas dans un environnement IoT.

Pour surmonter ces limites, une solution consiste à répartir la charge sur plusieurs courtiers coopérants, déployés de manière distribuée tout en agissant comme une entité logique unique. Cette architecture distribuée, appelée cluster, est particulièrement adaptée aux environnements cloud, car elle permet une montée en charge horizontale par ajout dynamique d'instances. Elle offre également des garanties en matière de mise à l'échelle, de réplication, de résilience aux pannes et d'élasticité (Redondi, Arcia-Moret & Manzoni, 2019).

Ces clusters assurent une communication inter-courtiers permettant la redirection intelligente des messages, la découverte automatique des nœuds et la reprise automatique après incident. Toutefois, cette architecture, pouvant engendrer une surcharge réseau non négligeable. Pour limiter cet impact, les courtiers sont généralement déployés sur des machines ou conteneurs distincts interconnectés via des réseaux locaux ou régionaux à faible latence. C'est notamment le cas des déploiements répartis sur plusieurs zones de disponibilité au sein d'une même région, comme le propose Amazon MQ, qui prend en charge à la fois RabbitMQ et ActiveMQ (Amazon MQ, 2025). En l'absence de telles optimisations d'infrastructure, les délais de propagation et la consommation de bande passante peuvent rapidement devenir problématiques.

Comme mentionné précédemment, dans le contexte des réseaux 5G et des applications IoT à faible latence, il devient nécessaire de rapprocher les services cloud, y compris les courtiers, des utilisateurs, via les sites edge. Dans ces environnements contraints, le clustering classique devient difficilement applicable.

Dans ce cadre, des approches plus légères et adaptées à l'edge sont à privilégier. L'une d'elles est le bridging, utilisé par les courtiers MQTT. Cette technique permet d'interconnecter plusieurs instances de courtiers via des ponts réseau, en transférant les messages d'un courtier à un autre selon des règles de correspondance de sujets. Une approche plus générale, appelée fédération,

est utilisée dans les systèmes de messagerie multiprotocoles tels que RabbitMQ ou ActiveMQ Artemis. Elle consiste à établir des liens logiques entre courtiers distribués, permettant de rediriger dynamiquement les messages vers d'autres nœuds selon leur configuration. Dans les deux cas, ces liens conservent l'autonomie de chaque courtier.

Une étude comparative (Longo, Redondi, Cesana & Manzoni, 2022) portant sur les performances des courtiers MQTT avec support de clustering, par rapport à Mosquitto associé à MQTT-ST (utilisant le bridging), montre que le bridging surpasse le clustering en termes de surcharge réseau et de consommation de ressources machines.

Dans le cadre de notre travail, nous exploitons les liens de fédération afin de reproduire, du point de vue de l'utilisateur, un comportement équivalent à celui d'un cluster. Cela permet d'assurer l'acheminement des messages vers tous les clients intéressés et de répartir les messages entre les clients connectés à une même file logique, sur des nœuds géographiquement distribués.

## 1.5 Styles de communication

Dans cette section, nous présentons les deux styles de communication supportés par les courtiers ainsi que les travaux associés à chacun d'eux.

### 1.5.1 Point à point avec file d'attente

La transmission traditionnelle de messages en mode point à point présente plusieurs limitations dans les environnements distribués à grande échelle, notamment la congestion du réseau et un taux élevé de perte de messages (Hong, Deng, Li & Wang, 2021). Pour remédier à ces problèmes, des solutions reposant sur le modèle de file d'attente ont été largement développées.

Dans ce modèle, un producteur (ou émetteur) génère un message, généralement associé à une commande ou une action spécifique à effectuer. Ce message est ensuite envoyé vers une file d'attente, où il est stocké en attente de traitement. La livraison est assurée selon le modèle point à point, c'est-à-dire que chaque message est destiné à un seul consommateur. Toutefois,

plusieurs consommateurs peuvent être connectés à une même file d'attente, et des mécanismes d'équilibrage de charge permettent de répartir les messages entrants entre eux afin d'optimiser le débit global et d'éviter les goulots d'étranglement.

La littérature regroupe diverses stratégies d'équilibrage de charge entre consommateurs. Certaines reposent sur des approches simples, telles que le round-robin ou le round-robin pondéré, tenant compte de la disponibilité, de la charge instantanée ou encore du temps de réponse des consommateurs. D'autres travaux s'appuient sur des techniques plus avancées, notamment des approches adaptatives basées sur l'apprentissage automatique (Lei & Liu, 2024). Par ailleurs, une modélisation couramment utilisée repose sur les chaînes de Markov pour représenter le comportement des files d'attente (Santos *et al.*, 2021).

Dans les systèmes de l'Internet des objets (IoT), de nombreux travaux se sont concentrés sur l'*offloading* des tâches, soit vers des nœuds périphériques, soit vers le nuage (Islam, Debnath, Ghose & Chakraborty, 2021). Il est important de noter que notre approche ne s'inscrit pas dans cette logique. En effet, nous réalisons un équilibrage de charge entre plusieurs sites sans considérer la nature exacte des tâches à exécuter ni la capacité matérielle des hôtes. Contrairement au *task offloading*, qui nécessite une connaissance fine du type de tâche et des ressources matérielles requises, notre stratégie considère uniquement les messages comme des unités de traitement abstraites, sans inspection de leur contenu.

Par ailleurs, l'adoption du modèle point à point avec file d'attente et équilibrage de charge demeure relativement peu explorée. Selon les auteurs de l'étude (Yoshino, Watanobe & Naruse, 2021), la majorité des systèmes IoT actuels reposent essentiellement sur des communications de type capteur-vers-nuage, les interactions directes entre capteurs étant rares. Dans cette perspective, un protocole léger tel que MQTT en mode pub/sub est considéré comme suffisant pour assurer la transmission des données.

Nous considérons toutefois que cette vision est de plus en plus restrictive au regard de l'évolution des architectures IoT. Avec l'émergence du cloud hiérarchique, composé de plusieurs niveaux de traitement (périphérie, brouillard, nuage), les nœuds intermédiaires assument des rôles croissants

dans le traitement, le filtrage et la coordination des données. Dans ce cadre, l'intégration de modèles de communication point à point avec files d'attente et équilibrage de charge s'avère pertinente, notamment pour répondre aux enjeux de mise en échelle, de résilience, et de latence réduite.

Par ailleurs, la communication dans les environnements IoT ne se limite plus à un schéma unidirectionnel capteur-vers-nuage. Elle englobe désormais des flux bidirectionnels, des interactions entre dispositifs hétérogènes (capteurs, actionneurs, passerelles), ainsi que des communications locales au sein de la périphérie. Dans ce contexte, il est essentiel que les courtiers de messages soient capables de s'adapter aux différents styles de communication, dont le modèle point à point avec file d'attente.

### 1.5.2 Publication/abonnement

Le modèle publication/abonnement (pub/sub) est l'un des paradigmes de communication les plus répandus dans les systèmes distribués (Eugster, Felber, Guerraoui & Kermarrec, 2003). Il permet aux abonnés d'exprimer leur intérêt pour un ou plusieurs types d'événements auprès d'un intergiciel médiateur, chargé de notifier ces abonnés dès qu'un événement correspondant est publié par un producteur.

Le modèle pub/sub est par ailleurs particulièrement bien adapté aux applications de l'Internet des objets (IoT), dans la mesure où il assure un découplage spatial, temporel et contextuel entre producteurs et consommateurs (Nast *et al.*, 2023).

Trois variantes principales de ce modèle sont généralement distinguées :

- le modèle basé sur les sujets (*topic-based*), où les abonnés s'inscrivent à un canal identifié par un sujet explicite ou un modèle de sujet (par exemple via des expressions régulières) ;
- le modèle basé sur le contenu (*content-based*), où l'abonnement est défini par des filtres appliqués au contenu des messages ;
- le modèle basé sur le type (*type-based*), où les abonnés expriment leur intérêt pour des types d'événements définis dans un schéma ou une hiérarchie.

Grâce à sa flexibilité et à sa capacité d'adaptation aux environnements dynamiques, le paradigme pub/sub a fait l'objet d'un intérêt scientifique considérable.

À titre d'exemple, Dynamoth (Gascon-Samson, Garcia, Kemme & Kienzle, 2015) propose un intergiciel de type pub/sub fondé sur les sujets, intégrant une répartition dynamique de charge entre plusieurs courtiers déployés dans le nuage. Cette répartition, sensible aux caractéristiques des flux en temps réel, est organisée de manière hiérarchique, à la fois au niveau des sujets (macro-niveau) et au sein de chaque sujet (micro-niveau). Bien que ces travaux visent à améliorer les systèmes pub/sub dans des environnements centralisés, ce n'est pas l'objectif du présent travail, qui s'inscrit dans une logique différente, orientée vers des environnements périphériques.

Par ailleurs, plusieurs d'autres travaux se sont intéressés à l'optimisation de la distribution des messages dans des systèmes pub/sub basés sur des journaux (log-based) tels que Kafka. Ces approches proposent des alternatives aux mécanismes de routage par défaut, notamment le hashing cohérent pour l'acheminement des messages vers les partitions, et l'attribution round-robin pour la distribution des partitions entre courtiers (Dedousis, Zacheilas & Kalogeraki, 2018; Xie, Ji, Xu, Xia & Cao, 2023). Cependant, ces contributions s'inscrivent principalement dans une logique de traitement de flux à grande échelle, qui dépasse le périmètre de notre étude. Notre travail se concentre exclusivement sur les modèles de messagerie applicables aux environnements distribués de type périphérique, sans viser les problématiques propres à l'analyse de flux continus.

## 1.6 Plateformes publication/abonnement en périphérie

Le contexte de *edge computing* introduit de nouvelles contraintes pour les plateformes pub/sub traditionnelles conçues pour le cloud. En effet, les limitations en termes de latence, de bande passante et l'hétérogénéité des dispositifs imposent une révision des modèles de communication. L'envoi massif de données brutes vers les centres de données du nuage peut entraîner une congestion du réseau, notamment dans les systèmes IoT où les objets connectés génèrent de grandes quantités de données à des fréquences élevées. Ce phénomène peut non seulement

augmenter la latence, mais aussi soulever des enjeux de confidentialité et de sécurité des données, lorsqu'elles sont traitées ou transférées en dehors des zones de confiance. De plus, les coûts élevés de certaines technologies de communication (comme la LTE ou la 5G), ainsi que la variabilité significative de la bande passante, constituent des contraintes supplémentaires à prendre en compte.

### 1.6.1 Réseau dédié des courtiers (*overlay network*)

La diffusion efficace des messages dans les systèmes distribués pub/sub a suscité un intérêt croissant dans la littérature scientifique au cours des deux dernières décennies. Ces systèmes reposent généralement sur des réseaux dédiés, qui permettent aux nœuds de s'organiser dynamiquement dans un réseau virtuel, indépendamment de l'infrastructure physique sous-jacente. Parmi les protocoles les plus utilisés, MQTT s'est imposé comme une solution de référence, en raison de sa légèreté et de sa grande popularité dans les environnements contraints, notamment dans le domaine de l'IoT.

Pour répondre aux exigences croissantes des applications distribuées, plusieurs travaux ont exploré des mécanismes visant à rendre la diffusion plus locale et adaptative. À ce titre, Teranishi et al. (Teranishi, Banno & Akiyama, 2015) introduisent la notion de *location awareness*, dans laquelle les messages sont directement échangés entre éditeurs et abonnés lorsqu'ils résident dans le même réseau physique. Cette approche permet de réduire significativement la latence et la charge.

Dans cette même logique d'optimisation géographique, Kawaguchi et al. (Kawaguchi & Bandai, 2020) proposent une architecture reposant sur des topics hiérarchiques géographiques, organisés selon un *Z-ordering*. Ce mécanisme permet de structurer efficacement les abonnements par zones, sans qu'il soit nécessaire de synchroniser explicitement les informations entre courtiers. Chaque courtier gère ainsi une région géographique spécifique et redirige les messages vers d'autres courtiers en cas de non-responsabilité. Toutefois, cette approche présuppose que les producteurs soient capables d'identifier les zones cibles lors de la publication.

D'autres travaux se sont penchés sur l'optimisation du routage thématique. L'article de Longo et al. (Longo & Redondi, 2023) introduit une architecture où chaque sujet est associé à un réseau dédié construit sous forme de *Spanning Tree*. Cette organisation garantit que les messages ne sont relayés que par les nœuds réellement intéressés, ce qui permet d'éviter toute surcharge liée à des transmissions inutiles via des nœuds intermédiaires.

À l'inverse, des solutions comme (Banno, Sun, Takeuchi & Shudo, 2019) visent à interconnecter des courtiers hétérogènes à l'aide de nœuds intermédiaires placés entre clients et courtiers. Bien que cette architecture facilite la communication entre instances distinctes, elle repose sur une configuration manuelle lourde, difficilement évolutive et peu résiliente face aux défaillances dynamiques du réseau.

Pour remédier à ces limitations, certaines approches proposent d'introduire une supervision centralisée dynamique. Par exemple, (Park, Kim & Kim, 2018) utilise un contrôleur *SDN* capable de collecter des informations sur les clients et leurs abonnements pour créer des groupes *multicast* par sujet, optimisant ainsi la distribution des messages. D'autres travaux explorent la reconfiguration à l'exécution de la topologie des courtiers grâce à des messages spécifiques envoyés par une entité de contrôle. À titre d'exemple, le système EMMA (Rausch, Nastic & Dustdar, 2018b) intègre un contrôleur qui redirige dynamiquement les clients vers les courtiers les plus appropriés selon la charge et la latence, contribuant à une amélioration notable de la qualité de service. De manière similaire, EdgePub (Bouallegue & Gascon-Samson, 2022) propose un mécanisme d'équilibrage de charge intelligent, en attribuant dynamiquement à chaque sujet un courtier responsable, capable de s'adapter en fonction de l'évolution du trafic.

Dans cette perspective, nous proposons une solution réseau géodistribuée reposant sur le protocole AMQP 0-9-1, capable de gérer simultanément le modèle pub/sub ainsi qu'une adaptation du modèle point à point avec file d'attente, réinterprété en une variante pub/sub. Du côté du contrôle, notre approche s'appuie sur un orchestrateur central qui supervise dynamiquement le système et, à partir de données collectées en temps réel, génère une configuration réseau ajustée aux conditions courantes. Ce mécanisme assure une adaptation continue face aux fluctuations de

charge et à la dynamique propre aux environnements edge et IoT. Concrètement, la solution proposée introduit une structuration du réseau overlay en sous-topologies logiques, chacune dédiée à un groupe de messages, afin de garantir une diffusion ciblée et localement optimisée.

Nous avons choisi un contrôle centralisé afin de maximiser la visibilité sur l'ensemble de la topologie, ce qui permet de générer des configurations plus appropriées et de simplifier la gestion, tout en restant réaliste pour un contexte industriel. Bien qu'une approche décentralisée soit envisageable, comme le suggèrent certains travaux existants, elle serait plus complexe à implémenter et à administrer.

## 1.7 Protocoles de communication IoT

Contrairement au Web, qui repose essentiellement sur le protocole HTTP, l'IoT utilise une diversité de protocoles de communication, chacun conçu pour répondre à des exigences spécifiques. Dans ce qui suit, nous présentons les deux protocoles les plus couramment utilisés dans ce domaine : MQTT et AMQP 0-9-1.

### 1.7.1 Protocole MQTT

Le Message Queuing Telemetry Transport (MQTT) est un protocole de messagerie léger spécifiquement conçu pour les environnements contraints en ressources, tels que les systèmes embarqués, les objets connectés et les réseaux de capteurs. Il repose sur une architecture client–courtier et implémente le modèle publication/abonnement.

MQTT s'est imposé comme une référence dans le domaine de l'Internet des objets (IoT). Sa simplicité d'implémentation, sa faible empreinte mémoire, ainsi que sa capacité à fonctionner sur des réseaux peu fiables ou intermittents en font un protocole particulièrement bien adapté aux contraintes des applications IoT.

Le protocole propose trois niveaux de qualité de service (QoS), permettant de moduler le compromis entre fiabilité et performance selon les exigences applicatives :

- QoS 0 – au plus une fois : le message est transmis sans garanties de livraison ;
- QoS 1 – au moins une fois : le message est garanti d'être livré, mais peut être reçu en double ;
- QoS 2 – une seule fois : le message est livré de manière fiable, avec un mécanisme de contrôle garantissant l'unicité.

La majorité des plateformes cloud dédiées à l'IoT prennent aujourd'hui en charge MQTT, notamment Microsoft Azure IoT Hub, Google Cloud IoT Core, AWS IoT, parmi de nombreuses autres(Al-Masri *et al.*, 2020).

Par ailleurs, de nombreuses implémentations du protocole MQTT sont disponibles, qu'elles soient open source ou commerciales. Parmi les plus populaires, on peut citer Mosquitto, EMQX, HiveMQ, ou encore VerneMQ. Ces courtiers diffèrent selon plusieurs dimensions, notamment en termes de performance, de mise en échelle, de résilience, de sécurité, d'extensibilité et de facilité d'intégration (Koziolek *et al.*, 2020).

### 1.7.2 Protocole AMQP 0-9-1

Une confusion fréquente existe dans la littérature, ainsi que dans les pratiques industrielles, entre les deux versions du protocole AMQP : AMQP 0-9-1 et AMQP 1.0. Contrairement à ce que leur numérotation pourrait laisser entendre, il ne s'agit pas de deux versions successives d'un même protocole, mais bien de deux protocoles fondamentalement différents (AMQP, 2025).

AMQP 1.0 est un protocole de communication de niveau réseau, orienté pair-à-pair, qui définit les échanges entre deux entités symétriques. Il est standardisé par l'OASIS et supporté par plusieurs courtiers tels que Microsoft Azure Service Bus, Apache ActiveMQ Artemis et IBM MQ. Ces implémentations peuvent varier considérablement sur le plan architectural, car le protocole ne spécifie pas une structure de courtier centralisé. (Chen, 2025)

En revanche, AMQP 0-9-1 est un protocole léger de la couche application, fondé sur un modèle client-courtier, où le courtier joue un rôle central dans la réception, le routage et la distribution des messages. Ce protocole prend en charge à la fois les modèles de communication point à

point avec file d'attente et publication/abonnement. Le modèle de routage défini par le protocole AMQP 0-9-1 repose sur trois entités fondamentales :

- **Échanges (*exchanges*)** : points d'entrée des messages dans le système, ils hébergent la logique de routage et hébergent la base du modèle pub/sub.
- **Files d'attente (*queues*)** : structures de stockage temporaire des messages en attente de consommation.
- **Liaisons (*bindings*)** : règles qui relient les échanges aux files d'attente.

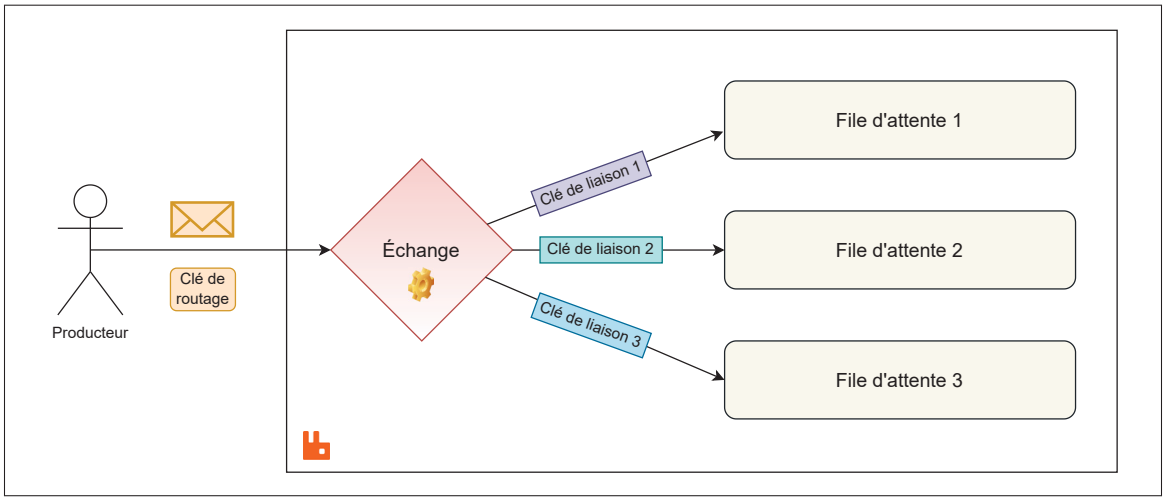



Figure 1.2 Architecture du protocole AMQP 0-9-1

Lorsqu'un message est publié, il est envoyé vers un échange. Le routage est alors déterminé en fonction de plusieurs éléments : le type de l'échange, la clé de routage associée au message, ainsi que les clés de liaison (*bindings*) configurées. Comme illustré dans la figure 1.2, un échange peut être relié à plusieurs files d'attente, chaque file avec sa propre clé de liaison. De plus, une file d'attente peut être connectée à plusieurs échanges, et il est possible pour une même file d'avoir plusieurs liaisons avec un même échange.

AMQP 0-9-1 définit quatre types principaux d'échanges (RabbitMQ, 2023) :

- **Direct** : le message est acheminé vers les files d'attente dont la clé de liaison correspond exactement à la clé de routage du message.

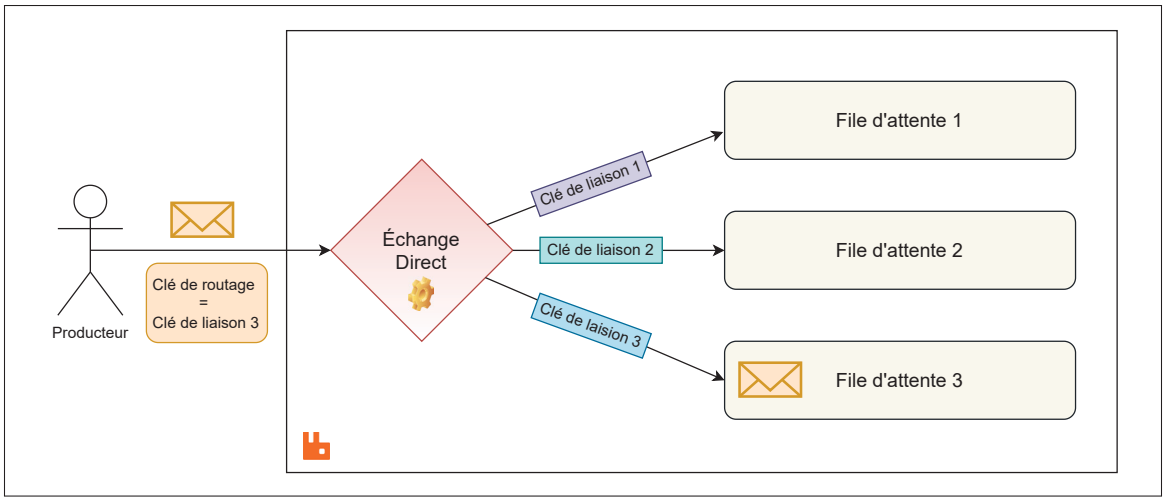



Figure 1.3 Échange de type *Direct*

Dans l'exemple de la figure 1.3, la clé de routage du message correspond exactement à la clé de liaison de la file d'attente 3, ce qui explique le routage vers cette file. Il est important de noter que plusieurs files d'attente peuvent partager la même clé de liaison, entraînant ainsi la duplication du message.

- **Fanout** : le message est distribué à toutes les files d'attente liées à l'échange, sans considération de la clé de routage.

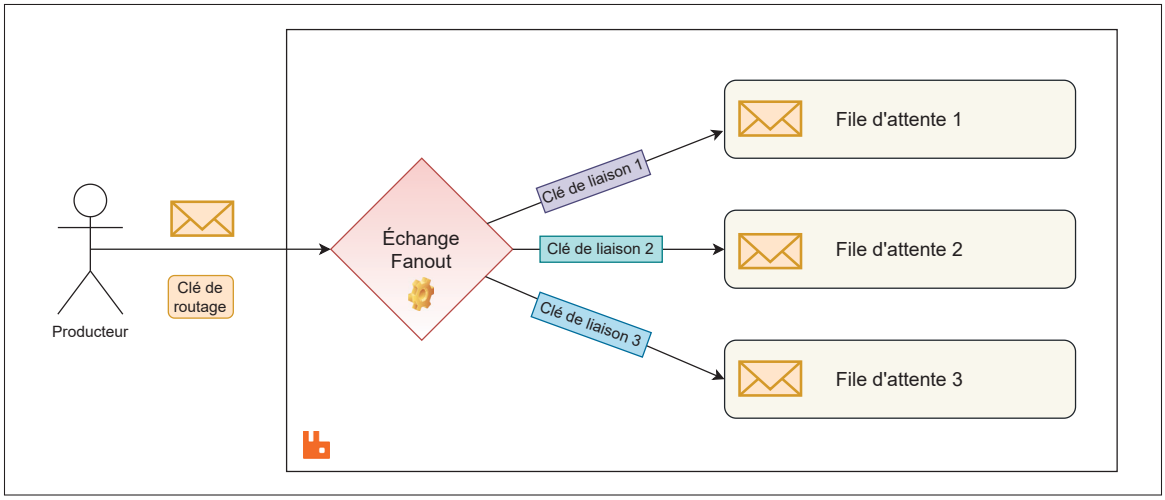



Figure 1.4 Échange de type *Fanout*

- **Sujet (*topic*)** : le message est routé en fonction d'un motif hiérarchique exprimé dans la clé de liaison, à l'aide des caractères spéciaux * (remplace un seul niveau) et # (remplace un ou plusieurs niveaux).

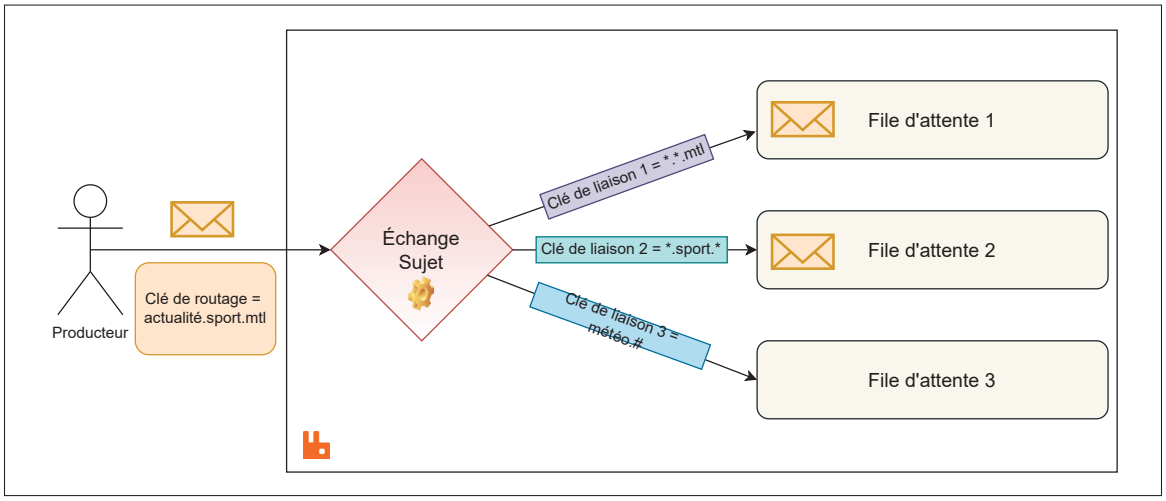



Figure 1.5 Échange de type Sujet

Comme illustré dans la figure 1.5 :

- La file 1, avec la clé `*.*.mtl`, reçoit tous les messages dont le sujet contient exactement trois niveaux hiérarchiques, avec « mtl » en troisième position.
- La file 2, avec `*.sport.*`, s'abonne à tous les sujets ayant « sport » en deuxième position, également sur trois niveaux.
- La file 3, avec `météo.#`, est intéressée par tous les sujets commençant par « météo », indépendamment du nombre de niveaux supplémentaires.

- **En-tête (*Headers*)** : le routage repose sur les en-têtes du message plutôt que sur une clé de routage explicite. Chaque liaison spécifie des paires clé-valeur à faire correspondre, ainsi qu'un attribut spécial `x-match`, qui peut prendre la valeur *all* (tous les critères doivent être remplis) ou *any* (au moins un critère doit l'être).

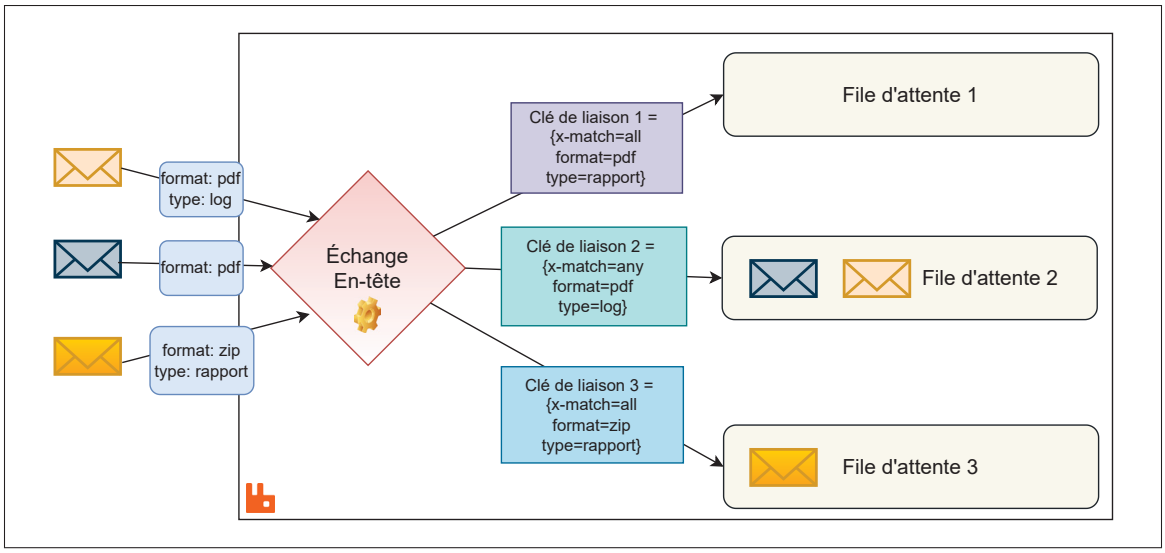



Figure 1.6 Échange de type *Headers*

Dans l'exemple de la figure 1.6 :

- La file 1, avec `x-match=all`, reçoit uniquement les messages ayant les en-têtes `format=pdf` et `type=rapport`.
- La file 2, avec `x-match=any`, accepte les messages ayant soit `format=pdf`, soit `type=log`.
- La file 3 filtre les messages avec `format=zip` et `type=rapport` en `x-match=all`.

Ce type d'échange est plus coûteux en performances et est généralement moins utilisé que les échanges *direct*, *topic* ou *fanout*.

### Modèle point à point avec file d'attente

Dans ce modèle, plusieurs consommateurs peuvent se connecter à une même file (figure 1.7). Toutefois, pour chaque message donné, un seul consommateur est sélectionné parmi l'ensemble des consommateurs disponibles. Cette propriété rend le modèle particulièrement adapté aux scénarios de répartition de charge, où des tâches soumises à des microservices doivent être distribuées de manière équilibrée entre plusieurs instances.

Ce modèle repose sur l'utilisation de l'échange par défaut, un échange *direct* prédéfini par le courtier. Le producteur publie directement un message vers une file d'attente en utilisant son nom comme clé de routage, sans spécifier de nom d'échange (chaîne vide ""). La file d'attente est reliée à cet échange avec une clé de liaison égale au nom de la file. Ainsi, plusieurs consommateurs peuvent se connecter à une même file.

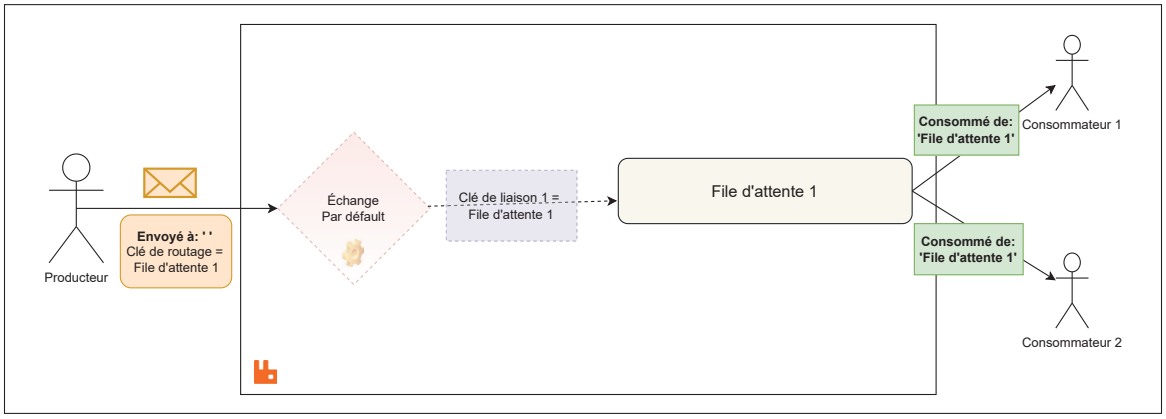



Figure 1.7 Fonctionnement du modèle point à point avec file d'attente

Comme nous le verrons dans les sections suivantes, il est possible de créer des consommateurs concurrents sur une même file en exploitant d'autres types d'échanges. Notre système LuffyMQ généralise ce principe en permettant la concurrence de consommateurs géo-distribués, quel que soit le type d'échange utilisé.

### Modèle publication/abonnement

Le modèle *pub/sub* est généralement implémenté à l'aide des échanges de type *topic*. Les échanges *fanout* et *direct* peuvent être vus comme des cas particuliers de ce type : un échange *topic* avec une clé de liaison égale à # se comporte comme un échange *fanout*, tandis qu'un échange *topic* avec une clé sans motif se comporte comme un échange *direct*.

Dans ce modèle, lorsqu'un consommateur s'abonne à un sujet sans spécifier de file nommée, le courtier crée automatiquement une file temporaire dédiée pour ce consommateur, afin qu'il reçoive uniquement les messages correspondant au motif souhaité.

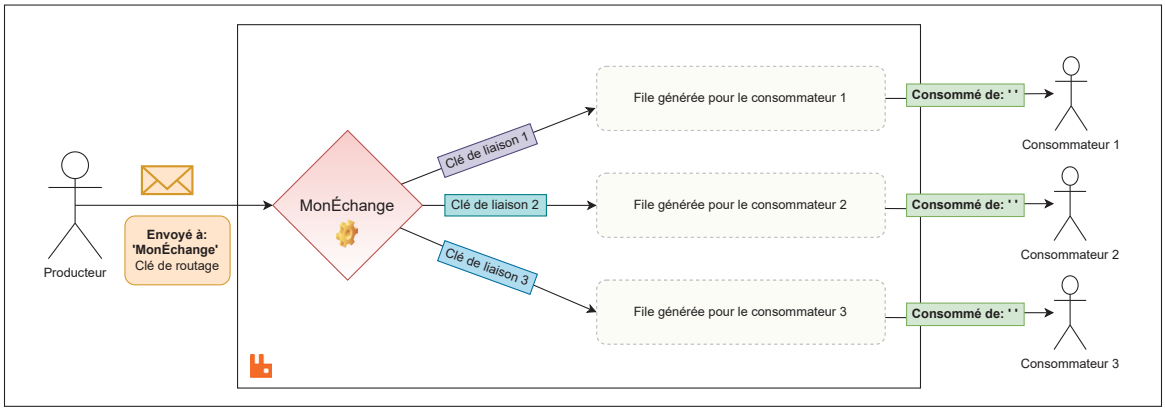



Figure 1.8 Fonctionnement du modèle publication/abonnement

### Modèle hybride

Le protocole AMQP 0-9-1 offre un modèle hybride combinant les deux modèles fondamentaux. Dans ce modèle, les consommateurs sont reliés à une file d'attente spécifique tandis que les producteurs publient leurs messages vers un échange particulier. Comme illustré sur la figure 1.9, toutes les composantes du système, files, échanges et liaisons, sont ainsi explicitement déclarées et nommées. Ce mécanisme permet de mettre en place un routage des messages plus avancé et flexible.

RabbitMQ est le courtier de messages le plus largement utilisé implémentant le protocole AMQP 0-9-1. Il enrichit les possibilités de routage en offrant, par exemple, le routage d'un échange vers un autre, et il prend en charge des modules complémentaires intégrant d'autres logiques de routage au-delà des quatre types d'échanges standards présentés. Quelques modules supplémentaires seront détaillés dans la section 1.10.

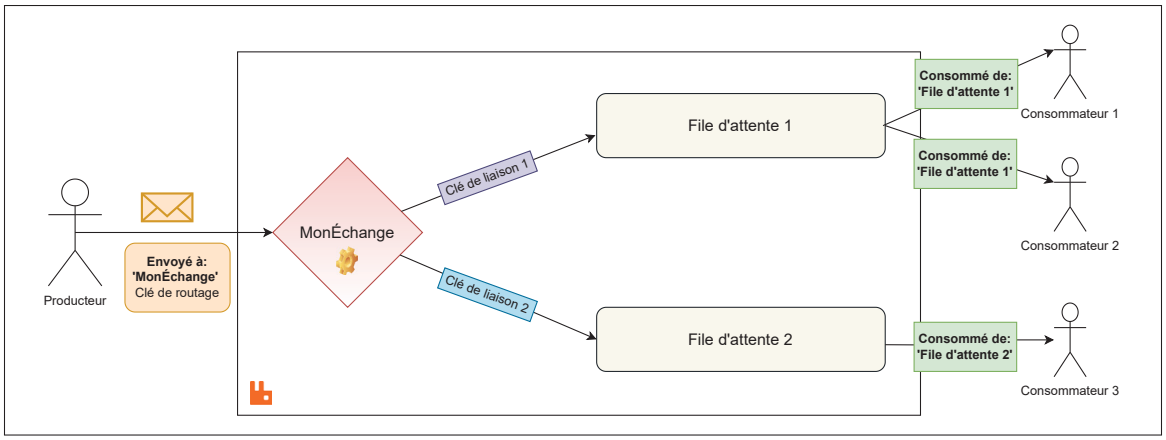



Figure 1.9 Fonctionnement du modèle hybride

## 1.8 Systèmes de messagerie multiprotocoles

Les systèmes de messagerie multiprotocoles offrent une flexibilité d'intégration élevée dans des environnements hétérogènes, en permettant la communication entre composants utilisant des protocoles différents. Dans cette section, nous étudions deux solutions open source largement utilisées : ActiveMQ Artemis et RabbitMQ.

Ces deux intergiciels prennent en charge les modèles de communication point à point avec file d'attente et publication/abonnement, tout en assurant une interopérabilité transparente entre les protocoles pris en charge. Ils offrent également des mécanismes de routage , adaptés à une grande variété de cas d'usage allant des architectures de messagerie traditionnelles jusqu'aux systèmes plus dynamiques tels que les microservices ou le edge computing.

### 1.8.1 ActiveMQ Artemis

ActiveMQ Artemis est un système de messagerie hautes performances, développé dans le cadre du projet Apache ActiveMQ. Il constitue l'évolution directe de la version classique d'ActiveMQ, avec une architecture repensée, modernisée et optimisée pour les environnements distribués.

Contrairement à son prédécesseur, principalement conçu comme serveur JMS (Java Message Service), ActiveMQ Artemis prend en charge plusieurs protocoles de communication, soit de manière native, soit via des extensions. Il prend notamment en charge AMQP 1.0, CORE (protocole interne optimisé), OpenWire (assurant la compatibilité avec la version classique d'ActiveMQ), ainsi que MQTT et STOMP.

L'une des spécificités clés d'Artemis réside dans son architecture interne unifiée, indépendante du protocole utilisé. Cette architecture repose sur deux concepts fondamentaux : les adresses et les files d'attente. Elle permet de prendre en charge aussi bien les communications point à point avec file d'attente que publication/abonnement, en fonction du type d'adresse utilisé. Une adresse de type multicast permet de diffuser un message à l'ensemble des abonnés, chacun recevant une copie via une file interne distincte. À l'inverse, une adresse de type anycast dirige les messages vers une unique file partagée, garantissant qu'un seul consommateur traite chaque message.

### 1.8.2 RabbitMQ

RabbitMQ est l'un des courtiers de messages les plus populaires dans le domaine des systèmes distribués. Il se distingue par sa prise en charge de multiprotocole assuré nativement ou à travers des modules complémentaires *(plugins)*. Parmi les protocoles pris en charge, on retrouve AMQP 0-9-1, AMQP 1.0, MQTT, ainsi que STOMP.

L'architecture de base de RabbitMQ repose sur le modèle défini par AMQP 0-9-1. Comme discuté auparavant, la messagerie est fondée autour des entités suivantes : échanges *(exchanges)*, files d'attente *(queues)* et liaisons *(bindings)*. Les autres protocoles, comme MQTT, sont implémentés en s'appuyant sur cette infrastructure.

Dans ses versions les plus récentes, RabbitMQ introduit également un mode de traitement de flux *(stream processing)*, reposant sur un modèle de journal *(log-based)*. Ce mode de fonctionnement s'écarte de l'architecture AMQP 0-9-1 et vise à répondre aux besoins de cas d'usage nécessitant une persistance durable, une relecture de messages, ou un traitement continu à haute performance,

similaire à ce que propose Apache Kafka. Cette fonctionnalité relève d'un paradigme différent de celui étudié dans le cadre du ce travail, et ne sera donc pas abordée.

### 1.8.3 Comparaison des fonctionnalités distribuées entre RabbitMQ et ActiveMQ Artemis

RabbitMQ et ActiveMQ Artemis intègrent des fonctionnalités avancées de déploiement distribué. Bien qu'ils partagent des caractéristiques communes, telles que la prise en charge de multiples protocoles, du clustering et de la fédération, ils se distinguent par leur modèle architectural, leur flexibilité de configuration, ainsi que la manière dont ils gèrent le routage des messages et la répartition de la charge.

Le tableau ci-dessous présente une synthèse comparative des principales fonctionnalités distribuées offertes par ces deux systèmes :

Tableau 1.2 Comparaison des fonctionnalités distribuées entre RabbitMQ et ActiveMQ Artemis

| **Fonctionnalité** | **RabbitMQ** | **ActiveMQ Artemis** |
|---|---|---|
| **Protocoles pris en charge** | AMQP 0-9-1, AMQP 1.0, MQTT, STOMP, WebSockets | AMQP 1.0, MQTT, STOMP, OpenWire, CORE |
| **Clustering** | • Réseau maillé avec communication entre pairs<br>• Support des files de type quorum basées sur Raft | • Topologie souple et modulable<br>• Prise en charge avancée du rééquilibrage des messages entre nœuds |
| **Fédération** | • Réplication unidirectionnelle des messages<br>• Déplacement unidirectionnel des messages | • Réplication unidirectionnelle des messages<br>• Déplacement unidirectionnel ou bidirectionnel des messages |
| **Gestion des politiques (policy)** | Configuration via interfaces en ligne de commande et interface graphique | Configuration déclarative via fichiers XML (ex. `broker.xml`) avec paramétrage détaillé |

Bien qu'ActiveMQ Artemis offre des fonctionnalités plus avancées et une architecture modulaire, la gestion des politiques, y compris celles liées à la fédération, reste statique à l'exécution.

Autrement dit, toute modification de configuration nécessite un redémarrage du courtier pour être prise en compte.

En revanche, RabbitMQ permet une modification dynamique de la plupart des paramètres par des politiques, sans interruption de service. Cette souplesse en fait une solution particulièrement adaptée aux environnements dynamiques et évolutifs.

C'est dans cette optique que nous avons choisi d'utiliser RabbitMQ pour la mise en œuvre de notre système. Les sections suivantes présentent les modules d'extension de RabbitMQ que nous avons intégrés pour réaliser notre architecture.

## 1.9 Gestion des messages non distribués dans RabbitMQ

RabbitMQ propose un mécanisme de gestion des messages non traités à travers un système d'échange appelé *Dead Letter Exchange (DLX)*. Lorsqu'un message ne peut pas être livré ou traité correctement, il peut être redirigé vers un DLX, qui le republie dans une autre file d'attente nommée *Dead letter queue (DLQ)* pour un traitement ultérieur, une analyse ou une inspection.

Un message peut être redirigé vers un DLX dans les cas suivants :

- Il est explicitement rejeté par un consommateur AMQP 0.9.1 avec le paramètre *requeue=false* ;
- Il expire suite à un TTL *(Time-To-Live)* défini au niveau du message ;
- Il est supprimé, car la file d'attente a dépassé sa limite de longueur ;
- Il a été redélivré plus de fois que la limite autorisée dans une file.

Les DLX sont des échanges standards. Ils peuvent être de n'importe quel type habituel et sont déclarés comme tout autre échange dans RabbitMQ.

## 1.10 Modules d'extension de RabbitMQ (*Plugins*)

RabbitMQ propose de nombreuses extensions sous forme de modules d'extension permettant d'enrichir ses fonctionnalités de base. Ces modules offrent des capacités supplémentaires, notamment la prise en charge de protocoles supplémentaires, la surveillance de l'état du système,

de nouveaux types d'échanges AMQP 0-9-1, ainsi que la fédération de nœuds. De nombreuses fonctionnalités sont intégrées sous forme de module inclus dans la distribution de base. RabbitMQ prend également en charge les modules tiers (*third-party*), permettant ainsi aux utilisateurs d'étendre encore davantage ses fonctionnalités pour répondre à des besoins spécifiques. Nous nous concentrerons uniquement sur les modules essentiels pour une meilleure compréhension de notre système.

### 1.10.1 Liens de fédération

Le plugin de fédération permet le transfert de messages entre des courtiers distribués. Contrairement au *clustering*, la fédération est particulièrement adaptée aux réseaux étendus (*WAN*) et peut tolérer des connexions intermittentes. Elle gère automatiquement les pannes de liaison et tente une récupération en cas d'échec.

Ce plugin prend en charge deux types de fédération :

- La fédération de files d'attente (*queue federation*),
- La fédération d'échanges (*exchange federation*).

Une file d'attente ou un échange fédéré peut recevoir des messages provenant d'un ou plusieurs instances distantes, appelées upstreams. Ces upstreams correspondent à des échanges ou des files d'attente situés sur les instances RabbitMQ distantes.

Les liens de fédération se connectent aux upstreams de manière similaire à une application classique, en utilisant le protocole AMQP 0.9. Cette architecture permet également de connecter des courtiers exécutant différentes versions de RabbitMQ, offrant ainsi une flexibilité dans la gestion de l'infrastructure.

Comme toute configuration, les liens de fédération peuvent être définis statiquement dans le fichier de configuration. Toutefois, la méthode recommandée consiste à utiliser des règles de configuration dynamiques (*policies*), qui permettent une gestion plus flexible. Ces règles peuvent être modifiées à chaud (*runtime*), permettant ainsi d'ajouter, mettre à jour ou supprimer des liens

de fédération en temps réel, sans interruption du service. Cette approche permet d'adapter la fédération de manière dynamique en fonction des besoins opérationnels.

### 1.10.2 Fédération de files d'attente

Un lien de fédération de files d'attente permet de relier deux files situées sur des nœuds distincts mais représentant une même file logique, dans le but de répartir la charge de traitement et d'améliorer les performances globales du système.

Lorsque la file en aval (appelée *downstream*) ne dispose plus de messages locaux à consommer mais que des consommateurs y sont encore actifs, elle interroge activement la file en amont (*upstream*) pour récupérer de nouveaux messages. Contrairement à un mécanisme de réplication ou de diffusion automatique, ce modèle repose sur une logique de récupération à la demande (*pull-driven*), où l'initiative du transfert revient au site demandeur.

Ce comportement de fédération des files d'attente est présenté dans la figure 1.10.

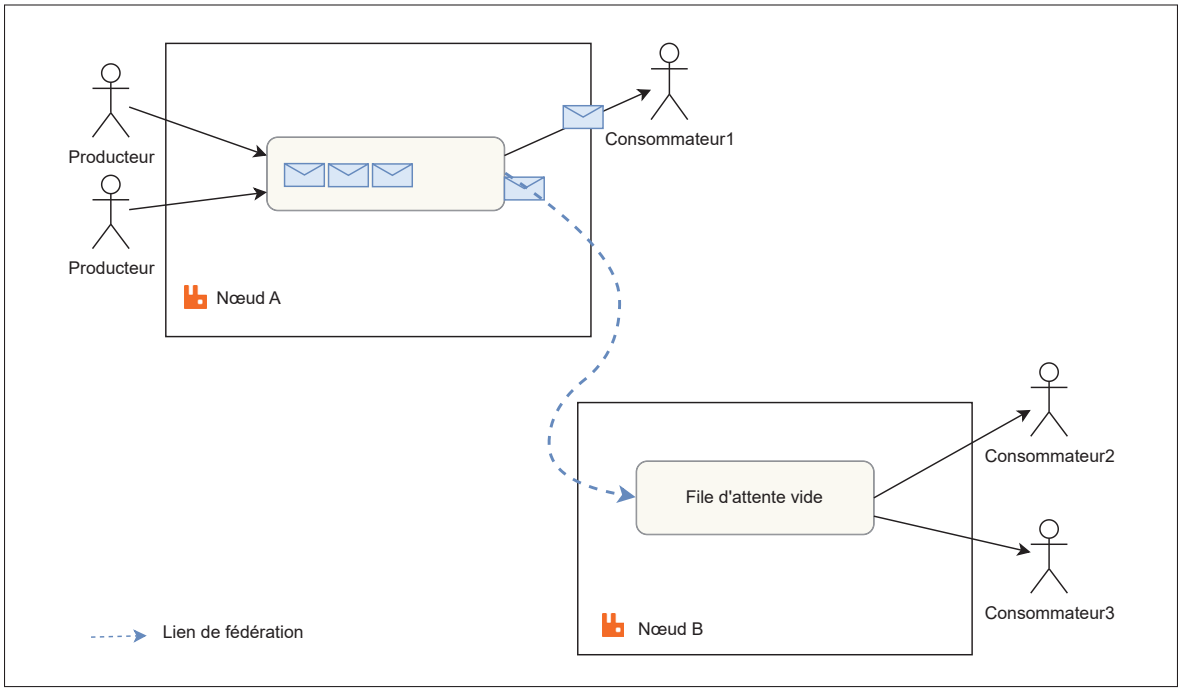



Figure 1.10 Fédération des files d'attente

### 1.10.3 Fédération des échanges

Contrairement à la fédération de files d'attente, qui déplace les messages en fonction de la disponibilité des consommateurs, la fédération des échanges réplique les messages publiés dans un nœud en amont vers les nœuds en aval.

Ce mécanisme repose sur la propagation des liaisons : les abonnements des clients en aval sont transmis au nœud en amont, permettant ainsi à l'échange amont de connaître les intérêts des consommateurs situés en aval.

En pratique, le lien de fédération agit comme un client applicatif pour le nœud en amont. Il s'abonne aux messages correspondant aux intérêts des clients en aval. Ainsi, dès que l'échange en amont reçoit un message correspondant à ces abonnements, il l'envoie au client fédéré, qui le transmet ensuite au nœud en aval.

La figure 1.11 illustre le fonctionnement de ce type de fédération.

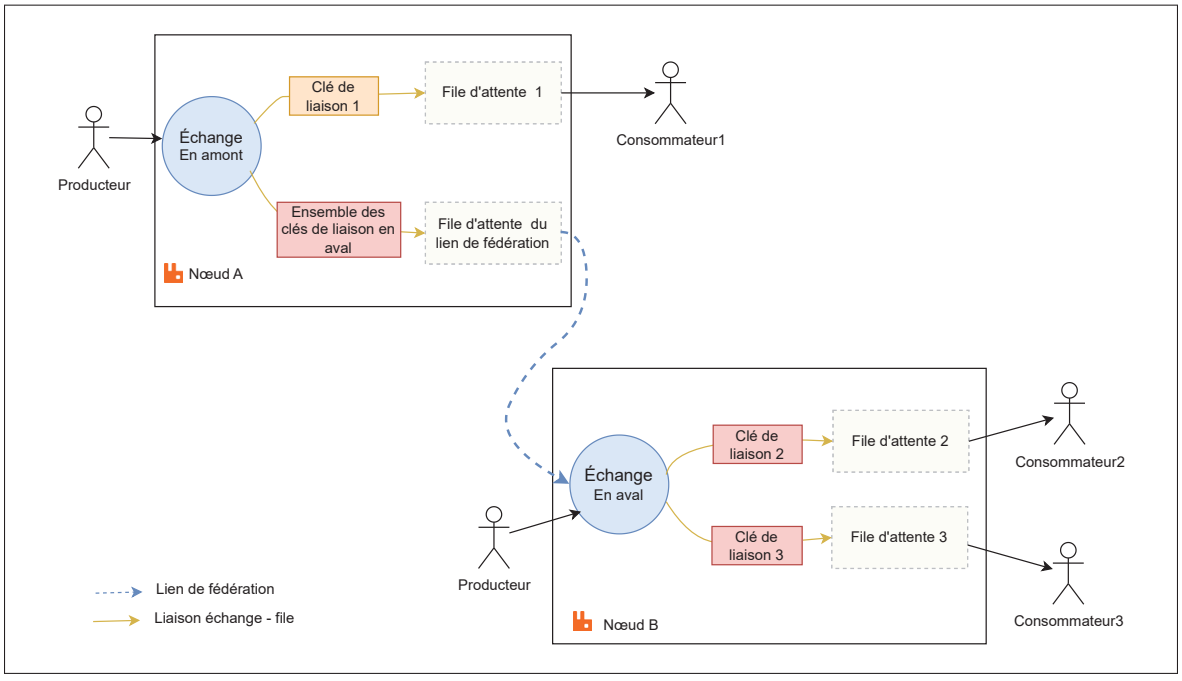



Figure 1.11 Fédération des échanges

### 1.10.4 Échange d'événement

Le RabbitMQ Event Exchange permet aux utilisateurs de suivre les événements internes du courtier, tels que la création et la suppression de files d'attente, les connexions et déconnexions des clients, ainsi que l'expiration des messages. Il repose sur un échange de type sujet, où les clés de routage des messages publiés indiquent des événements spécifiques, comme queue.created ou binding.deleted. Ces messages sont ensuite acheminés vers des files d'attente ciblées en fonction des liaisons configurées. Ce mécanisme est particulièrement utile pour la surveillance, l'audit et l'automatisation des actions en réponse aux changements du système.

### 1.10.5 Échange hachage cohérent

L'algorithme de hachage cohérent proposé par (Karger *et al.*, 1997), a été initialement conçu pour résoudre le problème des Hot Spots sur Internet. Grâce à ses propriétés d'équilibrage de charge, sa tolérance aux pannes et sa mise en échelle, il s'est imposé comme une solution largement adoptée dans les systèmes distribués et continue à être utilisé dans divers domaines (Wu, Li & Wei, 2023).

Le fonctionnement du hachage cohérent repose sur la création d'un anneau de hachage, où les données et les nœuds sont positionnés à l'aide d'une fonction de hachage. Le principe de fonctionnement de l'algorithme de hachage cohérent est illustré dans la figure 1.12 , et son déroulement spécifique est le suivant :

1. Création de l'anneau de hachage.
2. Mappage des nœuds sur l'anneau : Chaque nœud (ex : serveur ou file d'attente) est assigné à une position sur l'anneau en appliquant une fonction de hachage sur une valeur caractéristique (ex : ID du serveur).
3. Mappage des données sur l'anneau : Les données (ex : messages dans RabbitMQ) sont également hachées et placées sur l'anneau selon le même principe.
4. Attribution des données aux nœuds : En parcourant l'anneau, le nœud le plus proche de l'emplacement où les données ont été mappées les stockera.

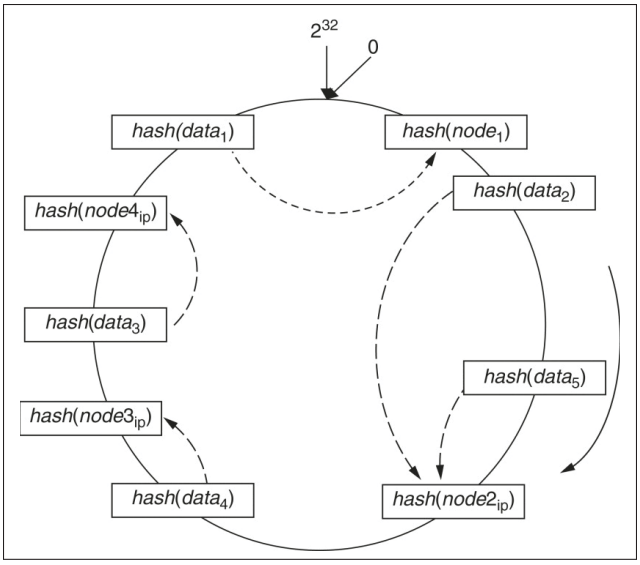



Figure 1.12 Hachage Cohérent
Tirée de Wu, Li & Wei(2023)

L'échange de hachage cohérent de RabbitMQ applique ce principe pour répartir dynamiquement les messages entre plusieurs files d'attente de manière équilibrée et prévisible.

Chaque file d'attente associée à l'échange reçoit un nombre de partitions déterminé par son poids de liaison, défini par sa clé de liaison qui doit obligatoirement être un entier. Plus le poids d'une file d'attente est élevé, plus elle reçoit de messages, ce qui permet un équilibrage de charge ajustable.

La distribution des messages repose sur une propriété du message, généralement sur la clé de routage, utilisée comme clé de hachage. Ainsi, dans un système où de nombreuses clés de routage sont utilisées, les files d'attente avec les poids plus élevés sont plus susceptibles de recevoir un plus grand volume de messages. Nous exploitons ce principe pour transformer le modèle point à point en une variante du modèle pub/sub, où le poids reflète la capacité de chaque file à traiter les messages. Ce comportement est présenté en détail dans la section 2.2.2.

## 1.11 La pertinence du projet

Notre solution LuffyMQ met en place des réseaux pub/sub dédiés à chaque groupe de messages via les échanges du protocole AMQP 0-9-1, dans le but de synchroniser l'ensemble des sites en périphérie. Elle transforme également le mode de communication point à point avec file d'attente en un modèle pub/sub, ce qui permet de répartir intelligemment la charge entre les consommateurs répartis sur différents nœuds tout en priorisant le traitement local des messages.

En parallèle, LuffyMQ vise à réduire la latence moyenne des échanges de données entre les nœuds périphériques, afin de répondre aux exigences des applications critiques en matière de temps de réponse, tout en respectant les contraintes de bande passante.

À notre connaissance, il n'existe actuellement aucune autre solution qui propose un système de messagerie distribué unifié avec une telle approche.

# CHAPITRE 2

# APPROCHE

Ce chapitre présente en détail l'approche méthodologique adoptée dans le cadre de ce travail. Il s'articule autour de plusieurs sections. La première introduit les objectifs de recherche, déclinés en un objectif global et en sous-objectifs spécifiques. La section 2.2 décrit l'architecture statique du système, tandis que la section 2.3 expose l'aspect dynamique des flux de messages. Nous consacrons ensuite une section à la formulation du problème étudié, suivie de la présentation de la méthode heuristique retenue pour sa résolution. Enfin, nous détaillions successivement l'architecture logicielle du courtier, puis celle de l'orchestrateur *LuffyMQ*.

## 2.1 Objectifs de recherche

L'objectif général de cette étude est de concevoir, d'implémenter et d'évaluer un système de messagerie en files d'attente géodistribuée, capable de prendre en charge à la fois les modèles publication/abonnement et point à point, et spécifiquement adapté aux environnements en périphérie. Ce système vise à réduire le temps d'attente dans la file et la latence de transmission des messages entre sites distribués, tout en respectant les contraintes de bande passante. Afin d'atteindre cet objectif, quatres sous-objectifs sont définis, portant notamment sur les performances en matière de latence et de débit :

1. Évaluer l'efficacité de la stratégie d'équilibrage de charge de *LuffyMQ* dans le modèle point à point avec files d'attente.
2. Analyser les performances de *LuffyMQ* dans le modèle publication/abonnement.
3. Examiner la scalabilité du système.
4. Étudier la sensibilité du système en faisant varier ses paramètres expérimentaux.

## 2.2 Architecture de LuffyMQ

La figure 2.1 présente une vue d'ensemble de l'architecture proposée, mettant en évidence ses principaux composants :

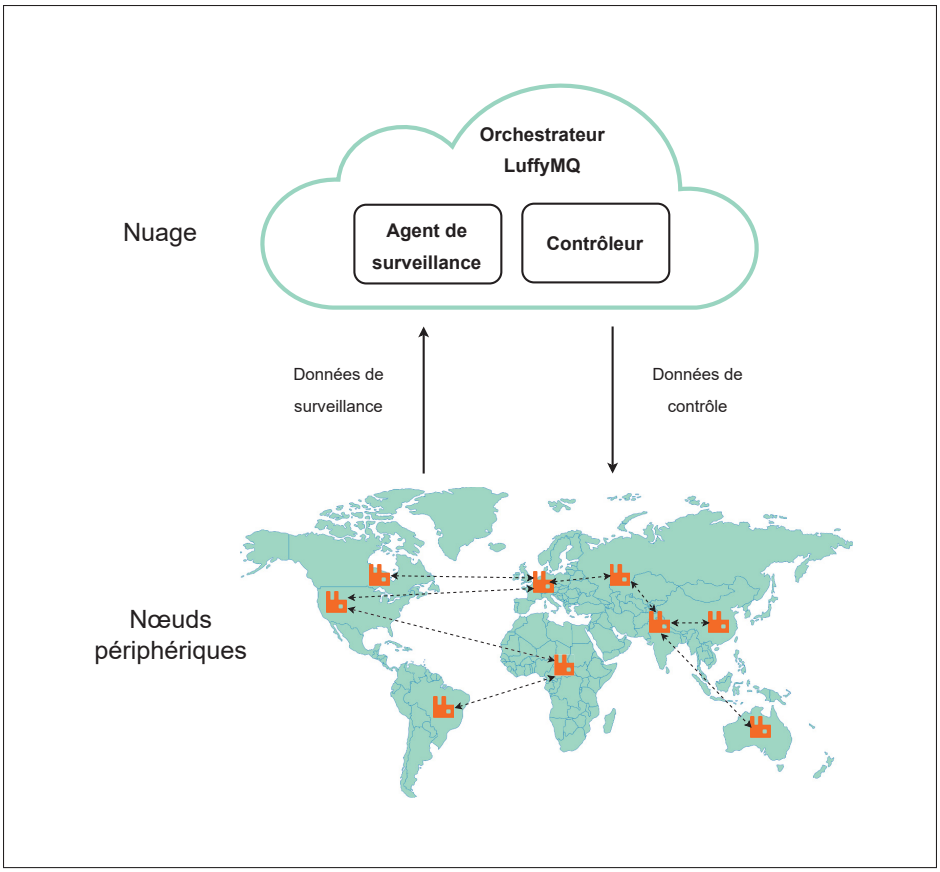



Figure 2.1 Vue d'ensemble du système

- Nœuds edge : Ce sont des nœuds distribués géographiquement, chacun hébergeant une instance logique du courtier de messages, en tant qu'instance unique ou en cluster. Le terme *nœud edge* désigne, par simplification, le courtier déployé sur ce site. Chaque nœud peut communiquer avec les autres si nécessaire. Les clients (producteurs et consommateurs) se connectent au nœud edge le plus proche.
- Surveillance : Un agent de surveillance collecte des données en temps réel sur l'état des nœuds edge, incluant des métriques clés telles que le nombre moyen de messages reçus et envoyés aux clients locaux, les messages échangés entre les nœuds, les débits de transmission de données, et d'autres indicateurs de performance.
- Contrôleur : À partir des données collectées par l'agent de surveillance, le contrôleur détermine la configuration optimale des connexions entre échanges. L'algorithme de prise de

décision est détaillé dans les sections suivantes. Le contrôleur publie ensuite les configurations de liens nécessaires dans une file de configuration dédiée sur chaque nœud. Chaque nœud exécute un agent de configuration abonné à cette file, qui traite les mises à jour et ajuste dynamiquement les connexions internœuds.

Dans les sections suivantes, nous détaillons l'architecture interne de chacun des composants du système. Nous commençons par les nœuds edge, en mettant en évidence leur prise en charge des deux styles de messagerie : la publication/abonnement et le mode point à point avec file d'attente.

### 2.2.1 Modèle publication/abonnement

Le système proposé repose sur les échanges RabbitMQ, en synchronisant les échanges portant le même nom sur différents nœuds edge. Chaque nœud peut héberger plusieurs échanges. Pour chaque échange réparti sur plusieurs nœuds, un nœud de synchronisation est désigné. En plus de son rôle habituel de servir les clients locaux, ce nœud agit comme coordinateur central : il recueille les abonnements des clients situés sur les autres nœuds et reçoit les messages publiés localement par ces derniers. Il se charge ensuite de redistribuer les messages aux nœuds auxquels des clients sont abonnés à l'échange concerné.

La figure 2.2 illustre une topologie composée de cinq nœuds edge RabbitMQ, chacun participant à un ou plusieurs groupes d'échanges, représentés par des couleurs distinctes. Chaque groupe correspond à un échange global logique et contient exactement une instance de synchronisation (symbolisée par un fond en zigzag). À chaque échange, des liens de fédération bidirectionnels relient l'instance de synchronisation aux autres instances de l'échange, assurant ainsi une synchronisation complète entre les nœuds.

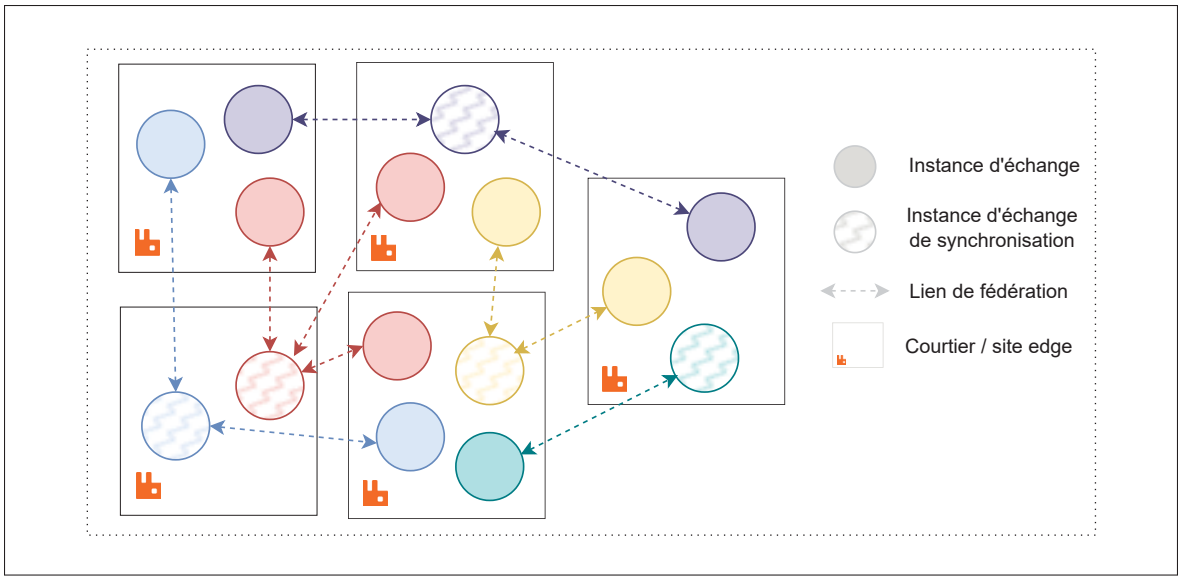



Figure 2.2 Topologie des nœuds edge

### 2.2.2 Modèle point à point avec file d'attente

Comme indiqué précédemment, le protocole AMQP 0-9-1 prend en charge à la fois le modèle pub/sub et le modèle point à point avec file d'attente. Dans cette section, nous nous intéressons à l'interconnexion de files d'attente réparties sur plusieurs sites edge, mais représentant une même file logique, c'est-à-dire servant une même application. Autrement dit, lorsque des files d'attente identiques, associées à une même logique applicative, sont déployées sur différents courtiers de messages distribués géographiquement (sous forme d'instances ou de clusters), l'enjeu consiste à les relier pour former une file logique répartie sur plusieurs sites edge.

Pour ce faire, nous proposons un flux de messages transformant le modèle point à point en un modèle analogue au pub/sub fondé sur l'entité échange, en nous appuyant sur le mécanisme de hachage cohérent décrit en section 1.10.5. En effet, le modèle point à point avec file d'attente ignore l'existence de l'échange et repose uniquement sur la file. Un exemple représentatif est la fédération de files d'attente implémentée par RabbitMQ (section 1.10.2) : elle permet de créer des consommateurs concurrents répartis sur deux sites différents en reliant directement les

files entre elles, sans passer par un échange. À l'inverse, la logique du modèle pub/sub s'appuie explicitement sur les échanges pour assurer le routage des messages.

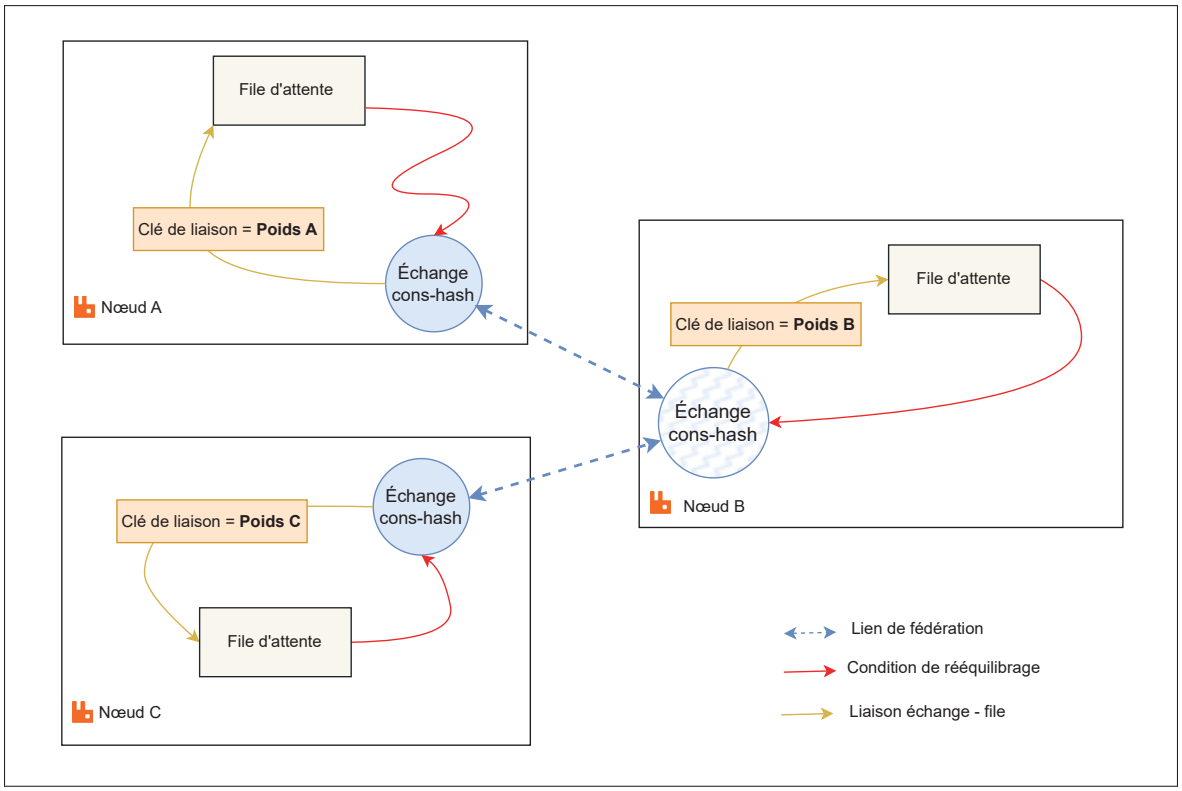



Figure 2.3 Conversion du modèle point à point avec file d'attente en modèle *Pub/Sub*

Comme illustré à la figure 2.3, chaque instance d'une file d'attente géodistribuée est associée à un échange de type hachage cohérent. Cet échange est relié à la file par une clé de liaison représentant son poids, c'est-à-dire sa capacité de traitement (le calcul du poids sera détaillé ultérieurement). Cet échange est ensuite traité comme un échange de type *pub/sub*, avec une instance de synchronisation chargée d'agréger, via des liens de fédération, les clés de liaison (poids), puis de les redistribuer à l'ensemble des instances concernées.

À la différence du comportement pub/sub classique des liens de fédération basés sur les échanges de type direct, fanout, topic ou headers qui dupliquent les messages vers l'ensemble des nœuds en aval disposant de clients intéressés, l'utilisation d'un échange à hachage cohérent permet d'acheminer chaque message vers une unique file cible. Cette file est choisie en fonction des

poids déclarés, en privilégiant celle dont la capacité est la plus élevée. En effet, si un message représentant une tâche dans le modèle point à point avec file d'attente était routé vers plusieurs files, il serait exécuté plusieurs fois par des clients distincts. Un tel comportement contredit notre objectif, qui vise à répartir intelligemment la charge en délégant chaque tâche à un unique client distant, le plus apte à la traiter efficacement lorsque nécessaire.

## 2.3 Flux des messages

Dans cette section, nous décrivons le comportement dynamique du système en détaillant le flux des messages dans les deux modèles de messagerie pris en charge : publication/abonnement (pub/sub) et point à point avec file d'attente.

### 2.3.1 Modèle publication/abonnement

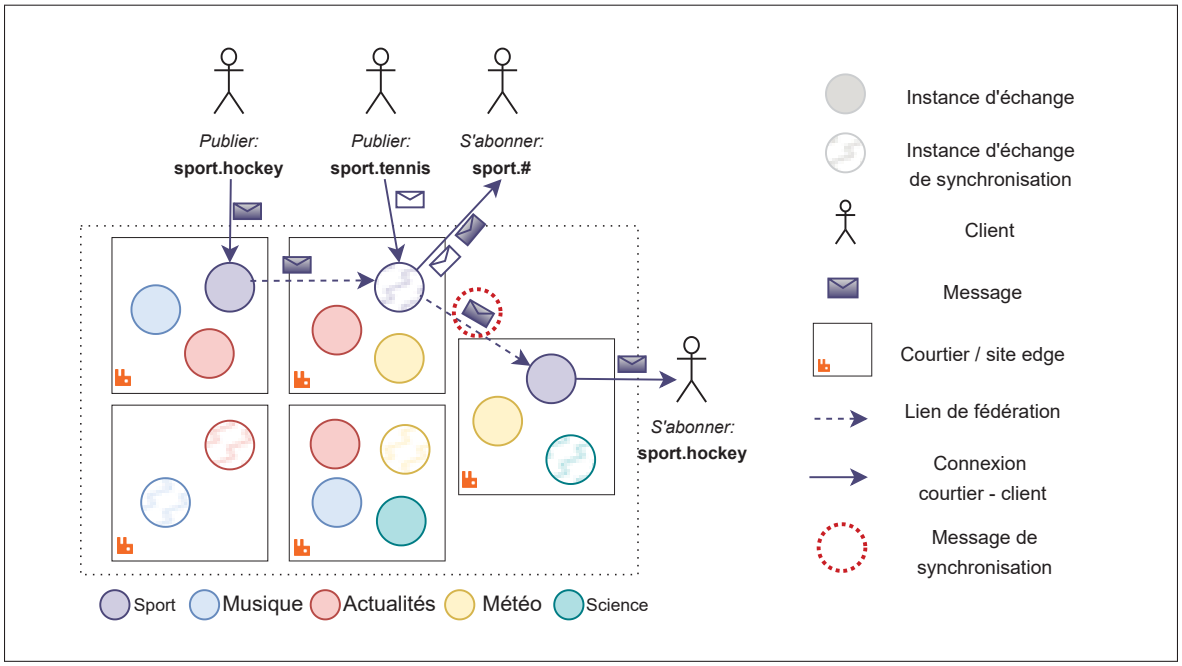



Figure 2.4 Flux de messages dans le modèle Pub/Sub

La figure 2.4 illustre un exemple comportant cinq échanges, chacun correspondant à un sujet d'application distinct. L'échange *sport*, de type *topic*, reçoit des messages de deux producteurs

publiant sur les sujets `sport.hockey` et `sport.tennis`. Deux abonnés sont présents : l'un s'abonne à `sport.#`, l'autre à `sport.hockey`.

Les messages sont tout d'abord livrés aux clients locaux, puis transférés au nœud de synchronisation (indiqué en zigzag), qui les redistribue à ses propres abonnés ainsi qu'aux autres nœuds contenant des clients intéressés.

Par exemple, un message publié avec la clé de routage `sport.hockey` est :

- livré à l'abonné de `sport.#` sur le nœud de synchronisation via un seul transfert intersites (du nœud source vers le nœud de synchronisation),
- puis redirigé vers l'abonné de `sport.hockey` via un message de synchronisation, impliquant ainsi deux transferts intersites (du nœud source vers le nœud de synchronisation et du nœud de synchronisation vers le nœud destination), ce qui constitue le pire scénario en termes de latence.

### 2.3.2 Modèle point à point avec file d'attente

Lorsqu'un message dépasse sa durée de vie (TTL) ou que la file locale atteint sa capacité maximale, le message est redirigé vers un DLX (présenté dans la section 1.9), qui est l'échange de type hachage cohérent associé à cette file. Les paramètres TTL et la taille maximale de la file sont adaptés dynamiquement selon le nombre de consommateurs actifs et leur capacité de traitement. Les clés de liaison entre chaque file et son DLX, représentant les poids des files, sont échangées dynamiquement entre les échanges DLX via les liens de fédération à travers l'échange de synchronisation (liens en vert sur la figure 2.5). Comme illustré dans la figure 2.5, le processus se déroule en trois étapes :

- Étape 1 : un message dans la file d'attente du nœud A est évacué, soit parce qu'il a dépassé sa durée de vie, soit parce que la file a atteint sa capacité maximale.
- Étape 2 : le message est publié dans l'échange de hachage cohérent lié à cette file.

- Étape 3 : étant donné que le nœud C dispose de deux consommateurs (contre un seul pour les autres), sa file a un poids plus élevé, indiquant une plus grande capacité de traitement. L'échange achemine le message vers la file du nœud C, où il sera traité.

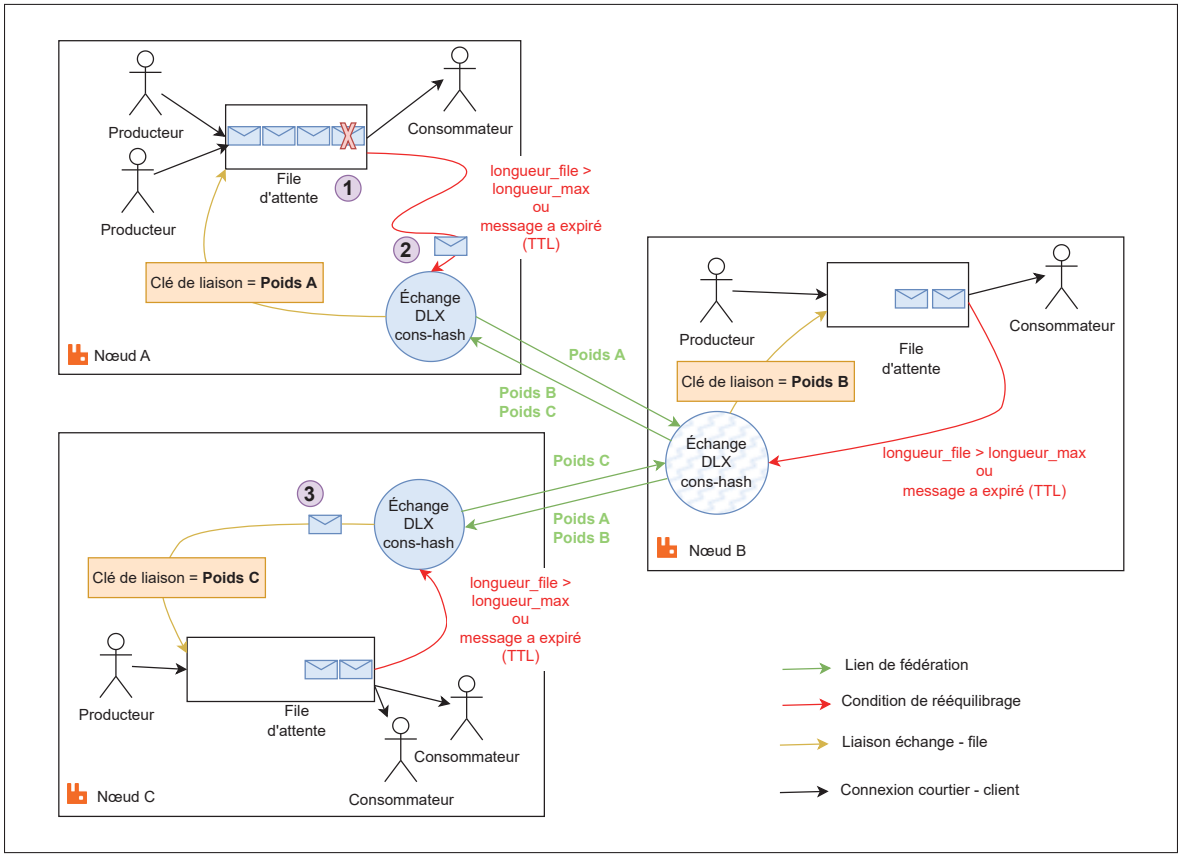



Figure 2.5 Flux de messages dans le modèle point à point

## 2.4 Formulation du problème de placement des échanges de synchronisation

Afin de déployer l'architecture et les flux de données proposés, et d'atteindre notre objectif qui consiste à réduire la latence de transmission des messages entre sites distribués tout en respectant les contraintes matérielles, nous formulons un problème de placement des échanges de synchronisation.

L'enjeu est d'optimiser la latence de transmission tout en tenant compte des capacités limitées des nœuds. Comme expliqué précédemment, le positionnement d'un échange de synchronisation

influence directement le nombre de transitions intersites, et par conséquent la latence globale du système. Concrètement, pour un échange donné, les producteurs et les consommateurs sont répartis sur plusieurs sites, chacun hébergeant une réplique de cet échange. Le défi consiste à désigner, parmi ces sites, celui qui jouera le rôle de site de synchronisation. Le problème revient alors à déterminer l'emplacement qui minimise le nombre de transitions intersites nécessaires pour acheminer les messages des producteurs vers les consommateurs.

En l'absence de contraintes, la résolution serait triviale : il suffirait de sélectionner, pour chaque échange, le site offrant le placement le plus avantageux. Toutefois, une telle approche peut violer les contraintes matérielles propres aux environnements edge. Pour cette raison, nous proposons un algorithme heuristique (présenté dans la section suivante) visant à trouver une configuration globale satisfaisante. Cette approche ne garantit pas que chaque échange soit toujours placé sur le site localement le plus optimal, mais elle permet d'obtenir une solution équilibrée qui respecte les contraintes autant que possible.

La définition formelle du problème est la suivante : étant donné un ensemble d'échanges $\mathbb{E}$ et un ensemble de nœuds $\mathbb{N}$, chaque échange $E_i$ est présent sur un sous-ensemble de nœuds $\mathbb{N}_i$. Pour chaque échange $E_i$, l'objectif est d'identifier le nœud optimal au sein de $\mathbb{N}_i$ à désigner comme nœud de synchronisation de $E_i$. L'objectif global est de minimiser la latence moyenne du système, tout en respectant les contraintes de capacité matérielle des nœuds. La latence est définie comme le délai de propagation entre le nœud où un message est publié et le nœud auquel un consommateur est connecté.

Étant donné que l'environnement est dynamique, toutes les variables présentées dans le tableau 2.1 dépendent du temps $t$. Pour simplifier la notation, cette dépendance est omise dans les formules.

Ce problème est formulé comme un programme linéaire en nombres entiers (ILP). La fonction objectif 2.1 vise à déterminer la valeur de la variable binaire $x_{ij}$, telle que définie dans le tableau 2.1, afin de minimiser le coût tout en respectant les contraintes (2.4) à (2.5).

Tableau 2.1 Variables du ILP

| Symbole | Description |
|---|---|
| $\mathbb{E} = \{E_1, E_2, \ldots, E_J\}$ | Ensemble des échanges. |
| $\mathbb{N} = \{n_1, n_2, \ldots, n_m\}$ | Ensemble de tous les nœuds. |
| $\mathbb{N}_i$ | Sous-ensemble de $\mathbb{N}$ contenant l'échange $E_i$. |
| $\mathbb{P}_i = \{p_{i1}, p_{i2}, \ldots, p_{iQ}\}$ | Ensemble des publications associées à l'échange $E_i$. |
| $\mathcal{R}(n_i)$ | Capacités matérielles du nœud $n_i$ (bande passante, mémoire, CPU, etc.). |
| $\mathbb{C}_i$ | Ensemble des consommateurs de l'échange $E_i$. |
| $d(n_i, n_j)$ | Délai de propagation entre le nœud $n_i$ et le nœud $n_j$. |
| $\theta^i_{k,l}$ | Variable binaire valant 1 si le consommateur $c_{il}$ est intéressé par la publication $p_{ik}$. |
| $n_{\mathrm{s}}(p_{ik})$ | Nœud source de la publication $p_{ik}$. |
| $n_{\mathrm{c}}(c_{il})$ | Nœud auquel est connecté le consommateur $c_{il}$. |
| $x_{i,j}$ | Variable binaire valant 1 si la synchronisation de l'échange $E_i$ est hébergée sur le nœud $n_j$. |
| $\lambda$ | Coût de reconfiguration. |
| $l_{ij}$ | Somme des latences de tous les messages envoyés à l'échange $E_i$ si le nœud $n_j$ est le nœud de synchronisation. |
| $h_i$ | Variable binaire valant 1 si le nœud de synchronisation de l'échange $e_i$ est inchangé par rapport à $t-1$. |
| $D_i(n_j)$ | Impact matériel prévu de l'échange $E_i$ sur le nœud $n_j$. |

Le terme $l_{ij}$ (2.2) dans la fonction objectif représente la somme des latences pour tous les messages envoyés à l'échange $E_i$ en passant par $n_j$ en tant que nœud de synchronisation. Comme un consommateur d'un échange peut ne s'intéresser qu'à un sous-ensemble de publications, une variable binaire $\theta^i_{kl}$ est introduite pour indiquer si le consommateur $c_{il}$ est intéressé par la publication $p_{ik}$. Le calcul de la latence tient compte du délai de propagation entre le nœud émetteur et le nœud $n_j$, puis de $n_j$ jusqu'au nœud du consommateur.

Afin d'éviter des reconfigurations fréquentes sans réel gain, une pénalité de reconfiguration $\lambda$ est soustraite et appliquée uniquement lorsque le nœud de synchronisation change entre $t-1$ et $t$ (2.3).

La contrainte (2.4) garantit que chaque échange dispose d'un seul nœud de synchronisation. La contrainte (2.5) impose le respect des capacités matérielles disponibles sur chaque nœud.

$$\min \sum_{i=1}^{|\mathbb{E}|} \left( \frac{1}{|\mathbb{P}_i| \cdot |\mathbb{C}_i|} \sum_{n_j \in \mathbb{N}_i} \left( l_{ij} + \lambda \cdot h_i \right) \cdot x_{ij} \right) \tag{2.1}$$

$$l_{ij} = \sum_{p_{ik} \in \mathbb{P}_i} \sum_{c_{il} \in \mathbb{C}_i} \theta^i_{kl} \cdot \left( d(n_s(p_{ik}), n_j) + d(n_j, n_c(c_{il})) \right) \tag{2.2}$$

$$h_i = |x_{ij}(t) - x_{ij}(t-1)| \tag{2.3}$$

$$\sum_{n_j \in \mathbb{N}_i} x_{ij} = 1 \quad \forall \mathbb{N}_i \subseteq \mathbb{N} \tag{2.4}$$

$$\sum_{i=1}^{\mathcal{N}_e} D_i(n_j) \cdot x_{ij} < \mathcal{R}(n_j) \quad \forall n_j \in \mathbb{N} \tag{2.5}$$

## 2.5 Méthode heuristique

Étant donné que ce problème est de type NP-difficile (He, Khamfroush, Wang, La Porta & Stein, 2018), le résoudre avec une approche ILP entraîne une complexité exponentielle. Pour y remédier, nous proposons une méthode heuristique reposant sur un algorithme glouton d'optimisation combinatoire, destiné au placement des échanges dans un graphe sous contraintes de capacité, en s'appuyant sur une métrique de centralité pondérée.

Par définition, un algorithme glouton ne garantit pas l'obtention de la solution optimale, mais fournit une bonne solution satisfaisante, adaptée à la résolution pratique du problème de placement. Dans le cadre de ce travail, l'objectif n'est pas d'identifier l'algorithme le plus performant, mais de proposer une solution fonctionnelle intégrée à la globalité de notre système.

Comme indiqué précédemment, cette approche est spécifiquement conçue pour les environnements edge, où les ressources matérielles sont généralement limitées. Dans ce cadre, nous introduisons des contraintes visant à limiter l'impact de notre solution sur les ressources du système. En

pratique, il est souvent possible d'allouer à moindre coût des ressources suffisantes (CPU, mémoire, stockage) pour exécuter un courtier de messages en périphérie. En revanche, la bande passante demeure une contrainte critique, difficilement maîtrisable (RabbitMQ, 2020a). Les sites edge sont en effet souvent reliés par des liaisons à capacité restreinte ou coûteuses (par exemple : satellite, LTE et 5G), où les coûts élevés et la variabilité du débit constituent un facteur limitant majeur. De plus, l'évaluation précise de la charge induite par la couche overlay sur l'usage du CPU et de la mémoire reste complexe.

Par conséquent, notre étude se concentre exclusivement sur les contraintes liées à l'utilisation de la bande passante, aussi bien en réception qu'en émission. Cette focalisation se justifie par le fait que les sites edge sont fréquemment connectés via des technologies à capacité limitée ou coûteuses, telles que les réseaux cellulaires ou satellites. L'analyse de l'impact sur les autres ressources est laissée comme perspective pour des travaux futurs.

Il convient également de souligner que ces contraintes sont souples : notre algorithme heuristique génère une solution même lorsqu'elles ne sont pas entièrement respectées. Dans ce cas, la solution cherchera à équilibrer la charge de manière avantageuse et à minimiser autant que possible les violations de contraintes.

### 2.5.1 Calcul de centralité

L'algorithme débute par un tri des échanges en fonction du volume total de messages échangés, c'est-à-dire la somme des messages reçus depuis les producteurs et envoyés vers les consommateurs sur l'ensemble des nœuds, conformément aux équations (2.6) et (2.7). Cette étape vise à identifier les échanges les plus sollicités en priorité.

$$\sum_{n_j \in \mathbb{N}_i} \mathcal{P}_{ij}, \quad \forall \mathbb{N}_i \subseteq \mathbb{N} \tag{2.6}$$

$$\mathcal{P}_{ij} = \mathcal{P}_{ij}^{in} + \mathcal{P}_{ij}^{out} \tag{2.7}$$

Tableau 2.2 Résumé des variables de l'heuristique

| Symbole | Description |
|---|---|
| $\mathbb{G}_i = (\mathbb{N}_i, \mathbb{L}_i)$ | Graphe complet composé du sous-ensemble de nœuds $\mathbb{N}_i$ et des liens $\mathbb{L}_i$. |
| $\mathcal{P}_{ij}^{in}$ | Nombre de publications entrantes provenant des clients locaux vers l'échange $e_i$ au nœud $n_j$. |
| $\mathcal{P}_{ij}^{out}$ | Nombre de publications sortantes vers les clients locaux depuis l'échange $e_i$ au nœud $n_j$. |
| $\mathcal{T}_{ij}^{in}$ | Nombre de publications transférées entrantes depuis d'autres nœuds vers le nœud $n_j$ via l'échange $E_i$. |
| $\mathcal{T}_{ij}^{out}$ | Nombre de publications transférées sortantes depuis le nœud $n_j$ vers d'autres nœuds via l'échange $E_i$. |
| $C_i(n_j)$ | Centralité de proximité pondérée du nœud $n_j$ au sein du sous-ensemble $\mathbb{N}_i$. |
| $\mathcal{B}_o^{in}(n)$ | Bande passante overlay entrante au nœud $n$. |
| $\mathcal{B}_o^{out}(n)$ | Bande passante overlay sortante au nœud $n$. |
| $\mathcal{F}_{ij}^{in}$ | Débit overlay estimé en entrée vers l'échange $e_i$ au nœud $n_j$. |
| $\mathcal{F}_{ij}^{out}$ | Débit overlay estimé en sortie depuis l'échange $e_i$ au nœud $n_j$. |
| $\mathcal{D}(E_i)$ | La taille moyenne des messages envoyés à l'échange $E_i$ |
| $\mathcal{S}_t$ | Plan de synchronisation à l'instant $t$. |
| $n_{is}$ | Nœud de synchronisation de l'échange $E_i$ à l'instant $t-1$, $\mathcal{S}_{t-1}(E_i)$. |

Pour chaque échange $E_i$, dans l'ordre décroissant de charge, nous examinons le sous-graphe des nœuds $\mathbb{N}_i$. Dans ce sous-graphe, nous calculons la centralité de proximité *(closness centrality)* de chaque nœud. Elle mesure le degré de centralité d'un nœud dans le réseau, défini comme l'inverse de la somme des distances des plus courts chemins entre ce nœud et tous les autres :

$$C(u) = \frac{1}{\sum_{v \in V \setminus \{u\}} d(u,v)} \tag{2.8}$$

- $C(u)$ : centralité de proximité du nœud $u$.
- $V$ : ensemble des nœuds du graphe.
- $d(u,v)$ : distance du plus court chemin entre les nœuds $u$ et $v$.

Une centralité plus élevée indique que le nœud est proche de tous les autres. Dans notre cas, la distance $d(u, v)$ est mesurée par le délai de propagation entre les nœuds $u$ et $v$.

Afin de prendre en compte l'activité du nœud, nous multiplions la centralité par le nombre total de messages entrants et sortants en local sur ce nœud :

$$C_i(n_j) = \frac{\mathcal{P}_{ij}}{\sum_{n_k \in \mathbb{N}_i \setminus \{n_j\}} d(n_k, n_j)} \tag{2.9}$$

Après avoir calculé la centralité pondérée, un score de pénalité est attribué à chaque nœud, à l'exception de celui déjà utilisé comme nœud de synchronisation à l'instant $t - 1$.

Les nœuds sont ensuite triés selon leurs scores, les scores les plus élevés étant prioritaires. Nous parcourons cette liste triée, nœud par nœud, vérifiant s'il respecte les contraintes matérielles en lui ajoutant la charge de synchronisation. Le processus s'arrête dès qu'un nœud valide est trouvé. Ce nœud est alors désigné comme nouveau nœud de synchronisation pour l'échange $E_i$.

### 2.5.2 Vérification des contraintes

Au début de l'algorithme, il est essentiel d'initialiser les débits estimés (en entrée et en sortie) générés par chaque instance d'échange sur chaque nœud, via la couche *overlay*. Pour un échange $E_i$, tout nœud appartenant à l'ensemble $\mathbb{N}_i$ peut potentiellement être sélectionné comme nœud de synchronisation.

Sur chaque nœud, le volume des messages reçus ou émis via la couche *overlay* $\mathcal{T}_{ij}^{\text{in/out}}$ correspond à la somme :

- des messages publiés ou consommés par les clients locaux ;
- et des messages de synchronisation, c'est-à-dire les messages que le nœud doit échanger uniquement s'il est désigné comme nœud de synchronisation. Sur la figure 2.4, seul le message entouré en rouge est un message de synchronisation.

Cependant, comme il est difficile sur le plan pratique de distinguer précisément les messages liés à la synchronisation de ceux produits ou consommés localement, nous adoptons l'hypothèse suivante : pour le nœud de synchronisation actuel $n_{is}$, la charge *overlay* $\mathcal{T}_{is}^{\text{in/out}}$ est uniquement due à la synchronisation.

Ainsi, pour initialiser le débit de chaque nœud potentiel autre que $n_{is}$, nous multiplions le nombre de messages *overlay* par la taille moyenne des messages de l'échange $E_i$ :

$$\mathcal{F}_{ij}^{\text{in/out}} = \mathcal{T}_{ij}^{\text{in/out}} \times \mathcal{D}(E_i). \tag{2.10}$$

Quant au nœud de synchronisation $n_{is}$, nous l'initialisons à partir du débit estimé d'un autre nœud sélectionné aléatoirement parmi $\mathbb{N}_i \setminus \{n_{is}\}$.

Pour vérifier si le nœud $n_j$ satisfait aux contraintes matérielles pour être un nœud de synchronisation de l'échange $E_i$, nous remplaçons le débit overlay estimé $\mathcal{F}_{ij}^{\text{in/out}}$ par l'activité réelle observée sur le nœud de synchronisation de cet échange à l'instant t-1, $n_{is}$. Nous calculons ensuite le débit total cumulé pour tous les échanges présents sur $n_j$. Si ce total dépasse la bande passante disponible $\mathcal{B}_o^{\text{in/out}}$, le nœud est écarté ; sinon, il est retenu.

Le débit total estimé au nœud $n_j$ est donné par l'équation suivante :

$$\Big(\sum_{E_k \in \mathbb{E} \setminus E_i} \mathcal{F}_{kj}^{\text{in/out}}\Big) + \mathcal{T}_{is}^{\text{in/out}} \times \mathcal{D}(E_i). \tag{2.11}$$

Si le nœud est admissible, le débit estimé initial est remplacé par le débit de synchronisation. Dans le cas où aucun nœud n'est retenu, l'algorithme sélectionne celui présentant la charge la plus faible, c'est-à-dire le nœud ayant, jusqu'à présent, le débit le plus bas.

### 2.5.3 Algorithme heuristique

La stratégie complète adoptée par le contrôleur pour générer un nouveau plan de synchronisation est présentée dans l'Algorithme 2.1. Dans un premier temps (lignes 1 à 3), le système initialise les débits estimés pour chaque instance d'échange sur chaque nœud, en considérant que toutes les instances ne sont pas des instances de synchronisation. Ensuite (lignes 4 à 6), pour chaque ensemble d'échanges (échanges globaux), le volume total de messages est calculé, ce qui permet d'identifier les échanges les plus sollicités. Ces échanges sont ensuite triés par ordre décroissant de charge (ligne 7). À partir de la ligne 8 débute l'algorithme de placement. Chaque échange global est traité successivement, en commençant par les plus occupés. À la ligne 9, le contrôleur récupère le nœud de synchronisation précédemment associé à l'échange. L'étape suivante consiste à évaluer les nœuds candidats en calculant leur centralité de proximité pondérée (ligne 10). Afin de limiter les reconfigurations redondantes, une pénalité est appliquée aux nœuds différents du nœud de synchronisation précédent (lignes 12 à 16). Les nœuds sont ensuite triés par ordre décroissant de score (ligne 17), et le premier respectant les contraintes de capacité est sélectionné comme nouveau nœud de synchronisation (lignes 18 à 24). Si aucun nœud candidat ne respecte les contraintes, l'échange est attribué au nœud le moins chargé (lignes 25 à 28). Grâce à la souplesse des contraintes, l'algorithme garantit toujours une solution, en limitant l'impact des violations. Les débits estimés de l'échange sur le nœud choisi sont mis à jour (ligne 29). Enfin, le plan de synchronisation est retourné.

Algorithme 2.1 Algorithme d'hébergement des échanges de synchronisation

**Input :** Ensemble des échanges $\mathbb{E}$ et des nœuds $\mathbb{N}$;
Plan de synchronisation précédent $\mathcal{S}_{t-1}$;
$\forall E_i$, la taille moyenne des messages échangés par $E_i$ $\mathcal{D}(E_i)$;
$\forall E_i \in \mathbb{E}$ et $\forall n_j \in \mathbb{N}_i$, les messages locaux $\mathcal{P}_{ij}^{in}$, $\mathcal{P}_{ij}^{out}$ et les messages overlay $\mathcal{T}_{ij}^{in}$, $\mathcal{T}_{ij}^{out}$;
$\forall n_j \in \mathbb{N}$, les bandes passantes overlay disponibles $\mathcal{B}_o^{in}(n_j)$, $\mathcal{B}_o^{out}(n_j)$;
**Output :** Plan de synchronisation mis à jour $\mathcal{S}_t$

**foreach** *échange* $E_i \in \mathbb{E}$ **do**
    Initialiser les débits estimés $\mathcal{F}_{ij}^{in}$, $\mathcal{F}_{ij}^{out}$ pour tous les $n_j \in \mathbb{N}_i$;
**end foreach**
**foreach** *échange* $E_i \in \mathbb{E}$ **do**
    Calculer le volume total de messages : $\mathcal{P}(E_i) \leftarrow \sum_{n_j \in \mathbb{N}_i} \mathcal{P}_{ij}$;
**end foreach**
Trier les échanges par $\mathcal{P}(E_i)$ en ordre décroissant;
**foreach** *$E_i$ dans la liste triée* **do**
    Récupérer le nœud de synchronisation précédent : $n_{is} \leftarrow \mathcal{S}_{t-1}(E_i)$;
    Calculer les scores de centralité de proximité pondérée $C_i$;
    affecté $\leftarrow$ faux;
    **foreach** *nœud* $n_j \in C_i$ **do**
        **if** $n_j \neq n_s$ **then**
            Appliquer une pénalité : $C_i(n_j) \leftarrow C_i(n_j) - \lambda$;
        **end if**
    **end foreach**
    Trier les nœuds $n_j$ par $C_i(n_j)$ en ordre décroissant;
    **foreach** $n_j \in C_i$ **do**
        **if** $n_j$ *respecte les contraintes* **then**
            $\mathcal{S}_t(E_i) \leftarrow n_j$;
            affecté $\leftarrow$ vrai;
            **break**;
        **end if**
    **end foreach**
    **if** *affecté = faux* **then**
        Sélectionner $n_j$ comme le nœud le moins chargé (débit total minimal);
        $\mathcal{S}_t(E_i) \leftarrow n_j$;
    **end if**
    Mettre à jour les débits estimés : $\mathcal{F}_{ij}^{in} \leftarrow \mathcal{T}_{ij}^{in} \times \mathcal{D}(E_i)$, $\mathcal{F}_{ij}^{out} \leftarrow \mathcal{T}_{ij}^{out} \times \mathcal{D}(E_i)$;
**end foreach**
**return** $\mathcal{S}_t$;

## 2.6 Architecture logicielle du courtier *LuffyMQ*

Le courtier *LuffyMQ* repose sur une instance RabbitMQ enrichie par trois agents clients autonomes chargés de modifier dynamiquement sa configuration selon les besoins du système. L'interaction entre ces agents et le courtier est illustrée à la figure 2.6.

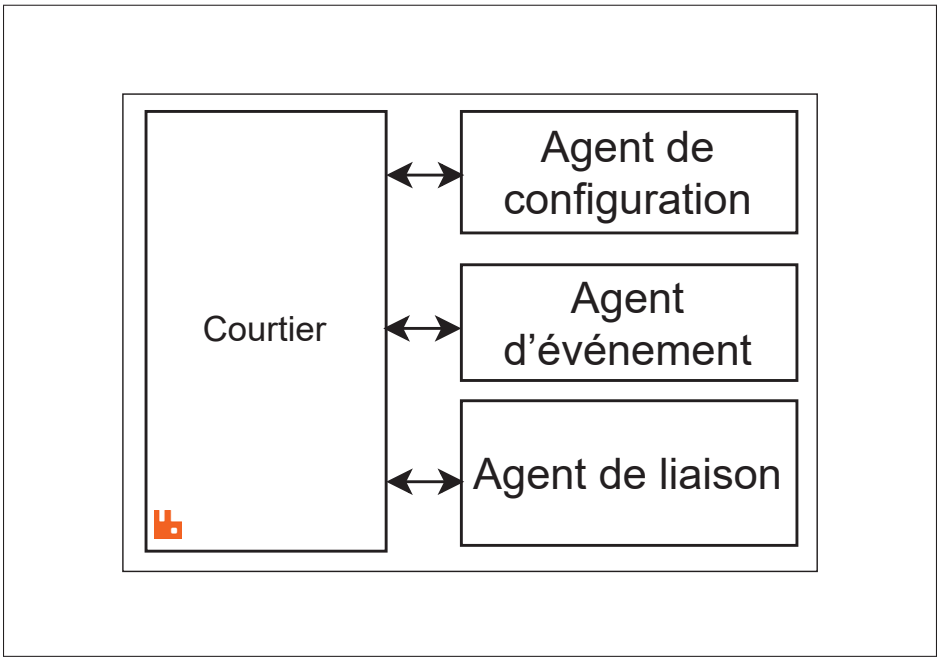



Figure 2.6 Architecture logicielle du courtier LuffyMQ

### 2.6.1 Agent de configuration : *Federation Setup Agent*

Cet agent est abonné à une file dédiée à la configuration. Il reçoit les instructions envoyées par un orchestrateur central. Un exemple de configuration reçu est illustré dans la figure 2.7.

La structure des messages de configuration suit le modèle illustré par le diagramme de classes (figure 2.8) : une politique est associée à un ensemble de fédérations, chacune contenant un ou plusieurs liens (agrégation).

Le fonctionnement global de cet agent est détaillé dans le diagramme d'activités (figure 2.9).

```
{
    "pubsub_news": [
        {"uri": "amqp://172.18.0.4:5672"},
        {"uri": "amqp://172.18.0.2:5672"}
    ],
    "pubsub.weather": [
        {"uri": "amqp://172.18.0.4:5672"}
    ]
}
```


Figure 2.7 Message de configuration reçu de l'orchestrateur

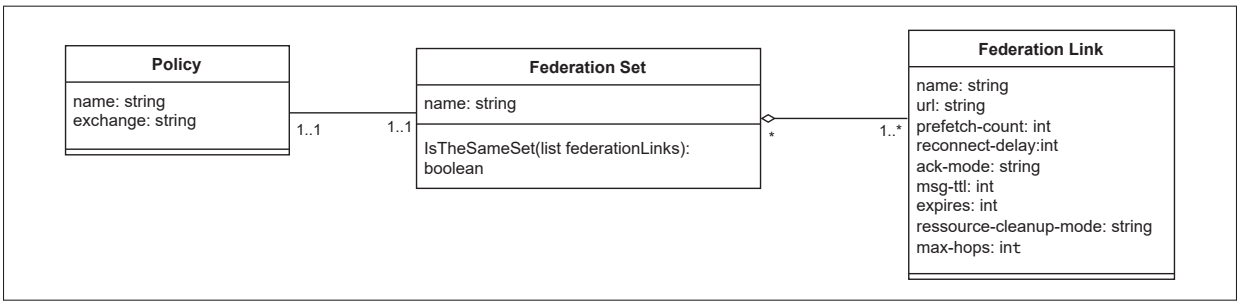



Figure 2.8 Diagramme de classes de la configuration de fédération

### 2.6.2 Agent d'événement

Cet agent est abonné à un échange d'événements, tel que défini dans la section 1.10.4, et écoute les messages dont le sujet `queue.#`, incluant donc les événements `queue.created` et `queue.deleted`.

Son rôle est de transformer dynamiquement une file point à point en un modèle analogue pub/sub comme expliqué dans la section 2.3.2. Lorsqu'une file d'attente préfixée par `wq` est créée (utilisée pour le mode point à point), l'agent crée un échange `DLX` (Dead Letter Exchange) de type hachage cohérent, connecte la file à cet échange avec un poids aléatoire, et applique une politique pour définir le TTL et la taille maximale.

Lorsque cette file est supprimée, l'agent nettoie également l'échange et la politique associés.

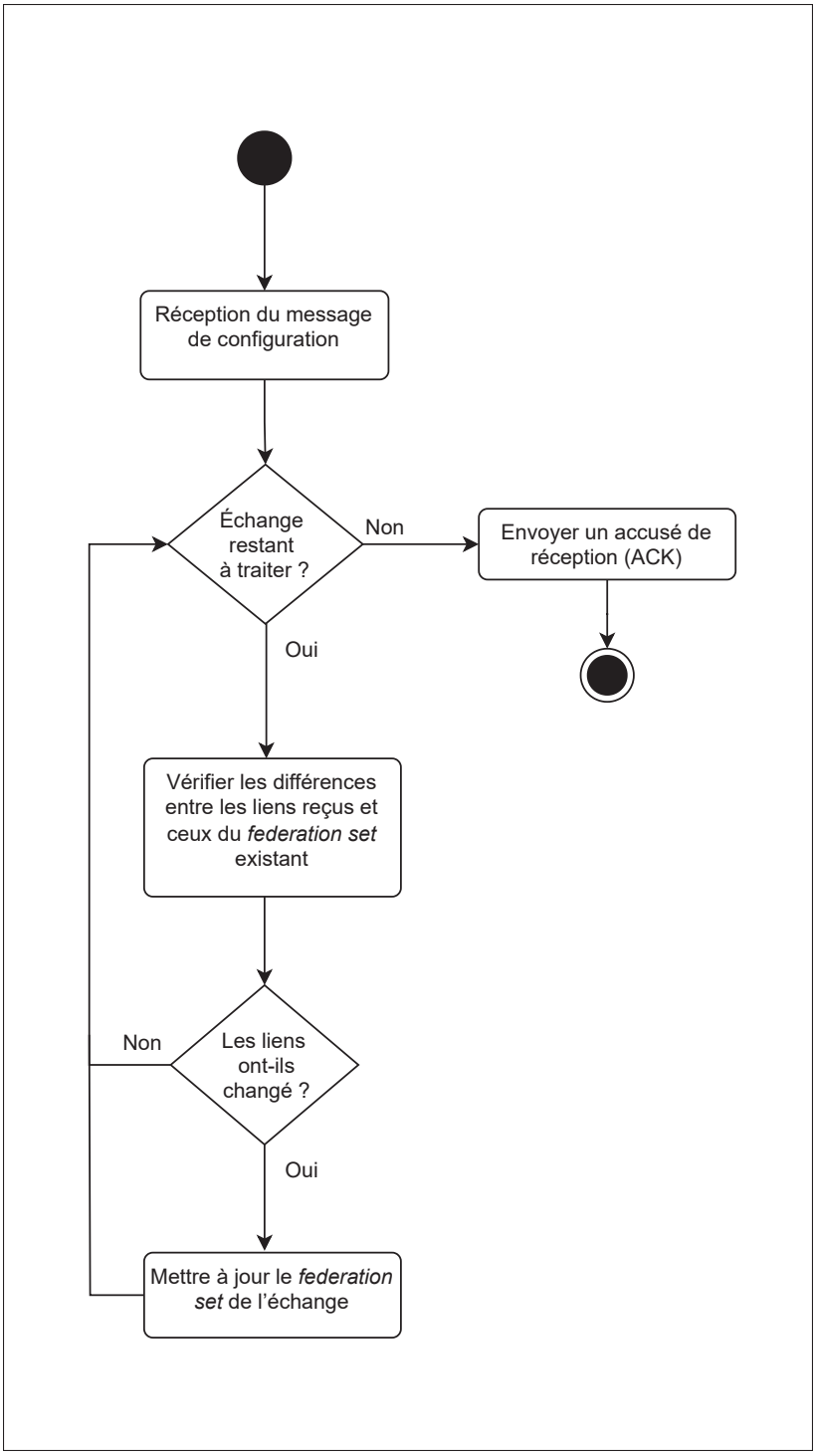



Figure 2.9 Fonctionnement de l'agent de configuration

### 2.6.3 Agent de liaison : *Binding Agent*

Cet agent a pour rôle d'ajuster dynamiquement, à intervalles réguliers, les paramètres des files d'attente utilisées dans le modèle point à point avec file, à savoir : le *Time-To-Live* (TTL), la taille maximale de la file, et le poids associé à la liaison avec l'échange DLX. Ces ajustements sont calculés à l'aide de la formule 2.12, en fonction de la capacité de traitement des consommateurs.

La variable `UtilisationConsommateur`, fournie par RabbitMQ, reflète la disponibilité des consommateurs à traiter de nouveaux messages. Elle prend une valeur entre 0% (saturation) et 100% (capacité pleinement disponible). La constante utilisée varie selon le paramètre concerné (TTL ou taille maximale) et selon le type de flux. Par exemple, dans le cas d'un flux élevé, la constante liée à la taille de la file peut être fixée à 3000 messages, tandis que celle liée au TTL peut être fixée à 500 ms.

$$\max\left(\frac{\texttt{Constante} \times \texttt{UtilisationConsommateur} \times \texttt{NombreConsommateurs}}{\texttt{100}},\ 1\right) \quad (2.12)$$

Le poids de la file est utilisé comme clé de liaison vers l'échange de hachage cohérent associé à la file. Il quantifie la capacité effective de traitement de la file et intervient dans l'équilibrage de charge. Ce poids est directement dérivé de la longueur maximale, et donc dépendant du nombre et de la disponibilité des consommateurs. Il est calculé selon les équations suivantes :

$$\texttt{Poids} = \max\left(\texttt{LongueurMaximale} - \texttt{MessagesEnFile},\ 1\right) \quad (2.13)$$

$$\texttt{MessagesEnFile} = \texttt{MessagesNonEnvoyés} + \texttt{MessagesEnvoyésSansAccusé} \quad (2.14)$$

## 2.7 Architecture logicielle de l'orchestrateur *LuffyMQ*

L'orchestrateur *LuffyMQ* repose sur une architecture modulaire structurée en plusieurs étapes successives : collecte des données, structuration, puis analyse et prise de décision.

### 2.7.1 Collecte des données

L'orchestrateur interagit avec RabbitMQ afin de récupérer les métriques nécessaires au pilotage du système. Les données extraites concernent principalement :

- les échanges dont le nom commence par `pubsub`, représentant les échanges déclarés comme pub/sub dans le contexte du cluster géodistribué
- les échanges par défaut du système RabbitMQ, identifiables par le préfixe `amq`.

Afin d'évaluer le trafic circulant à travers le réseau de fédération, l'orchestrateur interroge également les files d'attente associées aux liens de fédération, identifiables par le préfixe `federation`.

#### 2.7.1.1 Structuration des données

Les données collectées sont nettoyées et transformées afin d'être organisées dans une structure cohérente, illustrée par le diagramme 2.10. Bien que la majorité des données soient directement fournies par RabbitMQ, certaines métriques doivent être calculées manuellement, notamment :

- le nombre de messages entrants et sortants sur le réseau dédié (`overlayMsgIn` et `overlayMsgOut`)
- la taille moyenne des messages par échange (`avgSizeMsg`)

Pour un nœud donné, le nombre de messages envoyés par une instance d'échange vers ses répliques distantes correspond à la somme des messages publiés dans les files de fédération locales associées à cette instance. De manière symétrique, le nombre de messages reçus par une instance d'échange correspond à la somme des messages publiés dans les files de fédération rattachées aux répliques distantes dans la direction de cette instance.

La taille moyenne des messages peut être estimée en divisant le volume total d'octets envoyés (ou reçus) sur l'ensemble des connexions actives par le nombre total de messages transmis. Étant donné qu'un échange ne modifie pas le contenu des messages, la taille moyenne des messages reçus devrait être équivalente à celle des messages envoyés :

$$\texttt{avgSizeMsg} = \frac{\sum \texttt{octetsEnvoyés (ou reçus)}}{\texttt{nombre de messages envoyés}} \tag{2.15}$$

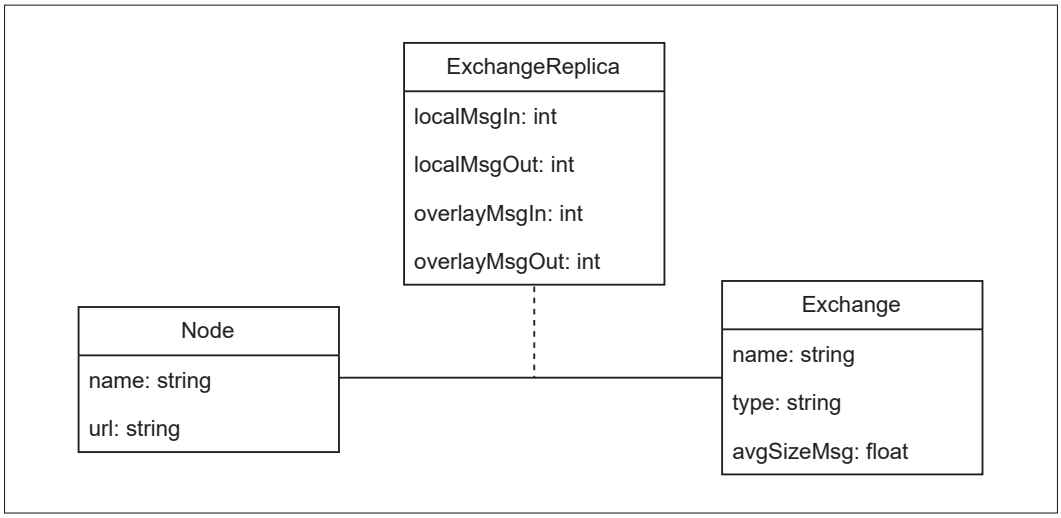



Figure 2.10 Structuration des données collectées par l'orchestrateur

## 2.8 Haute disponibilité des messages sur le réseau *overlay*

Un changement de configuration peut entraîner une perte de messages. Bien que les interruptions de réseau ou de connectivité soient gérées par les mécanismes de fédération intégrés à RabbitMQ, un changement de configuration présente une particularité : le site A continue logiquement à transmettre ses messages vers le site de synchronisation, mais en réalité, la cible a changé par exemple, le site de synchronisation passe de B à C. Dans ce cas, les mécanismes de haute disponibilité classiques ne sont plus applicables.

La solution proposée est illustrée à la figure 2.11. Dans RabbitMQ, la file d'attente associée à un lien de fédération est une file interne, gérée automatiquement et supprimée dès que le lien correspondant est interrompu. Nous proposons de modifier ce comportement en introduisant un délai d'expiration (TTL) appliqué uniquement après la perte du lien (étape 1). Cette mesure permet de conserver temporairement les messages non encore transmis, ainsi que ceux envoyés mais dont l'accusé de réception n'a pas été reçu. Dans un second temps, ces messages sont réinjectés dans l'échange d'origine afin d'être publiés à nouveau (étape 2). Ils peuvent ainsi être redirigés vers la nouvelle file d'attente créée suite à l'établissement d'un nouveau lien de fédération. Enfin, la file temporaire est automatiquement supprimée à l'expiration du TTL (étape 3). Cette approche permet de préserver les messages en cas de perte de lien, mais elle introduit un retard significatif sur leur livraison. Ce délai correspond à la durée du TTL configuré, à laquelle s'ajoute le temps de transmission. La stratégie demeure optionnelle et peut être désactivée par l'administrateur si la perte de messages n'est pas jugée critique.

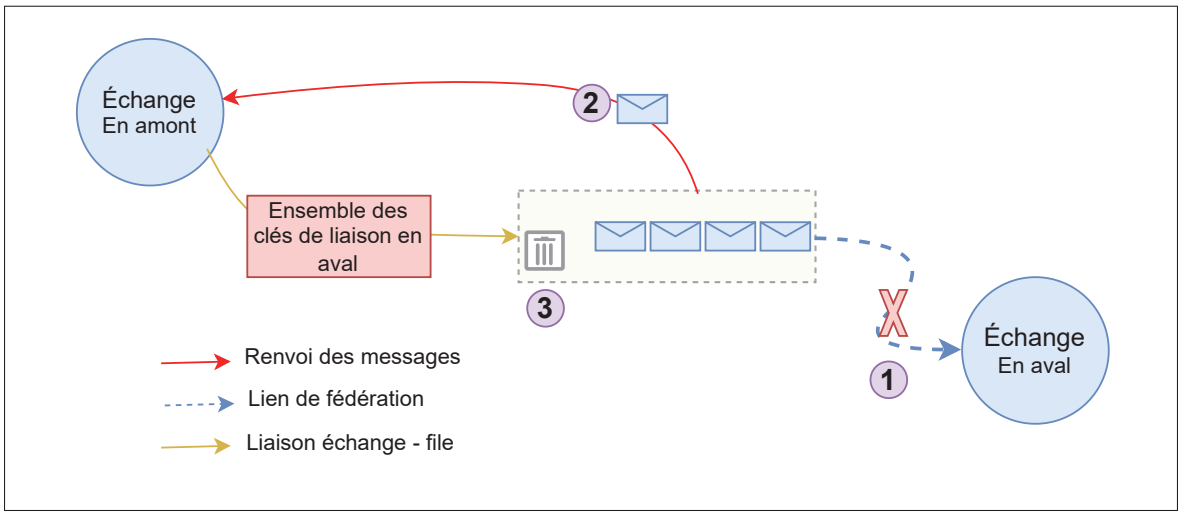



Figure 2.11 Stratégie de conservation des messages

## 2.9 Intéropirabilité de LuffyMQ avec les clients RabbitMQ

Le système LuffyMQ assure une transparence totale pour les clients RabbitMQ existants, ce qui garantit une intégration fluide. Les clients de LuffyMQ sont, par conception, des clients

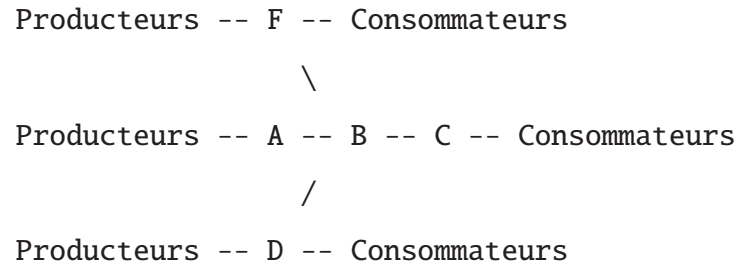
RabbitMQ standard. Les seules contraintes imposées sont des restrictions minimes au niveau des politiques de courtier (policies), lesquelles sont nécessaires au maintien des liens de fédération et au bon fonctionnement interne de LuffyMQ. Ces restrictions sont comparables à celles généralement imposées par les fournisseurs infonuagiques tels que Amazon MQ (2025).

Étant donné que RabbitMQ est un courtier multiprotocole dont l'architecture interne est basée sur le protocole AMQP 0-9-1 vers lequel tous les protocoles sont traduits en interne, le système LuffyMQ supporte nativement l'ensemble des protocoles pris en charge par RabbitMQ.

Par conception, LuffyMQ ne synchronise pas l'intégralité de ses échanges et de ses files d'attente à travers le réseau overlay. Le système distingue clairement deux catégories d'entités : les échanges et files d'attente locaux et les échanges et files d'attente globaux. Conformément aux descriptions fournies dans les sections précédentes, seules les entités désignées comme globales nécessitent une synchronisation intersites : il s'agit spécifiquement des échanges dont le nom commence par le préfixe `pubsub` et des files d'attente dont le nom commence par le préfixe `wq`.

La puissance architecturale de RabbitMQ réside dans sa capacité à établir une logique de routage avancée et complexe. Ceci est réalisé en chaînant et en reliant plusieurs échanges pour former un véritable réseau d'échanges connectés.

Par exemple, un réseau peut être structuré comme suit :




LuffyMQ supporte intégralement ce type de réseau d'échanges. Pour que cette topologie fonctionne au niveau distribué, il suffit de :

- Créer la même structure de réseau sur chaque courtier edge.

- En ne synchronisant que les points d'entrée (les échanges globaux), LuffyMQ assure la propagation des messages dans le réseau complexe vers les consommateurs distribués.

En ne synchronisant que les points d'entrée (les échanges globaux), LuffyMQ assure la propagation des messages vers les consommateurs distribués.

# CHAPITRE 3

# IMPLÉMENTATION ET ÉVALUATION

Ce chapitre présente l'implémentation du système proposé ainsi que son évaluation expérimentale. Dans une première partie, nous détaillons les technologies utilisées, les différents composants logiciels, ainsi que l'architecture du système. La seconde partie est consacrée à la description de l'environnement de test et aux expériences réalisées afin d'évaluer les capacités de notre solution dans des scénarios distribués.

## 3.1 Implémentation

### 3.1.1 Outils et technologies utilisés

Le développement du système s'est appuyé sur un ensemble d'outils et de langages. Les principaux outils utilisés sont :

- `Python` : langage principal pour le développement du système *LuffyMQ* et des clients de test. Il a été choisi pour sa simplicité, sa rapidité de développement et la richesse de son écosystème, notamment en matière de bibliothèques de communication synchrone et asynchrone.
- `Bash` : utilisé pour automatiser les scénarios expérimentaux et orchestrer les déploiements locaux.
- `RabbitMQ` : courtier de messages utilisé comme noyau du système *LuffyMQ*.
- `Docker` : l'ensemble des composants (nœuds, agents, orchestrateur) est conteneurisé pour garantir portabilité et isolation.
- `Dockerfile` et `docker-compose` : utilisés pour construire et orchestrer les différentes instances du système dans un environnement reproductible.

**Modules Python utilisés**

- `pika` : client AMQP 0-9-1 synchrone.

- `aio_pika` : client AMQP 0-9-1 asynchrone, utilisé notamment par l'orchestrateur pour envoyer des messages simultanément à plusieurs nœuds, et par le module de test pour générer plusieurs clients en parallèle.
- `aiohttp` : utilisé par l'orchestrateur pour collecter de manière asynchrone des métriques depuis les différents nœuds.
- `asyncio` : bibliothèque cœur du modèle asynchrone de Python, utilisée dans *LuffyMQ* pour faire fonctionner plusieurs agents en parallèle sans blocage.

**Configuration réseau dédiée**

Afin de simuler un réseau edge géodistribué, l'objectif est d'introduire des latences différenciées entre les nœuds, représentant des emplacements géographiques réels (ex. : Montréal, Berlin, Tokyo). Chaque lien entre deux sites est caractérisé par une latence propre, reflétant les conditions d'un réseau réel.

Pour cela, l'outil système `tc` (Traffic Control) est l'outil de référence. Plusieurs projets open source proposent des interfaces simplifiées pour `tc`, notamment :

- `docker-tc` : adapté à l'environnement Docker, mais applique les règles (latence, bande passante, etc.) à l'ensemble du trafic d'un conteneur, ce qui n'est pas adapté à notre besoin de latences différenciées par lien.
- `tcconfig` : un autre outil basé sur `tc`, plus souple, permettant de filtrer selon la direction (entrant/sortant) et selon les adresses sources ou destinations. Toutefois, il présente une limitation majeure : il ne permet pas de configurer plusieurs filtres complexes simultanément.

Nous avons également expérimenté l'application directe de `tc` sur les interfaces réseau `bridge` associé aux réseaux Docker. Cependant, la gestion simultanée de multiples règles complexes a rapidement montré ses limites, tant en termes de stabilité que de complexité de configuration. Cette piste n'a donc pas été approfondie davantage.

**Solution adoptée : utilisation de paires `veth`**

Par défaut, Docker bridge network connecte chaque conteneur à une interface `veth`, dont la contrepartie est attachée à un pont virtuel (bridge) sur l'hôte. Ce pont, qui joue le rôle d'un commutateur logiciel, permet d'acheminer le trafic entre les conteneurs ou vers l'extérieur (figure 3.1).

Dans notre approche, nous contournons cette topologie standard en créant manuellement des paires `veth` point à point entre les conteneurs eux-mêmes (veth 1-2 et veth 2-1 sur la figure 3.1). Cela nous permet de :

- simuler des liens directs entre nœuds géographiques (par exemple, Montréal–Berlin)
- appliquer les règles de latence via `tc` directement sur les interfaces concernées, sans avoir à configurer de filtres complexes
- diriger le trafic en fonction de la destination à l'aide de règles de routage, en associant chaque flux sortant à l'interface `veth` appropriée.

### 3.1.2 Clients de test

Bien que RabbitMQ fournisse un outil de benchmark performant, (PerfTest, 2025), celui-ci présente certaines limitations dans le cadre de notre étude. Notamment, il ne permet pas la création ni l'évaluation d'échanges personnalisés autres que l'échange direct par défaut.

Pour pallier ces restrictions, nous avons conçu un client de test sur mesure, entièrement configurable en fonction du scénario d'évaluation visé. Les paramètres requis sont spécifiés dans la variable d'environnement Docker et incluent notamment :

- `brokerUrl` : adresse du courtier RabbitMQ.
- `exchangeName`, `exchangeType` : nom et type de l'échange à tester.
- `queueName` : nom de la file utilisée (dans le cas point à point).
- `numProducers`, `numConsumers` : nombre de producteurs et de consommateurs à lancer.
- `messagesParSeconde` : fréquence d'envoi des messages.

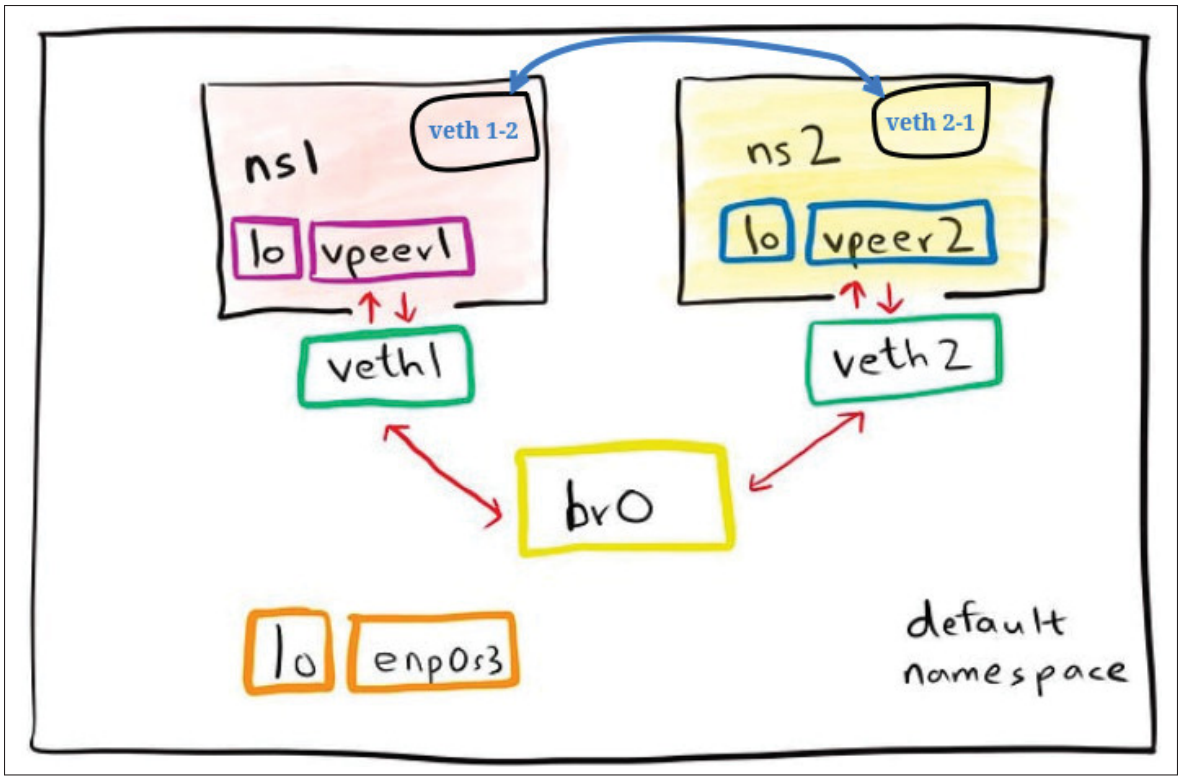



Figure 3.1 Configuration des interfaces `veth`
(adaptée de Khan (2022))

- `consumerProcessingDelay` : délai de traitement simulé par les consommateurs.
- `routingKeys`, `bindingKey` : clé de routage et clé d'attachement utilisées respectivement par les producteurs et les consommateurs.
- `isPubSub` : indique si le scénario correspond à un modèle publication/abonnement.

Le paramètre `isPubSub` joue un rôle central dans la configuration du test :

- Lorsque sa valeur est `true`, le système adopte un modèle publication/abonnement :
  - `queueName` est ignoré
  - le `prefetchCount` est élevé
  - l'accusé de réception des messages est automatique ou avec un délai minimal
  - chaque consommateur crée sa propre file, liée à l'échange

- Lorsque sa valeur est `false`, le système fonctionne en mode point à point :
  - un `queueName` unique est partagé par tous les consommateurs ;
  - le `prefetchCount` est faible ;
  - l'accusé de réception est manuel, avec un délai de traitement plus important.

Dans RabbitMQ, le paramètre `prefetchCount` définit le nombre maximal de messages qu'un consommateur peut recevoir sans avoir accusé réception des précédents. Un exemple de configuration Docker d'un client de test est illustré à la figure 3.2.

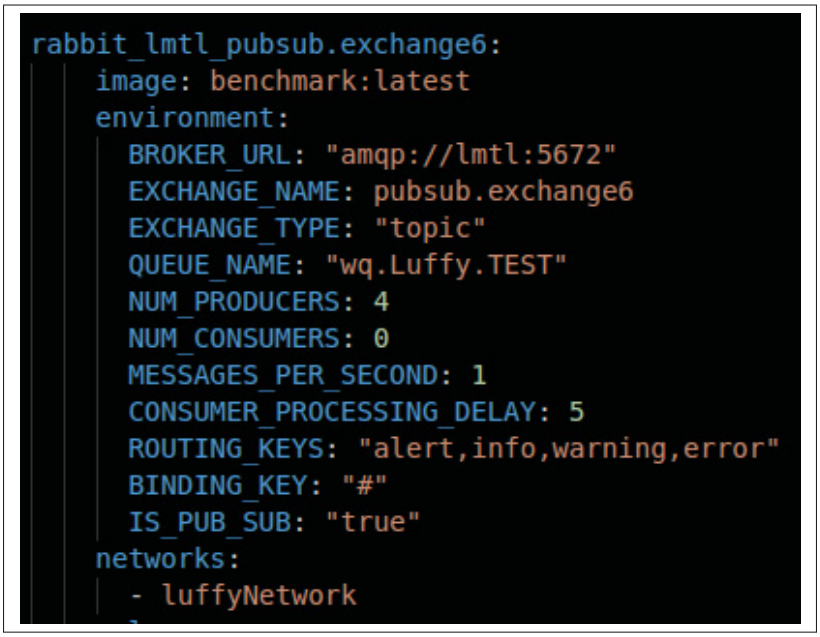



Figure 3.2 Paramètres de configuration d'un client de test

## 3.2 Évaluation et résultats

Conformément aux sous-objectifs définis dans la section 2.1, nous évaluons les performances de LuffyMQ selon plusieurs axes d'analyse. Les expérimentations portent sur deux modes de communication : le modèle point à point avec file d'attente et le modèle publication/abonnement.

La première série d'évaluations (section 3.2.2), consacrée au modèle point à point avec file d'attente, compare LuffyMQ à une configuration en maillage complet reposant sur le mécanisme

de fédération de files d'attente de RabbitMQ, dans un contexte géodistribué. LuffyMQ est également évalué face à des configurations centralisées infonuagiques.

La seconde série d'évaluations (section 3.2.3), centrée sur le modèle publication/abonnement, confronte LuffyMQ à une variante simplifiée de configuration aléatoire, à deux topologies classiques (maillage complet et étoile), ainsi qu'au modèle centralisé infonuagique. L'analyse inclut également l'étude de la scalabilité du système.

Enfin, la troisième série d'évaluations (section 3.2.4) examine plus en détail certaines décisions architecturales du système : la stratégie de conservation des messages, l'adaptation aux contraintes de bande passante, l'impact de la fréquence des reconfigurations et l'efficacité du mécanisme de pénalité dans la recherche d'un compromis entre stabilité et performance.

### 3.2.1 Environnement d'évaluation

Pour évaluer les performances et les fonctionnalités de notre système, nous l'avons déployé sur une plateforme de test en conditions réelles. Celle-ci repose sur une machine virtuelle à 12 cœurs, de 16 Go de mémoire vive et fonctionnant sous Ubuntu 24.10 LTS. La machine virtuelle est configurée comme un cluster, chaque nœud étant exécuté dans un conteneur Docker.

Le choix d'une plateforme en conditions réelles se justifie par la nécessité de mesurer la performance et la robustesse du système dans un environnement représentatif. Une telle configuration permet de reproduire un contexte proche d'un déploiement réel, tout en bénéficiant d'un cadre expérimental contrôlé.

L'environnement expérimental mis en place correspond à une architecture edge/cloud. Ses principales caractéristiques sont les suivantes :

- **Limite d'utilisation de bande passante inter-noeuds** : 3000 Kb/s.
- **Taille des messages** : 30 KB.
- **Pénalité de reconfiguration** : 100 unités
- **Fréquence de collecte des données par l'orchestrateur** : toutes les 2 minutes

- **Topologie expérimentale** : cinq nœuds distribués géographiquement à Berlin, Tunis, Montréal, Mexico City et Tokyo. Les latences entre nœuds ont été déterminées à partir des mesures réelles de temps de réponse ICMP (ping) obtenues via la plateforme (WonderNetwork, 2024). Cette topologie est présentée dans la figure 3.3

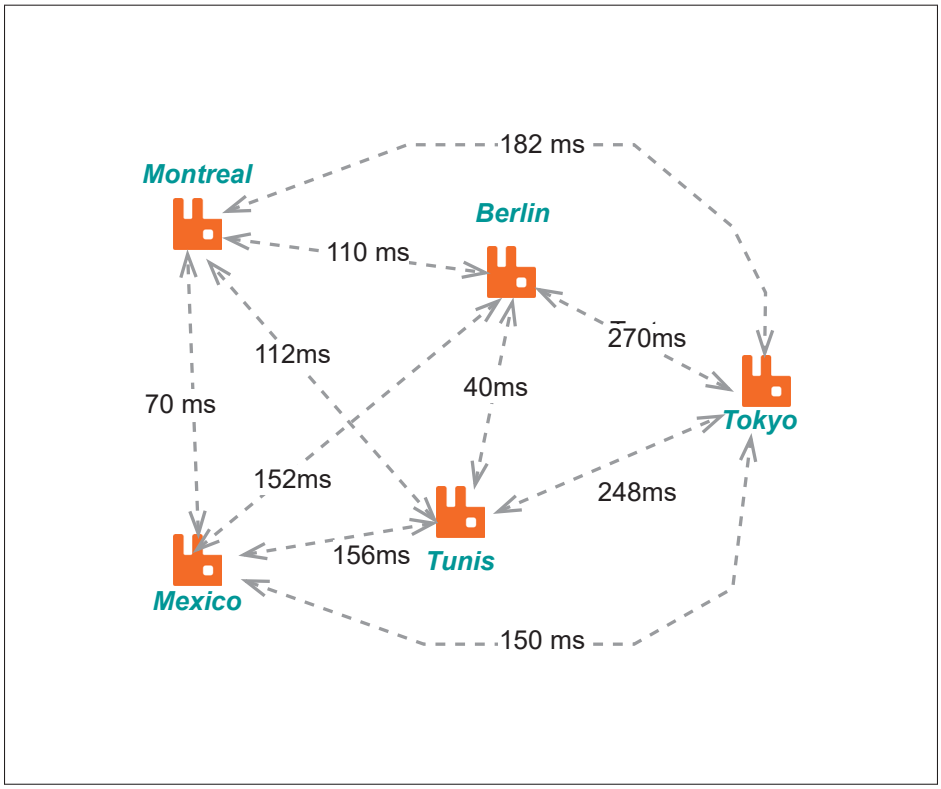



Figure 3.3 Topologie expérimentale avec les latences RTT

Compte tenu de la complexité du système étudié, chaque scénario d'expérimentation est répété cinq fois afin de renforcer la fiabilité des résultats. Pour chaque répétition, les mesures collectées sur les cinq sites géographiques sont agrégées. Les indicateurs de performance globaux sont ensuite calculés à partir de l'ensemble des résultats ainsi obtenus.

### 3.2.2 Évaluation du modèle point à point avec file d'attente

Cette section vise à évaluer la capacité du système à rééquilibrer la charge de travail entre différents sites edge. Dans ce modèle, les messages sont assimilés à des tâches qui doivent être exécutées une seule fois, c'est-à-dire par un unique consommateur. Il s'agit ainsi de modéliser un scénario typique AMQP où des producteurs génèrent des lots de tâches, tandis que des nœuds de traitement concurrents, par exemple des microservices, assurent leur exécution dans une logique de répartition de charge.

Nous mesurons la latence des messages dans une file d'attente distribuée sous LuffyMQ, en comparaison avec différentes configurations de référence basées sur RabbitMQ. Chaque scénario est exécuté pendant 10 minutes. Dans ce contexte, un message demeure en attente dans la file jusqu'à ce qu'un consommateur soit disponible pour le traiter. La latence est ainsi définie comme l'intervalle de temps entre l'émission du message par le producteur et sa réception effective par le consommateur, en incluant le délai éventuel d'attente en file. Par ailleurs, dans nos scénarios expérimentaux, chaque consommateur introduit volontairement un délai de traitement d'une seconde après la réception de chaque message. Ce comportement accentue directement le temps d'attente des messages suivants dans la file.

#### 3.2.2.1 Scénarios évalués

- **Charge équilibrée** : le volume de messages traités localement est équivalent à celui traité par des consommateurs distants.
- **Charge locale** : la majorité des messages sont consommés localement.

Chaque producteur publie un message par seconde, et chaque consommateur traite un message en une seconde. Théoriquement, dans les deux scénarios, la capacité globale du système est suffisante pour absorber le flux total publié.

#### 3.2.2.2 Configurations de référence (baselines)

La fédération de files d'attente proposée par RabbitMQ (présentée dans la section 1.10.2) permet, tout comme notre système LuffyMQ, d'appliquer le modèle des consommateurs concurrents sur une file unique répartie entre plusieurs sites géographiques. Concrètement, un message est transféré d'un site *upstream* vers un site *downstream* lorsqu'il y a des consommateurs actifs et que la file locale est vide.

À notre connaissance, aucun travail dans la littérature directement comparable à LuffyMQ n'a été identifié dans la littérature scientifique. Pour pallier cette absence, nous avons retenu des *baselines* représentatives des déploiements les plus courants : des configurations centralisées de type infonuagique, ainsi qu'une configuration distribuée FullMesh basée sur le mécanisme de fédération de files d'attente. Nous montrons par la suite que, bien que largement utilisées, ces approches s'avèrent insuffisantes dans un contexte géodistribué et contraint tel que l'edge computing. Les détails des configurations de références sont comme suit :

- **Modèle infonuagique centralisé** : Cette configuration illustre l'architecture classique edge/cloud, dans laquelle l'ensemble du traitement est centralisé dans le cloud, tandis que les nœuds edge se limitent à la collecte et à la transmission des données. Un cas d'usage typique dans l'IoT est celui de capteurs déployés en périphérie générant des données, ensuite traitées par plusieurs instances de microservices en concurrence, hébergées dans le cloud. Le cluster RabbitMQ, composé de cinq nœuds, est déployé dans le cloud avec un équilibreur de charge (`HAProxy`) placé en frontal pour diriger le trafic vers le cluster. Les producteurs, répartis sur l'ensemble des sites edge, publient leurs messages via cet équilibreur. Les consommateurs (microservices) sont hébergés à proximité immédiate du cluster dans le cloud, ce qui rend négligeable la latence entre les courtiers et les services consommateurs.
  Pour limiter la latence moyenne entre les nœuds edge et le cloud, nous avons positionné le site cloud sur le nœud géographiquement le plus central du système, à savoir le site de Berlin, qui correspond au nœud minimisant la distance moyenne aux autres nœuds dans le graphe du réseau, comme expliqué en section 2.8. Le choix du nœud central correspond à une

hypothèse réaliste : dans la pratique, les déploiements industriels sélectionnent généralement la région cloud en fonction de la localisation majoritaire du trafic généré.

- **Modèle infonuagique avec clients distribués** : Cette configuration reprend le modèle infonuagique centralisé, à la différence que les consommateurs, tout comme les producteurs, sont répartis sur les différents sites edge, tandis que le courtier demeure centralisé dans le cloud. Ce modèle permet une gestion centralisée de la distribution des tâches, tout en maintenant les nœuds de traitement distribués. Un cas d'usage représentatif consiste à déployer des unités de traitement sur des serveurs *on-premise* répartis sur plusieurs sites, le courtier jouant alors le rôle de passerelle centrale pour l'équilibrage des tâches.
- **FullMesh** : Cette configuration illustre la variante distribuée des *baselines*, dans laquelle chaque file d'attente est synchronisée avec l'ensemble de ses répliques distantes par le biais du mécanisme de *fédération de file d'attente* de RabbitMQ (voir section 1.10.2). Le transfert des messages entre les files est entièrement pris en charge par le plugin de fédération natif, sans intervention manuelle. En pratique, la fédération de files d'attente peut être organisée selon différentes topologies (graphe orienté, anneau, etc.). Nous avons retenu la configuration FullMesh, car elle englobe tous les cas possibles de transfert d'un message entre deux nœuds. Toute autre topologie peut être vue comme un sous-graphe, et donc comme un sous-ensemble des transferts possibles modélisés par le maillage complet.

#### 3.2.2.3 Résultats expérimentaux

Les figures se concentrent sur la partie principale de la distribution des messages, où la médiane et la moyenne sont bien visibles. Certaines valeurs extrêmes, issues de la majorité des configurations, tirent les diagrammes vers le haut, mais elles ne sont pas toutes représentées afin de préserver la lisibilité des graphiques.

La figure 3.4 présente les statistiques de latence observées dans le scénario *local*, où la majorité des messages sont pris en charge par les consommateurs locaux. Quatre configurations sont

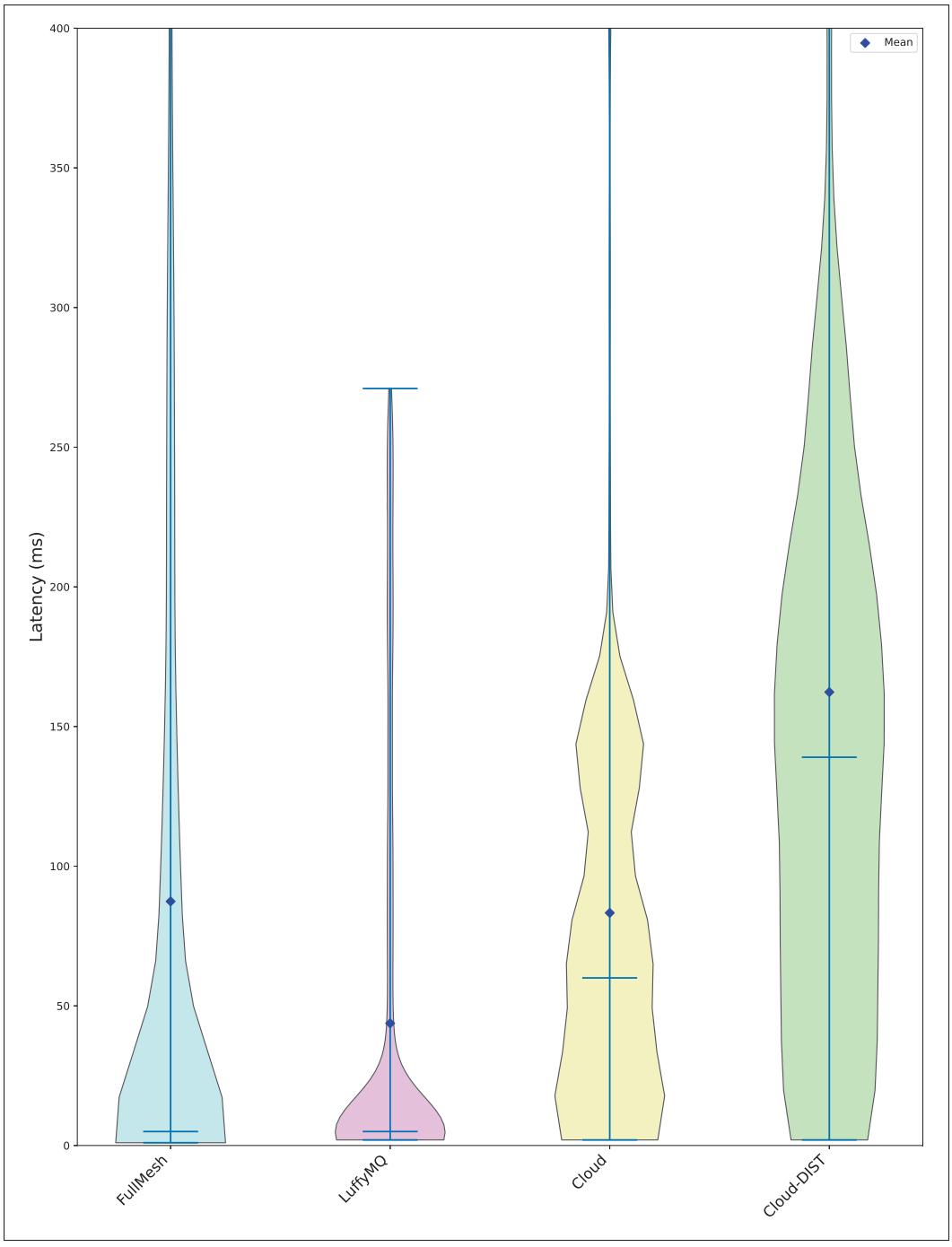



Figure 3.4 Latence du modèle point à point avec file d'attente dans le scénario *local*

comparées : **LuffyMQ**, **FullMesh**, **Cloud** (modèle infonuagique centralisé) et **Cloud-DIST** (cloud avec clients distribués).

Parmi ces configurations, LuffyMQ affiche les meilleures performances globales. La latence moyenne y est d'environ 40 ms, avec une médiane (50e percentile) particulièrement basse à 5 ms, ce qui signifie que la moitié des messages sont livrés en moins de 5 millisecondes. De plus, 75 % des messages sont acheminés en moins de 41 ms, et même les cas extrêmes restent sous la barre des 300 ms, ce qui démontre une grande stabilité.

Le modèle FullMesh, quant à lui, présente une latence moyenne plus élevée, avoisinant les 90 ms. Bien que sa médiane reste identique à celle de LuffyMQ (5 ms), la variabilité est plus marquée : de nombreux points aberrants (outliers) viennent tirer la moyenne vers le haut, témoignant d'une distribution plus étalée des délais. Nous formulons l'hypothèse que ces messages atypiques correspondent à ceux qui doivent être traités par des nœuds distants. Le modèle *Cloud* enregistre une latence moyenne de 83,5 ms, avec une médiane bien plus élevée à 60 ms soit douze fois supérieure à celle des configurations distribuées. Cela traduit un surcoût notable dû au passage par le nœud central du cloud, même si les performances restent acceptables.

Enfin, la configuration *Cloud-DIST* présente les latences les plus élevées : une moyenne proche de 200 ms et une médiane autour de 250 ms, accompagnées de nombreux outliers. Cette dégradation s'explique par les allers-retours systématiques entre les sites edge et le cloud centralisé, induisant une surcharge réseau significative.

Dans l'ensemble, ces résultats soulignent les avantages de l'architecture distribuée, en particulier dans les contextes où une grande partie des messages est consommée localement.

La figure 3.5 montre les résultats de latence dans le scénario équilibré, où le volume de messages traités localement est équivalent à celui traité par des consommateurs distants. Dans ce scénario, LuffyMQ surpasse nettement les trois configurations de référence : les deux modèles centralisés ainsi que le modèle entièrement décentralisé *FullMesh*.

La médiane de LuffyMQ demeure très basse, autour de 15ms, et la moyenne conserve également une position avantageuse, similaire à celle observée dans la figure précédente. Toutefois, la distribution présente un plus grand nombre de valeurs aberrantes (outliers), dont

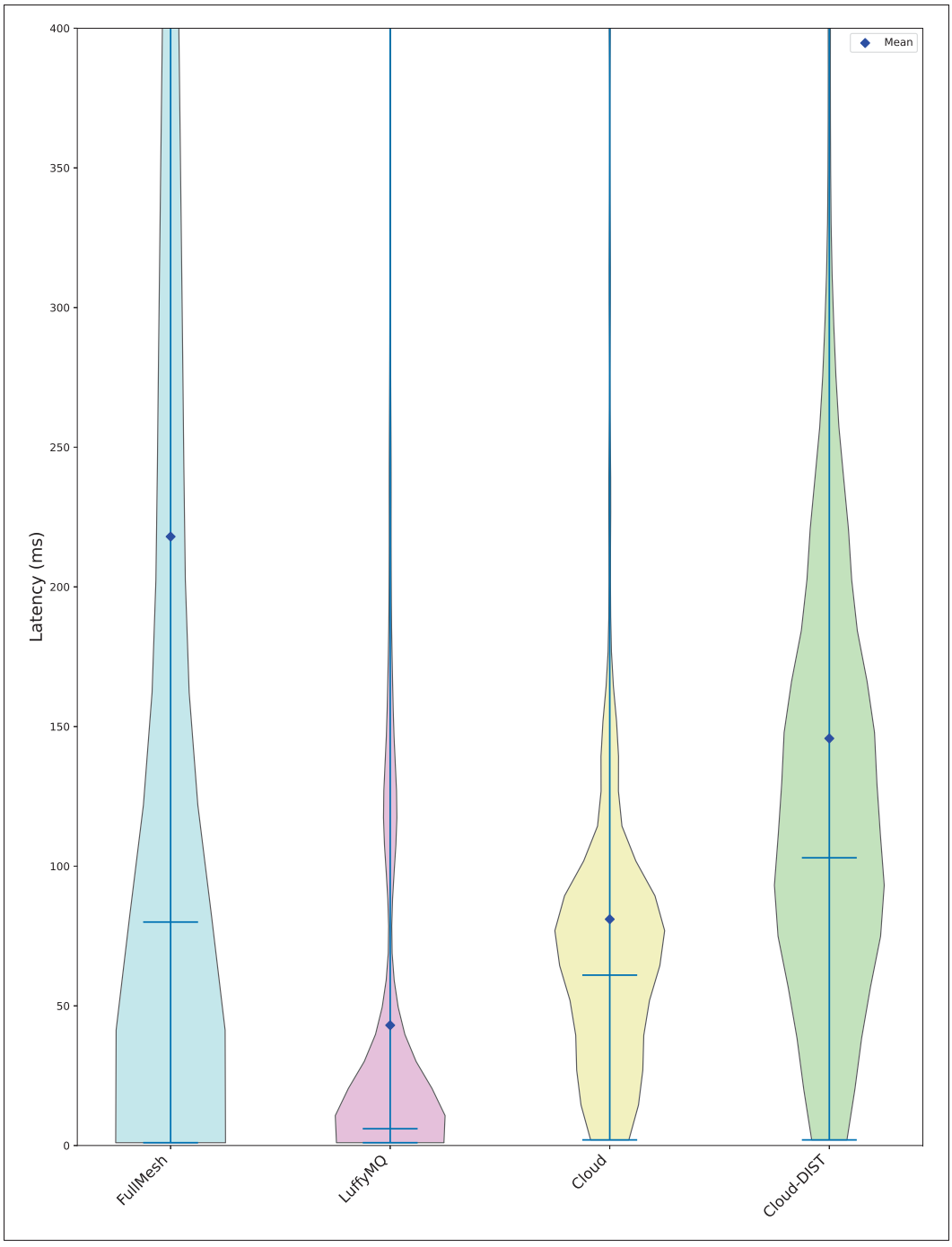



Figure 3.5 Latence du modèle point à point avec file d'attente dans le scénario équilibré

certaines dépassent les 400ms, soit au-delà de la limite affichée dans la figure. En revanche, les configurations *Cloud* et *Cloud-DIST* conservent une forme de distribution similaire à celle déjà

analysée. La performance de *FullMesh*, en revanche, se dégrade considérablement, avec une médiane avoisinant les 80ms et une moyenne qui dépasse les 200ms, la plus élevée parmi toutes les configurations testées.

Conformément à notre hypothèse, cette dégradation s'explique par la stratégie d'acheminement adoptée par FullMesh. En effet, le routage repose exclusivement sur la configuration statique du *plugin* de fédération de file d'attente de RabbitMQ, sans possibilité d'ajustement dynamique par l'utilisateur. Comme expliqué en section 1.10.2, cette stratégie suit un mécanisme de type *pull-driven*, où les messages ne sont transférés qu'à condition que la file aval *(downstream)* soit vide et que des consommateurs y soient toujours actifs.

Dans nos scénarios de test, où chaque site héberge à la fois des producteurs et des consommateurs actifs, il peut arriver que certains consommateurs se retrouvent temporairement sans messages locaux à traiter, ce qui déclenche une demande de messages distants. Cependant, cela ne se produit que ponctuellement, laissant la possibilité à une file d'attente de se remplir excessivement sur un site, sans assistance des autres nœuds pour absorber l'excès de charge.

LuffyMQ adopte une approche opposée, dite *push-driven*, plus proactive et préventive. Les messages sont envoyés vers les autres nœuds dès qu'un seuil est atteint, sans attendre que les files soient vides. L'administrateur peut configurer explicitement les conditions de transfert en ajustant les constantes de l'équation 2.12, en fonction des caractéristiques du trafic. Ces paramètres permettent notamment de définir :

- la durée maximale de rétention locale des messages, via le paramètre TTL des files d'attente.
- la capacité maximale autorisée pour une file avant que le transfert vers un autre nœud ne soit déclenché.

En résumé, les performances des scénarios centralisés *Cloud* et *Cloud-DIST* demeurent similaires dans les deux contextes. Dans le scénario local, les configurations distribuées *FullMesh* et LuffyMQ surpassent nettement les modèles centralisés. Cependant, lorsque la proportion de messages traités sur des sites distants augmente (scénario équilibré), *FullMesh* révèle ses limites et enregistre une forte dégradation. À l'inverse, LuffyMQ maintient des performances stables

grâce à sa stratégie d'équilibrage de charge proactive, qui anticipe la congestion au lieu de la subir.

### 3.2.3 Évaluation du modèle publication/abonnement

Nous avons approfondi l'évaluation dans le cas du modèle *publication/abonnement*, car dans notre architecture LuffyMQ, le modèle point à point avec file d'attente constitue un cas particulier du modèle pub/sub. Dans l'ensemble des scénarios évalués, tous les abonnés sont supposés recevoir l'intégralité des messages publiés à un échange donné. L'expérimentation porte sur 6 échanges *pub/sub* distincts.

L'évaluation se divise en deux volets : le premier consiste à comparer LuffyMQ à des configurations de référence, tandis que le second s'attache à analyser la scalabilité du système.

#### 3.2.3.1 Étude comparative de LuffyMQ avec les configurations de référence (baselines)

Comme indiqué précédemment, aucun travail directement comparable à LuffyMQ n'a été identifié dans la littérature scientifique. Nous avons évalué les performances de LuffyMQ en le comparant à quatre configurations de référence représentatives : une configuration centralisée de type infonuagique, deux topologies classiques reposant sur le mécanisme de fédération des échanges, *FullMesh* et *Hub and Spoke* (RabbitMQ, 2020b), ainsi qu'une variante simplifiée de LuffyMQ.

- **Modèle infonuagique avec clients distribués** : une architecture similaire à celle étudiée dans la section précédente, où les clients sont répartis sur plusieurs sites, mais les courtiers restent centralisés.
- **FullMesh** : inspirée de l'architecture interne d'un cluster RabbitMQ, qui repose sur une topologie en maillage complet dans laquelle chaque nœud est connecté à tous les autres (RabbitMQ, 2025). Nous reproduisons ce principe dans un contexte géodistribué, où chaque échange est fédéré avec toutes ses répliques distantes par l'intermédiaire de liens de fédération identiques à ceux utilisés par LuffyMQ.

- **Étoile (*Hub and Spoke*)** : topologie très répandue où un nœud central (le *hub*) assure la coordination, tandis que les autres nœuds (les *spokes*) y sont connectés. Concrètement, tous les échanges de synchronisation sont centralisés sur ce nœud central, qui agit comme point unique de coordination.
- **Aléatoire (*Random*)** : une variante simplifiée dans laquelle le placement des échanges de synchronisation est déterminé de façon aléatoire sur l'ensemble des nœuds du système, sans application d'une stratégie particulière.

#### 3.2.3.1.1 Méthodologie d'évaluation

Les scénarios de test sont les suivants :

- **Charge équilibrée (B)** : un nombre comparable de producteurs et de consommateurs est considéré. En raison de la duplication propre au modèle pub/sub, le volume de messages consommés dépasse celui des messages produits, puisque chaque publication est transmise à l'ensemble des consommateurs abonnés.
- **Producteurs majoritaires (P)** : le nombre de producteurs est largement supérieur à celui des abonnés. Le volume de messages produits reste proche du volume de messages consommés (peu de duplication).
- **Consommateurs majoritaires (S)** : inversement, le nombre d'abonnés dépasse largement celui des producteurs, engendrant un fort volume de duplication au niveau des nœuds.
- **Charge variable (dynamique)** : succession de phases alternant les trois scénarios précédents, selon la séquence suivante :

`P (4min)` → `S (4min)` → `B (4min)` → `S (4min)` → `P (4min)` → `B (4min)`

La durée des expérimentations est comme suit :

- **Charge équilibrée**, **producteurs majoritaires**, **consommateurs majoritaires** : chaque scénario est exécuté pendant 6 minutes.
- **Charge variable** : chaque phase de charge dure 4 minutes.

#### 3.2.3.1.2 Distribution des clients

Notre stratégie de distribution des clients s'appuie sur le fait que, dans les systèmes publish/subscribe à grande échelle, les sujets ne sont pas répartis de manière homogène d'un point de vue géographique. Les sujets populaires *(hot topics)* reflètent souvent des centres d'intérêt fortement localisés, comme l'illustrent les tendances de recherche observées sur des plateformes comme (Google Trends, 2025), où l'intérêt pour un même mot-clé peut varier considérablement d'une région à l'autre.

Pour reproduire cette dynamique dans notre configuration expérimentale, nous connectons 70% des clients (éditeurs ou abonnés) à seulement 25% des nœuds edge. De plus, chaque échange est répliqué sur un nombre variable de sites, allant de 2 à 5. Cette distribution implique que certains échanges bénéficient d'une présence plus marquée à l'échelle du système, tandis que d'autres restent plus localisés.

#### 3.2.3.1.3 Résultats expérimentaux

Les figures 3.6a, 3.6b, 3.7a et 3.7b présentent les cartes de chaleur *(heatmaps)* de la latence moyenne observée au fil du temps pour les quatre scénarios : consommateurs majoritaires, charge équilibrée, producteurs majoritaires et dynamique. La latence est représentée à l'aide d'un dégradé de couleurs, allant du jaune clair (latence faible) au bleu foncé (latence élevée). Les lignes rouges verticales visibles dans la figure 3.7b indiquent approximativement les transitions de phase.

Il convient de noter que la configuration *FullMesh* constitue ici une borne inférieure théorique en termes de latence, dans la mesure où chaque réplique reçoit instantanément les messages, sans latence supplémentaire. Toutefois, comme démontré en section 3.2.3.2, cette configuration révèle rapidement ses limites en matière de scalabilité.

Dans l'ensemble des scénarios, la configuration *Cloud* affiche une latence moyenne nettement plus élevée que les modèles distribués. En raison des contraintes expérimentales liées à l'ajout

de nouvelles latences entre les clients et le cloud à chaque changement de phase, des zones vides apparaissent en fin de chaque carte de chaleur associée à la configuration *Cloud*.

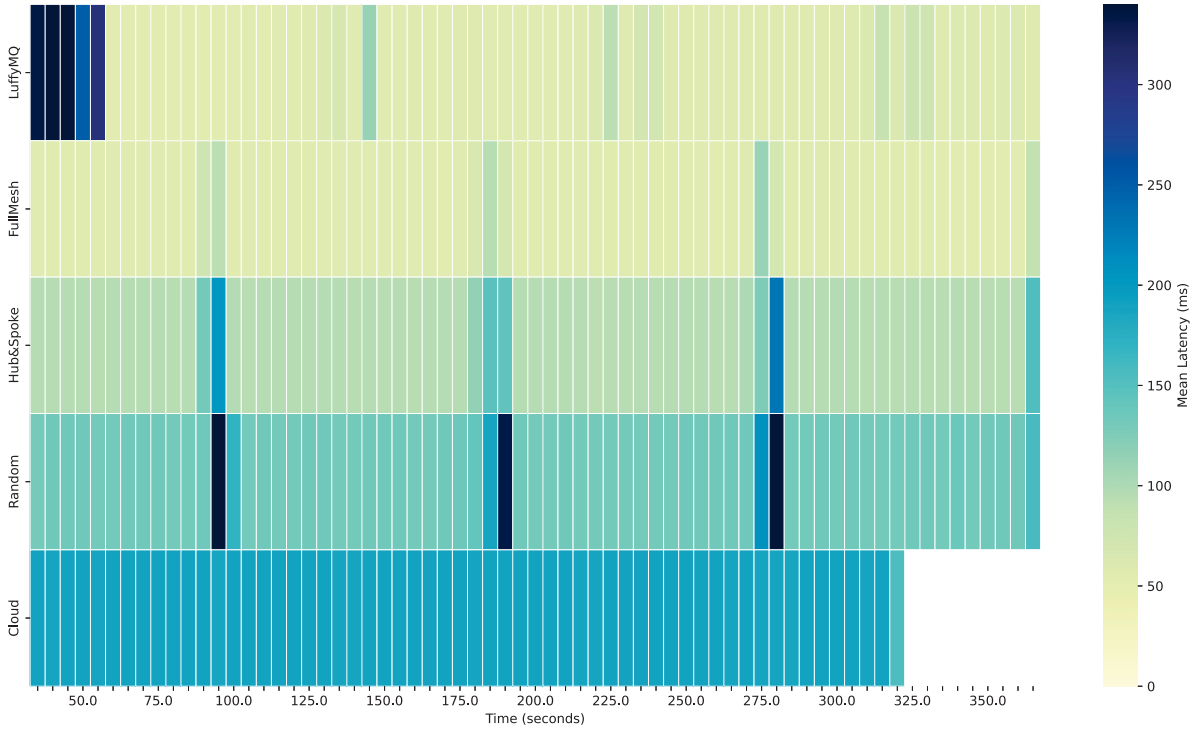
Les configurations *Aléatoire* et *Étoile* affichent des zones plus foncées sur les cartes de chaleur, signe d'une latence moyenne supérieure à celle des modèles *FullMesh* et LuffyMQ. Ce dernier présente généralement un niveau de latence proche de *FullMesh*, mais avec un pic d'initialisation du système de durée de 25 s au début des scénarios consommateurs majoritaires, charge équilibrée et producteurs majoritaires, ainsi que des pics récurrents dans le scénario dynamique.

Ces résultats conduisent à l'hypothèse suivante, qui sera examinée dans la section suivante : les pics de latence seraient liés aux opérations de reconfiguration dynamique du système. Plus précisément, la stratégie de conservation des messages adoptée pendant ces reconfigurations contribuerait à l'augmentation temporaire de la latence.

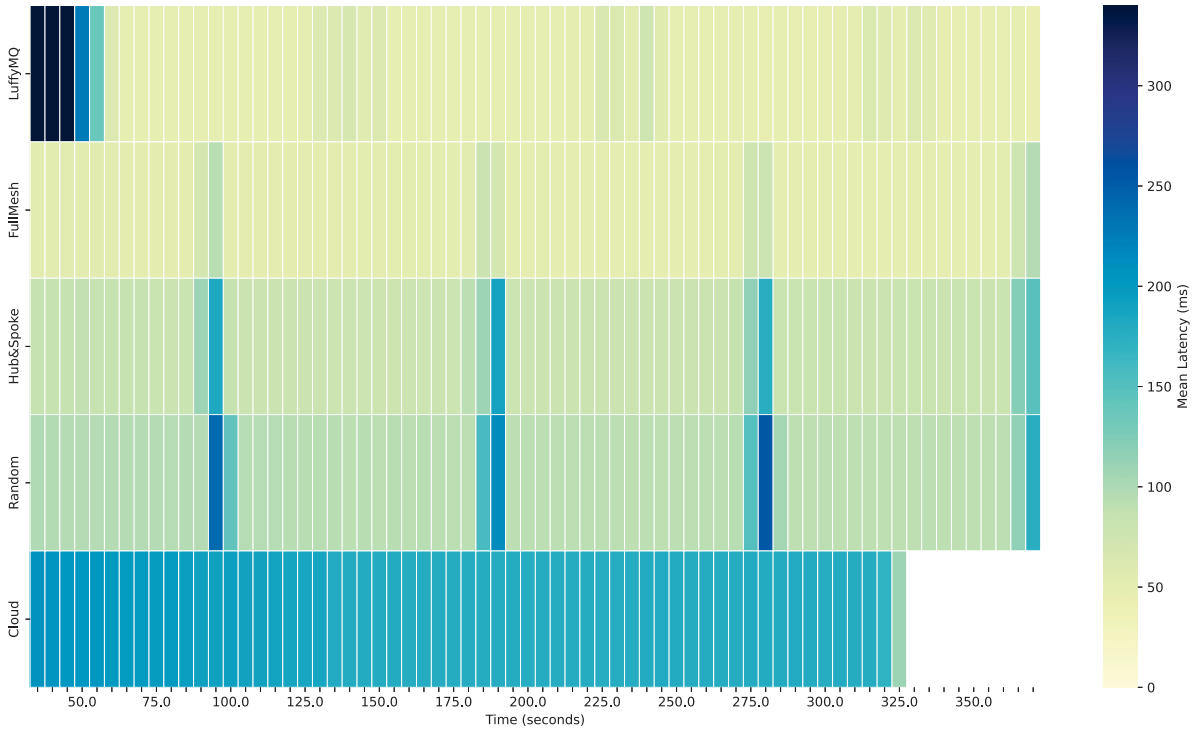
Comme indiqué lors de la présentation des scénarios, celui basé sur les consommateurs génère le trafic le plus faible sur le réseau *overlay*. En effet, pour transmettre un message aux abonnés distants, une seule copie est envoyée vers chaque site hébergeant des abonnés, puis la duplication s'effectue localement sur ce site. Par exemple, si un producteur publie un message destiné à 1000 abonnés répartis sur 10 sites edge distants, seuls 10 messages seront transmis sur le réseau pour cette publication.

Le scénario équilibré produit un volume de trafic plus élevé que le scénario consommateurs majoritaires, tout en conservant une certaine duplication locale. À l'inverse, le scénario basé sur les producteurs engendre le trafic le plus important, avec de nombreux messages distincts circulant sur le réseau *overlay* et très peu de duplication locale.

Cette dynamique explique les variations de taux de trafic générées dans la figure 3.8, qui illustre l'évolution du trafic par site au fil du temps dans le scénario dynamique qui enchaîne successivement les trois configurations de base. Il convient de noter que l'échelle de l'axe Y varie selon les graphiques. Plus en détail :

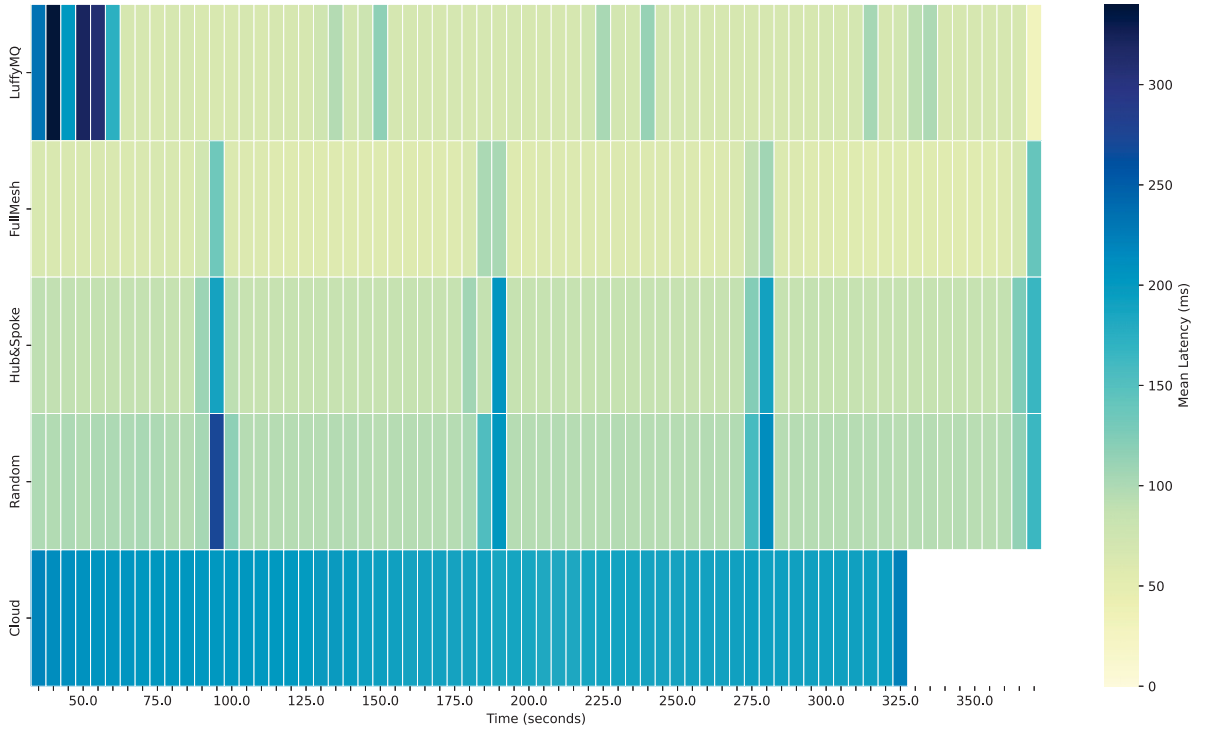



a) Évolution de la latence moyenne au fil du temps – scénario : Consommateurs majoritaires

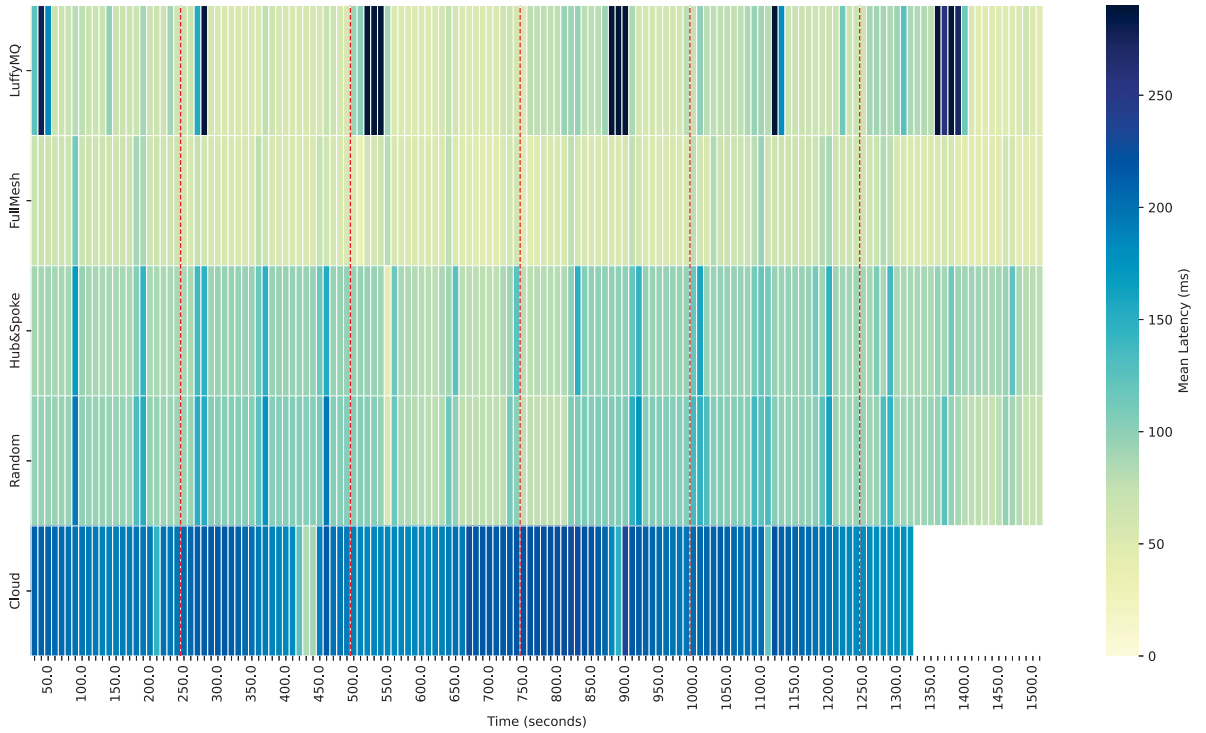



b) Évolution de la latence moyenne au fil du temps – scénario : Charge équilibrée

Figure 3.6 Évolution de la latence (1/2)




a) Évolution de la latence moyenne au fil du temps – scénario : Producteurs majoritaires




b) Évolution de la latence moyenne au fil du temps – scénario : Dynamique

Figure 3.7 Évolution de la latence (2/2)

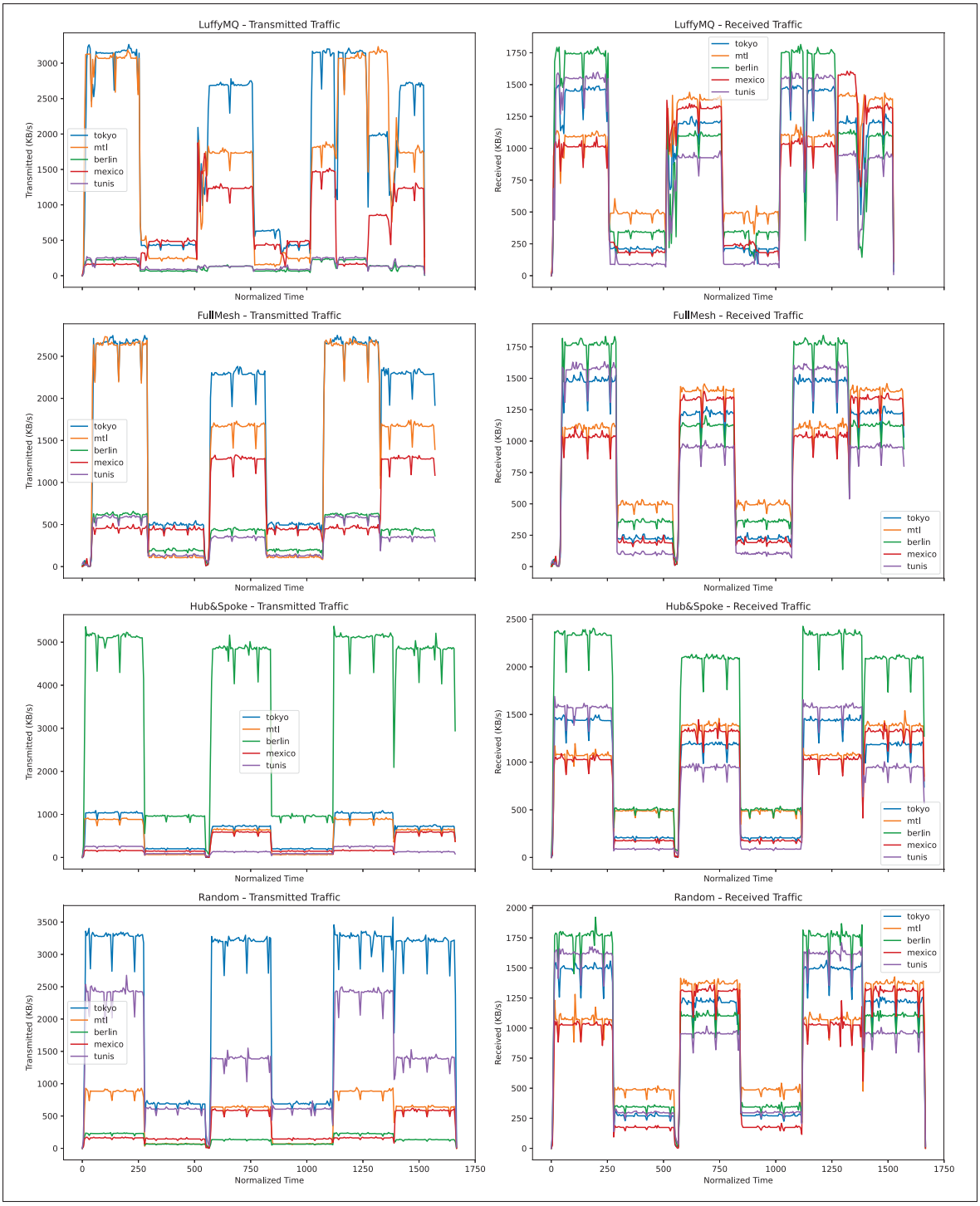



Figure 3.8 Évolution du trafic par site au fil du temps – scénario : *Dynamique*

- la première et la cinquième phases correspondent au scénario basé sur les producteurs, caractérisé par le volume de trafic le plus élevé ;
- la deuxième et la quatrième phases représentent le scénario basé sur les consommateurs, générant un trafic très faible ;
- la troisième et la sixième phases illustrent le scénario équilibré, avec un niveau de trafic intermédiaire.

Comme prévu, dans la configuration *Étoile*, on observe une concentration marquée du trafic sur le nœud *Berlin*, qui regroupe l'ensemble des instances de synchronisation. Sur ce nœud, le trafic dépasse largement la contrainte théorique initialement fixée, ce qui pourrait constituer une limite dans des environnements edge, où une concentration excessive du trafic sur un seul nœud est difficilement tolérable.

Par ailleurs, le modèle *Aléatoire* génère un trafic légèrement supérieur à celui des autres configurations, puisqu'il ne prend pas en compte les contraintes de bande passante. Ce résultat met en évidence son inadéquation à un environnement edge, où les ressources en bande passante sont limitées et toute surcharge de trafic demeure difficilement acceptable.

On constate également que la charge est plus uniformément répartie dans *FullMesh* que dans LuffyMQ. Cette différence s'explique par le fait que, dans LuffyMQ, pour chaque échange, un nœud unique est désigné pour assurer le transfert des messages vers les nœuds de destination, tandis que dans *FullMesh*, chaque nœud envoie directement les messages à l'ensemble des autres.

Comme indiqué précédemment, la configuration *FullMesh* constitue une borne inférieure en termes de latence et de trafic, difficile à surpasser. Toutefois, ses limites apparaîtront dans la section suivante.

#### 3.2.3.2 Étude de la mise à l'échelle

Afin d'évaluer la mise en échelle de notre système ainsi que des configurations de référence distribuées, nous avons créé un environnement de test distinct sur la même machine hôte, mais hébergé dans une machine virtuelle disposant de 60 cœurs et de 32 Go de mémoire RAM.

Dans ce contexte, nous avons déployé 29 instances RabbitMQ, chacune représentant un site edge situé dans une zone géographique distincte. Les villes retenues comme sites edge sont les suivantes : Adelaide, Dar es Salaam, Bucarest, Buenos Aires, Le Caire, Le Cap, Dallas/Fort Worth, Dubaï, Hong Kong, Lagos, Lima, Londres, Los Angeles, Montréal, Moscou, New Delhi, Orlando, Paris, San Juan, Santiago, São Paulo, Stockholm, Tokyo, Toronto, Tunis, Vancouver, Kuala Lumpur, Barcelone, Tirana, Athènes, Edmonton, Shanghai, Copenhague, Mexico City et Sydney.

Pour cette expérimentation, nous avons choisi d'utiliser 15 échanges au total. Chaque échange peut accueillir jusqu'à 140 clients, répartis sur l'ensemble des sites. Nous avons conservé les trois scénarios de test précédemment définis : équilibrée, producteurs majoritaires et consommateurs majoritaires. Chaque échange est répliqué sur un nombre variable de sites, allant de *4 à 20*.

Les défis de la scalabilité sont principalement liés au volume des messages échangés entre les sites et au nombre de liens de synchronisation à maintenir. En effet, les résultats de scalabilité sont directement influencés par le nombre de clients, de sites et d'échanges actifs. Étant donné que le système repose sur la synchronisation des échanges, le nombre d'échanges a un impact direct sur la complexité réseau. À titre d'exemple, dans une configuration FullMesh impliquant 15 échanges répliqués sur 10 nœuds chacun, chaque échange exige $10 \times 9 = 90$ liens de synchronisation, ce qui représente un total de 1350 connexions ($90 \times 15$) à établir et à maintenir.

Le tableau 3.1 présente les résultats de l'expérimentation en termes de pourcentage de fréquence des messages reçus par rapport au nombre de messages attendus. Les configurations FullMesh et aléatoire n'ont pas résisté à ces conditions extrêmes, affichant des pourcentages très faibles jusqu'à quatre fois moins que LuffyMQ.

Tableau 3.1 Taux de réception des messages par seconde selon l'architecture et le scénario

| **Architecture** | **Équilibrée (%)** | **Consommateurs maj (%)** | **Producteurs maj (%)** |
|---|---|---|---|
| LuffyMQ | 72.3% | 78.8% | 68.7% |
| FullMesh | 24.2% | 19.9% | 18.9% |
| Étoile | 52.9% | 54.7% | 44.2% |
| Aléatoire | 24.6% | 26.3% | 23.1% |
| *Attendu (100%)* | 20 933 | 23 108 | 21 211 |

Dans le cas de FullMesh, la dégradation des performances s'explique par le nombre très élevé de connexions à établir et à maintenir au sein de la couche *overlay*, avec un intervalle variant de 480 à 12 000 connexions. Quant à la stratégie aléatoire, elle se révèle inefficace dans un contexte à grande échelle, notamment lorsque la latence entre les sites peut être très élevée. À l'inverse, LuffyMQ réduit significativement le nombre de connexions nécessaires et sélectionne intelligemment les sites de synchronisation. Par exemple, le nœud du *Cap Town* présente une latence moyenne (ping unidirectionnel, sans réponse) de 139 ms vers les autres sites et une latence maximale de 220 ms vers Sydney. Un tel nœud ne devrait pas être sélectionné comme nœud de synchronisation. En revanche, dans un environnement minimal comprenant seulement cinq nœuds, avec une latence maximale de 120 ms, la stratégie aléatoire demeure tolérable.

Bien que le taux de réception de messages de LuffyMQ n'ait pas dépassé les 80% par seconde, nous estimons qu'il démontre une capacité de scalabilité satisfaisante. Cette conclusion s'appuie sur le caractère particulièrement exigeant du protocole expérimental, incluant de sévères contraintes matérielles et la gestion d'un volume de données très élevé, dépassant les 20 000 messages par seconde.

### 3.2.4 Étude du comportement du système LuffyMQ

Dans cette section, nous analysons le comportement de notre système dans différentes conditions de test, en faisant varier les paramètres afin d'évaluer l'impact et la pertinence de chaque décision architecturale. Nous nous intéressons particulièrement au respect des contraintes d'utilisation

de la bande passante, à la pénalité, à la fréquence de reconfiguration, ainsi qu'à la stratégie de conservation des messages.

Compte tenu de la pertinence des résultats obtenus, nous avons modifié la répartition des clients et l'ordre des scénarios de base composant le scénario dynamique, en adoptant la séquence suivante : $S \rightarrow P \rightarrow B \rightarrow S \rightarrow B \rightarrow P$.

Les lignes verticales rouges, visibles sur les figures d'évolution de la latence présentées ci-après, indiquent les changements de phase.

#### 3.2.4.1 Stratégie de conservation des messages

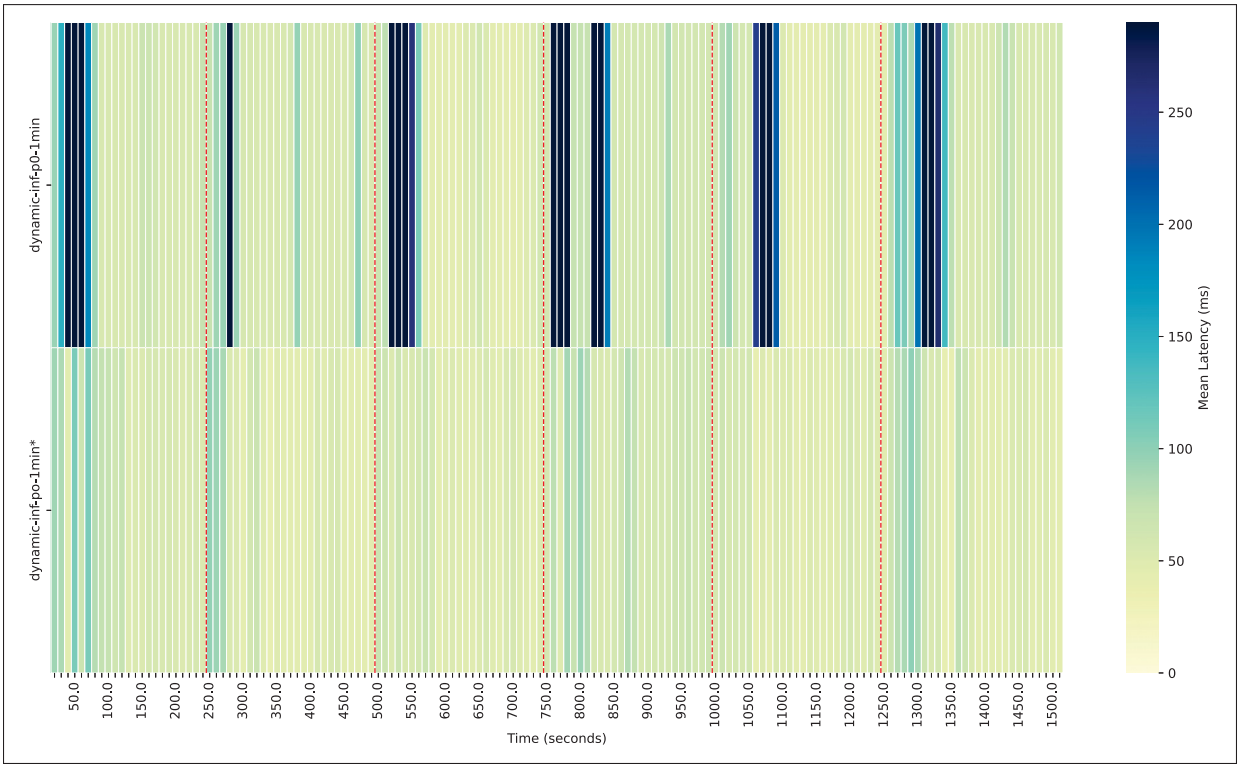



Figure 3.9 Évolution de la latence moyenne au fil du temps avec et sans stratégie de conservation des messages

Dans la section précédente, nous avons émis l'hypothèse que les pics de latence observés provenaient des opérations de reconfiguration dynamique. Plus précisément, la stratégie de

conservation des messages appliquée durant ces phases pourrait engendrer une augmentation temporaire de la latence.

Pour valider cette hypothèse, deux expériences ont été menées : l'une avec la stratégie activée, l'autre sans. Les tests ont été réalisés sans pénalité, avec une bande passante élevée et une fréquence de reconfiguration fixée à une minute. La figure 3.9 présente les résultats : la carte supérieure correspond à la stratégie activée, la carte inférieure à la stratégie désactivée.

Comme anticipé, les pics de latence (d'une durée d'environ 25 s) disparaissent lorsque la stratégie est désactivée. Toutefois, cette configuration provoque une perte d'environ 4 % des messages par rapport à l'approche de LuffyMQ. (tableau 3.2). Il existe donc un compromis à considérer selon la criticité des pertes de messages ponctuelles lors des phases de reconfiguration.

La figure 3.10 présente la médiane ainsi que les 75$^{e}$ et 85$^{e}$ percentiles. Les médianes des deux configurations sont quasiment identiques, mais les percentiles révèlent des pics plus marqués avec la stratégie de conservation, en particulier pour le 85$^{e}$, ce qui indique qu'une minorité de messages est impactée. Le pic le plus important apparaît vers 800 s, lors de la phase où les consommateurs sont majoritaires : bien que peu de messages soient retenus puis réémis (avec un délai d'environ 1,5s), chacun affecte simultanément un grand nombre de consommateurs. En revanche, sans la stratégie de conservation, ces pics disparaissent et la latence du 85$^{e}$ percentile ne dépasse pas 200 ms.

En conclusion, lors d'une reconfiguration, la grande majorité des messages reste peu affectée dans les deux configurations. Seuls les messages temporairement retenus par la stratégie de conservation, représentant environ 4 % du trafic, subissent un délai supplémentaire. Le compromis consiste donc à choisir entre tolérer une faible perte de messages ou accepter un retard de livraison affectant seulement un pourcentage minime du trafic

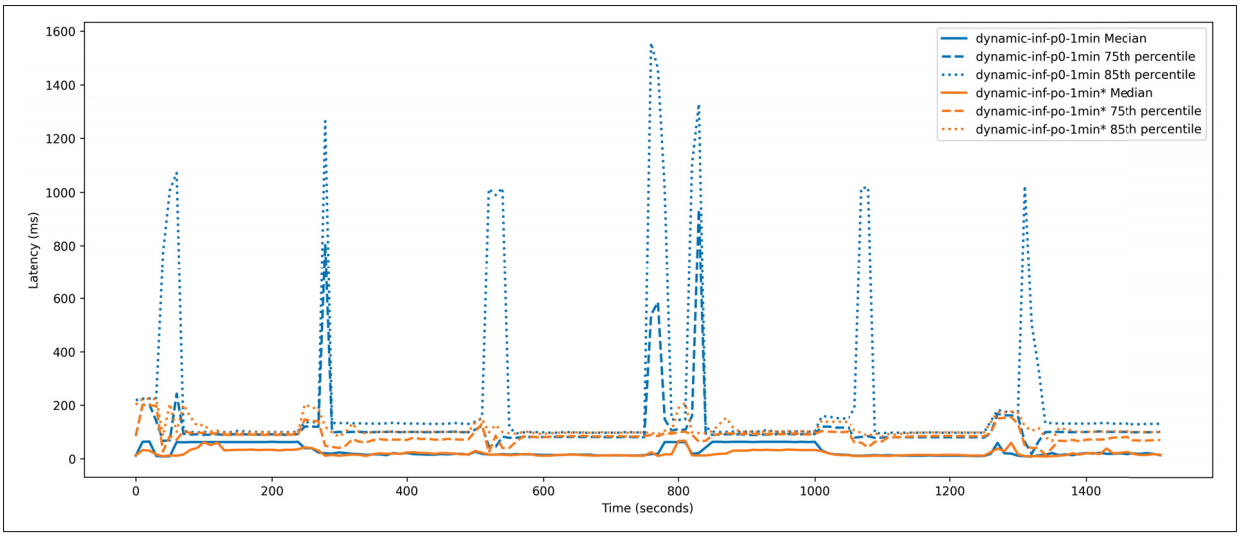



Figure 3.10 Évolution de la latence médiane, 75$^{e}$ et 85$^{e}$ percentiles avec et sans stratégie de conservation des messages

Tableau 3.2 Impact de la stratégie de conservation des messages sur le nombre de messages reçus

| **Expérience** | **Messages reçus par seconde** | **Pourcentage** |
|---|---|---|
| Stratégie activée (`dynamic-inf-p0-1min`) | 1381 | 100% |
| Stratégie désactivée (`dynamic-inf-p0-1min*`) | 1331 | 96% |

#### 3.2.4.2 Respect des contraintes

Nous avons modifié la répartition des clients de manière à garantir que, pour chaque échange, plusieurs sites soient fortement sollicités. L'objectif est d'analyser la latence et la répartition du trafic dans ce type de scénario.

Sans appliquer de pénalité et avec une fréquence de reconfiguration fixée à une minute, nous avons évalué notre système dans trois configurations :

- une limite de bande passante très élevée par rapport au trafic généré, notée *inf* (pour infinie),
- une contrainte intermédiaire de *3Mo*,
- une contrainte plus stricte, notamment pour les phases producteurs majoritaires et *balanced* de *2.5Mo*.

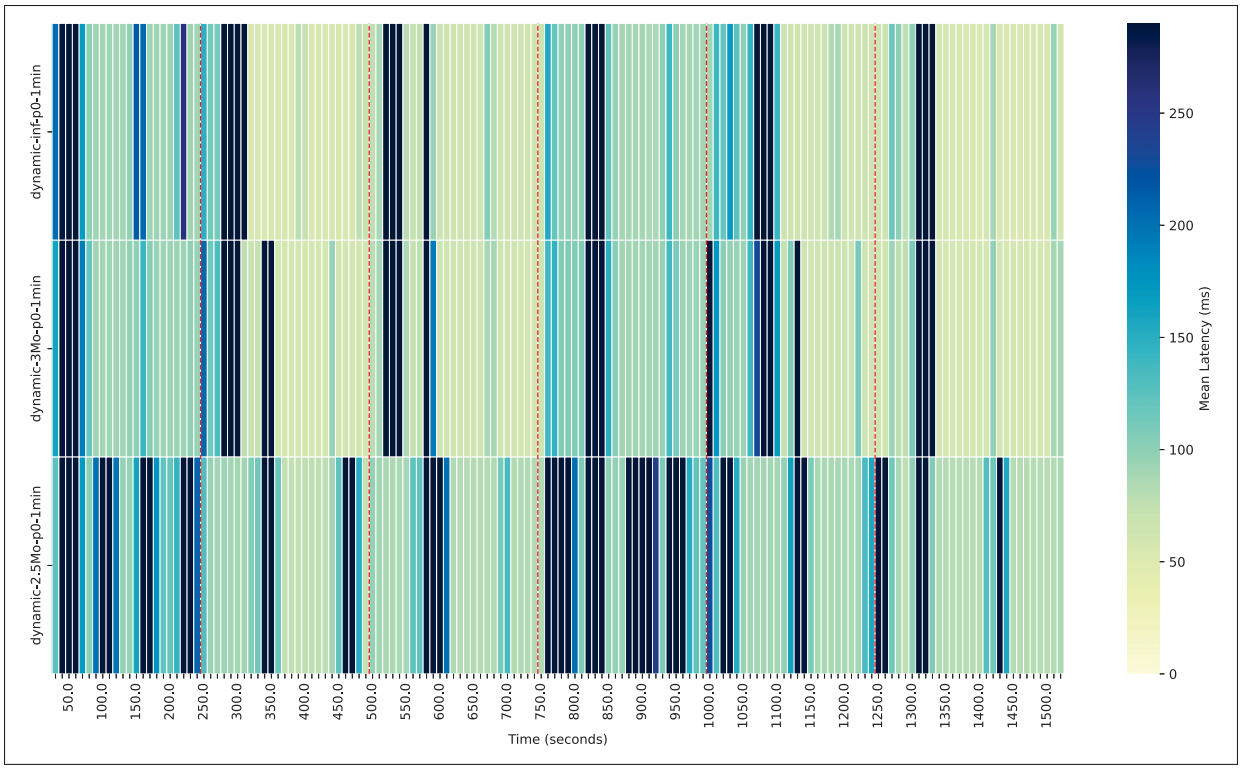



Figure 3.11 Évolution de la latence moyenne selon différentes contraintes de bande passante

La figure 3.11, qui illustre l'évolution de la latence au fil du temps, montre que dans le premier scénario, le système reste relativement stable : les reconfigurations ne surviennent qu'au changement de phase (toutes les 4 minutes), et ce, malgré une fréquence de contrôle d'une minute. En revanche, dans les deux autres scénarios, les reconfigurations deviennent de plus en plus fréquentes. Ce comportement s'explique par le fait que, plus la contrainte est stricte, plus le système doit faire des compromis sur la configuration choisie. Cette instabilité reflète la tendance du système à tenter régulièrement d'améliorer la configuration en place.

Concernant le trafic, le système parvient globalement à respecter les contraintes de bande passante, à l'exception de quelques pics isolés. Plus les contraintes sont strictes, plus la charge tend à s'homogénéiser entre les sites. Ce comportement s'explique par la stratégie adoptée par notre algorithme : dès que les contraintes ne sont plus respectées, l'échange de synchronisation est placé dans le site le moins chargé, sans considération de la latence. Cela justifie l'apparition

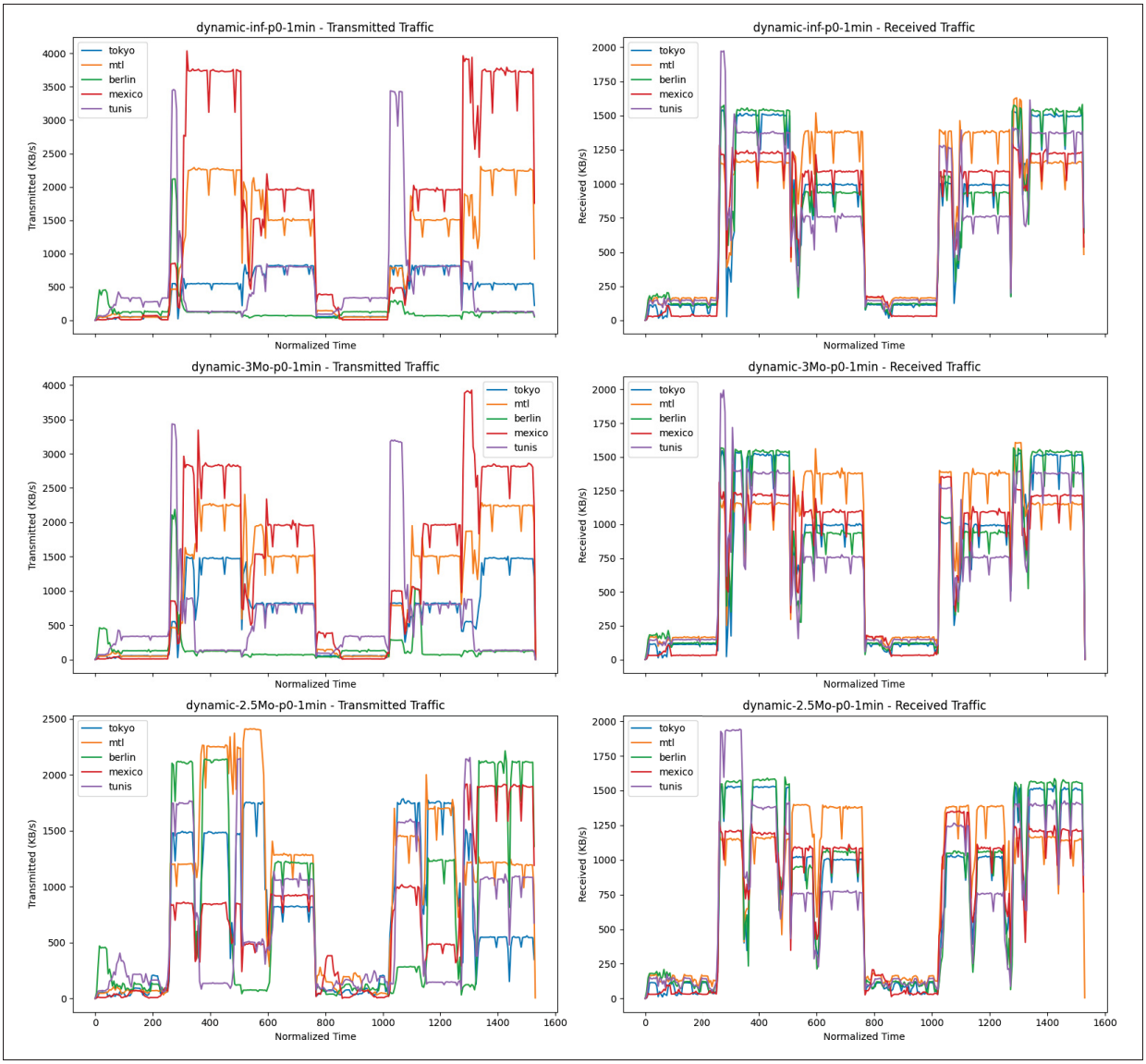



Figure 3.12 Évolution du trafic par site selon différentes contraintes de bande passante

de pics isolés dans la configuration intermédiaire, tandis que la charge est répartie de manière plus uniforme lorsque la contrainte est très stricte.

Comme l'illustre la figure 3.12, lors de la phase producteurs majoritaires avec une contrainte infinie, le trafic transmis est fortement concentré sur le nœud *Mexico*, qui permet de minimiser la latence. À l'inverse, avec la contrainte la plus stricte, le trafic transmis devient plus uniformément

réparti entre les sites. Cependant, cette contrainte serrée s'accompagne d'une instabilité du trafic, due principalement aux reconfigurations fréquentes du système.

#### 3.2.4.3 Fréquence de reconfiguration

Afin d'analyser l'impact de la fréquence de reconfiguration sur le comportement du système, nous avons mené trois expériences avec des intervalles distincts :

- 1 minute, soit largement inférieure à la fréquence de changement de phase
- 4 minutes, équivalentes à la fréquence de changement de phase
- 6 minutes, supérieures à la fréquence de changement de phase.

Ces tests ont été réalisés sans pénalité et avec une contrainte de bande passante très élevée.


Figure 3.13 Évolution de la latence moyenne au fil du temps selon la fréquence de reconfiguration

Lors du lancement de l'expérience, les courtiers sont mis en place avec une configuration initiale de fédération fournie par le contrôleur, avant le démarrage des clients. Ainsi, à l'instant $t = 0$ sur la figure des résultats 3.13, une configuration aléatoire d'initialisation est appliquée, puis les clients commencent à émettre.

Dans le scénario avec une fréquence de reconfiguration de 1 minute, le contrôleur ajuste la configuration presque immédiatement après le démarrage (en moins d'une minute), en tenant compte des clients actifs. Par la suite, chaque changement de phase (6 au total) est suivi d'une reconfiguration.

Avec une fréquence de 4 minutes, le contrôleur parvient à réviser la configuration pour chaque phase, mais avec un léger retard. On observe bien 6 reconfigurations, mais décalées par rapport au début de chaque phase. Ces décalages entraînent une augmentation temporaire de la latence pendant la période de transition.

Enfin, avec une fréquence de 6 minutes, le contrôleur ne peut pas suivre chaque changement de phase : seules 3 reconfigurations sont observées au lieu de 6. Ce rythme plus lent se traduit par des périodes prolongées avec des configurations moins adaptées. Étant donné que le contrôleur applique une approche gloutonne et ne recherche pas la solution optimale globale, certaines configurations peuvent rester imparfaites plus longtemps, ce qui contribue à des latences plus élevées que dans les deux autres scénarios.

#### 3.2.4.4 Pénalité de reconfiguration

Dans des conditions contraignantes, le système tend à optimiser en permanence sa configuration, ce qui entraîne un grand nombre de reconfigurations redondantes. Afin d'évaluer l'impact d'une pénalité sur ce comportement, nous avons mené des expérimentations avec une limite de bande passante fixée à 2,5 Mo.

La figure 3.14 présente l'évolution de la latence dans le scénario dynamique pour différentes valeurs de pénalité :

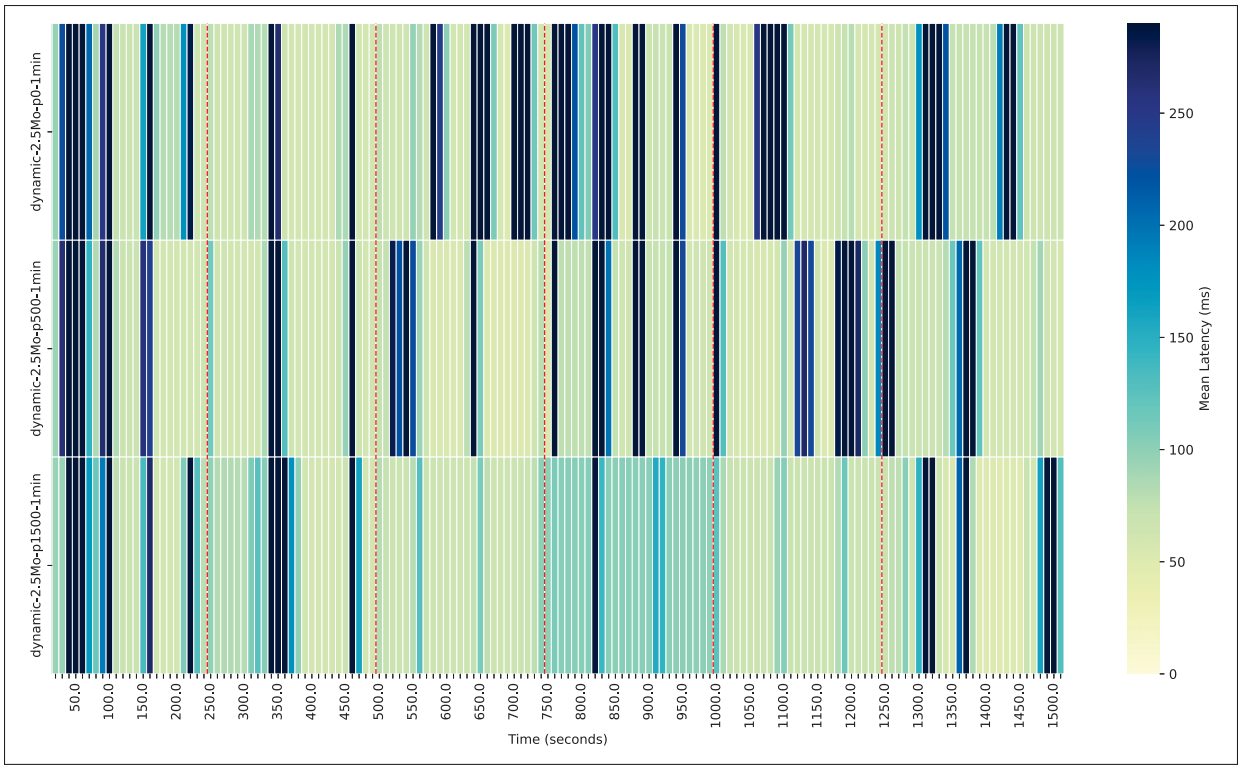



Figure 3.14 Évolution de la latence moyenne au fil du temps selon différentes valeurs de pénalité

- Sans pénalité : nombreuses reconfigurations redondantes, le système cherchant constamment à optimiser la configuration en place. En l'absence de gains significatifs, ces reconfigurations fréquentes peuvent déstabiliser le système et entraîner des retards ou des pertes ponctuelles sur les messages retenus, ce qui constitue un risque pour un déploiement en production.
- Pénalité modérée (500) : une réduction notable des reconfigurations interphases, tout en conservant des reconfigurations quasi synchrones lors des changements de phase.
- Pénalité élevée (1500) : la distinction entre les phases est moins marquée, les reconfigurations sont beaucoup plus rares et une augmentation de la latence moyenne (zones plus foncées).

Ainsi, une pénalité modérée permet de limiter efficacement les reconfigurations redondantes sans dégradation notable des performances. En revanche, une pénalité trop importante réduit la réactivité du système face aux variations de trafic, entraînant une augmentation globale des latences.

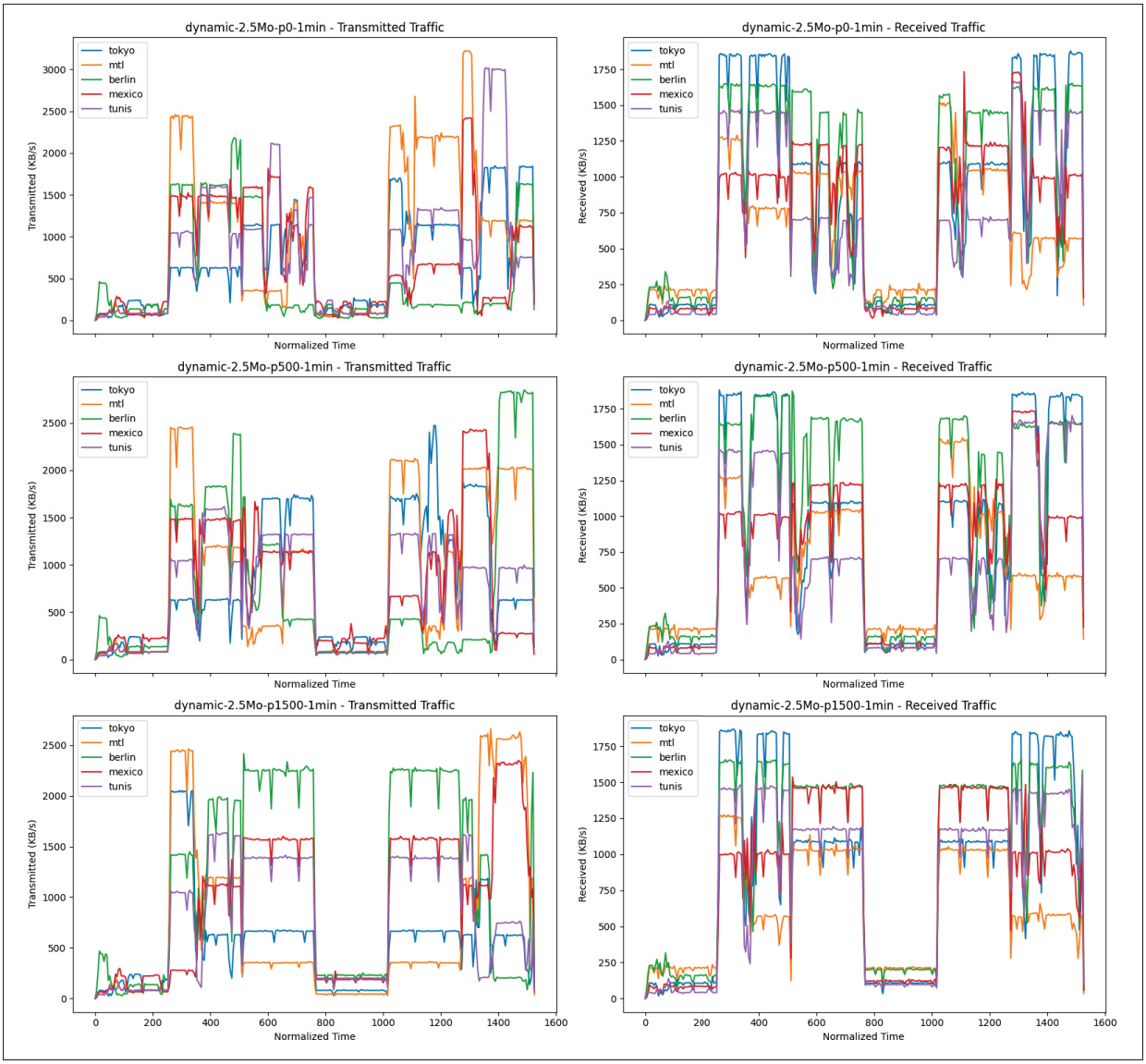



Figure 3.15 Évolution du trafic par site dans le scénario dynamique selon différentes valeurs de pénalité

Du point de vue du trafic (figure 3.15), on observe qu'à mesure que la pénalité augmente, le volume échangé tend à se stabiliser. Les phases de trafic deviennent plus nettes et régulières, conséquence directe de la réduction du nombre de reconfigurations.

## 3.3 Discussion

Cette section propose une synthèse des principaux résultats obtenus et met en évidence les limites de l'étude.

### 3.3.1 Synthèse des résultats

Nous présentons ci-dessous une synthèse des résultats obtenus lors de l'évaluation de notre solution, en lien avec les objectifs de recherche définis dans le chapitre Approche (section 2.1).

L'objectif général de cette étude était de concevoir, d'implémenter et d'évaluer un système de messagerie en files d'attente géodistribuée, capable de supporter à la fois les modèles publication/abonnement et point à point, et spécifiquement adapté aux environnements edge. Ce système visait à réduire le temps d'attente dans les files ainsi que la latence de transmission entre sites distribués, tout en respectant les contraintes de bande passante. Les résultats sont présentés ci-dessous, organisés par sous-objectif.

#### 3.3.1.1 Sous-objectif 1 : Évaluer l'efficacité de la stratégie d'équilibrage de charge de LuffyMQ dans le modèle point à point avec files d'attente

Dans le modèle point à point avec files d'attente, la comparaison avec une configuration FullMesh, qui couvre l'ensemble de transitions possibles pour un message via la fédération de files d'attente, et deux configurations centralisées (consommateurs distribués et centralisés) montre que LuffyMQ surpasse ces approches, aussi bien dans un scénario où la majorité des messages sont traités localement que dans un scénario où les volumes de traitement local et distant sont équivalents. Dans les deux cas, la réduction de la latence moyenne atteint jusqu'à 75%.

#### 3.3.1.2 Sous-objectif 2 : Analyser les performances de LuffyMQ dans le modèle publication/abonnement

LuffyMQ a été évalué dans quatre scénarios : charge équilibrée, producteurs majoritaires, consommateurs majoritaires et charge variable (dynamique). La configuration centralisée (cloud) présente systématiquement la latence moyenne la plus élevée. LuffyMQ surpasse également les topologies en étoile et aléatoire, tant en termes de latence que de répartition du trafic. L'approche aléatoire engendre un trafic plus important, tandis que la topologie en étoile concentre l'ensemble du trafic sur un seul nœud, ce qui la rend peu adaptée aux environnements edge. La configuration FullMesh reste néanmoins compétitive dans des environnements minimaux, puisqu'elle constitue une borne inférieure en termes de latence, mais elle présente des limites de scalabilité, comme démontré dans le sous-objectif 3.

#### 3.3.1.3 Sous-objectif 3 : Examiner la scalabilité du système

En situation de montée en charge, LuffyMQ démontre une nette supériorité. Dans les trois scénarios de base, il traite jusqu'à 200 % de messages supplémentaires par rapport à FullMesh. Les configurations FullMesh et aléatoire montrent une faible mise en échelle, tandis que la configuration en étoile se comporte mieux, mais demeure inférieure à LuffyMQ.

#### 3.3.1.4 Sous-objectif 4 : Étudier la sensibilité du système en faisant varier ses paramètres expérimentaux

Cette analyse de sensibilité vise à mesurer l'impact de différents paramètres expérimentaux sur la performance du système. Elle permet d'évaluer les compromis entre latence, stabilité et utilisation des ressources à travers quatre dimensions : conservation des messages, contraintes de bande passante, fréquence des reconfigurations et introduction d'une pénalité.

#### 3.3.1.4.1 Impact de la stratégie de conservation des messages

L'étude des distributions de latence (moyenne, médiane, 75e et 85e percentiles) et du nombre de messages reçus met en évidence un compromis. Avec la conservation activée, des pics temporaires apparaissent dans les percentiles élevés après les reconfigurations, dus aux messages retenus puis réémis. Avec la conservation désactivée, ces pics disparaissent, mais une perte de 4 % des messages est observée. Les messages responsables des pics correspondent donc à ceux retenus lors des reconfigurations. Le choix dépend ainsi de la criticité des messages.

#### 3.3.1.4.2 Adaptation aux contraintes de bande passante

Trois niveaux de contrainte ont été étudiés : élevée, intermédiaire et stricte. Avec une contrainte élevée, le système reste stable et ne se reconfigure qu'en cas de changement de phase. Plus la contrainte est stricte, plus les reconfigurations sont fréquentes, traduisant la tendance du système à optimiser continuellement sa configuration. Le trafic respecte globalement les limites de bande passante, à l'exception de quelques pics isolés. Plus la contrainte est stricte, plus la charge tend à s'homogénéiser entre les sites. LuffyMQ s'adapte ainsi efficacement en équilibrant la charge en fonction des ressources disponibles.

#### 3.3.1.4.3 Influence de la fréquence des reconfigurations

Trois intervalles ont été testés : 1 minute (inférieure à la durée d'une phase), 4 minutes (équivalente) et 6 minutes (supérieure). Un intervalle trop court entraîne des reconfigurations redondantes, tandis qu'un intervalle trop long réduit la réactivité. Un réglage adéquat permet de maintenir une bonne adaptabilité face aux variations de trafic sans excès de reconfigurations.

#### 3.3.1.4.4 Compromis entre stabilité et performance avec introduction d'une pénalité

Sous une contrainte stricte, trois configurations ont été évaluées : sans pénalité, avec pénalité modérée et avec pénalité élevée. Sans pénalité, le système réalise de nombreuses reconfigurations redondantes, entraînant des retards ou des pertes ponctuelles. L'introduction d'une pénalité

modérée réduit ces reconfigurations, tout en maintenant une bonne réactivité, ce qui stabilise le système sans impact significatif sur les performances. En revanche, une pénalité trop élevée limite fortement les reconfigurations, mais accroît la latence moyenne, réduisant la réactivité.

### 3.3.2 Limites de l'étude

Cette section présente les principales limites de notre travail, susceptibles d'influencer l'interprétation et la validité des résultats obtenus.

#### 3.3.2.1 Environnement de test

Bien que le système ait été implémenté et exécuté de manière distribuée sur une plateforme de test en conditions réelles, l'environnement reste contrôlé et ne reflète pas totalement les conditions d'un déploiement en production. En particulier :

- Les ressources matérielles (processeur, mémoire, bande passante) sont partagées sur un même hôte, ce qui peut affecter les performances mesurées et ne permet pas de refléter pleinement les contraintes d'un environnement distribué et hétérogène.
- Les paramètres de latence et de variabilité réseau sont simulés et ne capturent pas l'ensemble des aléas et complexités rencontrés dans un réseau réel à grande échelle (pannes, variations imprévues, congestions multiniveaux, etc.).

Par ailleurs, l'utilisation de la bande passante par le réseau overlay est limitée à des valeurs relativement restreintes (par exemple 3 Mb/s). Cette contrainte constitue une limite du banc d'essai. Toutefois, l'analyse repose moins sur les valeurs absolues que sur les tendances observées et sur la comparaison relative entre les différentes configurations de référence.

#### 3.3.2.2 Tests de mise en échelle

Les tests de montée en charge ont été restreints par des contraintes techniques propres à l'infrastructure expérimentale. En particulier, la pile réseau Docker s'appuie sur des *bridge networks* dont la stabilité se dégrade lorsque plus de 1000 conteneurs sont connectés à un même

réseau. Cette limitation, liée au noyau Linux, peut provoquer une instabilité des communications inter-conteneurs et entraîner des pertes ou interruptions de service, ce qui limite la portée et la validité des résultats pour le scénario de très grande échelle étudié.

#### 3.3.2.3 Variété des scénarios du modèle point à point avec file d'attente

Dans le cadre de cette étude, les expérimentations se sont concentrées sur des échanges reposant sur une seule file d'attente par expérience. L'impact d'une configuration intégrant plusieurs files simultanées n'a pas été exploré. Cette restriction réduit la représentativité des résultats, notamment dans un contexte de production où la concurrence entre files est un facteur déterminant pour les performances globales.

# CONCLUSION ET RECOMMANDATIONS

L'objectif principal de cette étude était de concevoir, implémenter et évaluer un système de messagerie en files d'attente géodistribué, capable de prendre en charge à la fois les modèles publication/abonnement et point à point, tout en étant adapté aux environnements *edge*. Pour le modèle point à point avec files d'attente, l'enjeu majeur consistait à réduire le temps d'attente dans la file, tandis que pour le modèle pub/sub, il s'agissait de diminuer la latence de transmission inter-sites tout en respectant leurs contraintes spécifiques.

Afin d'atteindre cet objectif, nous avons développé LuffyMQ, un système basé sur RabbitMQ, enrichi de modules supplémentaires implémentant les fonctionnalités requises. LuffyMQ reste totalement transparent et compatible avec les clients RabbitMQ. De plus, un orchestrateur a été conçu pour collecter des métriques et déterminer dynamiquement la configuration optimale des connexions entre les instances distribuées des échanges.

Les résultats obtenus valident les objectifs de recherche et démontrent la pertinence de l'approche proposée. LuffyMQ prend en charge de manière efficace les deux modèles, publication/abonnement et point à point avec files d'attente. Dans ce dernier cas, il surpasse les approches centralisées ainsi que la stratégie native FullMesh de RabbitMQ, avec une réduction de la latence moyenne pouvant atteindre 75 %. Pour le modèle pub/sub, les résultats montrent une latence comparable à celle du FullMesh qui constitue une borne inférieure tout en offrant une bien meilleure mise en échelle : LuffyMQ peut traiter jusqu'à 200 % (soit trois fois plus) de messages en situation de montée en charge. Comparé aux autres références (topologie en étoile et aléatoire), LuffyMQ présente à la fois une latence moyenne plus faible et une meilleure scalabilité.

L'analyse détaillée des décisions architecturales du système a mis en évidence que :

- la stratégie de conservation des messages assure une livraison complète, même si certains messages peuvent être retardés lors des phases de reconfiguration ;

- l'introduction d'une pénalité modérée réduit les reconfigurations redondantes sans gain significatif et contribue à stabiliser davantage le système ;
- le système tend à bien respecter les contraintes de bande passante et équilibre efficacement la charge en fonction des ressources disponibles ;
- un ajustement approprié de la fréquence de reconfiguration permet de conserver une bonne réactivité aux variations de trafic sans provoquer de reconfigurations excessives.

Comme perspectives, plusieurs pistes méritent d'être explorées. Il serait notamment pertinent d'enrichir le protocole expérimental en intégrant de nouvelles contraintes propres aux nœuds IoT telles que la consommation énergétique, l'utilisation du processeur et la mémoire, ainsi qu'en modélisant des latences intersites variables afin de mieux simuler les environnements réels. Concernant le modèle point à point avec file d'attente, un axe de recherche prometteur consisterait à identifier dynamiquement les conditions de rééquilibrage des files en fonction de leur charge effective. Par ailleurs, l'amélioration de l'algorithme de placement des échanges constitue une autre voie d'optimisation. Enfin, il serait intéressant d'évaluer les performances de LuffyMQ avec les différents protocoles supportés (AMQP 1.0, MQTT, STOMP), ainsi que d'étudier sa compatibilité avec le protocole de stream et les partitions de flux proposées par RabbitMQ.

## BIBLIOGRAPHIE


(2023a). Federation Plugin — RabbitMQ. Repéré le 2023-06-18 à https://www.rabbitmq.com/federation.html.

(2023b). RabbitMQ VS Apache Kafka - compare differences & reviews ? Repéré le 2023-05-10 à https://www.saashub.com/compare-apache-kafka-vs-rabbitmq.

(2025a). Address Federation. Repéré le 2025-02-05 à https://activemq.apache.org/components/artemis/documentation/latest/federation-address.html#address-federation.

(2025b). Federated Exchanges | RabbitMQ. Repéré le 2025-02-15 à https://www.rabbitmq.com/docs/federated-exchanges.

ActiveMQ. (2025). Messaging Concepts. Repéré le 2025-01-30 à https://activemq.apache.org/components/artemis/documentation/latest/messaging-concepts.html.

Akhlaqi, M. Y. & Mohd Hanapi, Z. B. (2023). Task offloading paradigm in mobile edge computing-current issues, adopted approaches, and future directions. *Journal of Network and Computer Applications*, 212, 103568. doi : 10.1016/j.jnca.2022.103568.

Al-Masri, E., Kalyanam, K. R., Batts, J., Kim, J., Singh, S., Vo, T. & Yan, C. (2020). Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access*, 8, 94880–94911. doi : 10.1109/ACCESS.2020.2993363. Conference Name : IEEE Access.

Amazon MQ. (2025). What is Amazon MQ ? - Amazon MQ. Repéré le 2025-06-25 à https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/welcome.html.

AMQP. (2025). AMQP Version 1.0 | AMQP. Repéré le 2025-05-23 à https://www.amqp.org/resources/specifications.

Azure, I. E. M. (2023). IoT Edge | Intelligence Cloud | Microsoft Azure. Repéré le 2023-04-16 à https://azure.microsoft.com/fr-fr/products/iot-edge.

Bagaskara, A. E., Setyorini, S. & Wardana, A. A. (2020). Performance Analysis of Message Broker for Communication in Fog Computing. *2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pp. 98–103. doi : 10.1109/ICITEE49829.2020.9271733.

Banno, R., Sun, J., Takeuchi, S. & Shudo, K. (2019). Interworking Layer of Distributed MQTT Brokers. E102.D(12), 2281–2294. doi : 10.1587/transinf.2019PAK0001.

Bonomi, F., Milito, R., Zhu, J. & Addepalli, S. (2012). Fog computing and its role in the internet of things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, (MCC '12), 13–16. doi : 10.1145/2342509.2342513.

Bouallegue, C. & Gascon-Samson, J. (2020). DynPubSub : A Peer To Peer Overlay For Topic-Based Pub/Sub Systems Deployed at the Edge. *Proceedings of the 21st International Middleware Conference Demos and Posters*, (Middleware '20 Demos and Posters), 7–8. doi : 10.1145/3429358.3429373.

Bouallegue, C. & Gascon-Samson, J. (2022). EdgePub : A Self-Adaptable Distributed MQTT Broker Overlay for the Far-Edge. *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 1–8. doi : 10.1109/FMEC57183.2022.10062858.

Chang, H., Hao, F., Kodialam, M., Lakshman, T. V., Mukherjee, S. & Varvello, M. (2022). SNAPS : Seamless Network-Assisted Publish-Subscribe. *2022 IFIP Networking Conference (IFIP Networking)*, pp. 1–9. doi : 10.23919/IFIPNetworking55013.2022.9829774.

Chang, H., Hao, F., Kodialam, M., Lakshman, T. V., Mukherjee, S. & Varvello, M. (2023). Towards network-assisted publish–subscribe over wide area networks. *Computer Networks*, 231, 109702. doi : 10.1016/j.comnet.2023.109702.

Chen, C., Jacobsen, H.-A. & Vitenberg, R. (2016). Algorithms based on divide and conquer for topic-based publish/subscribe overlay design. *IEEE/ACM Trans. Netw.*, 24(1), 422–436. doi : 10.1109/TNET.2014.2369346.

Chen, H., Qin, W. & Wang, L. (2022). Task partitioning and offloading in IoT cloud-edge collaborative computing framework : a survey. 11(1), 86. doi : 10.1186/s13677-022-00365-8.

Chen, X. [original-date : 2016-06-09T19 :42 :32Z]. (2025). A curated list of AMQP 1.0 resources. Unless explicitly stated, AMQP in this list refers to AMQP 1.0. Repéré le 2025-07-20 à https://github.com/xinchen10/awesome-amqp.

Dedousis, D., Zacheilas, N. & Kalogeraki, V. (2018). On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 76–86. doi : 10.1109/ICDCS.2018.00018.

Detti, A., Funari, L. & Blefari-Melazzi, N. (2020). Sub-Linear Scalability of MQTT Clusters in Topic-Based Publish-Subscribe Applications. *IEEE Transactions on Network and Service Management*, 17(3), 1954–1968. doi : 10.1109/TNSM.2020.3003535. Conference Name : IEEE Transactions on Network and Service Management.

Dobbelaere, P. & Sheykh Esmaili, K. (2017). Kafka versus RabbitMQ : A comparative study of two industry reference publish/subscribe implementations : Industry Paper. pp. 227–238. doi : 10.1145/3093742.3093908.

Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C. & Hu, B. (2015). Everything as a Service (XaaS) on the Cloud : Origins, Current and Future Trends. *2015 IEEE 8th International Conference on Cloud Computing*, pp. 621–628. doi : 10.1109/CLOUD.2015.88.

Escaleira, P., Cunha, V. A., Gomes, D., Barraca, J. P. & Aguiar, R. L. (2023). Moving Target Defense for the cloud/edge Telco environments. 24, 100916. doi : 10.1016/j.iot.2023.100916.

Eugster, P. T., Felber, P. A., Guerraoui, R. & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2), 114–131. doi : 10.1145/857076.857078.

Fidler, E., Jacobsen, H. A., Li, G. & Mankovski, S. (2005). The padres Distributed publish/subscribe system : 8th International Workshop on Feature Interactions in Telecommunications and Software Systems, ICFI 2005. 12–30. Repéré à http://www.scopus.com/inward/record.url?scp=84881590257&partnerID=8YFLogxK.

Fu, G., Zhang, Y. & Yu, G. (2021). A Fair Comparison of Message Queuing Systems. *IEEE Access*, 9, 421–432. doi : 10.1109/ACCESS.2020.3046503. Conference Name : IEEE Access.

Gascon-Samson, J., Garcia, F.-P., Kemme, B. & Kienzle, J. (2015). Dynamoth : A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. *2015 IEEE 35th International Conference on Distributed Computing Systems*, pp. 486–496. doi : 10.1109/ICDCS.2015.56.

Gascon-Samson, J., Kienzle, J. & Kemme, B. (2017). MultiPub : Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2075–2082. doi : 10.1109/ICDCS.2017.203.

Gauthier, J.-F. D. (2010). *A message oriented middleware for mobility*. (Thèse de doctorat, McGill University, Montreal).

Giannakopoulos, T. & Kalogeraki, V. (2022). An Elastic and Scalable Topic-Based Pub/Sub System Using Deep Reinforcement Learning. *Distributed Applications and Interoperable Systems*, pp. 167–183. doi : 10.1007/978-3-031-16092-9_11.

Google Trends. (2025). Google Trends. Repéré le 2025-08-05 à https://trends.google.com/trends?geo=CA&hl=en-US.

Greengrass, A. I. (2023). Manage data streams on Greengrass core devices - AWS IoT Greengrass. Repéré le 2023-04-15 à https://docs.aws.amazon.com/greengrass/v2/developerguide/manage-data-streams.html.

Hanon, W. & Salman, M. A. (2022). Review the deployment and role of broker in IoT platforms. *2022 5th International Conference on Engineering Technology and its Applications (IICETA)*, pp. 308–315. doi : 10.1109/IICETA54559.2022.9888675.

He, T., Khamfroush, H., Wang, S., La Porta, T. & Stein, S. (2018). It's Hard to Share : Joint Service Placement and Request Scheduling in Edge Clouds with Sharable and Non-Sharable Resources. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 365–375. doi : 10.1109/ICDCS.2018.00044.

Hmissi, F. & Ouni, S. (2022). TD-MQTT : Transparent Distributed MQTT Brokers for Horizontal IoT Applications. *2022 IEEE 9th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)*, pp. 479–486. doi : 10.1109/SETIT54465.2022.9875881.

Hoang, H., Cassell, B., Brecht, T. & Al-Kiswany, S. (2020). RocketBufs : a framework for building efficient, in-memory, message-oriented middleware. *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pp. 121–132. doi : 10.1145/3401025.3401744.

Hong, C.-H. & Varghese, B. (2020). Resource Management in Fog/Edge Computing : A Survey on Architectures, Infrastructure, and Algorithms. *ACM Computing Surveys*, 52(5), 1–37. doi : 10.1145/3326066.

Hong, L., Deng, L., Li, D. & Wang, H. H. (2021). Retracted : Artificial intelligence point-to-point signal communication network optimization based on ubiquitous clouds. *International Journal of Communication Systems*, 34(6), e4507. doi : 10.1002/dac.4507. _eprint : https ://onlinelibrary.wiley.com/doi/pdf/10.1002/dac.4507.

Islam, A., Debnath, A., Ghose, M. & Chakraborty, S. (2021). A Survey on Task Offloading in Multi-access Edge Computing. 118, 102225. doi : 10.1016/j.sysarc.2021.102225. Publisher : Elsevier BV.

Jaeger, P. T., ,Jimmy, L., & Grimes, J. M. (2008). Cloud Computing and Information Policy : Computing in a Policy Cloud ? 5(3), 269–283. doi : 10.1080/19331680802425479. Publisher : Routledge _eprint : https ://doi.org/10.1080/19331680802425479.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. & Lewin, D. (1997). Consistent hashing and random trees : distributed caching protocols for relieving hot spots on the World Wide Web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, (STOC '97), 654–663. doi : 10.1145/258533.258660.

Kawaguchi, R. & Bandai, M. (2020). Edge Based MQTT Broker Architecture for Geographical IoT Applications. *2020 International Conference on Information Networking (ICOIN)*, pp. 232–235. doi : 10.1109/ICOIN48656.2020.9016528.

Khan, F. A. (2022). Diving into Linux Networking and Docker — Bridge, vETH and IPTables. Repéré le 2025-06-13 à https://medium.com/techlog/diving-into-linux-networking-and-docker-bridge-veth-and-iptables-a05eb27b1e72.

Koziolek, H., Grüner, S. & Rückert, J. (2020). A Comparison of MQTT Brokers for Distributed IoT Edge Computing. *Software Architecture : 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14â€“18, 2020, Proceedings*, pp. 352–368. doi : 10.1007/978-3-030-58923-3_23.

Lei, J. & Liu, J. (2024). Reinforcement learning-based load balancing for heavy traffic Internet of Things. 99, 101891. doi : 10.1016/j.pmcj.2024.101891. Publisher : Elsevier BV.

Li, S., Zhang, N., Lin, S., Kong, L., Katangur, A., Khan, M. K., Ni, M. & Zhu, G. (2018). Joint Admission Control and Resource Allocation in Edge Computing for Internet of Things. *IEEE Network*, 32(1), 72–79. doi : 10.1109/MNET.2018.1700163. Conference Name : IEEE Network.

Longo, E. & Redondi, A. E. C. (2023). Design and implementation of an advanced MQTT broker for distributed pub/sub scenarios. *Computer Networks*, 224, 109601. doi : 10.1016/j.comnet.2023.109601.

Longo, E., Redondi, A. E., Cesana, M., Arcia-Moret, A. & Manzoni, P. (2020). MQTT-ST : a Spanning Tree Protocol for Distributed MQTT Brokers. *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pp. 1–6. doi : 10.1109/ICC40277.2020.9149046.

Longo, E., Redondi, A. E. C., Cesana, M. & Manzoni, P. (2022). BORDER : A Benchmarking Framework for Distributed MQTT Brokers. *IEEE Internet of Things Journal*, 9(18), 17728–17740. doi : 10.1109/JIOT.2022.3155872. Conference Name : IEEE Internet of Things Journal.

Luzuriaga, J., Perez, M., Boronat, P., Cano, J.-C., Calafate, C. & Manzoni, P. (2014). Testing AMQP Protocol on Unstable and Mobile Networks. 8729. doi : 10.1007/978-3-319-11692-1_22.

Mell, P. & Grance, T. *(2011)*. The NIST Definition of Cloud Computing. National Institute of Standards and Technology. Repéré le 2025-05-30 à https://csrc.nist.gov/pubs/sp/800/145/final.

Mortazavi, S. H., Salehe, M., Gomes, C. S., Phillips, C. & de Lara, E. (2017). Cloudpath : a multi-tier cloud computing framework. *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pp. 1–13. doi : 10.1145/3132211.3134464.

Naha, R. K., Garg, S., Georgakopoulos, D., Jayaraman, P. P., Gao, L., Xiang, Y. & Ranjan, R. (2018). Fog Computing : Survey of Trends, Architectures, Requirements, and Research Directions. 6, 47980–48009. doi : 10.1109/ACCESS.2018.2866491.

Nast, M., Raddatz, H., Rother, B., Golatowski, F. & Timmermann, D. (2023). A Survey and Comparison of Publish/Subscribe Protocols for the Industrial Internet of Things (IIoT). *Proceedings of the 12th International Conference on the Internet of Things*, (IoT '22), 193–200. doi : 10.1145/3567445.3571107.

Ning, Z., Dong, P., Kong, X. & Xia, F. (2019). A Cooperative Partial Computation Offloading Scheme for Mobile Edge Computing Enabled Internet of Things. *IEEE Internet of Things Journal*, 6(3), 4804–4814. doi : 10.1109/JIOT.2018.2868616. Conference Name : IEEE Internet of Things Journal.

Park, J.-H., Kim, H.-S. & Kim, W.-T. (2018). DM-MQTT : An Efficient MQTT Based on SDN Multicast for Massive IoT Communications. 18(9), 3071. doi : 10.3390/s18093071. Number : 9 Publisher : Multidisciplinary Digital Publishing Institute.

PerfTest. (2025). RabbitMQ PerfTest. Repéré le 2025-06-15 à https://perftest.rabbitmq.com/.

RabbitMQ. (2020a). Production Deployment Guidelines | RabbitMQ [RabbitMQ]. Repéré le 2025-09-23 à https://www.rabbitmq.com/docs/production-checklist.

RabbitMQ. (2020b). testing-docusaurus/docs/concepts/streaming-across-clusters.md at master · rabbitmq/testing-docusaurus. Repéré le 2025-09-02 à https://github.com/rabbitmq/testing-docusaurus/blob/master/docs/concepts/streaming-across-clusters.md.

RabbitMQ. (2023). AMQP 0-9-1 Model Explained â€" RabbitMQ. Repéré le 2023-04-19 à https://www.rabbitmq.com/tutorials/amqp-concepts.html.

RabbitMQ. (2025). cluster-operator/observability/prometheus/rule-file.yml at main · rabbitmq/cluster-operator. Repéré le 2025-07-03 à https://github.com/rabbitmq/cluster-operator/blob/main/observability/prometheus/rule-file.yml.

Rahmani, S. (2023). MQTT2EdgePeer : a robust and scalable peer-to-peer edge communication infrastructure for topic-based publish/subscribe [masters]. École de technologie supérieure. Repéré le 2024-02-08 à https://espace.etsmtl.ca/id/eprint/3294/.

Rausch, T., Dustdar, S. & Ranjan, R. (2018a). Osmotic Message-Oriented Middleware for the Internet of Things. *IEEE Cloud Computing*, 5(2), 17–25. doi : 10.1109/MCC.2018.022171663. Conference Name : IEEE Cloud Computing.

Rausch, T., Nastic, S. & Dustdar, S. (2018b). EMMA : Distributed QoS-Aware MQTT Middleware for Edge Computing Applications. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 191–197. doi : 10.1109/IC2E.2018.00043.

Redondi, A. E. C., Arcia-Moret, A. & Manzoni, P. (2019). Towards a Scaled IoT Pub/Sub Architecture for 5G Networks : the Case of Multiaccess Edge Computing. *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 436–441. doi : 10.1109/WF-IoT.2019.8767268.

Salehi, P., Zhang, K. & Jacobsen, H.-A. (2017). PopSub : Improving Resource Utilization in Distributed Content-based Publish/Subscribe Systems. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, (DEBS '17), 88–99. doi : 10.1145/3093742.3093915.

Santos, B., Soares, A., Nguyen, T.-A., Min, D.-K., Lee, J.-W. & Silva, F.-A. (2021). IoT Sensor Networks in Smart Buildings : A Performance Assessment Using Queuing Models. 21(16), 5660. doi : 10.3390/s21165660. Publisher : MDPI AG.

Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1), 30–39. doi : 10.1109/MC.2017.9.

Shi, W. & Dustdar, S. (2016). The Promise of Edge Computing. *Computer*, 49(5), 78–81. doi : 10.1109/MC.2016.145. Conference Name : Computer.

Subramoni, H., Marsh, G., Narravula, S., Lai, P. & Panda, D. K. (2008). Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand. *2008 Workshop on High Performance Computational Finance*, pp. 1–8. doi : 10.1109/WHPCF.2008.4745404.

Telecommunication Standardization Sector (ITU-T). (2012). *Recommendation ITU-T Y.2060 : Overview of the Internet of Things* (Rapport n° Y.2060). Repéré à https://www.itu.int/rec/T-REC-Y.2060-201206-I.

Teranishi, Y., Banno, R. & Akiyama, T. (2015). Scalable and Locality-Aware Distributed Topic-Based Pub/Sub Messaging for IoT. *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7. doi : 10.1109/GLOCOM.2015.7417305.

Toyohara, T. & Nishi, H. (2023). Distributed MQTT Brokers Infrastructure with Network Transparent Hardware Broker. *2023 Eleventh International Symposium on Computing and Networking (CANDAR)*, pp. 182–188. doi : 10.1109/CANDAR60563.2023.00032.

WonderNetwork. (2024). Global Ping Statistics. Repéré le 2025-06-20 à https://wondernetwork.com/pings.

Wu, G., Li, E. & Wei, T. (2023). Multimaster Node Byzantine Fault-Tolerant Consensus Algorithm Based on Consistent Hash Algorithm. *Computer*, 56(11), 48–63. doi : 10.1109/MC.2023.3255305. Conference Name : Computer.

Xie, Z., Ji, C., Xu, L., Xia, M. & Cao, H. (2023). Towards an Optimized Distributed Message Queue System for AIoT Edge Computing : A Reinforcement Learning Approach. 23(12), 5447. doi : 10.3390/s23125447. Publisher : MDPI AG.

Yi, S., Li, C. & Li, Q. (2015). A Survey of Fog Computing : Concepts, Applications and Issues. *Proceedings of the 2015 Workshop on Mobile Big Data*, (Mobidata '15), 37–42. doi : 10.1145/2757384.2757397.

Yoshino, D., Watanobe, Y. & Naruse, K. (2021). A Highly Reliable Communication System for Internet of Robotic Things and Implementation in RT-Middleware With AMQP Communication Interfaces. *IEEE Access*, 9, 167229–167241. doi : 10.1109/ACCESS.2021.3136855. Conference Name : IEEE Access.