Interaction entre KubeEdge et COOJA/Contiki

Encadrants : **Naceur MALOUCH, Firas ZAAROURI**

Etudiants : **Karim MEDJDOUB, Yorgo YOUNES, Ali EL BOUKILI, Ndeye Awa DIEME**

Table des matières

Table des figures

Liste des tableaux

Liste des acronymes

API	Application Programming Interface
CNCF	Cloud Native Computing Foundation
CoAP	Constrained Application Protocol
COOJA	Simulateur de réseaux IoT pour Contiki-NG
DMI	Device Management Interface
gRPC	Google Remote Procedure Call
HPA	Horizontal Pod Autoscaler
IoT	Internet of Things
K8s	Kubernetes
M2M	Machine-to-Machine
MQTT	Message Queuing Telemetry Transport
RPL	Routing Protocol for Low-Power and Lossy Networks
RTT	Round-Trip Time
VM	Virtual Machine

1 Cahier des charges

1.1 Introduction et Contexte

Aujourd'hui, nous sommes entourés d'objets connectés : montres intelligentes, capteurs urbains, caméras de surveillance, thermostats connectés, etc. Ces appareils collectent et envoient des données en permanence. Lorsque l'ensemble de ces données est envoyé vers des serveurs distants, souvent centralisés dans une même infrastructure Cloud, plusieurs problématiques apparaissent.

Premièrement, la latence induite par les communications avec des serveurs distants peut devenir importante pour certaines applications nécessitant une forte réactivité. Deuxièmement, le transfert massif de données peut entraîner une saturation des liens réseau reliant les infrastructures Cloud. Enfin, cette approche crée une forte dépendance vis-à-vis des infrastructures centralisées : en cas de perte de connectivité avec le Cloud, les objets connectés ne peuvent plus accéder aux services ou aux traitements distants.

Pour répondre à ces problématiques, le paradigme de l'*Edge Computing*, qui signifie littéralement « informatique en bordure » consiste à rapprocher les traitements au plus près des objets connectés, en déployant des capacités de calcul et de stockage à la périphérie du réseau. Cette approche permet de réduire la latence, de limiter les transferts de données vers le Cloud et d'améliorer la résilience du système.

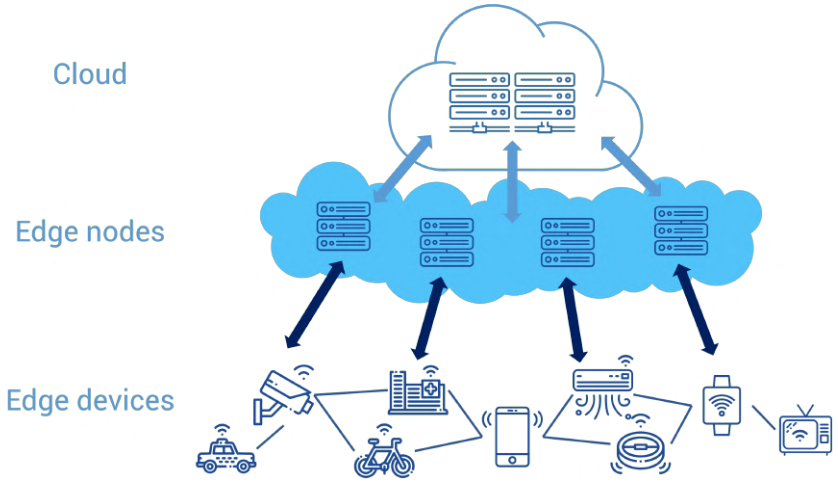

Figure 1 : Illustration simplifiée d'une architecture Edge Computing [1]

Dans ce contexte, il devient essentiel de disposer de solutions capables de gérer et d'orchestrer des applications distribuées entre le Cloud et la périphérie du réseau. Parallèlement, le développement de ces systèmes nécessite des environnements de test adaptés, permettant de simuler le comportement d'objets connectés sans recourir à du matériel physique.

Ce projet consiste à faire communiquer deux technologies importantes :

KubeEdge est une plateforme qui permet de gérer et déployer des applications à la fois dans le Cloud et à proximité des objets connectés, directement sur des machines situées en périphérie du réseau. Elle facilite ainsi la gestion d'applications distribuées en assurant la coordination entre une partie centrale (le Cloud) et des nœuds locaux plus proches des dispositifs IoT (Internet of Things). Dans ce type d'environnement, la communication avec les objets connectés repose généralement sur des protocoles légers adaptés à l'IoT, tels que *MQTT* et *CoAP* qui seront détaillés ultérieurement.

COOJA/Contiki-NG, quant à lui, est un environnement d'émulation permettant de tester des objets connectés sans recourir à du matériel physique réel. Les dispositifs IoT sont émulés au sein de COOJA, tandis qu'ils exécutent un système d'exploitation réel, à savoir Contiki-NG, ainsi que de véritables applications embarquées.

Faire interagir ces deux technologies permet de reproduire différents scénarios IoT dans un environnement de test réaliste et contrôlé.

1.2 Motivations et Problématiques

Le développement des architectures IoT combinées à l'Edge Computing représente aujourd'hui un enjeu majeur, notamment pour les applications nécessitant une forte réactivité et une gestion distribuée des ressources. Ces systèmes permettent d'améliorer les performances globales tout en réduisant la dépendance aux infrastructures centralisées.

Cependant, leur mise en œuvre et leur validation restent complexes. En effet, tester des systèmes IoT dans des conditions réalistes nécessite généralement du matériel spécifique, souvent coûteux et difficile à déployer à grande échelle. Cela constitue un frein important pour le développement, l'expérimentation et la validation de nouvelles solutions.

Par ailleurs, bien qu'il existe des outils permettant de gérer des applications distribuées entre le Cloud et l'Edge, ainsi que des simulateurs capables de reproduire le comportement d'objets connectés, ces deux types d'environnements sont généralement utilisés de manière indépendante. Cette séparation limite la possibilité de tester des scénarios complets, intégrant à la fois la gestion des applications et le comportement des dispositifs IoT.

La problématique principale de ce projet est donc la suivante : comment intégrer un système de gestion d'applications distribuées avec un environnement de simulation d'objets connectés, afin de permettre la validation de scénarios IoT réalistes dans un cadre contrôlé ?

Afin de répondre à cette problématique, ce projet propose de mettre en place une interaction entre ces deux environnements, permettant ainsi de créer une plateforme de test cohérente, flexible et adaptée aux besoins du développement de solutions IoT.

1.3 Besoins

Pour la réalisation de ce projet, les besoins suivants sont essentiels :

1. **Infrastructure matérielle et logicielle :**
   - Trois machines virtuelles dédiées disposant de ressources suffisantes (CPU, RAM).
   - Une connectivité réseau entre les machines virtuelles permettant la communication entre les différentes composantes du système.
   - Une plateforme permettant la gestion d'applications distribuées.
   - Un environnement de simulation d'objets connectés.
2. **Compétences techniques :**
   - Maîtrise de l'administration système sous Linux (Ubuntu).
   - Compréhension des architectures distribuées Cloud/Edge.
   - Connaissance des principes de communication entre objets connectés notamment MQTT et CoAP.
   - Compétences en développement logiciel pour l'implémentation de composants spécifiques.
3. **Documentation :**
   - Un rapport détaillant les étapes d'installation, de configuration et de test.
   - Une documentation du code permettant de comprendre le fonctionnement des composants développés
   - Un guide de reproduction de l'environnement

1.4 Périmètre

Exigences fonctionnelles :

- Le système doit permettre la communication entre une plateforme de gestion d'applications distribuées et des objets connectés simulés.
- Il doit être possible d'échanger des données entre les dispositifs IoT simulés et le Cloud.
- Les états des objets connectés doivent être synchronisés entre les différentes parties du système.

- Les applications déployées doivent être visibles et contrôlables depuis une interface centrale.

Exigences non fonctionnelles :

- Performance : Le système doit assurer une communication avec une latence raisonnable.
- Scalabilité : Il doit supporter plusieurs objets connectés simultanément.
- Fiabilité : Il doit gérer les pertes de connexion et assurer la continuité du service.
- Maintenabilité : Le système doit être structuré et documenté.
- Portabilité : Il doit fonctionner sur des machines virtuelles standards.

1.5 Contraintes

Le projet doit respecter plusieurs contraintes techniques. L'architecture repose sur l'utilisation de plusieurs machines virtuelles distinctes afin de reproduire un environnement distribué proche d'un cas réel. Les communications entre les différentes composantes doivent être fiables et respecter des protocoles standards utilisés dans les systèmes IoT.

Par ailleurs, les ressources matérielles étant limitées, une attention particulière doit être portée à l'optimisation du système afin de garantir son bon fonctionnement. Enfin, l'ensemble des composants doit être configuré de manière à assurer une communication stable entre les différentes parties du système.

1.6 Objectifs et Résultats attendus

Le projet consiste à déployer un environnement complet d'Edge Computing basé sur KubeEdge et à l'intégrer avec l'émulateur COOJA/Contiki-NG. Les objectifs principaux sont les suivants :

Objectifs primaires :

- Mettre en place une plateforme de test basée sur plusieurs machines virtuelles.
- Déployer une infrastructure KubeEdge fonctionnelle sur deux VMs distinctes.
- Installer et configurer COOJA/Contiki-NG sur une VM dédiée.
- Permettre la communication entre ces deux environnements.
- Développer des composants permettant cette interaction.
- Mettre en place un mécanisme permettant de mesurer le temps de réponse entre les objets connectés simulés et les applications déployées, et exploiter cette métrique pour adapter dynamiquement les ressources.

- Compiler localement les outils utilisés afin de garantir la reproductibilité et le contrôle des versions.

Résultats attendus :

- Un environnement de test complet et fonctionnel permettant de simuler des scénarios IoT réalistes.
- Une communication effective entre les différentes parties du système.
- Une visualisation des états des objets connectés.
- Des tests avec plusieurs objets émulés.
- Une documentation complète.
- Des machines virtuelles préconfigurées qui hébergent tout l'environnement de tests.

2 Plan de développement

2.1 Phase 1 : Préparation et étude initiale

Cette phase vise à comprendre les concepts et technologies nécessaires au projet.

- Étude des principes de l'Edge Computing et des systèmes IoT.
- Prise en main de la plateforme KubeEdge.
- Étude de l'environnement de simulation COOJA/Contiki-NG.
- Identification des mécanismes nécessaires à la communication entre les différents environnements.

2.2 Phase 2 : Mise en place de l'environnement

Cette phase consiste à créer l'infrastructure de test permettant de déployer et connecter les différents composants du projet.

- Création des machines virtuelles nécessaires (Cloud, Edge, Simulation).
- Compilation et mise en place de l'infrastructure KubeEdge.
- Installation et configuration de l'environnement de simulation COOJA.
- Configuration du réseau permettant la communication entre les différentes machines.

2.3 Phase 3 : Développement des mécanismes d'interaction

Cette phase est dédiée à la mise en place des composants permettant la communication entre les environnements.

- Conception des mécanismes de communication entre KubeEdge et les objets simulés.
- Mise en place de la communication en s'appuyant sur les protocoles MQTT et CoAP.
- Développement des composants assurant l'échange et la transformation des données.
- Intégration de ces composants dans l'architecture globale du système.

2.4 Phase 4 : Mise en place des métriques et adaptation dynamique

Cette phase vise à améliorer le système en introduisant un mécanisme d'analyse des performances.

- Mise en place d'un mécanisme de mesure du temps de réponse entre les objets simulés et les applications.

- Transformation de ces mesures en métriques exploitables.
- Utilisation de ces métriques pour adapter dynamiquement les ressources du système.

2.5 Phase 5 : Tests avancés et validation

Cette phase permet de vérifier le bon fonctionnement du système dans différentes conditions.

- Tests de communication entre les différents composants.
- Validation du fonctionnement avec plusieurs objets simulés.
- Analyse des performances (temps de réponse, stabilité).
- Tests de robustesse face à des conditions réseau dégradées.

2.6 Phase 6 : Documentation et finalisation

Cette phase consiste à finaliser le projet et produire les livrables.

- Rédaction de la documentation technique.
- Documentation du code développé.
- Préparation des machines virtuelles livrables.
- Validation finale du système.

DIAGRAMME DE GANTT · Projet 12 : KubeEdge / COOJA · 9 fév. - 24 avr. 2026

Groupe de 4 membres | Karim · Awa · Ali · Yorgo

#	Tâche	Participants	Début	Fin
1	**Étude initiale**		09/02/26	01/03/26
1.a	Compréhension des enjeux	Tous	09/02/26	15/02/26
1.b	Documentation	Tous	16/02/26	25/02/26
1.c	Cahier des charges	Tous	26/02/26	01/03/26
2	**Mise en place de la plateforme**		23/02/26	15/03/26
2.a	Installation des VMs et configuration réseau	Karim	23/02/26	27/03/26
2.b	Mise en place du cluster K3s / KubeEdge	Karim	28/02/26	08/03/26
2.c	Installation des outils	Tous	09/03/26	15/03/26
3	**Communication Edge ↔ Devices**		09/03/26	24/04/26
3.a	Installation mapper framework	Awa /Ali	09/03/26	15/03/26
3.b	Developpement et tests des motes MQTT et CoAP	Karim	09/03/26	15/03/26
3.c	Développement du mapper MQTT	Awa	16/03/26	24/03/26
3.d	Développement du mapper CoAP	Ali	16/03/26	25/03/26
3.e	Test des communications mappers ↔ motes	Tous	26/03/26	04/04/26
3.f	Finalisation & conteneurisation des mappers	Tous	06/04/26	21/04/26
3.g	Tests de bout en bout	Tous	20/04/26	24/04/26
4	**Compilation locale**		23/03/26	13/04/26
4.a	Récupération Code source K3s / KubeEdge	Yorgo	23/03/26	28/03/26
4.b	Recompilation K3s et EdgeCore	Yorgo / Ali	29/03/26	04/04/26
4.c	Reconstruction image CloudCore	Yorgo / Ali	03/04/26	07/04/26
4.d	Remplacement binaires/images	Yorgo / Ali	08/04/26	10/04/26
4.e	Tests binaires locaux	Tous	09/04/26	13/04/26
5	**Monitoring**		30/03/26	24/04/26
5.a	Modif. motes MQTT/CoAP (métrique RTT)	Karim	30/03/26	03/04/26
5.b	Adaptation des mappers (métrique RTT)	Karim	04/04/26	08/04/26
5.c	Développement & conteneurisation treatment-app	Karim	09/04/26	11/04/26
5.d	Déploiement treatment-app (Cloud/Edge)	Karim / Yorgo	12/04/26	12/04/26
5.e	Tests de la treatment-app	Karim / Yorgo	13/04/26	24/04/26
5.f	Installation Prometheus	Awa	13/04/26	13/04/26
5.g	Développement metrics-exporter	Awa	13/04/26	14/04/26
5.h	Adaptateur Prometheus et HPA	Awa	15/04/26	15/04/26
5.i	Tests d'auto-scaling	Tous	17/04/26	24/04/26
6	**Tests finaux**		24/04/26	24/04/26
6.a	Tests pipeline complet	Tous	24/04/26	24/04/26

Figure 2 : Diagramme de Gantt

3 Analyse

Cette section présente les principales technologies et mécanismes utilisés dans le cadre du projet. Elle introduit notamment l'architecture de KubeEdge, l'environnement d'émulation COOJA/Contiki-NG, les protocoles de communication utilisés ainsi que les mécanismes d'échange de données entre les différents composants du système.

L'objectif de cette analyse est d'identifier les éléments techniques nécessaires à la conception et à l'intégration de la plateforme. À l'issue de cette section, les principaux mécanismes intervenant dans le fonctionnement du système auront été présentés et analysés.

3.1 Architecture de KubeEdge

Kubernetes, souvent abrégé en *K8s*, est un moteur d'orchestration de conteneurs open source permettant d'automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Ce projet open source est hébergé par la Cloud Native Computing Foundation (CNCF) [2].

KubeEdge est un framework open source pour l'Edge Computing, basé sur Kubernetes et actuellement en phase d'incubation au sein de la CNCF [3]. KubeEdge étend K8s aux environnements Edge et permet l'orchestration d'applications sur des appareils et serveurs proches de l'utilisateur, appelés *Edge nodes*, tout en maintenant la cohérence et la synchronisation avec le cluster central, c'est-à-dire le groupe de machines principal de l'infrastructure, appelé le *Cloud*. Il peut communiquer avec les objets connectés à l'aide de composants capables de traduire différents protocoles en données compréhensibles par le système.

Pour gérer ces environnements Edge, KubeEdge propose différents composants permettant de gérer les aspects essentiels d'une architecture Cloud/Edge comme :

- **CloudCore** : Il s'agit de l'ensemble des composants exécutés dans le Cloud. Ils jouent le rôle de plan de contrôle et sont responsables de l'orchestration globale du système. Le CloudCore assure notamment la planification des applications sur les nœuds Edge ainsi que la synchronisation des données entre le Cloud et l'Edge.
- **EdgeCore** : Ce composant est déployé sur les nœuds Edge. Il agit comme un nœud de travail chargé d'exécuter localement les applications. Il permet également de collecter et traiter les données provenant des objets connectés, tout en assurant la communication et la synchronisation avec le CloudCore.
- **Mappers** : Ce sont des composants essentiels permettant de faire le lien entre KubeEdge et les dispositifs IoT. Leur rôle est de traduire les données échangées entre les objets connectés et la plateforme, en adaptant les différents protocoles de communication utilisés par les appareils en un format exploitable par le système.

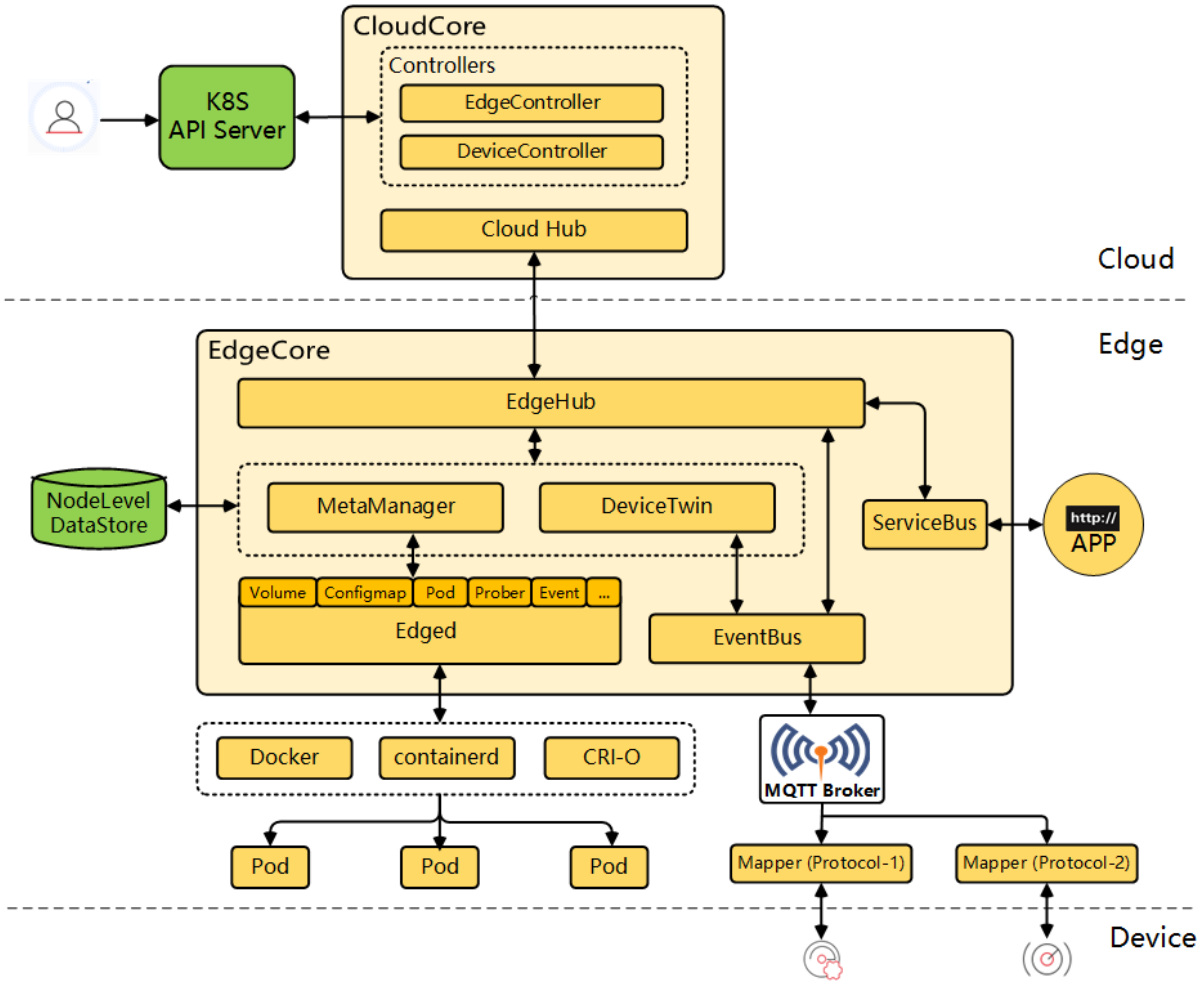

Figure 3 : Architecture d'un cluster KubeEdge [4]

Au sein de ces composants principaux, KubeEdge s'appuie également sur plusieurs sous-composants assurant des fonctions spécifiques. Parmi les plus importants, **CloudHub** et **EdgeHub** permettent d'assurer la communication entre le Cloud et les nœuds Edge en maintenant un canal d'échange bidirectionnel. Le composant **DeviceTwin** joue un rôle clé dans la gestion des objets connectés en maintenant une représentation virtuelle de leur état, synchronisée entre le Cloud et l'Edge. Enfin, d'autres modules internes participent à la gestion des données et à l'exécution des applications, contribuant ainsi au bon fonctionnement global du système.

3.2 Émulation IoT avec Contiki/COOJA

Contiki-NG est un système d'exploitation pour les dispositifs IoT à ressources limitées. Il intègre une pile de communication IPv6 basse consommation conforme aux RFC, permettant la connectivité Internet [5].

Contiki propose un simulateur de réseau appelé *COOJA*. Ce simulateur permet l'émulation de différents capteurs sur lesquels seront chargés un système d'exploitation et des applications. COOJA permet ensuite de simuler les connexions réseaux et d'interagir avec les capteurs. Cet outil permet aux développeurs de tester les applications à moindre coût [6].

Grâce à cet environnement, il est possible d'émuler des objets connectés appelés *motes*. Ces motes représentent des dispositifs IoT réels et peuvent exécuter des programmes, échanger des données et interagir au sein d'un réseau simulé.

Dans le cadre de ce projet, COOJA joue un rôle essentiel en fournissant une source de données simulées, permettant de tester l'interaction entre les objets connectés et l'infrastructure distribuée mise en place.

Distinction entre simulation et émulation
Il est important de distinguer les notions de simulation et d'émulation dans le fonctionnement de COOJA/Contiki-NG. COOJA agit comme un simulateur pour les aspects liés au réseau et à l'environnement, en reproduisant notamment les communications radio, les délais ou encore certaines interactions physiques entre les dispositifs.
En revanche, les motes exécutés dans COOJA sont émulés plutôt que simplement simulés. Cela signifie que les dispositifs virtuels exécutent un véritable système d'exploitation embarqué (Contiki-NG) ainsi que du code réel compilé pour l'architecture matérielle ciblée. Cette approche permet d'obtenir un comportement beaucoup plus proche d'un déploiement réel qu'une simple simulation abstraite du comportement des dispositifs.

3.3 Protocoles de communication

La communication entre les différents composants du système repose sur des protocoles légers adaptés aux contraintes des environnements IoT. Dans le cadre de ce projet, deux protocoles ont été utilisés : *MQTT* et *CoAP*. Ces derniers répondent à des besoins différents et permettent de couvrir plusieurs types de communication entre les objets connectés, les mappers et les applications.

3.3.1 MQTT

MQTT (*Message Queuing Telemetry Transport*) est un protocole léger de messagerie basé sur le modèle publish/subscribe, conçu pour les environnements IoT et M2M (*Machine-to-Machine*) où les ressources matérielles et la bande passante réseau sont limitées [7].

Dans ce modèle, les dispositifs ne communiquent pas directement entre eux, mais passent par un intermédiaire appelé **broker**. Les objets connectés peuvent publier des données sur des "topics", tandis que d'autres composants peuvent s'abonner à ces topics pour recevoir les informations.

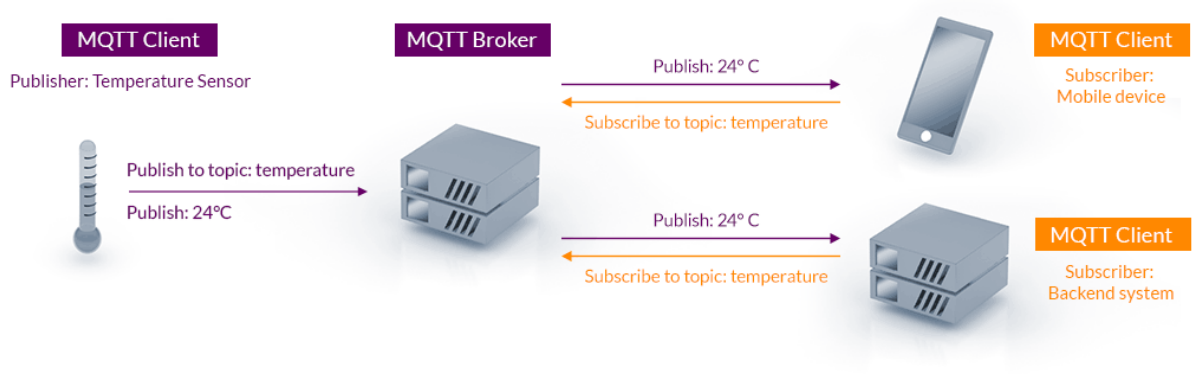

Figure 4 : Architecture Publish/Subscribe d'MQTT [8]

Ce mécanisme permet de découpler les émetteurs et les récepteurs, facilitant ainsi la gestion des communications dans des systèmes distribués. MQTT est particulièrement adapté aux environnements IoT en raison de sa faible consommation de bande passante et de sa simplicité.

3.3.2 CoAP

Le protocole CoAP (*Constrained Application Protocol*) est un protocole de transfert web spécialisé, conçu pour les nœuds et les réseaux aux ressources limitées. Ces nœuds sont souvent équipés de microcontrôleurs 8 bits avec de faibles quantités de mémoire ROM et RAM. Ce protocole est conçu pour les applications M2M, notamment pour la gestion intelligente de l'énergie et l'automatisation des bâtiments [9].

Il repose sur un modèle client/serveur, similaire à celui du protocole HTTP, mais allégé pour s'adapter aux contraintes des dispositifs IoT.

Dans ce modèle, un client peut envoyer des requêtes (par exemple GET, POST, PUT, DELETE) à un serveur pour accéder ou modifier des ressources. CoAP est particulièrement adapté aux communications directes entre dispositifs, notamment lorsque des interactions rapides et simples sont nécessaires.

3.4 Interaction entre les composants via les mappers

Afin de permettre la communication entre l'infrastructure KubeEdge et les objets connectés simulés dans COOJA, il est nécessaire de mettre en place un mécanisme d'interaction adapté. Dans ce projet, ce rôle est assuré par des composants appelés **mappers**.

Les mappers sont des composants intermédiaires qui assurent la communication entre les objets connectés et la plateforme KubeEdge. Leur rôle est de recevoir les données provenant des dispositifs IoT, de les transformer dans un format exploitable par l'infrastructure, puis de transmettre ces informations aux différents composants du système.

Ils permettent également d'envoyer des commandes depuis les applications vers les objets connectés, en effectuant la transformation inverse. Les mappers jouent ainsi un rôle essentiel en assurant la compatibilité entre les protocoles utilisés par les objets connectés et ceux utilisés par la plateforme.

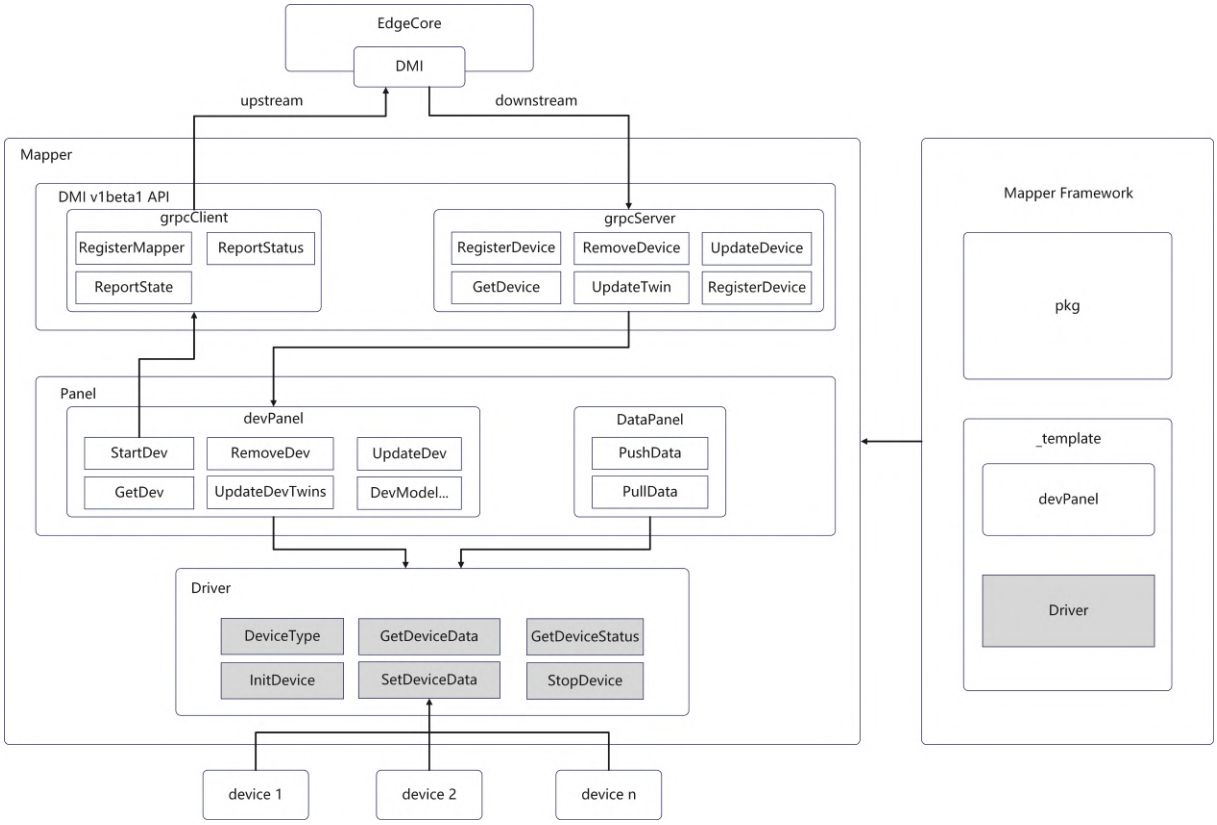

Figure 5 : Architecture d'un mapper KubeEdge [4]

EdgeCore et les mappers communiquent via gRPC à travers une interface standardisée côté EdgeCore appelée DMI (Device Management Interface). Cette interface permet aux mappers d'interagir avec le composant DeviceTwin, en envoyant et en recevant des informations relatives aux objets IoT. Le mapper est ensuite responsable de traduire ces données vers le protocole de communication utilisé par les dispositifs, tel que MQTT ou CoAP.

Afin de faciliter le développement de ces composants, KubeEdge propose un mapper framework qui prend en charge une grande partie de la logique interne nécessaire à l'intégration avec la plateforme et permet aux développeurs de se concentrer principalement sur l'implémentation du driver, c'est-à-dire la partie responsable de la communication avec les dispositifs IoT. Les aspects liés à l'interaction avec KubeEdge, notamment via la DMI, sont ainsi abstraits, ce qui simplifie le développement et améliore la modularité des mappers.

Les mappers proposent plusieurs fonctionnalités pour envoyer les données notamment le push HTTP qui permet d'envoyer les données à une application locale directement en HTTP sans passer par l'EdgeCore.

3.5 Analyse des flux

Le fonctionnement global du système repose sur des échanges de données entre les objets connectés émulés, les mappers et l'infrastructure KubeEdge. Ces échanges permettent d'assurer une communication bidirectionnelle entre les dispositifs IoT et les applications déployées.

Flux ascendant (objets → application) : correspond à la transmission des données générées par les objets connectés vers l'infrastructure.

1. Un mote génère une donnée (par exemple une mesure de capteur)
2. Cette donnée est envoyée via un protocole de communication adapté
3. Le mapper reçoit cette information et la transforme en un format exploitable par KubeEdge
4. Le mapper transmet ensuite la donnée à l'EdgeCore ou l'envoie directement à l'application qui tourne sur l'Edge
5. L'EdgeCore synchronise cette information avec le CloudCore
6. Une application sur le Cloud peut alors accéder à la donnée via kube-apiserver

Flux descendant (application → objets) : correspond à l'envoi de commandes depuis les applications vers les objets connectés.

Dans le cadre de ce projet, ce type de communication n'a pas été implémenté, l'objectif étant de collecter et traiter les données provenant des dispositifs IoT simulés. Toutefois, ce mécanisme est supporté par l'architecture et permet d'envoyer des commandes depuis les applications vers les objets connectés en passant par le mapper, qui assure la traduction vers le protocole utilisé.

Ce type de flux pourrait être utilisé dans des scénarios de contrôle ou d'automatisation des objets connectés.

3.6 Système de métriques et adaptation

Dans un système distribué combinant des objets connectés, des composants Edge et des composants Cloud, la latence constitue un indicateur important de performance. Il est donc pertinent de mesurer le temps de réponse du système afin d'évaluer son comportement dans différentes conditions.

Le temps de réponse correspond au délai entre l'émission d'une donnée par un objet connecté et la réception de la réponse associée. Cet indicateur permet de refléter l'ensemble des délais introduits par les différents composants du système, notamment les phases de transmission, de traitement et de synchronisation.

Afin de caractériser ces performances, il est possible d'exprimer ce temps comme la différence entre deux instants :

$$\Delta t = t_f - t_0$$

où t_0 représente le moment d'émission de la donnée et t_f le moment de réception de la réponse.

L'exploitation de cette mesure permet d'identifier les situations de surcharge ou de dégradation des performances. En effet, une augmentation du temps de réponse peut traduire une insuffisance de ressources ou une saturation du système.

Dans les architectures modernes, ces métriques peuvent être utilisées pour adapter dynamiquement les ressources allouées aux applications ou de déplacer le traitement entre le Cloud et l'Edge. Cette approche permet d'ajuster le comportement du système en fonction de sa charge réelle, en augmentant les ressources ou en rapprochant le traitement du Cloud lorsque cela est nécessaire.

4 Conception

Cette section présente les choix de conception réalisés dans le cadre du projet ainsi que l'architecture mise en place. Elle détaille notamment l'organisation des machines virtuelles, les mécanismes de communication entre les différents composants, la conception des mappers ainsi que le système de métriques mis en place pour l'adaptation dynamique des ressources.

L'objectif de cette section est de décrire la manière dont les différents éléments étudiés précédemment ont été intégrés afin de construire une plateforme cohérente, fonctionnelle et reproductible.

4.1 Architecture du système

L'architecture du projet repose sur trois machines virtuelles distinctes : **cloud-core-vm**, **edge-core-vm** et **cooja-vm**. Cette séparation permet de reproduire une architecture distribuée proche d'un environnement réel, avec une partie Cloud, une partie Edge et une partie dédiée à l'émulation des objets connectés.

La machine **cloud-core-vm** joue le rôle de nœud principal de l'infrastructure. Elle héberge le master Kubernetes et le composant CloudCore de KubeEdge.

La machine **edge-core-vm** représente le nœud Edge. Elle héberge EdgeCore et les mappers chargés de faire le lien avec les dispositifs IoT.

Enfin, la machine **cooja-vm** est dédiée à l'émulation des objets connectés à l'aide de COOJA/Contiki-NG. Elle permet de simuler les motes et de générer les données utilisées pour tester l'interaction avec l'infrastructure KubeEdge.

Les machines **cloud-core-vm** et **edge-core-vm** utilisent Ubuntu Server, tandis que **cooja-vm** utilise Ubuntu Desktop afin de faciliter l'utilisation de l'interface graphique de COOJA.

L'architecture réseau repose sur plusieurs réseaux distincts, chacun répondant à un besoin précis. Le réseau `192.168.201.0/29` assure la communication entre la machine Cloud et la machine Edge. Le réseau `192.168.202.0/29` a été prévu pour la communication entre l'Edge et COOJA, bien qu'il n'ait finalement pas été utilisé dans la solution finale.

En complément, deux réseaux IPv6 ont été mis en place. Le réseau `fc00::/64` relie la machine Edge et la machine COOJA. Il est nécessaire car les motes simulés utilisent IPv6. Le choix a été fait de conserver une communication IPv6 de bout en bout entre l'Edge et les motes, plutôt que de recourir à un mécanisme de traduction comme NAT64.

Le réseau `fe00::/64` relie quant à lui la machine Cloud et la machine COOJA. Il permet notamment à l'application de traitement, lorsqu'elle est déployée côté Cloud, de répondre directement aux requêtes CoAP émises par les motes.

Enfin, un réseau IPv6 interne `fd00::/64` est utilisé au sein de la simulation COOJA.

Ce réseau correspond au réseau RPL (IPv6 Routing Protocol for Low-power and Lossy Network) dans lequel évoluent les motes émulés. Il est relié à la machine hôte via l'RPL Border Router qui est connecté à l'interface `tun0`, qui permet de faire le lien entre le réseau simulé et l'infrastructure réelle.

Machine virtuelle	**OS**	**Rôle**
cloud-core-vm	Ubuntu Server	Master Kubernetes, CloudCore
edge-core-vm	Ubuntu Server	Worker Kubernetes, EdgeCore, mappers
cooja-vm	Ubuntu Desktop	COOJA/Contiki-NG et motes simulés

Table 1 : Rôle des machines virtuelles

Réseau	**Utilisation**
`192.168.201.0/29`	Communication entre cloud-core-vm et edge-core-vm
`192.168.202.0/29`	Réseau prévu entre edge-core-vm et cooja-vm, finalement non utilisé
`fc00::/64`	Communication IPv6 entre edge-core-vm et les motes COOJA
`fe00::/64`	Communication IPv6 entre cloud-core-vm et les motes COOJA
`fd00::/64`	Réseau RPL interne de la simulation COOJA

Table 2 : Réseaux utilisés dans l'architecture

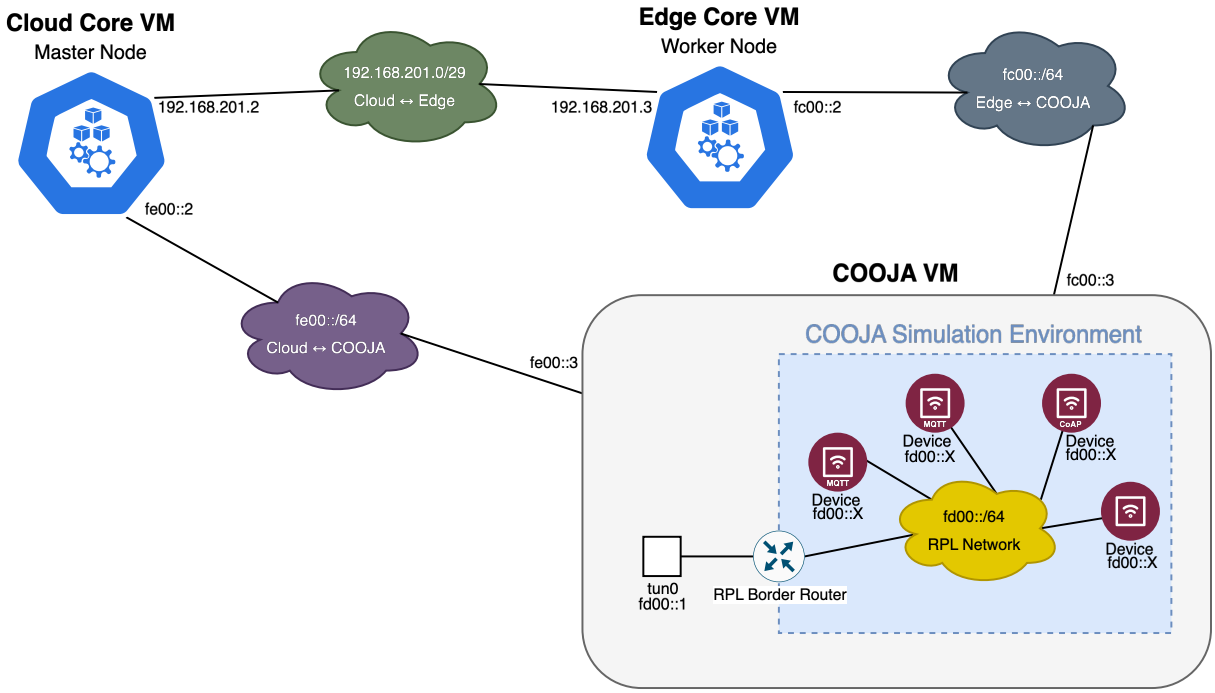

Figure 6 : Architecture de la plateforme de tests

4.2 Choix des technologies

Le principal choix technique a concerné la distribution Kubernetes utilisée pour déployer l'infrastructure. La solution **K3s** a été retenue en raison de sa légèreté et de sa simplicité de mise en œuvre, ce qui la rend particulièrement adaptée à un environnement basé sur des machines virtuelles avec des ressources limitées.

4.3 Conception des mappers

Dans le cadre de ce projet, les mappers ont été développés à partir du **mapper framework** fourni par KubeEdge. Ce framework, récupéré depuis le dépôt officiel du projet, permet de générer une structure de base facilitant le développement de nouveaux mappers.

Ce template fournit une architecture standard dans laquelle le développeur doit principalement implémenter un ensemble de méthodes appelées Driver qui définissent la manière dont le mapper contacte l'objet. `InitDevice`, `GetDeviceData`, `SetDeviceData`, `DeviceDataWrite`, `StopDevice` et `GetDeviceStates`.

Méthode	**Rôle**
`InitDevice`	Initialise la connexion avec le dispositif IoT.
`GetDeviceData`	Récupère les données exposée par le dispositif IoT pour les reporter vers l'infrastructure KubeEdge.
`DeviceDataWrite`	Permet l'écriture de données ou l'envoi de commandes vers le dispositif IoT.
`SetDeviceData`	Met à jour l'état des données du dispositif au sein du mapper.
`GetDeviceStates`	Récupère les états associés au dispositif IoT.
`StopDevice`	Assure l'arrêt propre des connexions et des mécanismes associés au dispositif.

Table 3 : Méthodes implémentées dans les drivers des mappers

Dans ce projet, ces méthodes ont été implémentées afin d'assurer la communication avec les objets connectés simulés, en tenant compte des spécificités de chaque protocole.

Pour le protocole MQTT, le mapper établit une connexion avec un broker central et s'abonne à des topics spécifiques associés à chaque mote. Les données reçues sont ensuite traitées et transmises à l'infrastructure KubeEdge.

Pour le protocole CoAP, le mapper communique directement avec les motes en utilisant des requêtes vers les endpoints exposés par ces derniers. Afin de permettre cette interaction, il a été nécessaire de modifier le code des motes simulés sous Contiki-NG, notamment en ajoutant des endpoints correspondant aux données à récupérer.

En complément de l'intégration avec KubeEdge, les mappers ont été conçus pour pouvoir transmettre directement les données reçues à une application via un mécanisme de push HTTP. Cette fonctionnalité permet de contourner certains délais liés à la synchronisation Cloud et d'obtenir un accès plus rapide aux données côté application.

Bien que le mapper framework soit conçu pour limiter les modifications à la seule implémentation du driver, certaines adaptations ont été nécessaires pour répondre aux besoins du projet. En particulier, le comportement par défaut du framework ne gérait pas correctement le cas où l'application cible du push HTTP n'était pas accessible, ce qui entraînait un arrêt du mapper. Ce scénario étant possible dans l'architecture retenue (notamment lorsque l'application est déployée côté Cloud), le code interne du mapper a été modifié afin de gérer cette situation de manière robuste.

De plus, le mécanisme de report utilisé pour remonter les données vers KubeEdge et le mécanisme de push HTTP reposaient initialement sur une même méthode `GetDeviceData`. Cette conception ne posait pas de problème particulier pour le protocole CoAP, dans lequel chaque appel à la méthode entraîne directement une nouvelle requête vers le dispositif IoT afin de récupérer la donnée courante.

En revanche, le fonctionnement du mapper MQTT reposait sur un mécanisme différent. Lorsqu'un message est reçu depuis le broker, la donnée est stockée localement par le mapper puis considérée comme une nouvelle donnée disponible. La méthode `GetDeviceData` récupère ensuite cette donnée locale afin de la transmettre à l'infrastructure KubeEdge, tout en évitant qu'une même donnée soit reportée plusieurs fois.

Dans cette architecture, l'utilisation simultanée de `GetDeviceData` pour le report vers KubeEdge et pour le mécanisme de push HTTP introduisait un conflit de consommation des données. Selon l'ordre d'exécution, une donnée pouvait être soit transmise au mécanisme de report, soit envoyée à l'application via le push HTTP, sans garantir que les deux traitements soient correctement effectués.

Afin de résoudre ce problème, une séparation des responsabilités a été introduite en distinguant `GetDeviceData`, dédiée au report vers KubeEdge, et `GetDeviceDataForPush`, utilisée exclusivement pour le mécanisme de push HTTP. Le framework a ensuite été modifié afin d'appeler la méthode appropriée en fonction du contexte d'exécution.

Cette évolution a permis d'améliorer la robustesse du système et de garantir un fonctionnement cohérent des différents flux de données au sein des mappers MQTT.

Ces adaptations ont permis de rendre les mappers plus robustes et mieux adaptés aux différents scénarios d'exécution du système.

4.4 Conception des flux

Le fonctionnement du système repose sur un flux de données montant, depuis les objets connectés simulés jusqu'aux applications.

Dans un premier temps, les motes émulés dans COOJA envoient leurs données aux mappers via les protocoles MQTT ou CoAP. Le mapper reçoit ces données et assure leur acheminement.

Une fois les données reçues, le mapper les transmet systématiquement à l'EdgeCore via la DMI, afin de les intégrer dans l'infrastructure KubeEdge. Ce mécanisme correspond au *report*, permettant de remonter l'état des dispositifs IoT vers la plateforme.

En parallèle, un mécanisme de **push HTTP** peut être activé pour certaines données. Dans ce cas, le mapper envoie directement les données à l'application. Ce mode est particulièrement utile lorsque l'application est déployée sur le nœud Edge, car il permet de réduire la latence en évitant les étapes de synchronisation avec le Cloud.

Lorsque l'EdgeCore reçoit les données, celles-ci sont ensuite synchronisées avec le CloudCore via les mécanismes internes de KubeEdge. Cette synchronisation permet de rendre les données disponibles au niveau du cluster Kubernetes.

Une fois les données présentes dans le Cloud, elles deviennent accessibles via l'API server Kubernetes. Ce mécanisme constitue le moyen le plus fiable pour une application déployée dans le Cloud d'accéder aux informations, car il garantit que les données récupérées correspondent à l'état le plus récent synchronisé.

Ainsi, en fonction de l'emplacement de l'application, deux chemins de données coexistent :

- un accès direct via le push HTTP lorsque l'application est déployée sur l'Edge ;
- un accès via l'API server Kubernetes lorsque l'application est déployée dans le Cloud.

Cette conception permet d'adapter dynamiquement le chemin des données en fonction du contexte d'exécution, tout en assurant la cohérence globale du système.

4.5 Conception du système de métriques

Afin d'évaluer les performances du système, un mécanisme simple de mesure du temps de réponse a été mis en place. Ce mécanisme repose sur un calcul effectué directement au niveau des objets connectés émulés.

Le principe consiste à mesurer le temps écoulé entre l'émission d'une donnée par un mote et la réception de la réponse associée. Pour cela, chaque mote envoie un timestamp initial t_0 en même temps que ses données.

Ces données sont ensuite reçues par le mapper, qui les transmet à l'application. Selon le mode de déploiement de l'application, deux cas sont possibles :

- si l'application est déployée sur le nœud Edge, les données lui sont envoyées directement via un mécanisme de push HTTP.
- si l'application est déployée dans le Cloud, elle récupère les données via l'API server Kubernetes.

Une fois les données reçues, l'application simule un temps de traitement en introduisant un délai aléatoire. À l'issue de ce traitement, elle renvoie une réponse au mote, contenant un nouveau timestamp.

Lorsque le mote reçoit cette réponse, il calcule un nouveau timestamp t_1 et en déduit le temps de réponse global du système :

$$\Delta t = t_1 - t_0$$

Cette valeur correspond à un **temps de réponse de type round-trip time (RTT)**, incluant à la fois les délais de communication réseau et le temps de traitement de l'application.

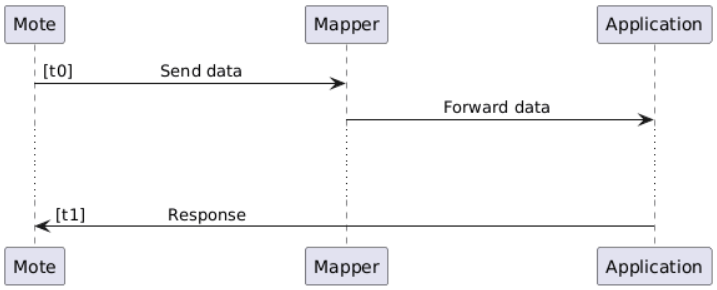

Figure 7 : Chronogramme de temps de réponse

Une fois ce calcul effectué, le mote expose cette nouvelle donnée comme une propriété supplémentaire, qui est ensuite transmise au mapper. Cette information suit alors le même chemin que les autres données : elle est envoyée à l'EdgeCore, puis synchronisée avec le CloudCore.

Afin d'exploiter cette mesure, un mécanisme de collecte de métriques a été mis en place au niveau du cluster Kubernetes. Une application dédiée agit comme un exporteur : elle interroge régulièrement l'API server Kubernetes afin de récupérer les données des dispositifs IoT (via les ressources de type *Device*).

Ces données sont ensuite exposées sous forme de métriques compatibles avec *Prometheus*. Le système de monitoring Prometheus, déployé dans le cluster, collecte ces métriques.

Enfin, un composant *Prometheus Adapter* est utilisé pour transformer ces métriques en ressources exploitables par Kubernetes. Cela permet de les intégrer au système de métriques du cluster et de les rendre utilisables par le mécanisme de *Horizontal Pod Autoscaler (HPA)*.

Cette approche permet ainsi d'adapter dynamiquement les ressources allouées aux applications en fonction du temps de réponse observé, en s'appuyant sur des métriques directement issues du comportement réel du système.

4.6 Maîtrise de l'environnement logiciel et compilation locale

Dans le cadre du projet, il nous a été demandé de travailler à partir du code source de la distribution Kubernetes choisie et de KubeEdge, et de compiler nos propres binaires. Cette exigence visait à garantir la stabilité du système et à éviter tout problème lié à des évolutions futures des versions utilisées.

Pour répondre à cet objectif, les codes sources de K3s et de KubeEdge ont été récupérés, ainsi que l'ensemble de leurs dépendances. Une attention particulière a été portée à la mise en place d'un environnement de compilation entièrement local, ne nécessitant aucun accès à Internet. Cette approche permet de garantir la reproductibilité du processus de build.

Les composants ont ensuite été compilés localement afin de produire de nouveaux binaires pour K3s et EdgeCore. Dans le cas de CloudCore, celui-ci étant déployé sous forme de pod dans la version utilisée de KubeEdge, une image conteneur a été construite localement à partir du code source.

Enfin, les binaires et l'image générés ont été intégrés à l'infrastructure en remplacement des versions précompilées. Cette démarche permet de disposer d'un environnement entièrement maîtrisé, basé sur des versions figées, et donc de garantir la stabilité et la reproductibilité du système à long terme.

Au-delà de l'exigence initiale, cette approche présente également un intérêt technique important, en offrant un meilleur contrôle sur le comportement des composants et en facilitant leur intégration dans l'architecture globale du projet.

5 Résultats et validation

Cette section présente les principaux résultats obtenus au cours du projet ainsi que les mécanismes de validation mis en place. Elle met notamment en évidence le fonctionnement de l'infrastructure déployée, la validation des flux de données et l'intégration du système de métriques au sein de l'environnement Cloud/Edge.

Les limites identifiées ainsi que les perspectives d'évolution envisageables pour prolonger ce travail sont également présentées.

5.1 Validation de l'infrastructure

L'infrastructure complète du projet a pu être déployée et validée avec succès. Les environnements KubeEdge et COOJA/Contiki-NG communiquent correctement à travers les différents mécanismes mis en place au cours du projet.

Les motes émulés dans COOJA/Contiki-NG sont capables d'échanger des données avec les mappers MQTT et CoAP, qui assurent ensuite leur intégration au sein de l'infrastructure KubeEdge. Les données remontées par les dispositifs IoT sont correctement synchronisées entre l'Edge et le Cloud puis accessibles depuis l'API Kubernetes.

Les applications de traitement ont également pu être exécutées aussi bien côté Cloud que côté Edge, validant ainsi les différents flux de communication de l'architecture.

Les figures suivantes illustrent notamment l'environnement COOJA/Contiki-NG utilisé ainsi que la remontée des données au sein de l'infrastructure Cloud/Edge.

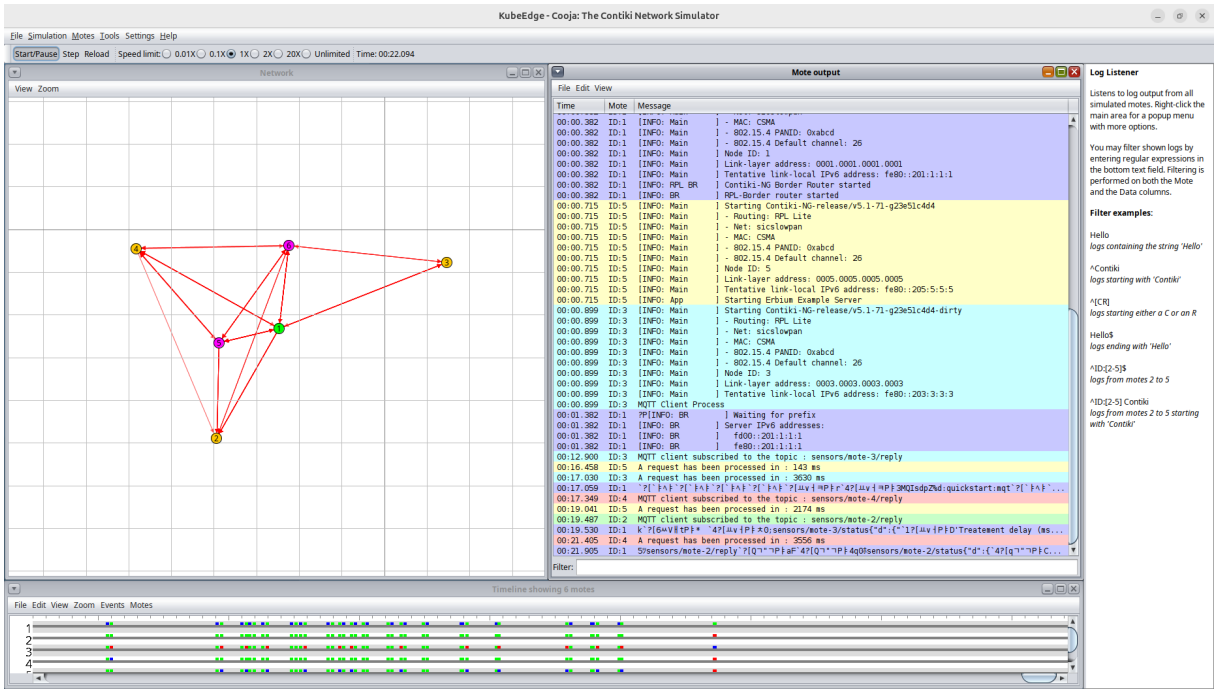

Figure 8 : Environnement d'émulation COOJA/Contiki-NG

La figure 8 présente l'environnement COOJA/Contiki-NG utilisé dans le cadre du projet. Dans la fenêtre de gauche, le mote vert correspond au *RPL Border Router*, tandis que les motes jaunes représentent les dispositifs MQTT et les motes roses les dispositifs CoAP. Les flèches rouges illustrent les communications réseau entre les différents motes du réseau RPL. La fenêtre située à droite affiche quant à elle les sorties produites par les motes au cours de leur exécution.

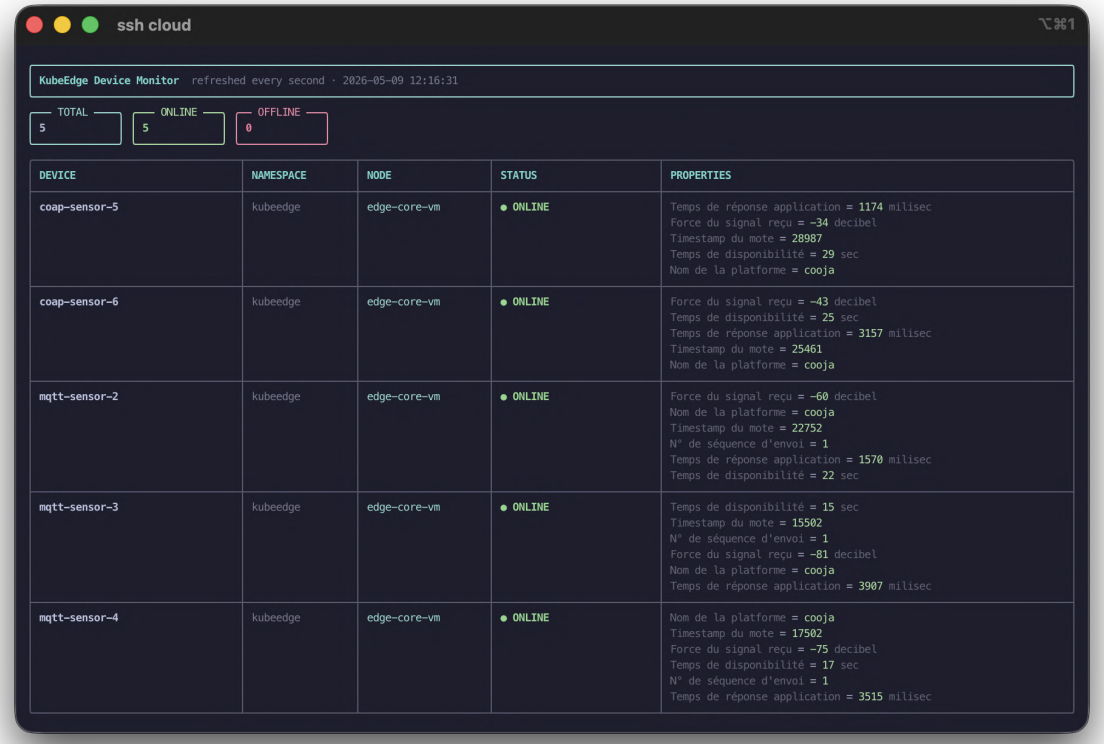

Figure 9 : Données des motes remontées au sein de l'infrastructure Cloud

La figure 9 montre les données remontées par les motes au sein de l'infrastructure KubeEdge. Cette interface, exécutée côté Cloud, permet de visualiser les différents dispositifs enregistrés ainsi que les propriétés synchronisées depuis les composants Edge vers le Cloud.

5.2 Validation du système de métriques

Le système de métriques basé sur le calcul du RTT (*Round-Trip Time*) a également été validé expérimentalement. Les timestamps générés par les motes sont correctement transmis aux applications de traitement puis exploités afin de calculer les temps de réponse observés au sein du système.

Les métriques produites sont remontées jusqu'au Cloud, exposées à Prometheus puis rendues accessibles au mécanisme de *Horizontal Pod Autoscaler*. Des variations artificielles du temps de traitement ont permis de vérifier le bon fonctionnement de la chaîne complète de monitoring.

Les figures suivantes montrent notamment l'exposition des métriques dans Prometheus ainsi que l'adaptation dynamique du nombre de pods en fonction des variations du RTT

observé.

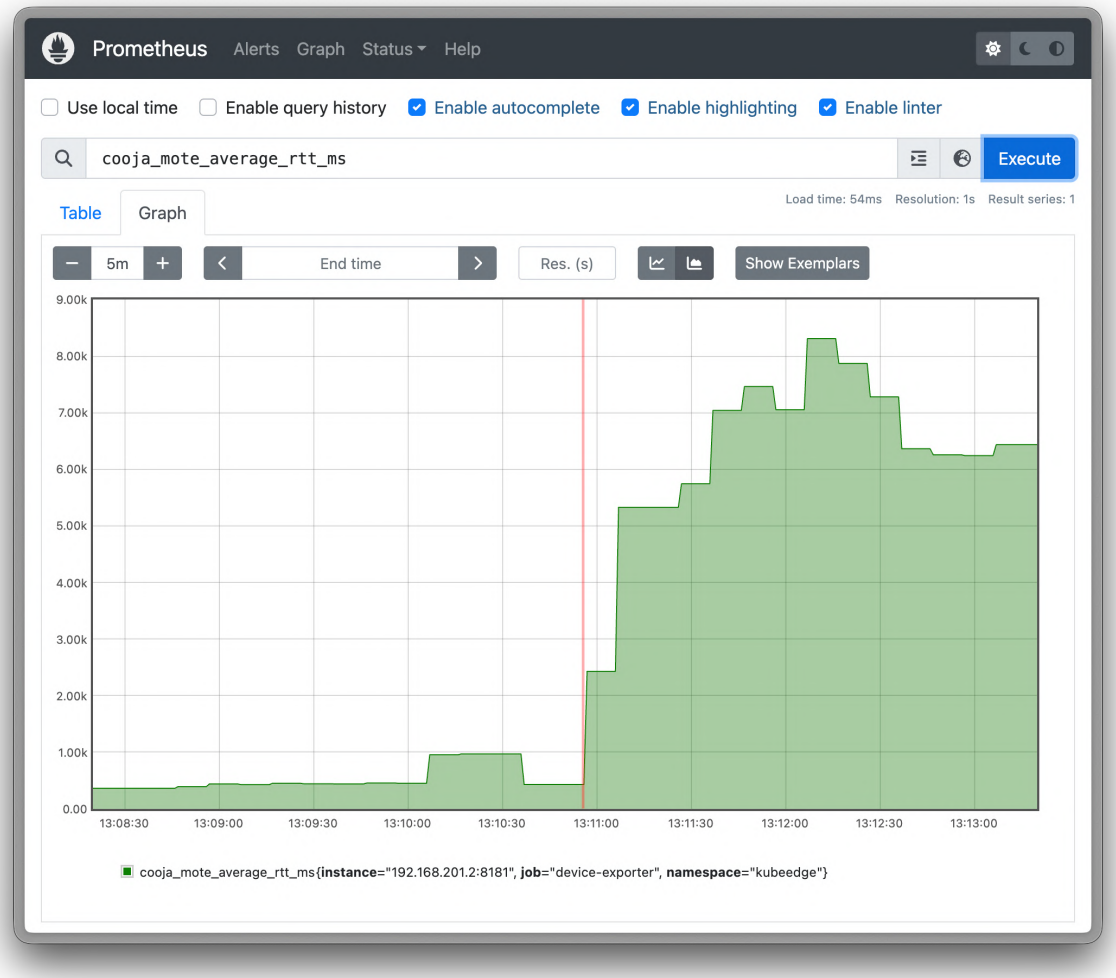

Figure 10 : Évolution de la métrique RTT moyenne calculée par les motes

Dans la figure 10, L'axe rouge correspond au moment où le temps de traitement de l'application a été artificiellement augmenté de 0 seconde à une valeur comprise entre 5 et 7 secondes. On observe alors une augmentation significative du RTT moyen, qui passe d'environ 400 ms à des valeurs oscillant entre 5000 et 8500 ms.

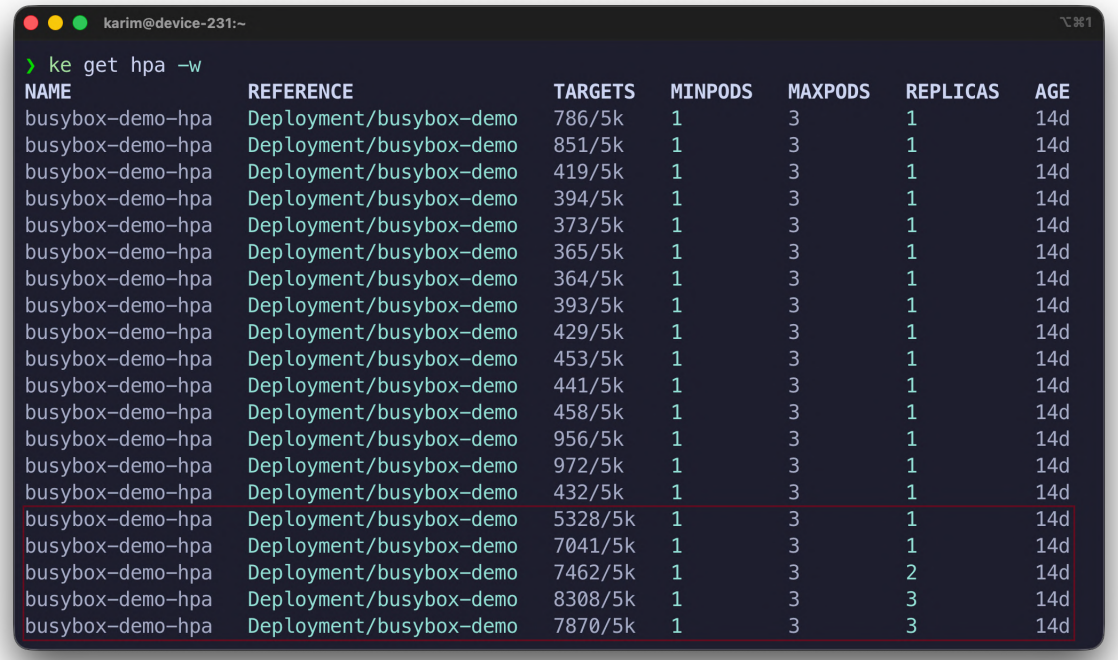

Figure 11 : Adaptation automatique du nombre de pods par le Horizontal Pod Autoscaler

Dans la figure 11, La zone encadrée en rouge correspond à l'augmentation artificielle du temps de traitement de l'application. On peut observer que le mécanisme de *Horizontal Pod Autoscaler* augmente automatiquement le nombre de pods associés au déploiement ciblé, passant de 1 à 3 pods en réponse à l'augmentation du RTT.

5.3 Limites et perspectives

Le projet a permis de valider le bon fonctionnement de l'ensemble des mécanismes mis en place. Cependant, les ressources matérielles disponibles pour les expérimentations sont restées limitées, notamment en termes de puissance de calcul et de mémoire. Cette contrainte n'a pas permis de réaliser des scénarios impliquant un très grand nombre de motes émulés afin d'étudier plus précisément l'évolution du RTT sous forte charge.

Une perspective intéressante consisterait à déployer l'infrastructure sur des machines plus puissantes ou dans un environnement distribué afin de réaliser des campagnes de tests à plus grande échelle.

Enfin, le projet constitue une base solide pour le développement de mécanismes d'orchestration dynamique capables d'adapter le placement des traitements entre le Cloud et l'Edge en fonction des performances observées à travers les métriques collectées.

6 Conclusion

Ce projet avait pour objectif de concevoir et mettre en place une plateforme permettant l'interaction entre une infrastructure Edge Computing basée sur KubeEdge et un environnement d'émulation IoT reposant sur COOJA/Contiki-NG. L'ensemble des composants nécessaires au fonctionnement du système a été intégré au sein d'une architecture distribuée composée de nœuds Cloud, Edge et de dispositifs IoT émulés.

Les principaux objectifs techniques ont été atteints. Une infrastructure KubeEdge fonctionnelle a été déployée sur plusieurs machines virtuelles, incluant les composants CloudCore et EdgeCore. Les mécanismes de communication entre les objets connectés et la plateforme ont été implémentés à travers le développement de mappers MQTT et CoAP, permettant l'intégration des dispositifs IoT au sein de l'infrastructure.

Le projet a également permis la mise en place d'un système de mesure du temps de réponse basé sur un mécanisme de type RTT. Les métriques obtenues ont été intégrées dans l'écosystème Kubernetes à l'aide de Prometheus, permettant leur exploitation par le mécanisme de *Horizontal Pod Autoscaler*.

Par ailleurs, une attention particulière a été portée à la reproductibilité et à la maîtrise de l'environnement logiciel. Les composants K3s et KubeEdge ont été compilés localement à partir de leur code source, permettant de disposer d'un environnement stable, entièrement contrôlé et indépendant des versions précompilées externes.

Les résultats obtenus montrent la faisabilité d'une intégration cohérente entre une infrastructure Cloud/Edge et un environnement d'émulation IoT réaliste. Ils mettent également en évidence l'intérêt de l'utilisation de métriques applicatives issues du comportement réel des dispositifs pour piloter dynamiquement les ressources du système.

Enfin, ce travail constitue une base pour des développements futurs autour de l'orchestration intelligente dans les environnements Cloud/Edge. Une perspective importante consisterait notamment à développer un ordonnanceur capable d'adapter dynamiquement le placement des traitements entre le Cloud et l'Edge en fonction des performances observées au sein du système.

7 Bibliographie

Références

[1] Alibaba Cloud. *What Is Edge Computing ?* 2021. url : `https://www.alibabacloud.com/fr/knowledge/what-is-edge-computing`.

[2] The Kubernetes Authors. *Kubernetes Documentation*. 2026. url : `https://kubernetes.io/docs/home/`.

[3] Charles Mahler. *Kubernetes on the edge : getting started with KubeEdge and Kubernetes for edge computing*. 2022. url : `https://www.cncf.io/blog/2022/08/18/kubernetes-on-the-edge-getting-started-with-kubeedge-and-kubernetes-for-edge-computing/`.

[4] *KubeEdge V1.22 Documentation*. KubeEdge Project Authors. 2023. url : `https://release-1-22.docs.kubeedge.io/docs/`.

[5] George Oikonomou et al. "The Contiki-NG open source operating system for next generation IoT devices". In : *SoftwareX* 18 (2022), p. 101089. issn : 2352-7110. doi : `https://doi.org/10.1016/j.softx.2022.101089`.

[6] Wikipedia contributors. *Contiki*. 2026. url : `https://fr.wikipedia.org/wiki/Contiki`.

[7] OASIS. *OASIS Message Queuing Telemetry Transport (MQTT) Technical Committee*. 2020. url : `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt`.

[8] OASIS MQTT Technical Committee. *MQTT : The Standard for IoT Messaging*. 2014. url : `https://mqtt.org/`.

[9] Z. Shelby, K. Hartke et C. Bormann. *The Constrained Application Protocol (CoAP)*. 2014. url : `https://www.rfc-editor.org/info/rfc7252`.