RIO203 - Smart Parking

DI COSTANZO Sacha
EYRAUD Côme
GAFFNEY Katharine
WEBERT Nans

Encadrant : LIM Keunwoo

Année : 2023-2024

Remerciements

Nous tenions à remercier notre encadrant sur ce projet M. LIM pour le partage de son expertise, l'accompagnement fourni tout au long du projet ainsi que l'enseignement des bonnes pratiques spécifiques à ce type de projet technique.

Nous tenions également à remercier l'association Rezel (`https://rezel.net`) pour l'hébergement du serveur et la réactivité du support.

Table des matières

Introduction

Dans le cadre de l'**UE RIO203**, nous avons eu à réaliser un projet en lien avec l'IoT et l'innovation. Ce projet a été mené en groupe de quatre étudiants. Notre groupe, constitué de quatre étudiants de troisième année en cursus d'apprentissage, a fait le choix de proposer une solution de parking intelligent.

Dans ce rapport, nous présenterons les moyens mis en œuvre afin de construire ce projet ainsi que les méthodes utilisées servant à vérifier son bon fonctionnement.

Durant la réalisation, notre projet a été hébergé sur l'URL : `https://parking-rio.rezel.net`

À titre d'information, ce rapport a été rédigé en LaTeX. Il est disponible ici : `https://github.com/Its-Just-Nans/rio203-report`

1 Présentation du Projet

Notre projet de smart parking vise à améliorer l'expérience de stationnement en utilisant une approche intégrant des capteurs de détection, un concept de réservation en ligne et une reconnaissance des plaques d'immatriculation. Face aux défis croissants de gestion des espaces de stationnement dans des environnements urbains denses, notre solution cherche à optimiser l'utilisation des places disponibles. Les capteurs de détection positionnés à chaque emplacement de stationnement assurent une surveillance en temps réel, indiquant clairement quels emplacements sont occupés et disponibles.

De plus, notre système de réservation en ligne permet aux utilisateurs de réserver à l'avance une place de stationnement, réduisant ainsi les incertitudes liées à la disponibilité des espaces. L'innovation clé réside dans la gestion des clients intégrée à la reconnaissance de plaques d'immatriculation, offrant une expérience pratique : l'entrée et la sortie du parking ne nécessitent pas de tickets physiques.

En abordant ces problèmes de gestion et d'accessibilité, notre solution vise à réduire les temps d'attente, améliorer la fluidité du trafic et optimiser l'espace de stationnement disponible, offrant ainsi une expérience plus pratique aux utilisateurs.

Architecture globale

Notre projet s'appuie sur plusieurs outils essentiels qui se combinent pour améliorer de manière significative l'expérience de stationnement.

Notre solution est composée de divers équipements qui interagissent entre eux pour réaliser trois fonctions que l'on détaillera par la suite. Chaque place de parking est équipée d'un dispositif composé d'un micro-contrôleur, de capteurs ainsi que de signalisation utilisateur de type LED. Un serveur cloud gère le lien entre les différentes parties du système. Il permet de relier les dispositifs de chacune des places de parking, d'offrir une interface web permettant la gestion du parking (mais aussi l'accès des utilisateurs) et enfin de communiquer avec le dispositif en charge de la reconnaissance de plaques d'immatriculation. Dans le schéma ci-dessous, nous ne matérialiserons que deux places de parking mais dans la pratique, chaque place aura un dispositif.

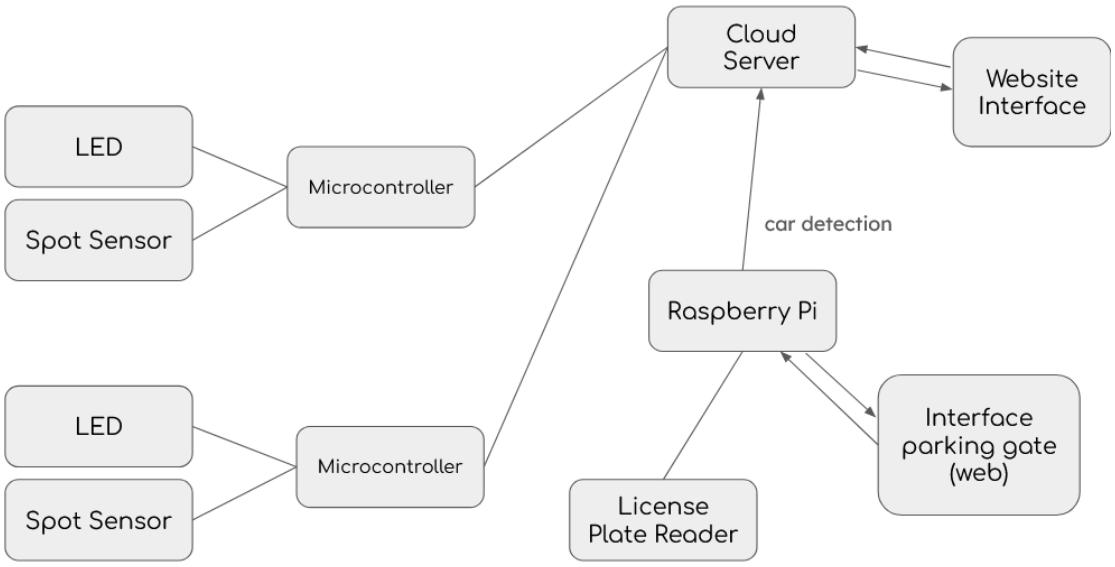

Fig. 1 : Architecture du système global

Dans notre cas, le détecteur de présence joue un rôle central en permettant de déterminer la disponibilité en temps réel d'une place de parking spécifique. Le système de détection de plaques d'immatriculation vient compléter cette infrastructure en simplifiant l'entrée et la sortie du parking grâce à la reconnaissance automatique des plaques associées aux réservations ainsi que la gestion de la facturation. Simultanément, notre serveur en tandem avec un site web dédié propose une plate-forme pour les réservations en ligne ainsi que pour le règlement des frais de stationnement. L'ensemble de ces outils collabore pour optimiser l'utilisation des espaces, réduire les temps d'attente et offrir une expérience de stationnement plus pratique et efficiente aux utilisateurs.

Architecture des technologies utilisées

Dans le cadre de ce projet, différentes technologies ont été mises en place. Lors de notre projet, nous avons pu tester deux micro-contrôleurs et pour chaque contrôleur, nous avons testé deux capteurs. L'un des micro-contrôleurs utilise Python tandis que l'autre se programme à l'aide du langage C++. Cette différence permet de montrer la compatibilité de notre système avec plusieurs types d'équipements (ce point sera détaillé ultérieurement). Les micro-contrôleurs utilisent le protocole WebSocket afin de communiquer avec le serveur cloud. Le serveur cloud est réalisé avec Hono, un framework permettant d'être utilisé avec plusieurs *cloud prodivers*. Celui-ci dispose d'une base de données avec SQLite avec laquelle il communique. Les interfaces web quant à elles sont codées avec le framework Svelte (en typescript). L'interface de gestion et d'accès utilisateur communique avec le serveur cloud à l'aide d'une API **REST** et du WebSocket (pour une communication en temps réel). Le module de reconnaissance de plaques d'immatriculation repose sur le langage Python en utilisant le framework **FastAPI** qui communique avec le serveur cloud en REST et la lecture de plaques d'immatriculation utilise OpenALPR. Le Raspberry Pi expose aussi une interface graphique en HTML et CSS et qui communique en **REST**.

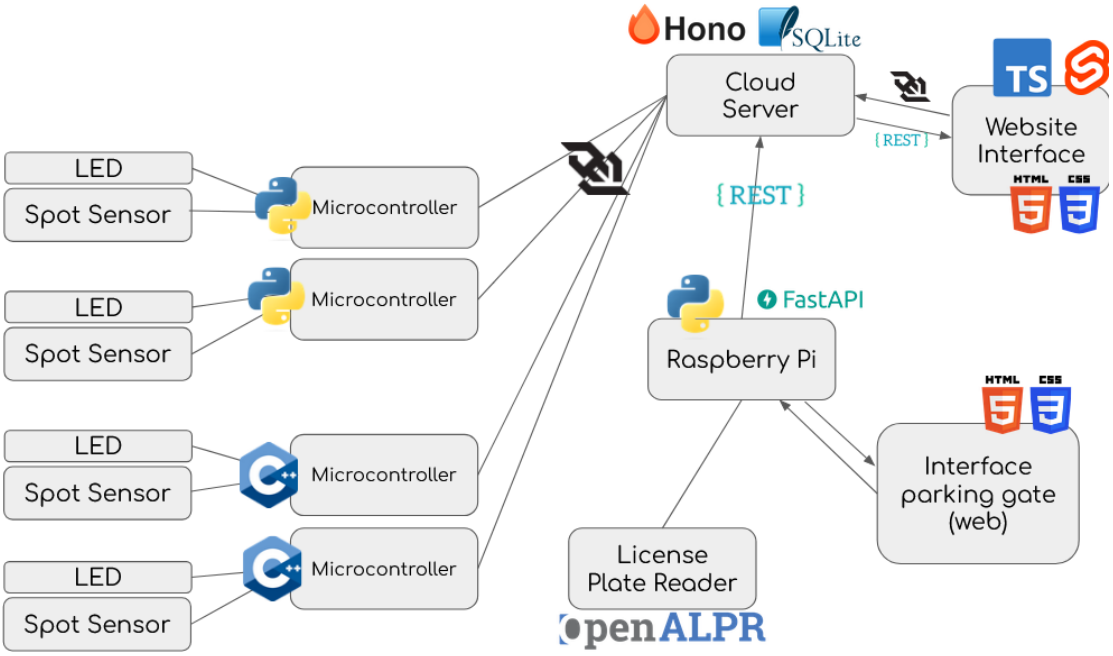

Fig. 2 : Architecture des technologies du système global

1.1 Fonction A

La première fonction que nous avons implémentée permet de visualiser l'état des places d'un parking à distance. En se connectant au site internet, un client peut ainsi vérifier la disponibilité des places ainsi que l'organisation de celles-ci. Il peut notamment voir si des bornes de recharge électrique sont disponibles, où se situent les places non occupées... Pour plus de simplicité, un indicateur (une LED) recense les places disponibles lorsqu'on est dans le parking. Dans ce cas de figure, les éléments mis en évidence sur la figure suivante sont ceux jouant un rôle dans l'exercice de cette fonction.

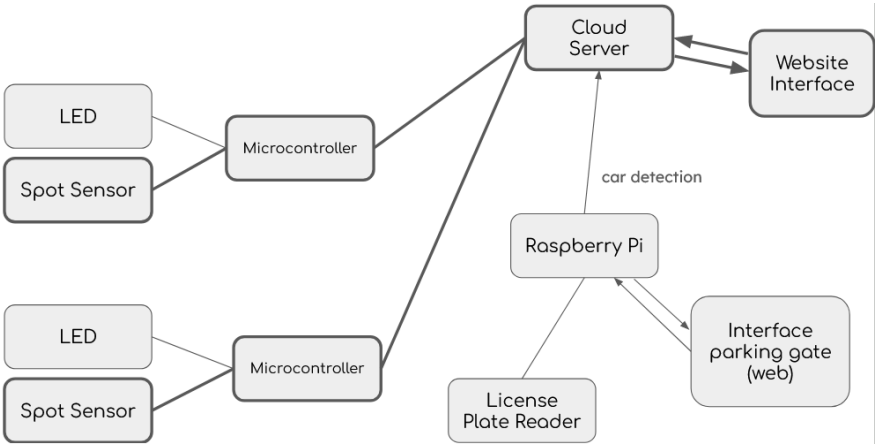

Fig. 3 : Architecture des composants du système utilisés pour la réalisation de la fonction A

1.2 Fonction B

La deuxième fonction mise en œuvre permet la réservation d'une place de parking. Cette fonction permet ainsi aux clients réguliers ou bien à ceux ne disposant que de très peu de temps pour rechercher une place une fois entré dans le parking de garantir la disponibilité d'une place au sein d'un parking. Il suffit au client de se connecter au site internet, sélectionner le parking voulu ainsi que la place désirée et si celle-ci est disponible, elle devient réservée. Lorsque la place est réservée, un marquage différent est utilisé sur le site internet ainsi que dans le parking. Cette fonction utilise les équipements mis en évidence sur la figure suivante.

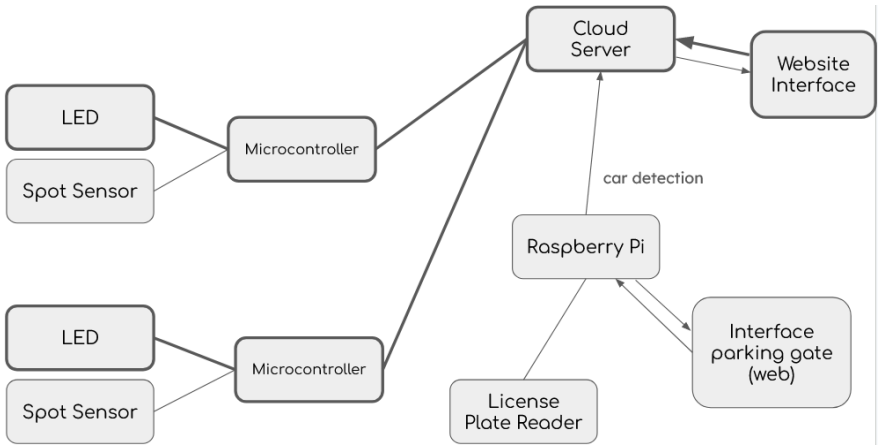

FIG. 4 : Architecture des composants du système utilisés pour la réalisation de la fonction B

1.3 Fonction C

La troisième fonction quant à elle a pour but de faciliter l'entrée, le paiement ainsi que la sortie des clients. En lisant la plaque d'immatriculation du client à l'entrée, elle permet une circulation sans ticket physique. Si de plus le client possède un compte sur le site internet, sa plaque d'immatriculation renseignée au moment de l'inscription est lue et associée à son compte au moment de son entrée dans le parking. De cette façon, lorsque le client souhaite sortir du parking, sa plaque d'immatriculation est lue à nouveau et l'argent déposé sur son compte internet est débité - si le solde est insuffisant ou bien si le client ne possède pas de compte sur le site internet, la somme à payer s'affiche sur l'écran. Cette fonction dont les composantes utiles sont mises en évidence sur la figure suivante permet de fluidifier le trafic en sortie du parking et accélérer le processus de paiement, d'entrée et de sortie.

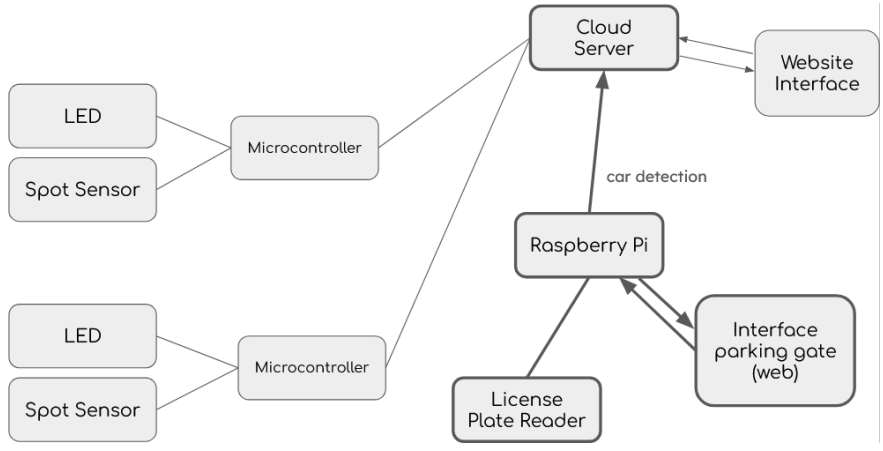

FIG. 5 : Architecture des composants du système utilisés pour la réalisation de la fonction C

2 Présentation des parties

Pour créer notre solution de parking intelligent, nous avons opté pour une architecture assez simple et logique. Nous prévoyons au début d'utiliser un serveur centrale par parking mais cela nous imposait des limites. Nous avons donc revu l'architecture avec un serveur dans le cloud et les autres équipements dans le parking.

Le parking possède donc des modules constitué d'un micro-contrôleur, d'une LED et d'un capteur. Chaque place de parking possède ce module capteur. Le parking possède également à l'entrée un module de lecture de plaque d'immatriculation. Ce module est fait à l'aide d'un Rapsberry Pi et de deux caméras (entrée et sortie). Le raspberry Pi affiche également une interface Web pour l'utilisateur a l'entrée ou pour payer à la sortie.

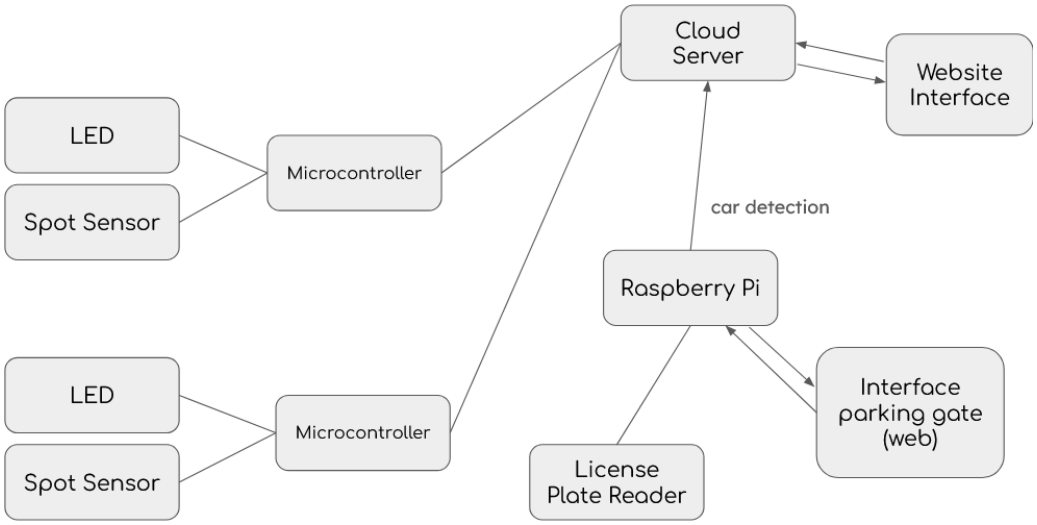

Fig. 6 : Architecture du système global

Afin de nous aider à nous organiser en amont du projet, nous avons réalisé un diagramme de Gantt. Cela nous a permis d'évaluer la durée de chaque tâche et de répartir le travail en fonction de la difficulté de la tâche et des aptitudes des étudiants.

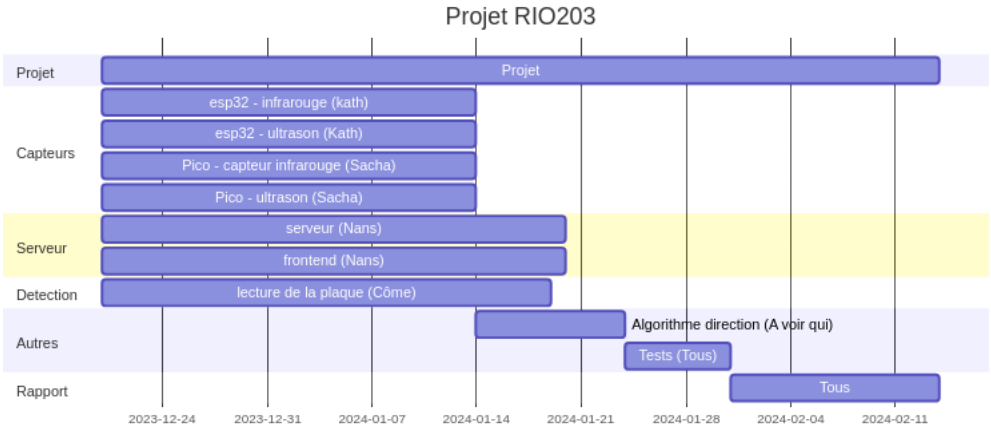

Fig. 7 : Diagramme de Gantt

Lors de la réalisation, les tâches ont généralement duré une semaine de plus. Cela n'a néanmoins pas changé le rendu final car nous avons choisi de ne pas réaliser la fonction d'algorithme de direction - étant donné que nous avions fait cela en première année dans le cadre de l'UE **INF103** (java).

2.1 Détecteur de présence

Afin de détecter la présence d'un véhicule sur une place du parking, nous installons un micro-contrôleur muni de divers capteurs. À l'aide d'une solution multi-plateformes et multi-capteurs, nous avons pris la décision de pouvoir offrir au client le choix du dispositif qu'il souhaite utiliser. Ainsi, notre produit ne nécessite pas forcément de nouvelles installations car si le client possède déjà des équipements, il pourra les réutiliser sans avoir à faire de nouvelles dépenses.

Chaque module sera composé
- d'un micro-contrôleur pouvant faire des requêtes sur internet
- d'une LED assurant la signalisation aux utilisateurs du parking
- d'un capteur permettant la détection

Dans notre cas, pour montrer la pluralité des solutions possibles, nous choisissons de tester quatre modules :
- un ESP32, une LED et un capteur infrarouge
- un ESP32, une LED et un capteur à ultrasons
- un Raspberry Pi Pico W, une LED et un capteur à infrarouge
- un Raspberry Pi Pico W, une LED et un capteur à ultrasons

Nous verrons dans les prochaines sections la réalisation et la comparaison des différents dispositifs ainsi que leur principe de fonctionnement.

Pour chacun des modules, les fonctionnalités ont été implémentées dans le langage de la plateforme : du Python pour le Rapsberry Pi Pico W et du C++ pour l'ESP32.

Le code pour chacun des capteurs est disponible sur github : `https://github.com/github.com/katheleligaf/rio203-sensors/`

Une idée intéressante qui aurait pu améliorer la démonstration aurait été la réalisation de boîte imprimée en 3D, qui contiennent et protègent les modules.

Protocole de communications

Comme décrit précédemment, chaque module devra offrir la possibilité de réaliser des requêtes sur internet dans le but de pouvoir communiquer avec le serveur. Dans un premier temps, nous avons choisi d'utiliser le protocole **HTTP** avec une **API** restful sur le serveur cloudifié. Cette **API** était très simple et permettait aux micro-contrôleurs de faire des requêtes vers le serveur central.

Après des tests fructueux, nous avons malheureusement rencontré un problème : le serveur ne pouvait pas envoyer de requêtes aux capteurs - or cette fonctionnalité est nécessaire (nous le verrons plus en détail dans les sections suivantes). En effet, pour pouvoir envoyer une requête depuis le serveur vers les capteurs, il faudrait tout d'abord que le capteur agisse comme serveur pour recevoir des requêtes. De plus, pour que le serveur puisse envoyer la requête au capteur, il faudrait connaître l'adresse IP du capteur.

Ainsi, cette solution n'est pas viable car il faudrait mettre en place un système de forwarding sur le routeur (**NAT** ou **PAT**) pour les adresses IP du parking. Cette solution ne nous convient pas car nous voulons que notre projet soit simple à mettre en place.

L'envoi et la réception de ces requêtes induisent cependant quelques contraintes. En effet, pour que la solution soit réactive, il est impératif que les transmissions soient réalisées en temps réel. De plus, puisque les interactions ont lieu dans les deux sens, il est nécessaire de mettre en place un système permettant les transmissions bidirectionnelles. Nous avons donc décidé d'utiliser le protocole WebSocket. Ce protocole permet de faire passer dans une connexion HTTP un flux d'information bidirectionnel et en temps réel entre le serveur et les capteurs.

Le protocole Websocket permet l'échange bidirectionnel entre un serveur et un client issus de deux réseaux différents. En utilisant un numéro de port ainsi qu'une adresse IP publique, le protocole Websocket contourne les problèmes liés au **NAT**/**PAT** ainsi que ceux de pare-feu. De plus, le protocole Websocket permet la transmission en temps réel. Cette solution est donc idéale dans notre contexte.

Pour mettre en place la communication Websocket, nous avons dû établir un protocole entre le serveur et les capteurs. Ce protocole simple permet d'envoyer et de recevoir des informations au format JSON.

Voici donc une explication des requêtes de notre protocole.

La première requête de notre protocole est envoyée par le serveur lorsque celui-ci enregistre une nouvelle connexion WebSocket. Le client doit répondre en indiquant son adresse MAC ainsi que, s'il existe, l'identifiant de la place de parking physique sur laquelle le dispositif est installé.

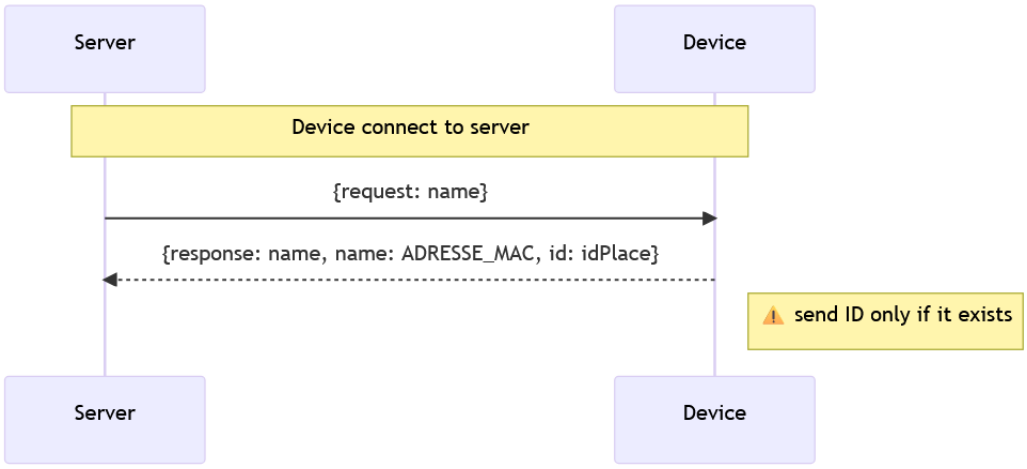

Fig. 8 : Structure de la réponse à la requête name

Lors de la première connexion au serveur au démarrage du micro-contrôleur, le module ne possède pas encore son identifiant. Il doit donc le demander au serveur en utilisant la requête **getId**.

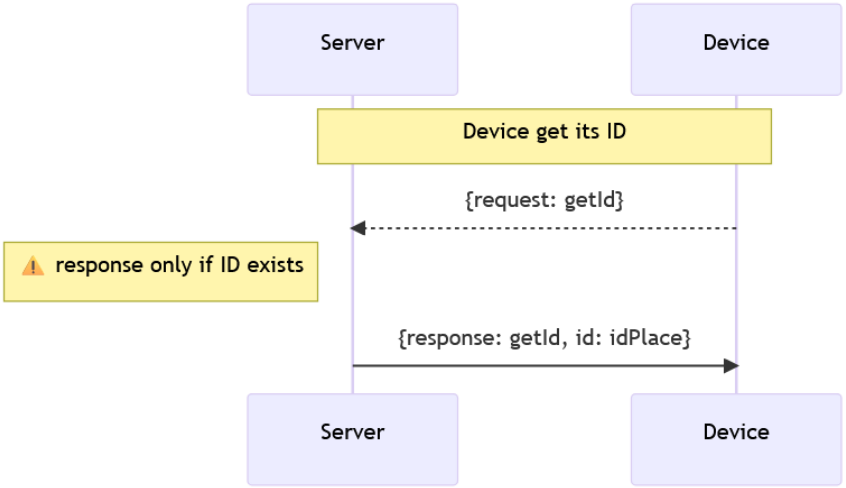

Fig. 9 : Structure de la requête getId

Une fois l'identifiant attribué, le module le garde en mémoire jusqu'à sa ré-initialisation. De ce fait, même si le serveur est redémarré, l'identifiant est toujours connu par le micro-contrôleur.

Grâce à la requête **getId**, le dispositif peut donc demander au serveur quel est l'identifiant de la place physique qui lui est attribuée. Pour pouvoir lier un ID de place et un capteur, le serveur

retient en mémoire l'attribution des places. S'il y a un lien, il pourra répondre à la requête **getId**. Autrement, le gestionnaire du parking est chargé d'attribuer manuellement un identifiant de place physique au dispositif - c'est la partie de mise en place du parking (elle n'arrive qu'une fois). Après avoir lié l'identifiant de place et le capteur, le serveur pourra

— renvoyer l'ID quand on lui fait une requête **getId**
— envoyer explicitement l'ID au capteur non lié avec l'appel **setId**

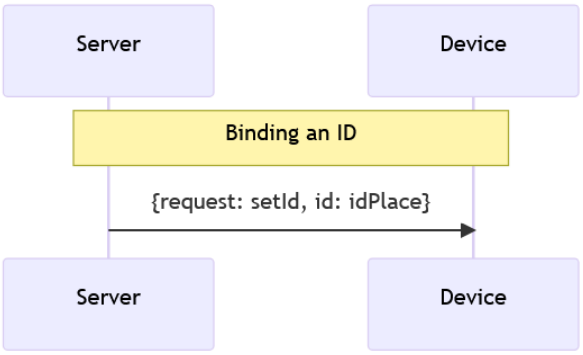

Fig. 10 : Structure de la requête setId

Après avoir reçu son ID, le dispositif peut commencer à détecter des présences et transmettre son état au serveur. Le micro-contrôleur agira ensuite comme une machine à état qui, selon le changement d'état, effectuera des actions (formuler une requête, changer la couleur de la LED, etc).

Lorsqu'un objet survole le dispositif, il détecte une présence et à l'inverse, lorsqu'aucun obstacle ne se trouve devant le dispositif, il ne détecte rien. Il met alors à jour son état (disponible, réservé, occupé) et conserve la valeur de son état précédent. Si l'état actuel est différent de l'état précédent, il envoie la requête **info** au serveur pour l'avertir du changement. Le capteur peut également être configuré plus intelligemment par exemple en calculant la durée de détection afin d'éviter les requêtes parasites lorsqu'un obstacle temporaire survole le capteur comme un piéton qui traverserait.

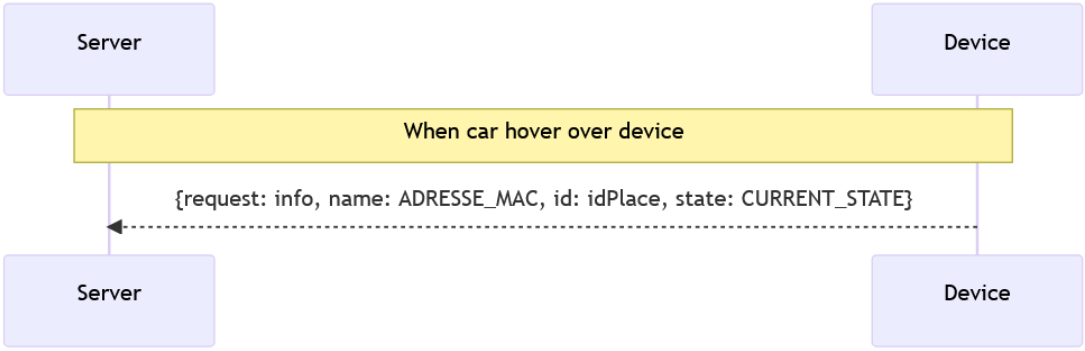

Fig. 11 : Structure de la requête info

Il est aussi possible que le serveur souhaite connaître l'état du dispositif. Pour cela, il peut envoyer la requête **state**. Le dispositif répond à la requête avec son adresse MAC, son ID de place ainsi que son état actuel.

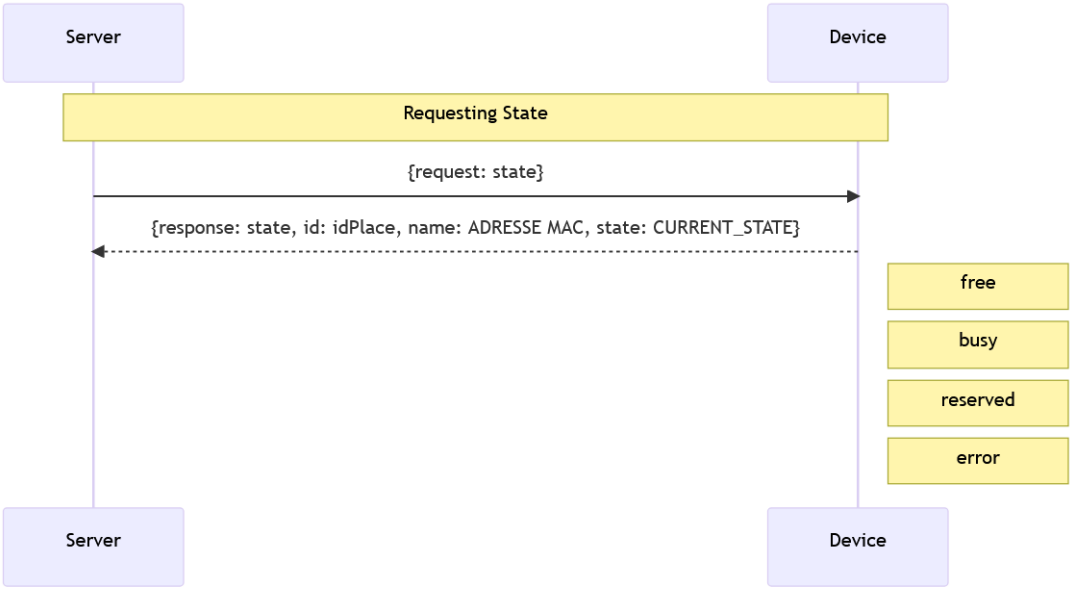

Fig. 12 : Structure de la réponse à la requête state

Enfin, la dernière requête de notre protocole permet la réservation de place. Pour cela, un utilisateur interagit avec le serveur qui va par la suite envoyer une requête **setState** au dispositif pour qu'il modifie son état actuel et qu'il affiche son état de réservation. À noter que cette requête peut aussi être utilisée lors du debug.

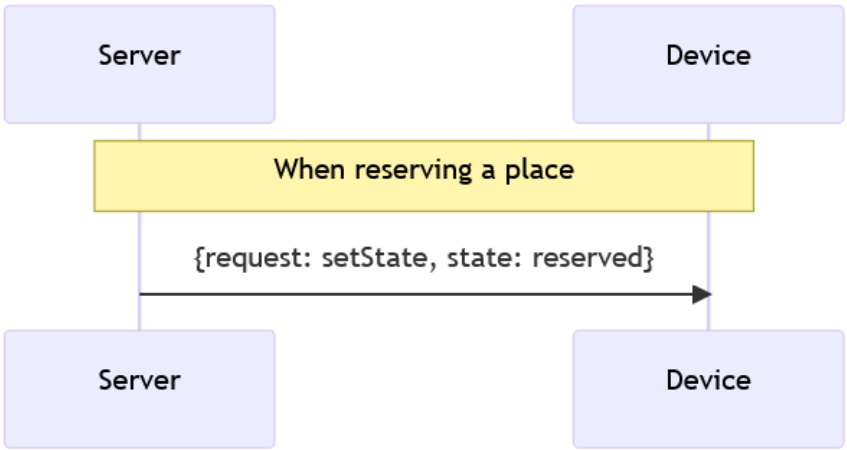

Fig. 13 : Structure de la requête setState

Version ESP32

Pour la version avec l'ESP32, nous testons deux capteurs différents : un capteur infrarouge et un capteur ultrason. L'ESP32 lit l'un des deux capteurs et détecte ou non la présence d'un véhicule. Il notifie ensuite le serveur de son état grâce à une requête et changera la couleur de la LED au besoin.

Pour communiquer avec le serveur qui se trouve dans le cloud, l'ESP32 a besoin d'être connecté à internet, il se connecte donc au WiFi du parking lors du démarrage, puis démarre la connexion WebSocket.

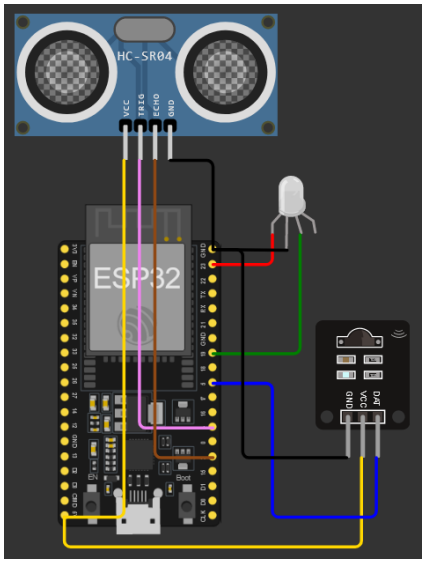

Fig. 14 : Architecture du circuit avec ESP32

Nous avons utilisé le capteur à ultrasons Sunfounder HC SR04 ainsi que le capteur infrarouge Sunfounder IR Obstacle Sensor. Enfin, afin d'indiquer l'état du micro-contrôleur, nous avons ajouté une LED bicolore Sunfounder Dual-Color LED. Ayant tout le matériel nécessaire, nous avons pu réaliser les tests sur du véritable hardware. Pour ce dispositif, nous avons utilisé l'IDE fournit par Ardunio : **Arduino IDE**. Le langage utilisé a donc été le C++.

Pour la communication WebSocket, nous avons profité de la communauté Arduino et avons utilisé plusieurs librairies :

— `https://github.com/gilmaimon/ArduinoWebsockets` pour la communication WebSocket
— `https://github.com/bblanchon/ArduinoJson` pour la gestion du JSON

Version ESP8266

Pour la version avec le Raspberry, le micro-contrôleur Raspberry Pico que nous avions n'avait pas la capacité de communiquer en WiFi. Nous l'avons donc connecté à une carte ESP8266 (WEMOS D1 WIFI) avec une liaison UART pour donner au Raspberry pico la capacité de communiquer par WiFi vers le serveur. Malheureusement nous avons rencontré des blocages dans le développement de cette première version. Premièrement, bien que nous ayons réussi à envoyer des messages vers l'ESP8266 en utilisant l'UART, nous n'avons pas été en mesure de les décoder correctement pour ensuite pouvoir les exploiter et les transmettre au serveur. De plus, les connexions entre les différents composants sur le breadboard (platine d'expérimentation) n'étaient pas stables et faisaient souvent défaut lors du développement de cette version (faux contacts, perte de détection des ports USB vers l'ordinateur ou bien même simplement des déconnexions) ce qui nous freinait dans l'avancement de cette version. À tel point que nous nous sommes retrouvés avec un composant (ESP8266) hors service suite à une fausse manipulation.

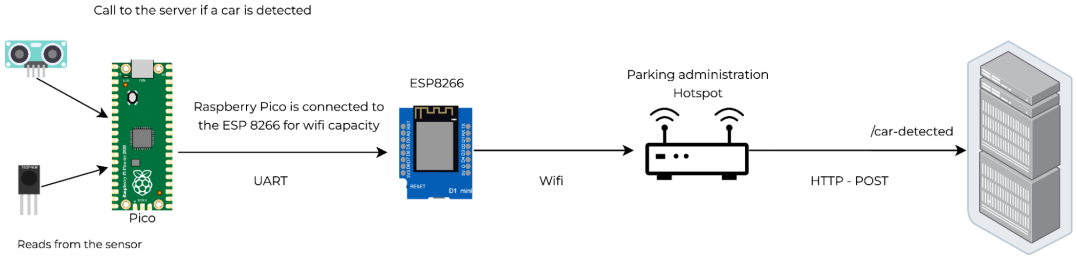

Fig. 15 : Architecture initiale du circuit avec ESP8266

Pour contourner l'échec de cette première tentative, nous avons finalement décidé de développer une solution via un simulateur de systèmes IoT disponible en ligne : Wokwi (`https://wokwi.com/`). En procédant ainsi, cela nous a permis de contourner les problèmes "matériels" (au sens hardware) ainsi que de simplifier la structure de ce module (un unique micro-contrôleur au lieu de deux). En effet, le site mettant à disposition de nombreux micro-contrôleurs et capteurs IoT différents, nous avons pu développer cette seconde version en utilisant uniquement un Rapsberry Pico WIFI pour à la fois collecter les données du capteur rattaché et les transmettre au serveur ensuite. Le programme développé pour cette seconde version est donc exclusivement en MycroPython (alors que la version initiale faisait en plus intervenir deux programmes : un en C++ et un en MicroPython).

Comme pour l'ESP32, il y a deux sous-versions : une avec un capteur infrarouge (Sunfounder IR Obstacle Sensor) et une avec un capteur à ultrasons (Sunfounder HC SR04). La logique est la même, si le micro-contrôleur considère qu'une voiture est garée à cette place, La LED change de couleur (passe au rouge) et il notifie alors au serveur que sa place est occupée via une requête WebSocket. Le serveur peut aussi envoyer des requêtes au micro-contrôleur pour mettre à jour certaines informations telle qu'indiquer qu'une place est désormais réservée.

Le code commenté destiné au fonctionnement du Raspberry Pico WIFI est disponible en annexe ou sur le repository GIT : `https://github.com/github.com/katheleligaf/rio203-sensors/`

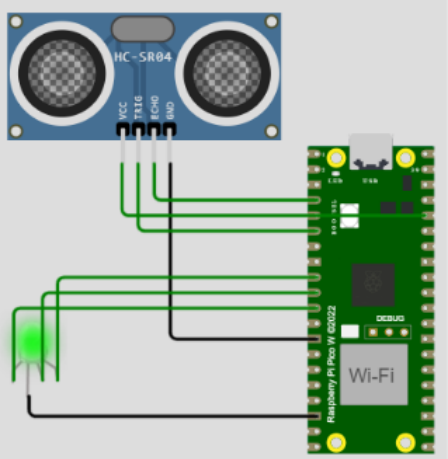

FIG. 16 : Architecture de la V2 avec Rapsberry Pico Wifi et le capteur ultrason

Les librairies systèmes utilisées dans le programme sont les suivantes :

- network : permet de gérer les fonctionnalités réseau comme la connexion WiFi.
- ujson : facilite la manipulation de données au format JSON (JavaScript Object Notation).
- time : offre des fonctionnalités pour la gestion du temps, comme la mesure du temps écoulé.
- uasyncio : fournit un support pour la programmation asynchrone en Python, permettant l'exécution de tâches simultanées.
- ubinascii : permet la conversion entre données binaires et leur représentation ASCII.
- machine : offre un accès aux fonctionnalités de bas niveau du matériel, comme la gestion des broches GPIO.

Certaines librairie possèdent un "**u**" devant le nom, cela signifie qu'elle sont spécifiques à MicroPython.

La librairie essentielle à notre projet est la librairie WebSocket.

Après avoir testé différentes librairies et clients websocket, nous avons retenu la librairie **AsyncWebsocketClient** (`https://github.com/Vovaman/micropython_async_websocket_client`). Cette librairie permet d'utiliser une unique classe "AsyncWebsocketClient", qui permet la gestion du protocole WebSocket. Cette librairie présente l'avantage d'être spécifique à MicroPython et c'est une "single file library" (librairie python compacte et autonome qui tient en un fichier, ce qui la rend facile à distribuer et à utiliser).

2.2 Lecteur de plaque d'immatriculation

Pour pouvoir gérer l'entrée et la sortie des voitures avec la reconnaissance des plaques d'immatriculation pour la gestion des frais liés au stationnement, il est nécessaire d'installer un dispositif à l'endroit destiné à accueillir les véhicules sortants ainsi que les véhicules entrants.

Architecture

L'outil de détection de plaques d'immatriculation a pour rôle d'envoyer une requête POST vers le serveur contenant l'identifiant unique du parking pour toute entrée ou sortie du parking. La requête POST doit donc préciser s'il s'agit d'une entrée dans le parking ou bien d'une sortie.

Système de détection

Pour réaliser la détection de plaque d'immatriculation, il est nécessaire de posséder une caméra ainsi qu'un ordinateur afin que ce dernier puisse valider la lecture supposée de la plaque d'immatriculation. Dans un premier temps, nous avons essayé d'utiliser la librairie opencv et de construire notre propre algorithme de détection. Nous avions initialement construit un algorithme basique qui ne présentait de bonnes performances que dans des conditions très précises et donc peu représentatives de la réalité. Après quelques recherches nous avons découvert OpenALPR (`https://github.com/openalpr/openalpr`), un programme open source qui utilise des algorithmes très performants de reconnaissance de plaques d'immatriculation (fait en C++). Après quelques tests, cette solution s'est avérée largement plus performante que le simple système de détection que l'on avait mis en place.

Implémentation

Après avoir choisi quel système de détection nous allions utiliser, il a fallu l'intégrer au reste de notre service. Pour ce faire, nous voulions utiliser un Raspberry Pi ainsi qu'un module caméra. N'ayant pas ce module, nous avons utilisé la Webcam d'un de nos ordinateurs portables qui jouera le rôle du module caméra pour le Raspberry Pi. Nous avons donc installé un serveur HTTP sur le Raspberry Pi, qui reçoit par POST une image encodée en base64 et traite l'image comme si celle-ci était envoyée depuis la caméra du Raspberry Pi. Après avoir effectué la détection, il enverra une requête POST vers le serveur, avec les informations nécessaires.

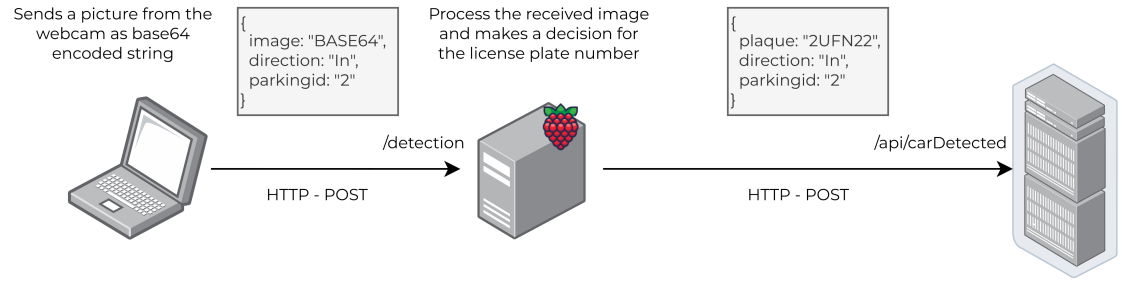

Fig. 17 : Schéma d'architecture du service de détection

Résultats

Ce service de détection de plaques d'immatriculation garantit de très bons résultats. Même avec de mauvaises conditions lumineuses, le système de détection est capable de lire correctement la plaque d'immatriculation présentée.

Il est cependant arrivé que des erreurs aient lieu, notamment lorsque le service confond le chiffre 0 et la lettre O. Mais puisque les plaques minéralogiques françaises sont sous le format AA-999-AA, il est donc possible d'améliorer la solution en chargeant le serveur de rectifier la mauvaise détection d'un zéro ou d'un O.

Le code de cette partie est dans ce repository :

— `https://github.com/comeyrd/rio203-image-detection`

2.3 Serveur

Le serveur est la partie centrale du projet. En effet, il permet de relier tous les équipements et de stocker les informations. Au début du projet, nous voulions avoir un unique serveur par parking. Après réflexion, nous avons constaté que cela nous imposait une limite : chaque parking devra posséder un serveur - cela signifie qu'on ne peut potentiellement pas agréger les parkings, ce qui pourrait être utile dans le cas où un client serait en fait un gestionnaire en charge de plusieurs parkings.

Nous avons donc choisi, à l'image des télécoms, de cloudifier notre infrastructure : le serveur est désormais dans le cloud et il reste seulement un Raspberry Pi dans le parking (et les capteurs). La partie serveur dispose donc de trois parties globales :

— la gestion d'un portail web pour les utilisateurs (admin ou non)
— la communication avec les capteurs
— la communication avec la barrière autonome

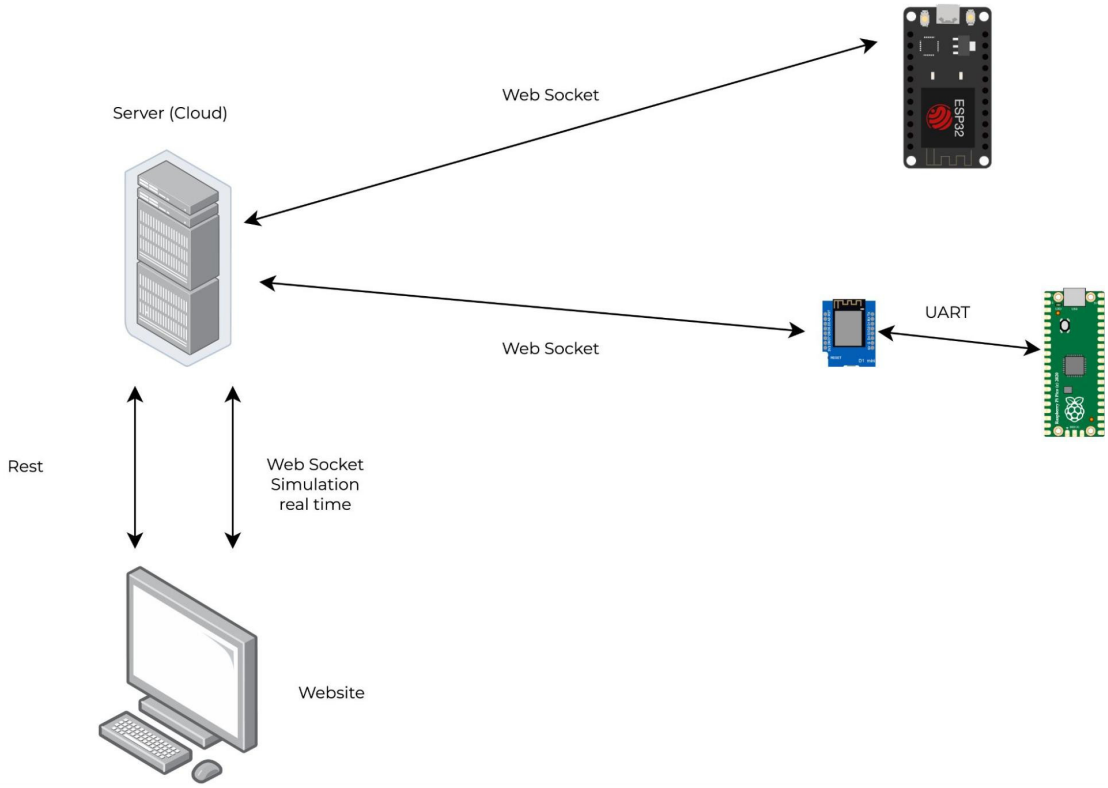

Fig. 18 : Architecture serveur

Communication avec les capteurs

La communication avec les capteurs est faite exclusivement en utilisant notre protocole (détaillé précédemment) sur WebSocket. Le serveur implémente donc ce protocole.

Communication avec la barrière autonome

La communication avec la barrière autonome se fait unidirectionnellement depuis le Raspberry Pi vers le serveur. Ainsi, la communication REST est adaptée à nos besoins.

Lors de la détection d'un véhicule, la barrière envoie sa requête contenant la plaque d'immatriculation. Le serveur enregistre donc la voiture en mode déplacement. Il vérifie également si la voiture est reliée à un compte client déjà existant.

Au moment de la sortie d'une voiture, il faut vérifier qu'un des capteurs ait préalablement envoyé une requête d'état (cela signifie qu'une voiture était sur une place et qu'elle l'a quittée). Après cette transition, le serveur connaît le nombre ainsi que les identifiant des véhicules se dirigeant vers la sortie. Ainsi lorsque la barrière détecte une voiture en direction de la sortie du parking, le serveur peut savoir de quelle voiture il s'agit.

Portail web

Le serveur a été réalisé en utilisant le framework javascript Hono (`https://hono.dev/`). Ce framework a la particularité de pouvoir fonctionner aisément sur la très grande majorité des clouds providers. Dans notre cas, comme nous n'utilisons pas de cloud provider mais un serveur généreusement prêté par Rezel, nous utilisons un *adapter* qui permet d'utiliser Hono avec NodeJS. Notre serveur est codé en typescript (superset de JavaScript), ce qui permet une certaine sécurité.

Pour le stockage des données, nous avons opté pour une solution éprouvée : SQLite. Cependant, pour faciliter son utilisation, nous avons eu recours à la librairie Drizzle, qui agit comme ORM (Operational Resource Management). Cet ORM permet de faciliter la création et la gestion de la base de données à l'aide de schéma SQL défini dynamiquement en TypeScript.

Pour le côté frontend, nous utilisons la remarquable librairie Svelte (`https://svelte.dev/`) qui permet d'allier simplicité et efficacité.

Le portail web possède deux parties :
- partie client
- partie administrateur

Partie client Dans cette partie, l'utilisateur peut se connecter et avoir accès à des informations basiques sur son compte : nom, plaque d'immatriculation enregistrée et solde. Il peut aussi créditer de l'argent sur son solde (simulé dans notre cas avec un bouton). L'utilisateur a également accès à un schéma des parkings présents sur la plate-forme et peut réserver une place s'il le souhaite.

Partie administrateur Dans cette partie , le ou les administrateurs ont accès aux différents parkings qu'ils possèdent. Cette partie permet également de faire la liaison des capteurs lors de l'installation du parking. En effet, lorsque le serveur reçoit une requête **getId** mais que la liaison capteur-place physique n'a pas été faite, il ne peut rien répondre, cependant, il peut enregistrer le fait qu'un capteur ait besoin d'être lié et ainsi remonter l'information à l'administrateur. L'administrateur peut par la suite lier une place physique et un capteur dans le schéma du parking.

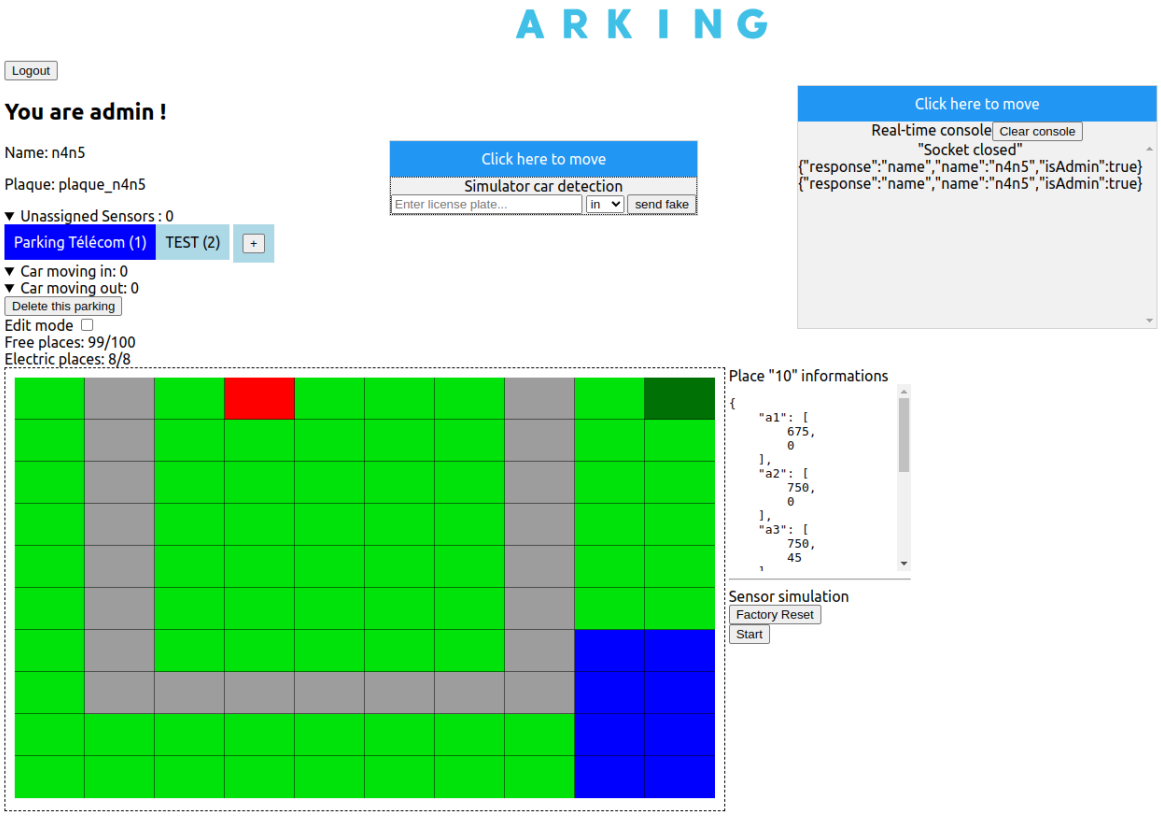

FIG. 19 : Partie administrateur

La partie administrateur possède également des éléments supplémentaires utiles dans notre projet : des simulateurs ! Un panel permet, par exemple, de simuler l'entrée d'un véhicule. Après avoir sélectionné une place, un autre panel permettant de simuler un capteur apparaît. Enfin une dernière fenêtre permet de voir en temps réel ce qu'il se passe au niveau du WebSocket.

Le code utilisé pour le serveur (frontend et backend) est dans ce repository : `https://github.com/Its-Just-Nans/rio203`

3 Présentation des résultats

Dans cette section, nous présenterons les résultats des implémentations des fonctions mentionnées précédemment.

3.1 Fonction 1 : Vérification des disponibilités à distance

Le serveur est la partie centrale du projet et il permet de tout réunir. Le serveur possède une interface web pour les utilisateurs. Sur cette interface, on peut y voir plusieurs informations (voir image ci-dessous).

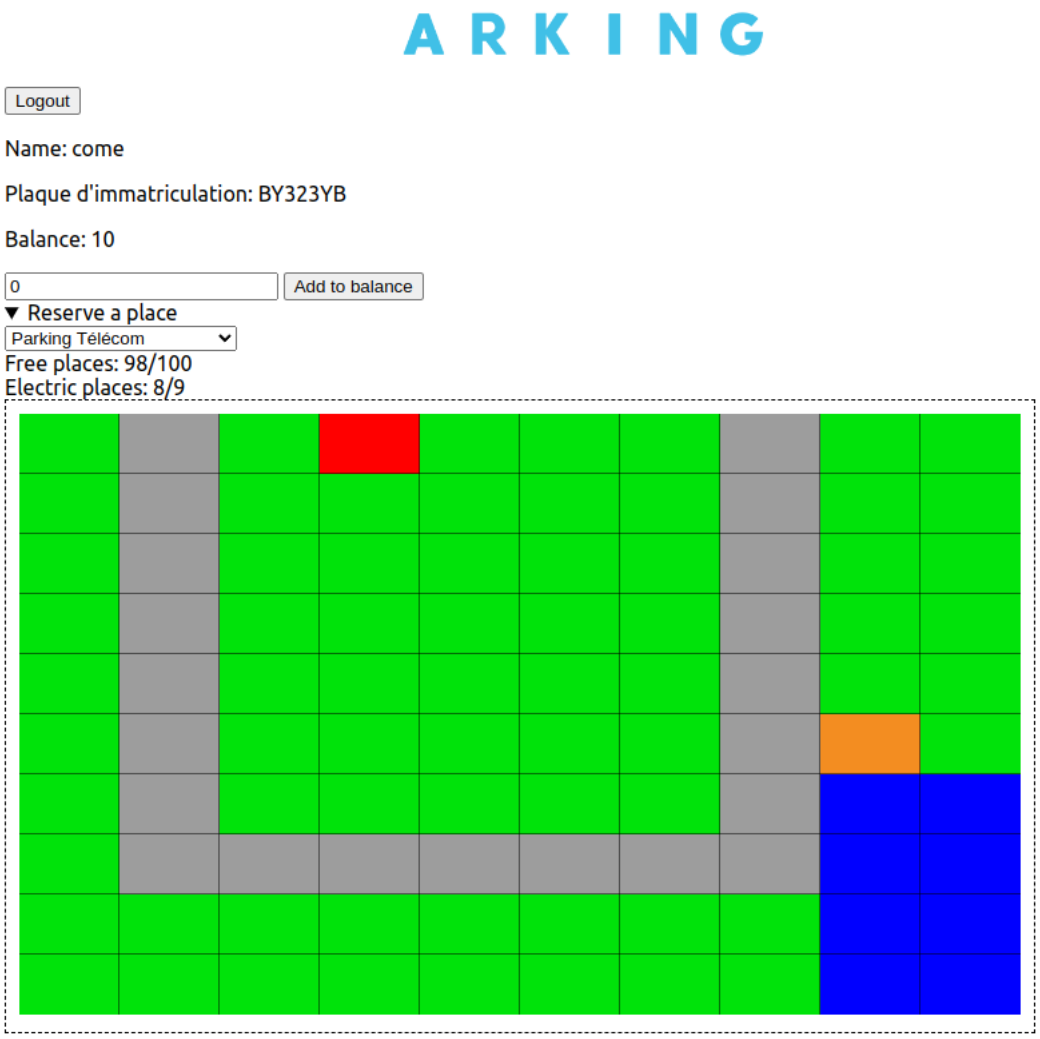

Fig. 20 : Affichage utilisateur du parking

On peut, par exemple, voir le solde de l'utilisateur et le bouton permettant de modifier celui-ci. Étant donné que nous sommes dans un Proof-of-Concept (PoC), nous n'avons pas mis en place de rechargement utilisant une **API** bancaire. En dessous, l'utilisateur peut sélectionner le parking qu'il souhaite visualiser, après la sélection, le parking s'affiche. L'application remplit donc bien la fonctionnalité de vérification des disponibilités. On peut également voir que les places ont des couleurs différentes.

Les différentes couleurs symbolisent :
- le gris pour la route (non réservable)
- le rouge pour une place occupée
- le orange pour une place réservée
- le bleu pour une place disposant d'une borne de recharge pour les véhicules électriques
- le vert pour une place disponible

Nous avons également pensé à une architecture assez souple, permettant par exemple d'ajouter très facilement un nouveau type de place, comme par exemple une place dédiée aux personnes à mobilité réduite.

3.2 Fonction 2 : Réservation d'une place

La réservation d'une place est possible dans l'interface de l'utilisateur. Lorsqu'il clique sur une place, un bouton de réservation apparaît. S'il clique dessus, la place sera réservée et deviendra orange.

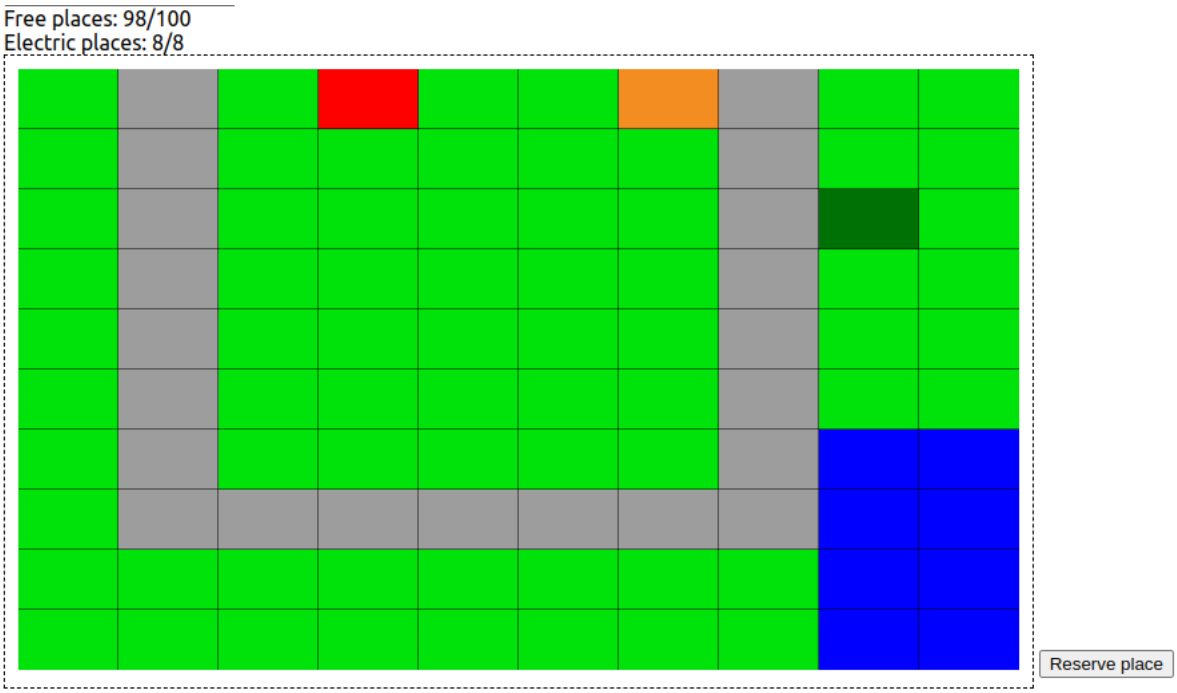

FIG. 21 : Réservation d'une place de parking

Cette fonctionnalité est donc bien réussie.

3.3 Fonction 3 : Lecture de plaque d'immatriculation et facturation

Comme expliqué précédemment, nous n'avions pas de caméra pour Raspberry Pi, nous avons donc utilisé la caméra de notre ordinateur. Cela est possible car le Raspberry Pi est lui-même un mini serveur en python, on peut donc avec notre ordinateur s'y connecter avec une interface web. Dans cette interface, du code JavaScript permet d'activer la caméra et d'envoyer les photos au serveur sur le Raspberry. Ensuite le Raspberry analyse l'image et envoie, s'il y a une plaque d'immatriculation détectée, la plaque au serveur.

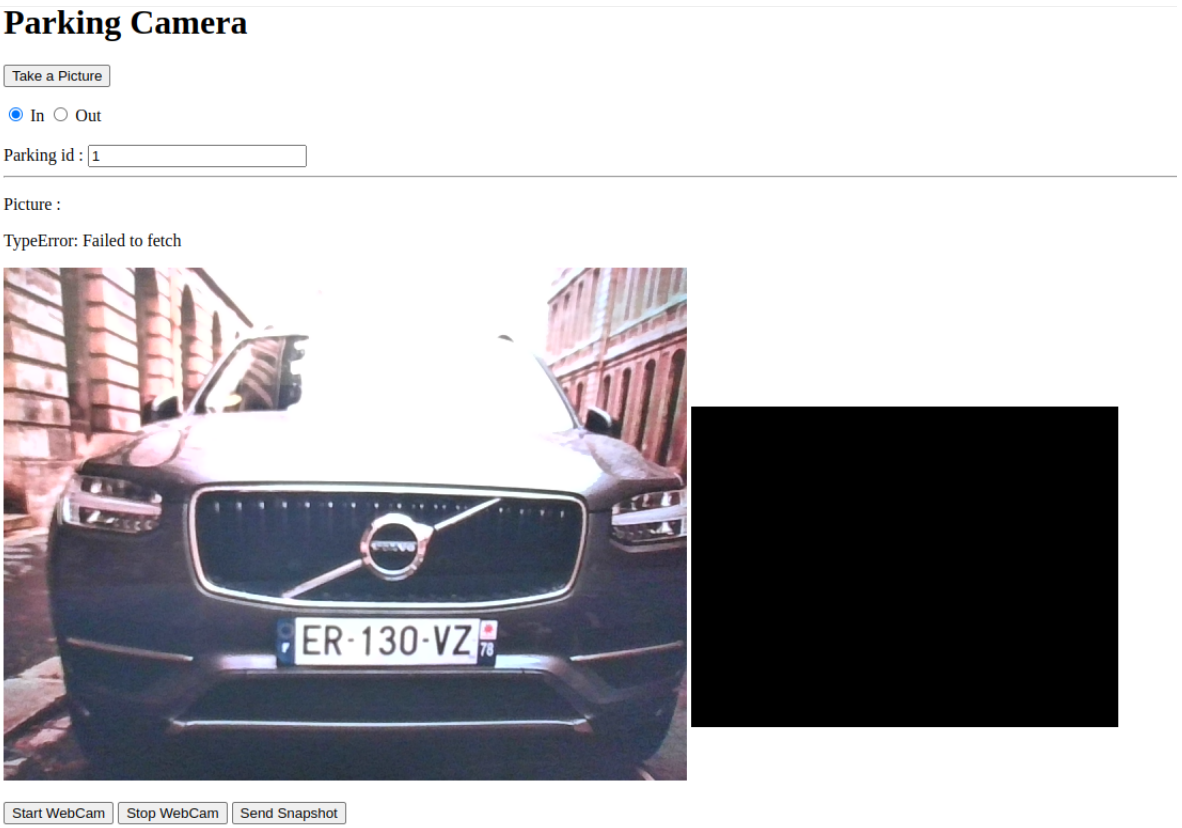

Fig. 22 : Détection de plaque

Sur l'interface, on peut voir l'image prise par notre caméra. Le carré noir représente le flux vidéo en live (il est caché dans cette capture). On indique l'ID du parking en haut, puis on envoie l'image au Rapsberry avec le bouton "Send Snapshot". Le Raspberry analyse l'image et fait une requête au serveur s'il y a une plaque. Enfin la réponse de la requête contient des informations à afficher à l'utilisateur comme par exemple le montant à payer.

De plus, le serveur possédant un système de solde utilisateur, lorsqu'une voiture sort du parking, qu'elle est reliée à un utilisateur et que celui-ci possède un solde suffisamment élevé, celui-ci est débité. S'il n'est pas assez élevé, l'utilisateur devra payer via l'interface du Raspberry Pi, mais comme expliqué précédemment notre projet n'utilise pas d'API bancaire pour plus de simplicité, ainsi, seul un message est affiché à l'utilisateur (pour simuler le paiement).

Ces fonctionnalités sont donc bien validées.

Conclusion

La réalisation du projet de smart parking a été une expérience enrichissante pour notre équipe, marquant la conclusion de notre parcours pratique dans le domaine de l'IoT au sein du programme RIO. Tout au long de cette aventure, nous avons intégré des technologies telles que les capteurs ESP, un serveur en Node.js, et une Raspberry Pi pour l'analyse d'image avec OpenALPR, afin de créer un système efficace de gestion de parking.

L'implémentation de ces technologies a permis à certains membres du groupe de développer leurs compétences dans la conception et la mise en uvre de solutions IoT concrètes. Les capteurs ESP ont été particulièrement utiles pour la collecte de données en temps réel, offrant une base solide pour la gestion intelligente des espaces de stationnement.

Le choix d'un serveur en Node.js a facilité la mise en place d'une architecture robuste et réactive, assurant une communication fluide entre les capteurs, la Raspberry Pi et d'autres composants du système. L'utilisation d'OpenALPR pour l'analyse d'image a permis d'automatiser la reconnaissance des plaques d'immatriculation, améliorant ainsi l'efficacité globale du système de smart parking.

Cependant, tout projet comporte des opportunités d'amélioration. Certains membres du groupe ont identifié des pistes d'optimisation telles que l'intégration d'une API bancaire pour une gestion plus avancée des transactions liées au stationnement. De plus, l'idée d'aller au-delà d'un simple proof of concept a été soulevée, suggérant la possibilité d'élargir les fonctionnalités du système pour répondre à des besoins plus complexes.

En ce qui concerne l'esthétique des interfaces, il a été noté que l'expérience utilisateur pourrait être encore améliorée pour rendre le système plus convivial. Des efforts supplémentaires pour affiner l'interface utilisateur contribueraient à renforcer l'acceptation et l'adoption du système par les utilisateurs finaux.

En conclusion, ce projet a été une étape significative dans notre parcours RIO, démontrant notre capacité à concevoir et mettre en œuvre des solutions IoT pratiques. Les leçons apprises, tant sur le plan technique que collaboratif, seront précieuses pour notre développement futur dans le domaine de l'IoT. Les opportunités d'amélioration identifiées offrent des perspectives passionnantes pour étendre et perfectionner notre système de smart parking, renforçant ainsi son utilité et son attrait sur le marché.

Liens

— `https://github.com/Its-Just-Nans/rio203` - Le web server
— `https://github.com/comeyrd/rio203-diagrams` - Diagrammes pour le rapport
— `https://github.com/katheleligaf/rio203-sensors` - Code des capteurs (ESP32, Rasperry Pico W)
— `https://github.com/comeyrd/rio203-image-detection` - Détection de plaque d'immatriculation
— `https://github.com/Its-Just-Nans/rio203-report` - Rapport
— `https://tinyurl.com/rio203` - Short link to this repo

Table des figures