Conception d'un module de passerelle pour objets connectés industriels multi-protocoles et services infonuagiques

par

Aurèle JACQUIN

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

MONTRÉAL, LE 9 AOÛT 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Tony Wong, directeur de mémoire
Département de génie des systèmes à l'École de technologie supérieure

M. Michel Rioux, président du jury
Département de génie des systèmes à l'École de technologie supérieure

M. Julio Montecinos, membre du jury
Département de génie des systèmes à l'École de technologie supérieure

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

LE 22 JUILLET 2022

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

J’aimerais dans un premier temps remercier Tony Wong, mon directeur de recherche, et lui exprimer toute ma reconnaissance. Tout ce travail accompli n’aurait pas été possible sans sa bienveillance, sa disponibilité, ses nombreux conseils et sa profonde gentillesse.

Je tiens aussi à remercier mes colocataires avec qui j’ai passé deux merveilleuses années. Leur présence était toujours une chance inouïe, que ce soit pour me procurer une motivation supplémentaire lors des sessions de travail à distance liées à la pandémie, ou pour apporter la joie et la bonne humeur le reste de la journée.

Un grand merci également à tous mes amis, ceux restés en France comme ceux rencontrés au Québec. Une pensée toute particulière pour ceux présents avec moi à Montréal avec qui j’ai pu vivre une expérience incroyable et effectuer de belles découvertes.

Et bien évidemment, je tiens à remercier ma famille, mes parents et mon frère qui ont toujours été une source d’inspiration et un moteur pour moi. Ils ont toujours su m’encourager, à chaque instant de ma vie, et c’est sans aucun doute grâce à eux que j’en suis là aujourd’hui.

Conception d'un module de passerelle pour objets connectés industriels multi-protocoles et services infonuagiques

Aurèle JACQUIN

RÉSUMÉ

L'apparition des nouvelles technologies liées au numérique procure de nouveaux outils à l'industrie, lui permettant d'être plus performante, plus efficace, et plus réactive. Il est en effet désormais possible de traiter les données en larges quantités, d'effectuer des observations à grandes échelles et de dresser des algorithmes pour optimiser les performances des machines industrielles. Le processus d'intégration de ces nouvelles technologies peut toutefois se révéler complexe, les objets industriels traditionnels communiquant via des protocoles bien particuliers le plus souvent incompatibles avec ceux utilisés par les services infonuagiques.

Ces dernières années, l'emploi de services infonuagiques est devenu très répandu au sein des entreprises, des solutions ont ainsi été développées pour permettre leur adoption à coûts raisonnables. On note alors l'apparition et l'emploi de systèmes nommés passerelles servant d'intermédiaires entre les objets industriels et le réseau. L'augmentation très prononcée des sujets de recherche sur ce type de dispositifs témoigne de l'intérêt croissant qui leur est porté, ce mémoire s'inscrit donc dans cet élan.

La grande diversité des protocoles industriels existants induit de facto la nécessité d'avoir un nombre conséquent de passerelles différentes. La passerelle présentée ici, développée à partir d'un Raspberry Pi, a ainsi pour objectif de répondre à ce besoin puisqu'elle implante plusieurs protocoles, à savoir Modbus, Canbus, MQTT et HTTP. Son coût modique, son développement, son caractère open source, ainsi que l'architecture de son programme en font un système abordable, adaptable et facilement adoptable. Un environnement d'expérimentation a été produit à l'aide notamment de microcontrôleurs Arduino afin de réaliser des essais dans le but de valider les performances de la passerelle. Plus d'une trentaine de tests ont été exécutés permettant alors de démontrer un fonctionnement sans faille des différentes options proposées par la passerelle, avec un taux d'erreur nul.

Mots-clés : passerelle, bus de terrain, protocoles industriels, internet industriel des objets, services infonuagiques

Design of a Gateway Module for Multi-Protocol Industrial Connected Objects and Cloud Services

Aurèle JACQUIN

ABSTRACT

The emergence of new digital technologies provides new tools to the industry, enabling it to be more efficient, more effective and more responsive. It is now possible to process huge amount of data, to make large-scale observations and to draw up algorithms to optimize the performance of industrial machines. However, the process of integrating these new technologies can be complex, as traditional industrial objects communicate via very specific protocols that are often incompatible with those used by cloud services.

In recent years, the use of cloud services has become widespread among companies, and solutions have been developed to enable their adoption at reasonable costs. We note the appearance of systems called gateways used as intermediaries between industrial objects and the network. The very pronounced increase of research topics on this type of devices testifies to the growing interest in them, and this thesis is therefore part of this dynamic.

The large diversity of the existing industrial protocols induces de facto the necessity to have a great number of different gateways. The gateway presented here, developed from a Raspberry Pi, aims to meet this need since it implements several protocols, namely Modbus, Canbus, MQTT and HTTP. Its low cost, its design, its open-source aspect, as well as the architecture of its program make it an affordable, adaptable and easily adoptable system. An experimental environment has been produced using Arduino microcontrollers in order to carry out tests to validate the performance of the gateway. More than thirty tests were performed, demonstrating its error-free operation for the various options offered by the gateway.

Keywords: gateway, fieldbus, industrial protocols, industrial internet of things, cloud services

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

6LoWPAN	*IPv6 over Low-Power Wireless Personal Area Networks*
ACK	*Acknowledge*
AM	*Asset Management*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
AWG	*American Wire Gauge*
BLE	*Bluetooth Low Energy*
CAN	*Controller Area Network*
CC	*Cloud Computing*
CIP	*Common Industrial Protocols*
CoAP	*Constrained Application Protocol*
CRC	*Cyclical Redundancy Check*
CRUD	*Create, Retrieve, Update, Delete*
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidance*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
CSMA/CR	*Carrier Sense Multiple Access with Collision Resolution*
CSV	*Comma-Separated Values*
DCS	*Distributed Control System*
DIY	*Do It Yourself*
DLC	*Data Length Code*

DP	*Decentralised Peripheral*
EC	*Edge Computing*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
EOF	*End of Frame*
ERP	*Enterprise Resource Planning*
E/S	Entrée/Sortie
GPIO	*General Purpose Input/Output*
HAT	*Hardware Attached on Top*
HTTP	*Hypertext Transfer Protocol*
I2C	*Inter-Integrated Circuit*
IA	Intelligence Artificielle
IDE	*Identifier Extension*
IEC	*International Electrotechnical Commission*
IFS	*Interframe Space*
IIoT	*Industrial Internet of Things*
IoT	*Internet of Things*
IP	*Internet Protocol*
IPv6	*Internet Protocol version 6*
ISO	*International Organization for Standardization*
JSON	*JavaScript Object Notation*
KNX	Konnex
LAN	*Local Area Network*
LCD	*Liquid Crystal Display*

M2M	*Machine-to-Machine*
MES	*Manufacturing Execution System*
MQTT	*Message Queuing Telemetry Transport*
NFC	*Near Field Communication*
NIST	*National Institute of Standards and Technology*
OASIS	*Organization for the Advancement of Structured Information Standards*
ODVA	*Open DeviceNet Vendors Association*
OPC	*Open Platform Communications*
OPC-UA	*Open Platform Communications – Unified Architecture*
OSI	*Open Systems Interconnection*
PHP	*Hypertext Preprocessor*
PLC	*Programmable Logic Controller*
PME	Petites et Moyennes Entreprises
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RPL	*Routing Protocol for Low-Power and Lossy Networks*
RTR	*Remote Transmission Request*
RTU	*Remote Terminal Unit*
Rx	*Receive*
RxD	*Receive Data*
SAC	*Standardization Administration of China*
SBC	*Single Board Computer*

SCADA	*Supervisory Control And Data Acquisition*
SEMI	*Semiconductor Equipment and Materials International*
SoC	*System on a Chip*
SOF	*Start of Frame*
SPI	*Serial Peripheral Interface*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
TTL	*Transistor-Transistor Logic*
Tx	*Transmit*
TxD	*Transmit Data*
UART	*Universal Asynchronous Receiver Transmitter*
UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
UTF-8	*Unicode Transformation Format, 8 bits*
XMPP	*Extensible Messaging and Presence Protocol*

LISTE DES SYMBOLES ET UNITÉS DE MESURE

°	Degré (angle)
Ω	Ohm (résistance électrique)
bit/s	Bits par seconde (débit binaire)
Hz	Herz (fréquence)
m	Mètre (longueur)
o	Octet (quantité de données numériques)
V	Volt (tension)

INTRODUCTION

0.1 Contexte et motivation

Le monde industriel tel que nous le connaissons aujourd'hui est voué à changer. Les progrès technologiques et les nouvelles attentes économico-sociales en font un secteur en constante évolution. Des changements notoires ont en effet été observés ces deux derniers siècles. L'industrie a ainsi déjà connu trois révolutions avec la mécanisation des procédés dès le XIX^{e} siècle, l'utilisation de l'électricité, entre autres, au début du XX^{e} siècle et plus récemment son informatisation vers la fin du XX^{e} siècle. Aujourd'hui, à l'ère du tout connecté, l'industrie s'apprête à franchir un nouveau cap avec l'incorporation de technologies reposant sur Internet. Le monde industriel entre dans sa quatrième révolution, c'est l'industrie 4.0. Cette initiative industrielle d'envergure repose principalement sur des technologies de l'Internet des objets appliquées au domaine industriel pour interconnecter les capteurs, les équipements industriels et les systèmes informatiques. Nous utiliserons dans ce mémoire la désignation anglophone « Industrial Internet of Things » (IIoT) pour se référer au concept de l'Internet industriel des objets.

Néanmoins, l'apparition de ces nouvelles technologies entraine certaines complications vis-à-vis de leur cohabitation avec les anciennes. Bien des entreprises disposent de systèmes de production existants qui ne sont pas adaptés à la communication via l'Internet. Elles n'ont pas de plateforme IIoT reliant des machines de production à des services infonuagiques à des fins de modélisation et d'analyse. Certes, des réseaux de communication sont déployés sur le plancher de production ainsi que dans les bureaux de supervision, mais l'utilité de ces réseaux de communication est davantage axée sur le contrôle et la gestion de la production. Les données générées et échangées servent à l'automatisation de la production et à la maintenance des machines. Il est donc essentiel de trouver des moyens permettant la cohabitation entre les différents acteurs industriels dans un souci de procéder à une transition douce et rentable vers l'avenir.

L'adoption d'une nouvelle approche où chaque élément est connecté avec son environnement, où les données sont accessibles à distance semble actée tant les avantages sont nombreux. Pourtant, l'implantation de ces nouvelles technologies n'est pas si évidente qu'il n'y paraît, nombreuses sont les entreprises qui sont réticentes à un tel changement. Accommoder une usine à l'industrie du futur demande souvent un investissement conséquent et peut nécessiter un niveau d'expertise assez élevé (Rüßmann et al., 2015). La multiplication des capteurs et des données échangées rend les systèmes beaucoup plus complexes et impose des changements drastiques d'organisation et de matériel au sein des usines. Il apparait que le risque est trop grand pour justifier un tel changement et les entreprises préfèrent continuer avec leur matériel actuel le temps d'atteindre la fin de leur cycle de vie et de rentabiliser leur précédent investissement. En effet, la durée de vie d'appareils tels que des automates programmables (PLC) peut être de plusieurs dizaines d'années. Les avancées technologiques sont si rapides que cette nouvelle révolution industrielle intervient quand les appareils déjà en place n'ont pas encore atteint leur durée d'amortissement, c'est pourquoi il est nécessaire de les intégrer et non simplement de les remplacer. Choisir la deuxième option reviendrait à se heurter à des obstacles financiers évidents (Calderón Godoy & González Pérez, 2018).

Cette réticence est d'autant plus marquée dans les petites et moyennes entreprises (PME). En matière d'investissements, ces dernières ont souvent une vision à courts et moyens termes. Il est aussi important de noter d'une manière générale qu'investir dans de nouvelles technologies est toujours risqué pour des PME (Moeuf, Pellerin, Lamouri, Tamayo-Giraldo, & Barbaray, 2017). Dans leur ouvrage, Matt et Rauch illustrent les réserves présentes à l'encontre d'une implantation de nouvelles technologies telles que l'IIoT au sein des PME puisque leur revue de littérature a révélé qu'aucune recherche sérieuse n'avait été faite à ce sujet avant 2017 (2020). Il faut toutefois mettre en exergue que les PME représentent une grande part de l'industrie, et ce sur tous les continents.

En revanche, la part grandissante accordée à l'informatique dématérialisée est indéniable. De ce fait, une phase de transition va devoir s'opérer, où les anciennes et les nouvelles technologies cohabiteront. Il n'est pour autant pas simple de concilier les différents protocoles

présents dans l'industrie. Chacun à sa propre structure et ils sont rarement compatibles entre eux. En effet, dans l'industrie, peu importe le secteur, la quasi-totalité des entités répond à des normes imposées par des organismes reconnus internationalement et doit respecter des contraintes bien spécifiques. Toutefois les normes qui régissent ces différentes entités sont rarement compatibles entre elles. Cela pose un réel problème d'interopérabilité dans l'industrie manufacturière actuelle où chaque objet tend à être connecté avec l'ensemble de son environnement (Zawra, Mansour, & Messiha, 2019).

0.2 Énoncé du problème

Encore aujourd'hui, un grand nombre d'équipements installés n'ont pas de connectivité à l'extérieur de l'environnement de production. Ces équipements, bien en place, reposent certes sur des technologies plus anciennes, mais elles sont très robustes et ont fait leurs preuves, à l'instar des bus de terrain. Ces derniers ne sont pas près de disparaitre et ont encore un long avenir devant eux. Garder ces technologies présente par conséquent un réel intérêt (Mamo, Sikora, & Rathfelder, 2017). Favoriser la migration de ces entreprises vers les nouvelles technologies de l'IIoT est donc un enjeu pour la modernisation de l'industrie manufacturière. Une approche prudente consiste à profiter des réseaux de communication industriels déjà en place et d'y ajouter des dispositifs IIoT en périphérie de ces réseaux (Krishnasamy, Varrette, & Mucciardi, 2020). L'infrastructure de communication existante est ainsi maintenue et rehaussée sans perturber le fonctionnement des systèmes de production.

Pour la connectivité avec le monde extérieur et en particulier avec des services infonuagiques, il faut recourir à une passerelle (*gateway*) IIoT. En réseautique, le rôle d'une passerelle est de transmettre et recevoir des données entre deux réseaux de communication. Dans le cas d'une passerelle IIoT, elle est placée en périphérie des réseaux de terrain et de contrôle. Les données en provenance de ces réseaux sont traitées par la passerelle puis transférées vers l'Internet. De cette façon, les entreprises conservent le contrôle sur la nature et la quantité de données à transférer à l'extérieur des lieux de production.

La passerelle IIoT étant reliée aux réseaux de terrain et de contrôle, elle doit aussi pouvoir transiger avec des systèmes utilisant différents protocoles de communication. La passerelle est également responsable du transfert des données reçues vers les services infonuagiques désignés. Ainsi, la passerelle doit être multi-protocole afin de répondre aux besoins des systèmes de production et pour pouvoir communiquer avec les services infonuagiques. Cette situation cause un problème d'ingénierie fort intéressant – comment concilier les caractéristiques disparates de ces protocoles de communication ? Le problème est exacerbé par le fait que les réseaux de terrain utilisent des interfaces matérielles (ports physiques) différentes. Enfin, la solution à ce problème doit comporter une combinaison judicieuse de composants matériels et logiciels.

0.3 Objectifs de recherche

L'objectif de cette recherche est de solutionner le problème de la conciliation des protocoles communication par la conception d'une passerelle IIoT multi-protocoles. Cette passerelle doit avoir l'habilité de communiquer avec des objets industriels et des services infonuagiques. De plus, elle doit être conçue à partir de matériels et logiciels « open source » afin de minimiser le coût de fabrication. Cette recherche vient donc répondre aux questions suivantes :

- Quelle est l'architecture de communication et quel est le rôle des passerelles industrielles ?

L'automatisation de la production nécessite une architecture de communication qui est à la fois rapide et efficace. Connaître cette architecture de communication, en particulier chez les PME, nous permettra de mieux cibler le rôle des passerelles et de leur emplacement logique parmi les réseaux de communication industriels.

- Comment transcrire les protocoles de communication ?

Les réseaux de terrain et de contrôle sont des réseaux en temps réel dédiés à l'automatisation des procédés de fabrication. Différents protocoles possèdent différentes exigences quant à la composition des trames de données binaires. En revanche, les protocoles utilisés par la communication Internet sont des protocoles de très haut niveau d'abstraction. Les données à

transférer sont encodées en mode texte adhérant à des styles qui sont proches des langages de programmation.

- Comment concevoir une passerelle IIoT ?

La conception d'une passerelle IIoT passe par un matériel adéquat satisfaisant les exigences électriques et mécaniques des ports de communication. Elle passe aussi par la maîtrise des outils de programmation qui réalisent correctement la transcription des protocoles de bas et de haut niveau. Bien orchestrer les interactions entre le matériel et le logiciel est un défi qu'il faut relever.

0.4 Plan du mémoire

Une revue de la littérature sera effectuée dans le premier chapitre. Nous observerons alors les changements instaurés par l'arrivée de nouvelles technologies et les possibilités que celles-ci engendrent. En outre, il sera également question d'observer les architectures de communication présentes dans l'industrie et les possibilités d'intégration des dispositifs IIoT. Nous présenterons également l'état actuel de la recherche et les solutions existantes.

Le chapitre 2 se concentre sur les technologies de communications. Dans un premier temps, il mettra en exergue l'industrie traditionnelle et les technologies déjà présentes en se concentrant tout particulièrement sur les bus de terrain. Par la suite, nous discuterons du fonctionnement des passerelles IIoT. Enfin, les différents protocoles qui seront incorporés à la solution proposée dans cette recherche et leur mode de fonctionnement seront étudiés en détails. Ces derniers, au nombre de quatre, sont Modbus, Canbus, MQTT et HTTP.

Le chapitre 3 présentera notre méthode de travail. Il introduira les composants sélectionnés pour la fabrication de la passerelle. Les divers montages agencés pour la réalisation des tests de performance seront également mis en évidence. Dans ce chapitre, nous aborderons également la réalisation du programme informatique. Ce programme est d'ailleurs accessible librement sur la plateforme GitHub à l'adresse https://github.com/ajacqu1/IIoT_Gateway.

Nous discuterons alors des bibliothèques utilisées, des fonctions à implanter ainsi que l'architecture générale du code Python et de ses avantages. Des explications quant au choix et à la configuration de la plateforme IIoT seront aussi apportées.

Enfin, dans le chapitre 4, nous pourrons observer les résultats et les performances de la passerelle IIoT. Ce chapitre sera également l'occasion de présenter en pratique le fonctionnement du programme et l'exécution des requêtes entre les différents acteurs de notre environnement expérimental.

CHAPITRE 1

REVUE DE LA LITTÉRATURE

1.1 Introduction

Le terme « Industrie 4.0 » est initialement apparu en 2011 Allemagne (Bassi, 2017) et reflète une industrie où les processus physiques sont surveillés et étudiés en temps réel afin d'en améliorer leurs performances. Ce terme est employé pour mettre en exergue sa continuité avec les trois précédentes révolutions que l'industrie a connues par le passé. La nouvelle vision apportée par cette quatrième révolution garde les acquis des précédentes en proposant de nouvelles approches stratégiques de gestion des données. Les informations sont directement fournies à l'utilisateur qui n'a plus besoin d'aller les recueillir de lui-même comme autrefois, les machines industrielles en deviennent connectées et autonomes. La mesure et la collecte de données sur ces appareils permettent de prendre des décisions prédictives et adaptatives en adéquation avec leurs besoins. Il s'agit d'un renouveau mondial, l'industrie 4.0 est également désignée par des appellations telles que « l'usine intelligente » ou encore « l'usine du future ». Pour répondre à ces nouveaux défis, des mouvements se mettent en place un peu partout dans le monde comme avec « l'Industrial Internet Consortium » aux Etats Unis (Lüder, Schleipen, Schmidt, Pfrommer, & Henßen, 2017).

Cette révolution implique un changement profond de l'industrie et de son mode de fonctionnement. De nouvelles technologies font leur apparition comme notamment « l'Internet des Objets » qui est un concept que l'on peut retrouver dans le monde anglophone sous la désignation de « Internet of Things » (IoT), la virtualisation des processus, la réalité augmentée, la fabrication additive (Lima, Massote, & Maia, 2019). Les systèmes présents se complexifient, le nombre de capteurs utilisés explose et les données échangées se décuplent. À terme, chaque objet sera conscient de son environnement et en mesure de communiquer avec autrui, les objets seront connectés, intelligents. D'après le rapport annuel de Cisco, le nombre d'appareils connectés sera, d'ici 2023, trois fois supérieur à la population mondiale, il y aura

en effet près de 29,3 milliards d'objets connectés par le biais du protocole Internet (IP). En outre, le nombre de connexions machines à machines (M2M) est estimé à 15 milliards pour 2023 (Cisco, 2020).

1.2 De nouvelles possibilités

L'interconnexion des appareils proposée par l'Industrie 4.0 permet d'élargir le champ des possibles. Nous sommes désormais en capacité de réaliser des opérations que les ressources d'autrefois ne permettaient pas. L'industrie 4.0 permet aux usines de mieux s'adapter aux demandes du marché et ainsi de produire mieux (Moeuf et al., 2017).

En outre, l'emploi de capteurs en tout genre est de plus en plus répandu. Ces derniers permettent de mesurer les données sensibles et contribuent à l'amélioration continue des performances d'un système. Ils peuvent par exemple aider à déterminer les origines d'un goulot d'étranglement au sein d'une chaine de production ou alors pointer certains dysfonctionnements et ainsi réduire les erreurs (Bassi, 2017). La nouvelle révolution technologique qui anime actuellement l'industrie permet non seulement d'améliorer l'organisation et les rendements d'une usine, elle entraine également l'apparition de nouveaux outils.

1.2.1 Internet Industriel des Objets

L'IoT, de l'anglais « *Internet of Things* », peut être considéré comme un écosystème où les différents objets physiques sont tous interconnectés par le biais d'Internet (Zawra et al., 2019). Nous devons ce terme à Kevin Ashton qui l'a utilisé pour la première fois en 1999. Le marché de l'IoT est actuellement très en vogue comme le laisse à présager sa grande inflation. Ainsi, si le revenu annuel des fabricants de matériel IoT s'élevait à 157 milliards de dollars en 2016, les estimations indiquent qu'il aura triplé en 2020 (Veneri & Capasso, 2018). D'après un récent rapport du cabinet McKinsey & Company, le marché de l'IoT représente au total près de 1,6

trillions de dollars en 2020, il est présumé atteindre entre 5,5 et 12,6 trillions de dollars d'ici la fin de la décennie (Chui, Collins, & Patel, 2021).

L'IoT est un monde en plein expansion avec un nombre d'objets connectés qui croît exponentiellement de jours en jours. Ces objets nous entourent déjà au quotidien, on les qualifie souvent de « *smart* » dans le monde anglophone. En français, on entendra alors souvent parler d'objet connectés ou d'objets intelligents. Les objets connectés sont présents dans à peu près tous les domaines envisageables, allant de l'électroménager avec un réfrigérateur intelligent par exemple à l'automobile avec les voitures intelligentes. L'IoT est très utilisé en médecine car il permet de procurer des soins à n'importe quel moment de la journée et en temps réel, d'effectuer des diagnostics et un suivi des patients à domicile.

Dans l'industrie, l'IoT permet bien évidemment de décupler les possibilités d'automatisation en entreprise et en usine. L'IoT dans l'industrie est d'ailleurs différenciée de l'IoT classique car les enjeux ne sont pas les mêmes. On parle dans ce cas de « l'*Industrial Internet of Things* » (IIoT). Les différences sont notables. Dans une usine, il est capital de s'assurer que les différents services fonctionnent en tout temps. Une panne temporaire, la plus infime qui soit, peut en effet avoir des conséquences économiques néfastes. A l'opposé de l'IoT classique, l'IIoT s'implante dans un environnement déjà en place, avec de nombreux appareils, souvent plus complexes que ceux du monde ordinaire, et fonctionnant avec divers protocoles. Par ailleurs, dans l'industrie la priorité sera plutôt mise sur la robustesse et les performances des systèmes, au détriment du confort de l'utilisateur. Enfin l'IIoT, et dans le milieu industriel de manière globale, un accent tout particulier sera mis sur la propriété intellectuelle et le partage des données, bien plus que pour un usage domestique (Veneri & Capasso, 2018).

La sécurité est un aspect essentiel dans l'Internet des objets puisque n'importe quel appareil a, potentiellement, la possibilité de se connecter à un réseau et d'échanger des informations, parfois sensibles, avec les autres appareils. Un immense accent est alors mis sur ce point. Cette considération est d'autant plus vraie pour l'IIoT, car le milieu industriel regorge d'appareils qui n'ont pas été conçus pour fonctionner avec Internet et qui sont par conséquent très

vulnérables (Veneri & Capasso, 2018). Développer une solution capable d'être robuste aux intrusions et aux actions malveillantes d'autrui présente toutefois un défi de taille qui requiert des connaissances très spécialisées qui dépassent le cadre de recherche de ce mémoire. Ainsi, nous avons choisi de ne pas traiter cet aspect dans notre recherche. Les considérations de sécurités ne seront donc plus mentionnées par la suite.

1.2.2 Architecture de l'IIoT

L'univers de l'IIoT est régulé par quelques règles qui permettent d'assurer son bon fonctionnement global. Dans un premier temps, il est important de noter que les réseaux industriels peuvent comporter plusieurs dizaines voire centaines d'appareils. Chaque appareil doit alors être identifié de manière unique au sein du réseau de sorte qu'il n'y ait aucune confusion. C'est souvent l'adresse IP ou *l'Uniform Resource Identifier* (URI) qui joue ce rôle. Ensuite, les appareils de l'IIoT doivent être en capacité de s'adapter d'eux-mêmes aux changements de conditions imposés par l'environnement ou l'utilisateur. Il est important également qu'ils puissent correctement se configurer lors de leur connexion au réseau. Il doit d'ailleurs être en mesure d'échanger des informations avec les autres appareils de l'IIoT, notamment sur l'état du réseau. Cela impose les différents appareils à utiliser des protocoles de communication respectant des contraintes d'interopérabilité (Divyashree & Rangaraju, 2018).

Plusieurs éléments différents constituent cet écosystème. Pour clarifier le fonctionnement, plusieurs modèles sont définis et permettent de répartir au mieux les éléments de l'IIoT en catégories suivant leurs fonctions. Il n'existe pas un seul modèle officiel mais bien plusieurs modèles qui ont été proposés par divers chercheurs et professionnels du milieu. Il est par exemple possible de trouver dans la littérature des modèles à trois couches, ou encore à cinq couches (Madakam, Ramaswamy, & Tripathi, 2015). C'est vers ce modèle que tendent à pencher des chercheurs de l'université de Pékin dans leur article de recherche sur les architectures de l'internet des objets (Miao, Ting-Jie, Fei-Yang, Jing, & Hui-Ying, 2010). La version à cinq couches est illustrée Figure 1.1 avec des exemples d'éléments les composants.

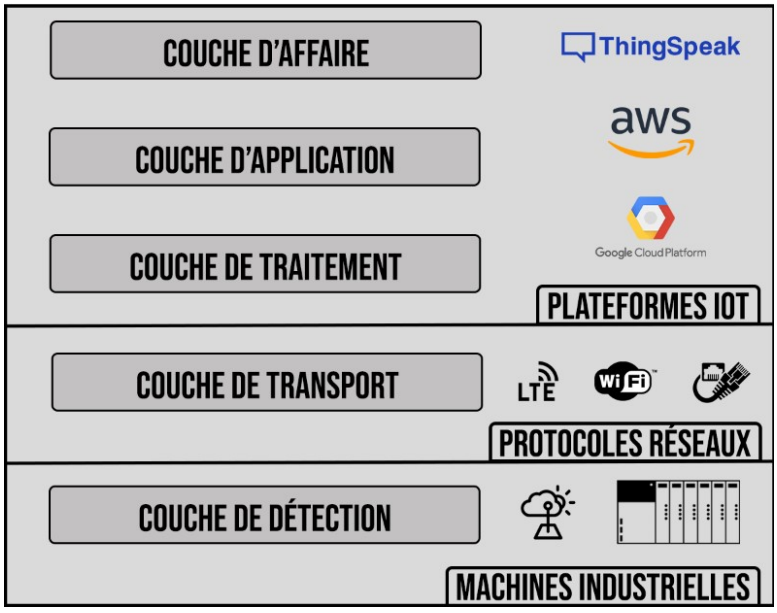


Figure 1.1 Architecture IIoT à cinq couches

Le Tableau 1.1 résume les fonctions de chaque couche de la Figure 1.1 en accord avec la description de Miao et al. (2010).

Tableau 1.1 Fonctions des couches d'une architecture IIoT

Nom de la couche	**Fonction**
Couche d'affaire	Gérer l'ensemble du Cloud et des données de l'utilisateur, réaliser des graphiques, etc.
Couche d'application	Effectuer toute sorte de traitement possible, plus poussés et plus complexes que ceux de la couche de traitement (IA, Big Data…), gérer la logistique et la sécurité.
Couche de traitement	Stocker l'information reçue, l'analyser et la traiter.
Couche de transport	Transmettre l'information entre la couche de détection et celle de traitement.
Couche de détection	Récolter les données du monde réel et les convertir en un signal numérique.

1.2.3 Constituants de l'IIoT

Différents acteurs de l'IIoT permettent de remplir les fonctions imposées par les différentes couches citées précédemment. Ainsi, les appareils intelligents seront associés à la couche de perception. Ensuite, il y a la couche de transport où se distinguent deux types de protocoles différents, remplissant chacun des fonctions distinctes. Il y a les protocoles réseaux et les protocoles d'application, qui eux servent d'interface entre les appareils IIoT et le réseau par lequel ils communiquent (Divyashree & Rangaraju, 2018). Enfin, les fonctionnalités des couches de traitement, d'application et d'affaire sont assurées respectivement par un « *Middleware* » et des logiciels d'application. Les plateformes IIoT sont des solutions regroupant le plus souvent ces couches, dans le cas où le traitement des données est léger. En revanche, si les données sont utilisées pour des opérations plus complexes alors des applications tierces seront considérées en aval de la plateforme pour traiter ces données (Guth, Breitenbücher, Falkenthal, Leymann, & Reinfurt, 2016).

Les Objets de l'Internet, ou objets intelligents, sont les composants de base, les objets physiques qui servent de frontière entre le monde réel et le monde numérique. Ils sont presque systématiquement équipés d'un microcontrôleur ainsi que d'un module de communication qui lui permet d'échanger avec le réseau. Le microcontrôleur a la responsabilité de commander les différents actionneurs et capteurs du système et de sélectionner les informations à partager.

1.2.4 Protocoles de l'IIoT

Une fois les données récoltées par les capteurs, ces données doivent être transmises à la plateforme afin qu'elles puissent être traitées. Pour ce faire, il est nécessaire d'être connecté à un protocole réseau, c'est via ce protocole que les différents acteurs sont en mesure de communiquer entre eux.

Il existe de nombreux protocoles de communication utilisés dans le domaine de l'IIoT. Dans un premier temps, nous pouvons bien évidemment penser aux protocoles réseaux populaires tels que Modbus, Canbus et DeviceNet (Galloway & Hancke, 2013). Il existe en revanche

également des réseaux moins connus mais parfois préférables à employer dans certains cas d'utilisation de l'IIoT. Ces protocoles respectent des contraintes de performances en étant notamment rapides, robustes, mais aussi légers pour fonctionner en congruence avec les systèmes embarqués et leurs ressources limitées. Parmi ces protocoles nous pouvons par exemple citer les protocoles BLE, NFC, 6LoWPAN, ZigBee, RPL, IPv6 (Divyashree & Rangaraju, 2018).

Quant aux protocoles d'application, il en existe une multitude, les plus connus étant AMQP, MQTT et XMPP. On retrouve également des protocoles reposant sur l'architecture REST tels que le populaire HTTP ou le protocole CoAP (Sharma & Gondhi, 2018).

Plusieurs études ont déjà été effectuées dans le but de présenter et de comparer ces protocoles. Ainsi, Nikolov a réalisé une étude comparative des protocoles MQTT, CoAP, HTTP et XMPP (2020), Gemirter, Şenturca et Baydere en ont fait de même pour les protocoles AMQP, MQTT et HTTP (2021). D'un point de vue architecturel, les protocoles MQTT et AMQP, d'un côté, fonctionnent suivant le même principe, et de l'autre les protocoles HTTP et CoAP présentent aussi des similitudes entre eux. Enfin, des travaux de recherche ont montré que les protocoles asynchrones de type MQTT ne peuvent pas remplacer les protocoles applicatifs tels que Modbus. Par contre, il est tout à fait convenable de compléter le protocole Modbus par le protocole MQTT dans le cas où le modèle de communication est de type évènementiel (*event-based*). Le désavantage majeur de l'agencement Modbus – MQTT provient de l'organisation des données qui est rigide chez Modbus. Il est nécessaire d'imposer une structure similaire au format préconisé par Modbus lors de la transmission des données par MQTT (Jaloudi, 2019).

1.2.5 Plateforme IIoT

Une plateforme IIoT est une application qui permet la gestion de l'ensemble des données collectées par les différentes machines du réseau. C'est en quelque sorte le logiciel qui va servir d'interface entre l'utilisateur et le Cloud. Il est possible de construire sa propre plateforme IIoT

mais il faut être bien renseigné sur le sujet et faire les bons choix car la solution peut vite devenir obsolète (Ravulavaru, 2018).

Le marché étant, comme nous l'avons vu, très prometteur, de nombreuses entreprises se sont lancées dans le domaine et proposent diverses solutions adaptées aux besoins de l'utilisateur. Il existe ainsi de nombreuses plateformes IIoT mises à disposition du public, parmi lesquelles nous pouvons par exemple citer AWS IoT, Google Cloud IoT, IBM Watson, Kaa IoT, Microsoft Azure ou encore ThingSpeak (Agarwal & Alam, 2020).

Chaque plateforme a ses avantages et ses inconvénients, et la sélection dépendra beaucoup des exigences et des goûts de l'utilisateur. Nous pouvons néanmoins classer les plateformes selon deux catégories, à savoir celles qui sont développées et supportées par des multinationales, des grands noms dans le domaine de l'informatique, et à l'inverse celles qui ne dépendent pas directement d'une entité et qui sont open source. Les plateformes IIoT de la première catégorie sont, de manière générale, plus simples à mettre en place puisque le support est important, elles sont également plus puissantes. En revanche, le coût entrainé sera conséquent et l'utilisateur ne sera pas maître du choix des composants utilisés. De plus, l'utilisateur doit accepter de ne plus avoir la mainmise sur ses données puisque ces dernières se retrouvent hébergées sur un serveur tiers appartenant à la compagnie fournissant la plateforme. Cela peut être très contraignant et se révéler un problème, a fortiori si les données de l'utilisateur sont sensibles et doivent rester privées. En effet, dans ce cas les données dépendent entièrement de la sécurité mise en place par l'hébergeur. L'avantage des solutions libres, en revanche, est de pouvoir s'affranchir de toute personne. Il est possible d'héberger le serveur sur son propre réseau et d'être l'unique détenteur des données téléchargées sur le service infonuagique, sans avoir recours à une entité tierce (Luis Bustamante, Patricio, & Molina, 2019).

1.3 Architectures de communication industrielle

L'interconnexion des objets permet de recueillir l'information de n'importe quelle machine et de centraliser les calculs. Les avantages d'une telle solution sont multiples, la mise en commun

des données permet d'avoir accès à bien plus d'informations, de les comparer, et réciproquement que plus d'appareils aient accès à ce nouveau savoir commun. De plus, la force de calcul peut alors être partagée entre les centres de calculs et n'a pas besoin d'être proche du terrain, dans un endroit où les contraintes physiques et de disponibilité peuvent être nombreuses. Nous obtenons ainsi une mise à disposition des données par le biais d'Internet. Ces données peuvent ensuite être stockées ou traitées par des logiciels. C'est ce qu'on nomme le *Cloud Computing* (CC), ou l'infonuagique en français (Qaisar, 2012). La société actuelle impose des fluctuations rapides du marché et la concurrence proposée entre les entreprises requiert de ces dernières une flexibilité maximale. L'emploi du CC et l'accès instantané à l'information décentralisée apportent des outils pour répondre à ces demandes (Rüßmann et al., 2015). Utiliser le CC pour un appareil peut être très bénéfique pour les raisons énoncées ci-dessus, mais en revanche cela peut aussi poser des problèmes lorsque l'accès au réseau est difficile ou que le réseau est en panne (Ravulavaru, 2018). De plus, à l'intérieur d'une usine, l'infonuagique n'est pas une solution adéquate pour le contrôle et la gestion des procédés. En effet, l'exigence de la communication en temps réel entre les équipements empêche le déploiement efficace des solutions de type CC. Pour pallier ces inconvénients, le traitement par *Edge Computing* (EC) est de plus en plus en vogue auprès des industriels (Krishnasamy et al., 2020).

Le EC est une architecture de technologie distribuée dans laquelle les données sont traitées à la périphérie du réseau local. Comme mentionné, le paradigme basé sur le CC n'est pas bien adapté aux besoins sans cesse croissants des entreprises. La bande passante limitée, une latence élevée et les perturbations imprévisibles du réseau Internet sont des problèmes qui nuisent à l'application du CC dans un contexte de production industrielle. En termes simples, le EC déplace une partie des ressources de stockage et de calcul plus près de la source de données elle-même. Plutôt que de transmettre des données brutes à un service infonuagique pour le traitement et l'analyse, ce travail est plutôt effectué là où les données sont réellement générées. Seul le résultat de ce traitement est renvoyé au service infonuagique pour le stockage à long terme et possiblement d'autres traitements plus exigeants. Ainsi, les entreprises conservent la

mainmise sur les données et peuvent ainsi anonymiser ces dernières avant de les transmettre sur le Cloud (Cao, Liu, Meng, & Sun, 2020).

Conceptuellement, le Cloud et le EC sont situés en périphérie et à l'extérieur des réseaux locaux de communication. La Figure 1.2, adaptée de l'ouvrage de Veneri et Capasso, montre une architecture de communication destinée à l'industrie manufacturière (2018).

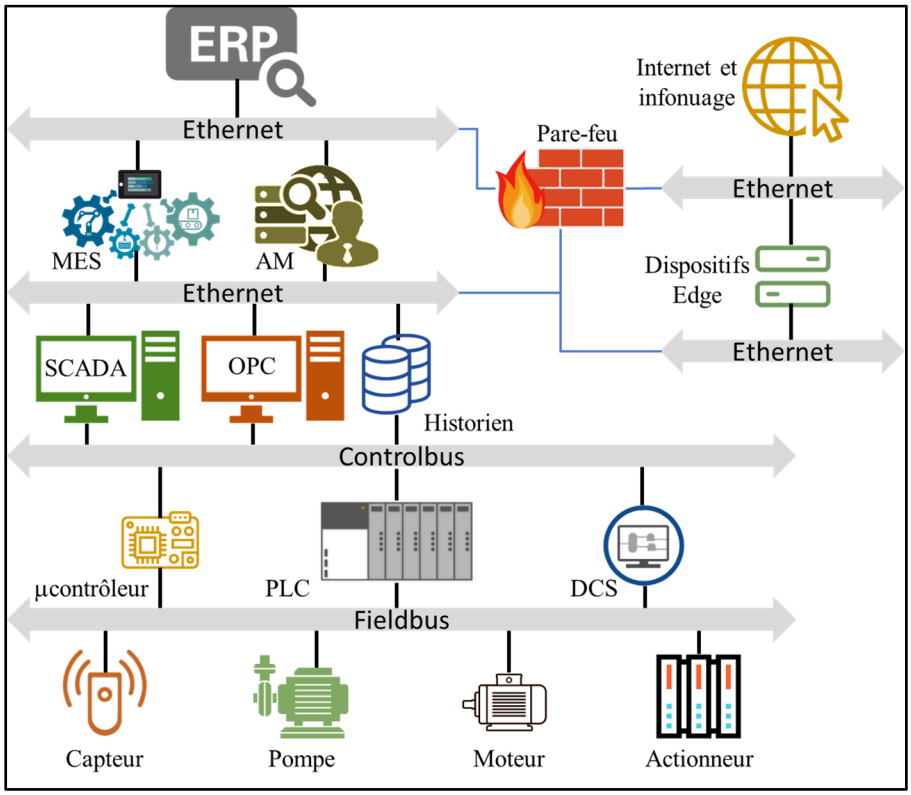


Figure 1.2 Architecture de communication industrielle

Les capteurs, les moteurs et les actionneurs sont au plus bas niveau de cette architecture. Ces équipements de production sont reliés à des bus de terrain (Fieldbus) permettant le transfert des données en temps réel. Des systèmes automatisés sont normalement installés afin de contrôler le fonctionnement de ces équipements. Dans le spectre des applications, le microcontrôleur convient pour des procédés simplistes. Pour des applications de plus grande complexité, l'automate programmable (PLC) avec ses régulateurs internes et ses nombreux

ports d'entrée-sortie, est le choix préféré des industriels. Le système distribué de contrôle et commande (DCS) est souvent déployé pour des applications critiques qui doivent avoir une bonne tolérance aux pannes. Dans l'architecture de la Figure 1.2, ces contrôleurs font le pont entre les dispositifs de production reliés au bus de terrain et les systèmes de supervision reliés à des bus de contrôle (*Controlbus*). On peut regrouper ces systèmes de supervision en deux grandes catégories, à savoir ceux destinés aux opérateurs et ceux destinés à d'autres systèmes. Le système de supervision et d'acquisition de données (SCADA) sert à gérer et présenter l'état de fonctionnement de l'ensemble des contrôleurs impliqués dans la production. Les opérateurs peuvent alors visualiser et contrôler l'état de fonctionnement des dispositifs de la chaîne de production. Ils peuvent ainsi consulter l'historique de fonctionnement des équipements par la fonction historien du SCADA. Enfin, les alarmes produites par différents dispositifs et contrôleurs sont acheminées vers des consoles du SCADA permettant par conséquent une gestion unifiée de la chaîne de production. L'autre catégorie des systèmes de supervision est représentée par l'*Open Platform Communications* (OPC) qui est destinée à être utilisée par d'autre systèmes. Les contrôleurs reliés aux bus de contrôle possèdent habituellement des interfaces matérielles et logicielles qui sont spécifiques au fabricant. Ces propriétés induisent souvent des problèmes d'interopérabilité, des contrôleurs de différents fabricants ne peuvent pas interagir entre eux. Le système OPC agit alors comme un traducteur permettant l'interopérabilité des contrôleurs sur le bus de contrôle.

Dans l'architecture illustrée en Figure 1.2, le SCADA est subordonné à un autre système appelé système de pilotage de la production (MES). Ce dernier assure l'ordonnancement des tâches de production et fait le suivi de la maintenance des équipements. En général, le MES réalise de concert avec le système de gestion des actifs (AM) qui tient à jour les registres de maintenance, les guides d'opération et le niveau de l'inventaire des pièces de rechange. À ce niveau hiérarchique, le réseau de communication est un réseau local de type Ethernet puisque le temps réel n'est plus exigé. Il en est de même pour le système de gestion intégré de l'entreprise (ERP). Les échanges de données entre l'ERP et le MES s'effectuent en temps différé et le réseau local est tout à fait suffisant. Les dispositifs du EC sont eux aussi reliés au réseau local de l'entreprise. Les données traitées par ces dispositifs proviennent directement

du système SCADA ou bien indirectement des réseaux de contrôle et de terrain par l'entremise de l'OPC. Naturellement, les données entreposées et traitées par les dispositifs Edge sont conservées à l'intérieur de l'entreprise.

L'architecture de la Figure 1.2 représente un déploiement complet des systèmes de contrôle et de gestion d'une entreprise de production d'envergure. Tous les éléments de communication et de traitement, incluant l'EC, sont montrés dans cette figure. Or, l'architecture de communication industrielle est souvent simplifiée chez les petites et moyennes entreprises (PME). Pour bien des PME, l'ajout d'un système ERP est un objectif à long terme. Le contrôle distribué DCS est couramment remplacé par des PLC qui sont moins coûteux et plus faciles à mettre en œuvre. L'ordonnancement des tâches et la gestion des actifs sont réalisés par des outils développés en interne. Les fonctions de MES et de AM sont en conséquence aussi intégrées à même ces outils développés.

La Figure 1.3 est une illustration de cette architecture simplifiée. Nous pouvons remarquer que les systèmes de contrôle et de gestion sont plus rudimentaires que ceux de la Figure 1.2. De plus, les éléments du EC sont remplacés par des dispositifs passerelles (*gateway devices*). Ces dispositifs réalisent certaines fonctions des systèmes OPC et EC qui sont inexistants dans l'architecture simplifiée.

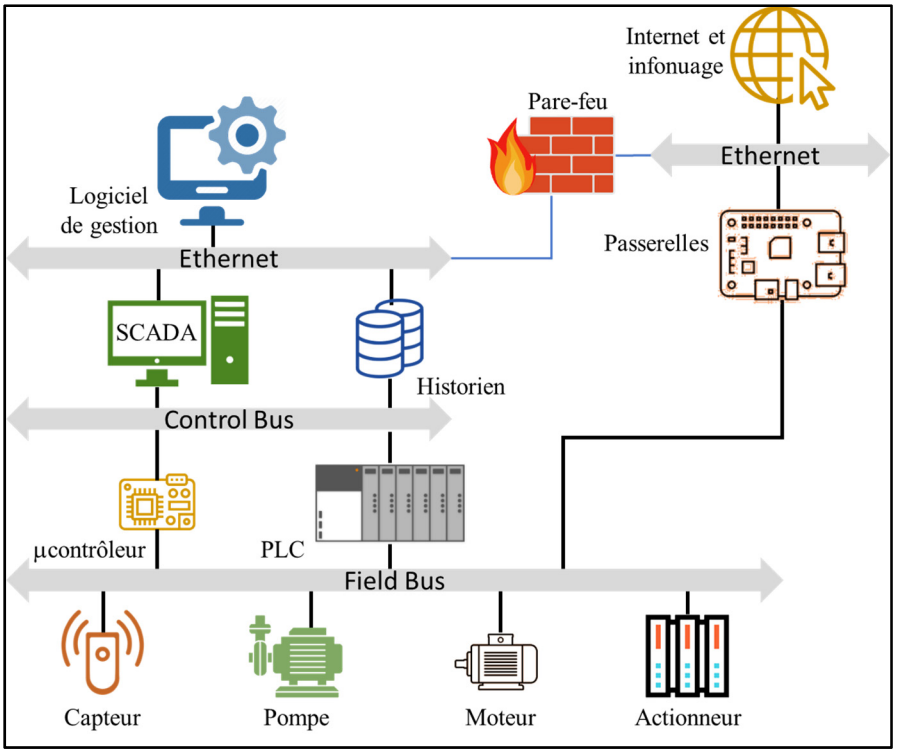


Figure 1.3 Architecture simplifiée de communication industrielle

1.3.1 Bus de terrain et passerelles

De par les enjeux gravitant autour de la transition vers les technologies du numérique, nombreux sont les chercheurs qui se sont intéressés aux bus de terrain et à leur développement futur. L'industrie 4.0 ne doit pas simplement intégrer les nouvelles technologies et assurer une compatibilité avec les anciens systèmes, elle doit par la même occasion garantir des performances stables et proposer une intégration simple (Zawra et al., 2019). Les passerelles sont des outils remplissant cette fonction. Il s'agit d'appareils physiques ou de plateformes virtuelles servant de point de connexion entre le serveur Cloud et les appareils de terrain. Les passerelles IIoT dédiées à l'industrie existent mais demeurent toutefois peu variées (Martinez et al., 2017).

Les passerelles IIoT ne sont pas toutes conçues sur base de licence libre. Le plus souvent, cela est dû à l'emploi de protocoles propriétaires. C'est, entre autres, le cas des infrastructures

opérant sous Profibus, le bus de terrain de l'entreprise Siemens. Nous pouvons par exemple citer le travail de Zhou, Li, Liu et Lin au travers duquel une passerelle IIoT pour le protocole Profibus-DP a été pensée (2016).

1.3.2 Passerelles de conversion de bus de terrain

Ce type de recherche a débuté il y a déjà plusieurs années, nous pouvons par exemple citer Xie et Gao qui ont réalisé une passerelle de conversion pour bus de terrain (2011). Cette passerelle permet la prise en charge de protocoles tels que Modbus RTU ou BACnet et leur conversion en Modbus TCP. Il ne s'agit pas d'une passerelle IIoT car l'information n'est pas destinée à être transmise sur Internet, en revanche le processus reste similaire. En outre, la conversion entre bus de terrain est un sujet de recherche important pour leur interopérabilité et leur intégration dans l'industrie à venir. Néanmoins, les moyens évoluent très vite ces dernières années et les solutions proposées antérieurement tendent à être rapidement obsolètes, c'est pourquoi nous ne tiendrons compte que des travaux ultérieurs à 2015.

Il est possible de trouver des passerelles de conversion de protocoles industriels ayant une approche didactique et visant à entrainer et perfectionner les capacités des étudiants dans l'automatisation industrielle (Cordeiro, Costa, Pires, & Foito, 2018). Ce travail offre un moyen de rendre compatibles des machines fonctionnant sous Modbus avec d'autres opérant sous Profibus.

Shen, Fan et Yan ont développé une passerelle basée sur l'architecture OPC UA et permettant l'acquisition de données au travers des protocoles Zigbee, BACnet et KNX (2016). Dans la même optique, Cupek, Ziebinski et Drewniak ont aussi opté pour une architecture OPC UA. Ce travail repose cette fois ci sur l'intégration des réseaux CAN (2017). Ye et Lei se sont attelés à la conversion du protocole Profibus-DP vers le protocole sans fil Zigbee dans un souci de développer un réseau industriel sans fil et de s'affranchir des contraintes imposées par les réseaux filaires (2016).

De leur côté, Wei, Xijun, Wenxia et Ruitao ont établi une passerelle de conversion entre Modbus et Profinet, l'homologue de Profibus dans sa version Ethernet (2016). Le contrôleur utilisé est basé sur une puce Siemens. Xu, Fei et Minrui ont échafaudé une passerelle comprenant trois protocoles, à savoir Modbus, Profibus et un réseau industriel sans fil (2016). Leur interopérabilité repose sur l'architecture de cette passerelle qui implante un format de données commun et un système de cartographie qui permet d'établir des correspondances entre les données propres aux protocoles et les données communes. Nous pouvons aussi mentionner Kim, Kim, Lee, Moon, et Jeon (2016). Afin de rendre les communications au sein des véhicules plus performantes, ils ont discuté d'une passerelle de conversion entre Canbus et un réseau Ethernet en temps réel.

1.3.3 Solutions virtuelles

Par ailleurs, il est important également de mentionner les solutions virtuelles. Pour créer un environnement d'expérimentation permettant l'étude de ces technologies, les chercheurs ont souvent recours à des simulateurs numériques. Il est en effet possible de trouver des simulateurs de bus de terrain. C'est par exemple le cas du travail de Viegas, Postolache, Girao et Pereira (2016) qui ont réalisé un programme permettant d'émuler le protocole Foundation Fieldbus. Il s'agit d'ailleurs d'une amélioration d'un simulateur conçu quelques années auparavant (Mossin, Pantoni, & Brandao, 2009). Ces programmes permettent l'entrainement de personnel de manière accessible et peu onéreuse.

Ce genre de maquette virtuelle peut aussi être réalisé avec le logiciel LabVIEW. On dénote par exemple une instance offrant la possibilité de simuler le protocole Modbus TCP (Mangkalajan, Koodtalang, Sangsuwan, & Pudchuen, 2019). Une passerelle IoT virtuelle a été réalisée en 2019 (Dirgantoro, Nwadiugwu, Lee, & Kim, 2020). Cette dernière permet de recueillir des informations d'un bus de terrain redondant, transmettant les données en parallèle via les protocoles Modbus et Canbus. L'instance Modbus est dite principale et les données Canbus sont analysées en cas de détection d'une erreur lors de la transmission Modbus.

Au vu des précédentes recherches énumérées ci-dessus, nous pouvons aisément observer l'intérêt apporté aux bus de terrains et à leur intégration. Notre contribution sera donc de prendre part à cette recherche en développant une passerelle IIoT multi-protocoles abordable. Cette passerelle aura également une architecture permettant aux usagers la possibilité d'incorporer d'autres protocoles industriels et protocoles de communication, afin de permettre un suivi et une amélioration continuelle du projet.

1.4 Conclusion

Ainsi, nous avons eu l'occasion d'observer un monde industriel en pleine révolution dans lequel les modes de fonctionnement traditionnels s'apprêtent à être totalement chamboulés par l'arrivée des nouvelles technologies. L'industrie devient une industrie connectée à tous les niveaux, c'est-à-dire un univers où chaque élément est interconnecté avec les autres, conscient de son environnement et évoluant en symbiose avec ce dernier. L'IIoT est un concept qui joue un rôle majeur dans cette nouvelle vision de l'industrie, et offre de nouvelles façons d'appréhender la gestion et le traitement des données.

À l'heure actuelle, l'industrie n'est pas en mesure d'adopter en masse ces nouvelles technologies, les diverses machines industrielles ne sont pas pensées pour fonctionner de la sorte. Il existe en effet une industrie traditionnelle déjà bien en place, avec des architectures de communication qui lui sont propres. On retrouve alors plusieurs niveaux de communications associés à des types de données et d'objets différents. Parmi ceux-ci, les Fieldbus sont les protocoles présents au plus près du terrain et contrôlant des objets tels que des actionneurs ou des capteurs. Ils assurent des échanges en temps réel. Un accent est mis sur leur rapidité et leur robustesse, la structure des données échangées n'est alors en aucun point compatible avec des protocoles de communication de haut niveau utilisés par des applications telles que des services infonuagiques. L'acheminement et la conversion des données entre ces deux environnements peuvent alors être particulièrement fastidieux et nécessitent de mettre en relation de nombreux acteurs. On note alors l'apparition d'architecture simplifiées mettant en scène des appareils destinés à réaliser un pont entre les bus de terrain et les objets de l'Internet. Il s'agit des

passerelles IIoT qui jouent un rôle important dans l'implantation des fonctions IIoT au sein des entreprises et en particulier des PME.

La revue de la littérature effectuée nous a permis de nous rendre compte des enjeux posés par la transition vers le numérique et d'observer les solutions existantes permettant de répondre à ces nouveaux défis.

CHAPITRE 2

TECHNOLOGIES DE COMMUNICATION INDUSTRIELLES

2.1 Introduction

Pour être en mesure d'effectuer des équivalences entre protocoles, des transcriptions permettant de passer de l'un à l'autre, encore faut-il comprendre leur mode de fonctionnement. Nous allons donc examiner les bus de terrain, leur rôle et leur fonctionnement. En outre, nous prêterons également une attention toute particulière aux passerelles IIoT et aux méthodes de conceptions employées.

Le contenu de ce chapitre aidera à répondre à la question de recherche « Comment transcrire les protocoles de communication ». Cette fonction de transcription joue un rôle important dans la passerelle IIoT multi-protocole à concevoir. Les protocoles ayant été retenus sont Modbus (section 2.6.1), Canbus (section 2.6.2), HTTP sous REST (section 2.7) et enfin MQTT (section 2.8). Les protocoles Modbus et Canbus sont chargés du transfert de données en temps réel entre les dispositifs interconnectés. HTTP et MQTT sont des protocoles de haut niveau utiles pour la communication avec des services infonuagiques. Les caractéristiques importantes de ces différents protocoles sont abordées dans ce chapitre.

2.2 Bus de terrain

Les bus de terrain sont apparus de manière conséquente dans les années 80. Nous pouvons cependant considérer que le premier bus de terrain est celui décrit par la norme MIL-STD-1553, en 1973. Ce protocole est utilisé dans les technologies avioniques et spatiales, notamment miliaires, et a été développé afin de réduire la masse des câbles embarqués (Zurawski, 2017). Comme leur nom le laisse présager, les bus de terrain sont utilisés au niveau du terrain, c'est-à-dire directement là où les machines et robots opèrent.

Les Fieldbus permettent ainsi de relier les automates aux composants de terrain, tels que des capteurs et des actionneurs. Ils créent un réseau industriel en reliant différents appareils qui se partagent ainsi un seul et même dispositif de transmission. Ces technologies ont par conséquent aidé à réduire drastiquement le nombre de câbles déployés dans des systèmes plus ou moins complexes. C'est d'ailleurs pour ce genre de considérations que les bus de terrain ont été pensés et développés en premier lieu. Autrefois, il était nécessaire d'avoir un câble dédié pour relier deux composants entre eux, par un système de point à point, voire même plusieurs câbles si différents types d'informations étaient échangés. Par exemple, dans un système comme une voiture, les longueurs cumulées de câbles pouvaient atteindre plusieurs kilomètres (Bera, 2021). Aujourd'hui, avec des fonctionnalités de plus en plus poussées et des capteurs se décuplant, une telle installation serait impensable. L'emploi des Fieldbus permet donc de pallier ce problème.

Au cours du temps, plusieurs définitions ont été attribuées à ces dispositifs, la définition de référence est énoncée par la norme IEC 61158-2:2003, « *A fieldbus is a digital, serial, multidrop, data bus for communication with industrial control and instrumentation devices such as – but not limited to – transducers, actuators and local controllers.* » (International Electrotechnical Commission, 2003, p. 15). On peut ainsi définir un bus de terrain comme étant un bus de données numérique, sériel, multipoint, destiné à la communication entre instruments et dispositifs de contrôle tels que des actionneurs, des contrôleurs, des capteurs. Le National Institute of Standards and Technology (NIST) s'est également attelé à donner une définition formelle à ces dispositifs et stipule les mêmes propriétés, « *A digital, serial, multi-drop, two-way data bus or communication path or link between low-level industrial field equipment such as sensors, transducers, actuators, local controllers, and even control room devices.* » (Stouffer, Pillitteri, Lightman, Abrams, & Hahn, 2015, p. 125).

Plusieurs facteurs permettent de mettre en avant leurs avantages et la nécessité de l'emploi des bus de terrain. Ces derniers sont un bon moyen d'obtenir de très bonnes performances avec des ressources de terrain limitées. Chaque bus de terrain a ses caractéristiques propres. Il existe

notamment de nombreuses topologies de réseau différentes. Parmi elles nous retrouvons les architectures en étoile, en ligne, en guirlande, en maillage, en anneau ou encore en arbre.

2.3 Classification

En vue d'avoir un seul et unique modèle de référence et ainsi de simplifier la catégorisation des différents procédés, l'*International Organization for Standardization* (ISO) a décidé dans les années 1970 de présenter un modèle normalisé, le modèle OSI, pour « *Open Systems Interconnection* ». Parfois appelé pyramide OSI, il s'agit donc d'un modèle permettant de classer les différents protocoles de communications en plusieurs catégories, ou couches, au nombre de sept. Il vise aussi à garantir une certaine compatibilité entre les différents systèmes par le biais de critères et exigences (Zimmermann, 1980). Les sept couches du modèle OSI sont présentées sur la Figure 2.1.

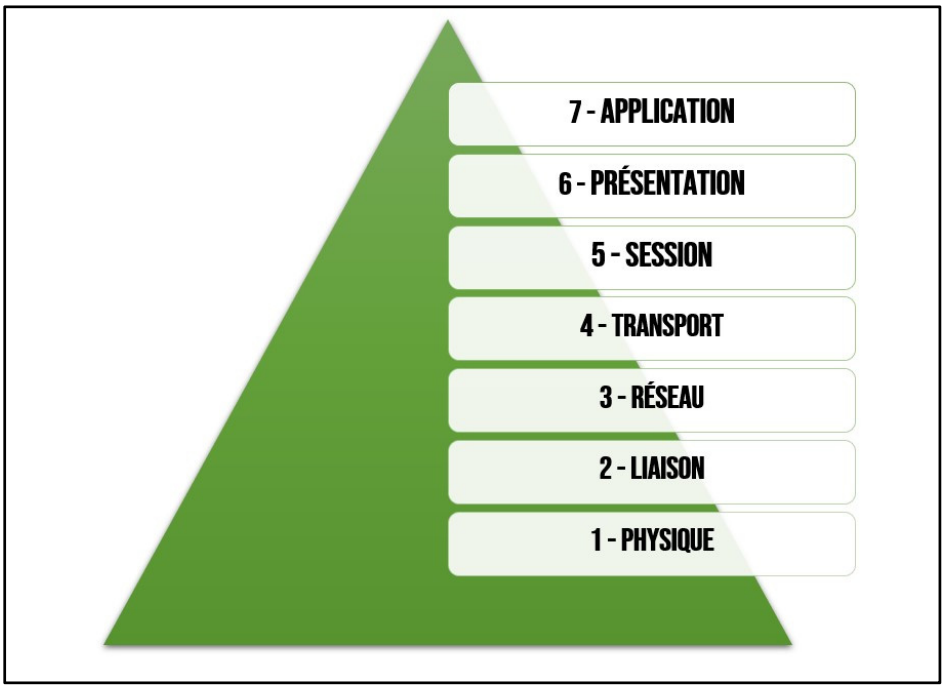


Figure 2.1 Représentation graphique des sept couches du modèle OSI

Avant la mise en place d'un tel concept, chaque système avait sa propre façon de fonctionner et ces derniers n'étaient souvent pas compatibles entre eux, cela limitait donc fortement leur

usage et leur capacité. Le contenu de la Figure 2.1 est brièvement présenté dans le Tableau 2.1 (Zurawski, 2017).

Tableau 2.1 Description des sept couches du modèle OSI

Nom de la couche	**Fonction**
1 – Physique	Échanger des données entre actionneurs et récepteurs. Les données sont brutes, les communications sont de l'ordre du bit.
2 – Liaison	Assurer la transmission de données entre deux nœuds et la formation de trames de données.
3 – Réseau	Acheminer les données sous forme de paquets. On retrouve souvent le protocole IP.
4 – Transport	Transmettre l'information à la cinquième couche. On retrouve souvent les protocoles TCP ou UDP.
5 – Session	Assurer l'identification et la connexion des appareils.
6 – Présentation	Interpréter et chiffrer les données pour la septième couche.
7 – Application	Assurer un point d'accès aux services réseaux, permettre une interaction entre ces services et les applications et offrir une interface utilisateur.

Avec cette classification, deux appareils de la même couche peuvent communiquer entre eux en utilisant le même protocole. Il est donc important de mentionner ce modèle pour évaluer la place des bus de terrain dans l'industrie. Les bus de terrain n'occupent pas toutes les couches de la pyramide, en général ils se trouvent au niveau des couches 1, 2 et 7. Le système OSI à sept couches est très complet, très détaillé. C'est pour cela que beaucoup d'installations ayant des bus de terrain fonctionnent en réalité seulement avec les couches physique, liaison de données et application. La norme IEC 61158-1:2019 sur les bus de terrain recommande l'incorporation des couches 3 et 4 au sein de la couche 2, et les couches 5 et 6 dans la couche 7 (International Electrotechnical Commission, 2019).

Les couches supérieures du modèle OSI supportent le traitement de grandes quantités de données alors que les bus de terrain étaient plutôt destinés à des petites quantités, avec l'accent mis sur le temps réel et la réactivité. Il faut toutefois noter que ces dernières années, avec les progrès effectués, les bus de terrain peuvent avoir des bandes passantes mesurables aux réseaux locaux (LAN). Par ailleurs, les réseaux Ethernets se démocratisent de plus en plus. Les technologies deviennent plus performantes, plus complexes et sont donc désormais en mesure d'assurer plusieurs rôles. La frontière entre les différentes couches du modèle semble être de plus en plus mince, certains protocoles opérants désormais souvent sur plusieurs couches (Rojas & Garcia, 2020).

2.4 Mode de fonctionnement

Il y a de nombreux bus de terrain sur le marché, ce qui induit une concurrence forte. Pour qu'un bus trouve sa place dans le monde industriel, il faut donc qu'il soit performant, cela passe en grande partie par ses aptitudes en termes de rapidité. Ainsi, on essaie en général, pour le transfert de données, d'avoir des messages transmis plutôt courts, ce sans quoi la gestion du temps réel peut vite devenir complexe. Par ailleurs, les protocoles doivent être optimisés au maximum car les ressources sur le terrain sont limitées.

Bien que chaque protocole ait son fonctionnement propre à lui, il existe certains points communs intrinsèques à ceux-ci qui découlent de lois physiques. C'est le cas de la relation entre la longueur d'un bus et le débit de transfert de données qu'il est capable de supporter. Plus l'on voudra une vitesse de transfert élevée, plus la distance séparant les appareils devra être faible, et inversement. Les bus de terrain sont des protocoles normalisés et doivent donc respecter certaines exigences vis-à-vis de ces caractéristiques. À titre indicatif, pour le protocole Profibus, si l'infrastructure du réseau atteint 1 000 mètres, le débit maximum toléré sera de 187,5 kbit/s. De la même manière, si on veut un gros débit de transmission, de l'ordre de 12 Mbit/s, le segment du bus ne pourra pas dépasser 100 mètres (Koch & Lueftner, 2019).

2.4.1 Coordonnateur/nœud

Le mode de fonctionnement le plus connu est celui du coordonnateur/nœud (*master/slave*). Le coordonnateur agira comme un cerveau, et il contrôlera toutes les actions des autres appareils du réseau, des autres nœuds. Ce processus peut se référer à la notion d'attente active (*Polling*), c'est-à-dire que le coordonnateur va sonder les différents appareils du bus. Le principe est illustré sur la Figure 2.2. Il s'adresse aux appareils désirés en mentionnant leur identifiant unique et ces derniers lui répondent par le biais d'une requête lorsqu'ils sont concernés. Il peut cibler un type de données et les appareils lui renvoient les informations concernant cette requête précise s'ils ont un lien avec la donnée sondée. Le *Polling* est adapté pour les trafics cycliques mais non les acycliques. Cette méthode ne sera alors pas recommandée dans certains cas. En outre, le nœud n'a aucun pouvoir, il n'est pas acteur et ne peut pas agir, seulement réagir. Les appareils tels que les capteurs peuvent tout de même procéder à une demande de requête, le coordonnateur envoie alors une requête pour savoir de quoi il en retourne. Il peut aussi accorder temporairement l'accès au bus à un nœud, la durée est forcément limitée et se finit par un délai d'expiration (Zurawski, 2017).

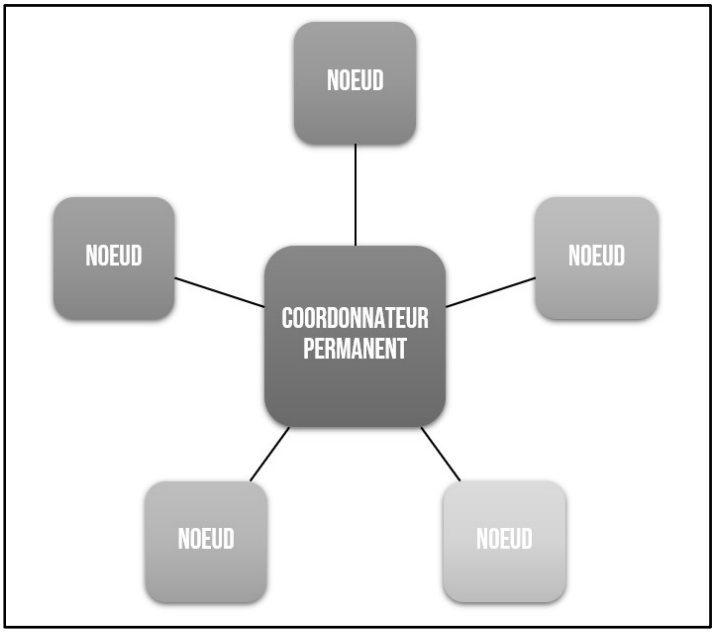


Figure 2.2 Schéma de principe d'un bus avec un unique coordonnateur

2.4.2 Multi-coordonnateurs

Une autre approche consiste à avoir plusieurs coordonnateurs qui administrent le bus à des moments distincts. Il ne peut en revanche n'y avoir qu'un coordonnateur à la fois pour éviter des problèmes de conflits. Pour répondre à cette condition, la technique du jeton (*token*) est adoptée. Un appareil ne pourra alors administrer le bus que lorsqu'il est en possession du jeton.

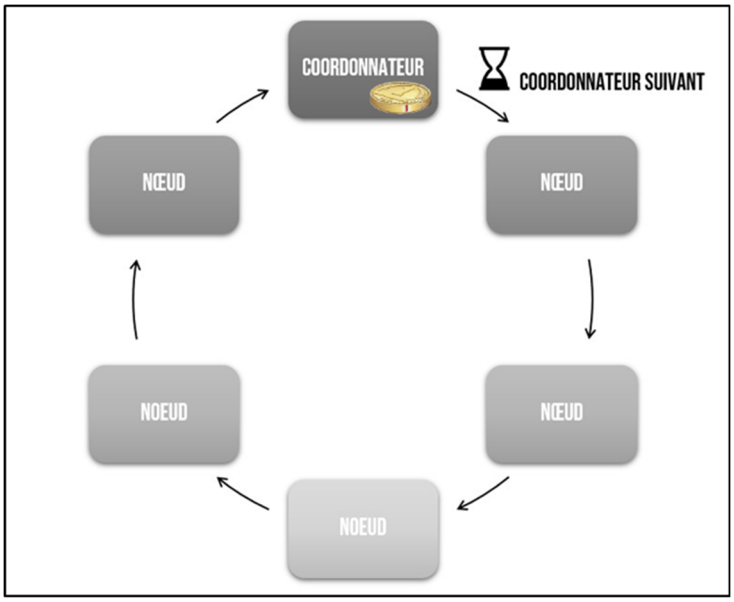


Figure 2.3 Schéma de principe d'un bus à multi-coordonnateurs avec jeton

Si le coordonnateur actuel n'a rien à transmettre, il va alors passer le jeton à l'appareil suivant qui sera à son tour le nouveau maître du bus. L'organisation d'un bus multi-coordonnateurs est présentée Figure 2.3. Pour s'assurer qu'un appareil ne monopolise pas le jeton, ce qui pourrait entrainer un blocage du bus, il y a un temps d'expiration défini au bout duquel le jeton doit être transmis si aucune action n'a été effectuée (Zurawski, 2017).

2.4.3 Accès universel

Enfin, les autres modes de fonctionnement existants sont ceux qui font abstraction du principe du coordonnateur comme décrit Figure 2.4. Il est possible de trouver des bus de terrain constitués d'appareils transmettant des requêtes dès qu'ils le désirent, à des moments aléatoires. On parle alors de « *Random Access* ». Son nom est assez explicite quant à son fonctionnement. Cela peut être intéressant pour les systèmes spontanés. Il peut toutefois avoir des problèmes de collisions. Il existe alors des structures à détection de collisions comme le CSMA-CD (*Carrier Sense Multiple Access with Collision Detection*). Cette méthode reste relativement peu utilisée mais il arrive par exemple que certains systèmes CAN (*Controller Area Network*) l'utilisent (Zurawski, 2017).

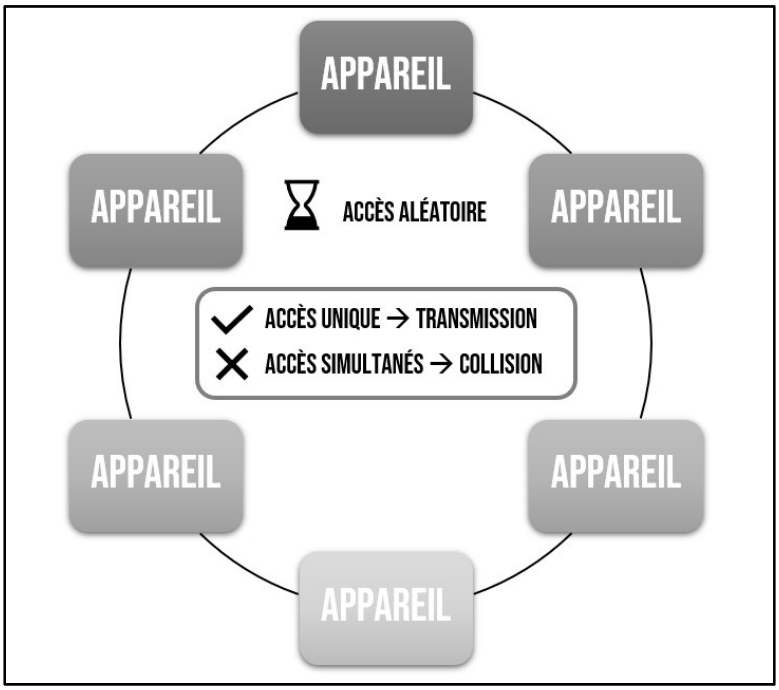


Figure 2.4 Schéma de principe d'un bus à accès universel

2.5 Fonctionnement des passerelles IIoT

Les problèmes de compatibilité, ou d'interopérabilité, entre deux protocoles industriels sont quelque peu similaires à ceux des langages humains. Pour que deux entités se comprennent, il

est parfois nécessaire d'avoir recours aux services d'un traducteur qui va établir une concordance entre chaque terme employé. Deux systèmes interopérables sont deux systèmes qui ont des interfaces en mesure de se comprendre, d'échanger des messages et de partager des fonctionnalités communes (Rojas & Garcia, 2020).

Les passerelles IIoT vont alors être les éléments intermédiaires entre l'usine et les services Internet, elles vont agir comme des interfaces, elles occupent donc un rôle de traducteur et sont en mesure d'établir un pont entre plusieurs protocoles. Toutes les informations transitent par la passerelle, ainsi si elle utilise un moyen de conversion non approprié, on pourra observer des instabilités au niveau de la transmission des données, cela peut alors entrainer des complications qui peuvent compromettre l'automatisation de l'installation. C'est ce qu'expliquent Dietrich, May, Hoyningen-Huene Mueller et Fohler à travers leur expérimentation entre le protocole CAN et un réseau Ethernet industriel (2017).

Il arrive souvent que les protocoles ne soient pas simplement différents de par les langages utilisés, mais que les architectures complètes soient inassimilables, c'est-à-dire que les façons de penser et de procéder des protocoles sont discordantes. Il existe ainsi tout un panel de passerelles différentes, incluant différents niveaux de complexité. Dans le cas des bus de terrain, l'emploi de telles solutions est obligatoire. Les protocoles de terrain ont été pensés pour des connexions rapides, en temps réel avec des volumes d'informations faibles, les données sont presque brutes et ne sont pas adaptées aux protocoles de transport par lesquels communiquent les plateformes IIoT.

Il est possible de distinguer majoritairement deux types de structures différentes, les autres structures existantes étant simplement des dérivées. On observe ainsi d'un côté les passerelles communiquant directement avec les PLC par l'intermédiaire des protocoles de terrain, et de l'autre les passerelles implantant une architecture OPC. Chacune de ces structures présente des avantages comme des inconvénients (Veneri & Capasso, 2018).

Les passerelles directement reliées aux bus de terrain présentent des meilleurs mécanismes d'acquisition des données et sont à l'origine de bases de données plus précises. Par ailleurs, aucun autre intermédiaire n'est rajouté ce qui permet de limiter au maximum les causes potentielles de dysfonctionnement. Comme cette passerelle n'instaure que deux interfaces, la mise en tampon des données est plus fiable et plus robuste. En revanche, il est nécessaire d'avoir un connecteur différent pour chaque protocole présent dans l'installation. Cela peut rapidement entrainer des complications, notamment lorsque ce nombre commence à être conséquent.

En ce qui concerne les passerelles implantant une architecture OPC, il est nécessaire d'instaurer des interfaces supplémentaires. L'architecture la plus répandue est OPC-UA (*Open Platform Communication – Unified Architecture*). Les normes OPC sont des normes permettant de connecter de nombreux systèmes industriels provenant de différents constructeurs et entreprises. Il s'agit des normes de communication les plus répandues et les plus acceptées. Elles offrent également la possibilité d'avoir une interface plus facile à utiliser que celles des automates traditionnels (Gang et al., 2020). Une fois le serveur OPC implanté, le fonctionnement de la passerelle s'en voit grandement simplifié. Cette dernière possède seulement une interface avec le serveur et peut alors faire abstraction de tous les bus de terrain en amont. Dans cette configuration, c'est le serveur OPC qui remplit la fonction de traducteur pour les différents protocoles de terrain. Les données échangées par la passerelle présentent alors toutes la même structure. En revanche, c'est le serveur OPC qui sera chargé de la collecte de tous les bus de terrain, s'il rencontre des problèmes de connexion ou des coupures le flux de données se verra paralysé. De plus, en cas de pertes de données, il est plus compliqué de trouver la source de l'incident (Veneri & Capasso, 2018).

2.5.1 Techniques de conception

Il est important de noter qu'il existe déjà diverses passerelles IIoT sur le marché, qu'elles soient industrielles et vendues par des fabricants reconnus ou bien que ce soit des œuvres réalisées par des chercheurs, des étudiants ou tout simplement des passionnés. Dans son tutoriel

expliquant comment bien choisir sa passerelle IIoT, Parikh nous explique qu'il y a essentiellement quatre types de passerelles différentes (2018). Les distinctions effectuées sont résumées dans le Tableau 2.2.

Tableau 2.2 Les différents types de passerelles IIoT

Type de passerelle	Remarques
Conception de A à Z	Le choix incombé à l'utilisateur est total, en revanche la complexité est grande.
Conception à partir de plateformes libres	La conception est relativement facile mais est limitée à de petites applications.
Conception fixe	Tout est déjà effectué et configuré par le vendeur, cependant la flexibilité est moindre.
Conception modulaire	Le corps est préconçu et le reste des composants est modulable, ils se présentent sous la forme de kits configurables. C'est la solution que l'auteur recommande.

Nous allons omettre les plateformes fabriquées entièrement par l'utilisateur du fait de la complexité de la tâche.

2.5.1.1 Conception fixe

Dans cette catégorie nous retrouvons donc les produits prêts à l'emploi. Nous pouvons par exemple mentionner l'entreprise Dell et leur gamme de passerelle IIoT « *Dell Edge Gateway 5000 series* ». D'après la fiche technique, cet équipement intègre des protocoles tels que BACnet, Modbus, Canbus ou encore Zigbee et 6LoWPWAN (Dell Inc., 2017).

Pour ce qui est des protocoles propriétaires, il faudra en général se fournir directement chez le fabricant. Par exemple, pour Profibus, qui est un bus de terrain propriétaire développé par

Siemens, il faudra plutôt s'orienter sur un appareil tel que le « *SIMATIC Cloud Connect 7* », de référence 6GK1411-5AC00 ("Industrial IoT Gateways SIMATIC CloudConnect 7," n. d.).

2.5.1.2 Conception modulaire

La conception modulaire requiert un peu plus travail de recherche qu'une solution déjà préconçue en amont, mais cela permet une meilleure personnalisation. Suivant la complexité des protocoles implantés et leur nombre, le prix des passerelles peut rapidement grimper. Cette tendance est d'autant plus vérifiée s'il s'agit de protocoles constructeurs. D'après Parikh, les leaders dans ce domaine sont Phytec, Volansys et Digi (2018). Tous ces sites proposent alors des solutions de passerelles sur devis, avec des prix très variables suivant les performances et les protocoles supportés. Sur le site de Volansys, un pack de débutant avec une passerelle IIoT est proposé pour une somme inférieure aux passerelles mentionnées dans la section précédente ("Modular IoT Gateway," n. d.). Avec cette passerelle, l'utilisateur pourra utiliser des protocoles tels que SPI (*Serial Peripheral Interface*), I2C (*Inter-Integrated Circuit*), UART (*Universal Asynchronous Receiver Transmitter*), NFC (*Near-Field Communication*) et Wifi.

Pour ce qui est des conceptions modulaires industrielles, il peut également être intéressant de présenter l'entreprise KUNBUS GmbH qui propose une passerelle IIoT modulable fonctionnant sur la base d'un Raspberry Pi 4, une plateforme libre (*open source*) que l'on présentera par la suite. On peut trouver cette passerelle sous le nom de « *RevPi Gate* » ("Integrate Revolution Pi into an industrial network," n. d.). Parmi l'ensemble des passerelles proposées par des entreprises, l'investissement dans ce produit est le moins onéreux, il faudra toutefois acheter le composant principal du produit, puis débourser de nouveau pour chaque module supplémentaire. Ces modules assurent la compatibilité avec de nombreux protocoles, parmi eux se trouvent Profibus, DeviceNet, Sercos III, Profinet, EtherCAT, CANopen, Modbus et POWERLINK.

2.5.1.3 Conception à partir de plateforme libre

Comme nous avons eu l'occasion de le remarquer dans les paragraphes précédents, les solutions proposées sont relativement onéreuses, peu importe leur type, surtout quand l'usager concerné s'avère être une petite entreprise aux moyens réduits ou bien un utilisateur indépendant, tel qu'un chercheur. Ce constat a poussé bon nombre de ces derniers à développer leur propre solution, à partir d'éléments abordables tels que les plateformes libres.

Dans cette optique, Des étudiants de l'université de Shiraz ont développé un système permettant d'automatiser le fonctionnement d'une maison à faible coût en utilisant un microprocesseur Arduino (Hassanpour, Rajabi, Shayan, Hafezi, & Arefi, 2017). Ce dernier gère la collecte des différents capteurs et actionneurs utilisant le protocole Modbus. Les données sont ensuite transmises au réseau et à l'application de contrôle grâce à un module Ethernet.

Corotinschi et Găitan ont utilisé un Raspberry Pi pour confectionner une passerelle entre un bus Modbus et le Cloud (2018). Un broker MQTT est hébergé sur le Raspberry Pi et permet d'établir un échange avec le service infonuagique. Un accent est mis sur les critères de sélection et la comparaison de différents brokers. Un broker, qui se traduit littéralement par « courtier » en français, n'a pas de réel équivalent dans le domaine de l'IIoT, c'est pourquoi nous adopterons le terme anglais. Ces derniers sont essentiels pour tout protocole basé sur le principe de publication/souscription et assurent la bonne distribution des données entre les différents nœuds impliqués. Ce sont eux qui sont chargés de publier les messages des divers nœuds du réseau, ils représentent ainsi une pièce centrale pour toute personne souhaitant concevoir une application mettant en jeu le protocole MQTT (Bertrand-Martinez, Dias Feio, Brito Nascimento, Kon, & Abelém, 2020).

Similairement, des chercheurs ont également opté pour la réalisation d'une passerelle de conversion entre le bus de terrain Modbus et le protocole de communication MQTT par le biais d'un Raspberry Pi 3 (Sun, Guo, Xu, Ma, & Hu, 2019). Pour ce faire, cet ordinateur mono-carte

est relié au bus par l'intermédiaire d'un connecteur RS-485. Dans le même élan, Khanchuea et Siripokarpirom ont réalisé une passerelle IIoT incorporant les protocoles Modbus et Zigbee (2019). La connectivité avec internet est assurée via Wifi, Ethernet et les communications sont assurées par un ordinateur mono-puce fonctionnant sous linux et simulant un client MQTT. Un réseau local faible consommation et sans fil est dispensé par un module Zigbee.

On trouve aussi dans la littérature une passerelle IIoT, dénommée DIIG, qui utilise le *middleware* open source Kaa IoT, ainsi qu'un Raspberry Pi (Hemmatpour, Ghazivakili, Montrucchio, & Rebaudengo, 2017). Cette solution permet l'intégration des protocoles Modbus et Siemens S7 au monde numérique. Nguyen-Hoang et Vo-Tan proposent une passerelle similaire, supportant d'une part les protocoles industriels Modbus et Siemens S7 et d'autre part les protocoles MQTT et HTTP (2019). Leur solution repose sur un microprocesseur Raspberry Pi et l'outil de programmation Node-RED. You et Ge ont réalisé une passerelle destinée à la gestion de bâtiments intelligents. Elle permet de passer du protocole Modbus RTU au protocole Modbus TCP et ainsi de bénéficier de la technologie Ethernet pour pouvoir communiquer avec l'application de contrôle (2019).

Enfin, nous pouvons mentionner Chen, Lin et Liu qui ont développé une passerelle IIoT innovante dans laquelle il n'y a pas non un microcontrôleur comme dans les architectures classiques mais au contraire plusieurs (2018). Cette architecture semblerait améliorer les performances de l'installation, que ce soit au niveau de la bande passante ou en termes de rapidité de calculs.

2.6 Sélection des protocoles

Dans la passerelle, nous allons implanter deux bus de terrains et deux protocoles de communication IIoT. La sélection de ces protocoles s'est trouvée influencée par leur notoriété, par leur caractère non propriétaire et par les divers travaux présents dans la littérature. Pour les bus de terrain le choix s'est porté sur Modbus et Canbus, deux protocoles très utilisés dans l'industrie. Modbus est l'un des bus de terrain le plus répandu dans les installations

industrielles. Même de nos jours, il arrive que ce protocole soit toujours implanté dans les nouveaux systèmes (Toc & Korodi, 2018). Canbus, quant à lui, n'a rien à envier à Modbus, il est, entre autres, présent dans la plupart des automobiles modernes (Kim et al., 2016).

En ce qui concerne les protocoles de communication de la couche application, nous nous intéresserons aux protocoles HTTP (avec l'architecture REST) et MQTT, ils ont deux architectures différentes et sont les protocoles les plus populaires dans leur domaine.

2.6.1 Modbus

Modbus est un bus de terrain conçu en 1978 par l'entreprise Modicon Inc, qui depuis a été rachetée par le groupe industriel français Schneider Electric. Modbus s'est vite répandu et est rapidement devenu un incontournable dans le monde de l'automatisation. C'est à l'heure actuelle le protocole compatible avec le plus d'appareils. C'est un protocole de la septième couche du modèle OSI, à savoir la couche application, mais il peut aussi s'utiliser sur les couches 1 et 2 (Zurawski, 2017).

En outre, ce protocole de communication présente de nombreuses fonctionnalités et avantages non négligeables. Contrairement à certains de ses concurrents, son implantation et sa configuration sont relativement simples et rapides. C'est également un protocole peu gourmand qui peut être adaptable suivant la complexité du réseau. Si la solution recherchée est basique, rien ne sert d'avoir une installation démesurée. Il en va de même pour le coût qui sera lui aussi relativement faible en comparaison. Modbus est homologué par de nombreuses institutions de normalisation telles que l'*International Electrotechnical Commission* (IEC), la *Standardization Administration of China* (SAC) et l'association *Semiconductor Equipment and Materials International* (SEMI) (Zurawski, 2017). Par conséquent, il est compatible avec de nombreux protocoles, notamment au niveau des couches inférieures.

Modbus est un protocole réputé pour sa grande compatibilité avec les autres technologies, notamment par le fait qu'il soit un protocole libre. En outre, en rejoignant l'*Open DeviceNet*

Vendors Association (ODVA) en 2007, Schneider Electric s'est assuré que Modbus soit pleinement compatible avec les protocoles gérés par cette association. Ces protocoles sont connus sous le nom de CIP (*Common Industrial Protocols*). En intégrant un module supplémentaire, un traducteur Modbus, il est alors possible pour une installation utilisant le protocole Modbus de communiquer avec les différents protocoles CIP, à savoir DeviceNet™, EtherNet/IP™, ControlNet™, et CompoNet™ (Zurawski, 2017). Au niveau du support physique, Modbus peut fonctionner avec différentes normes de câbles telles que les normes RS232, RS422, RS485 ou encore avec le protocole TCP/IP.

Pour ce qui est de l'organisation du bus, ce protocole se construit avec un ou plusieurs nœuds répondant aux demandes d'un coordonnateur, selon la démarche du *Polling*. Il s'agit plus spécifiquement d'une configuration client-serveur. Le client, qui est aussi le coordonnateur, va alors être l'entité qui enverra les requêtes aux nœuds, qui jouent le rôle de serveurs.

2.6.1.1 Modbus RTU

Il existe trois versions de Modbus différentes. Dans l'industrie, il est possible de retrouver des systèmes incorporant Modbus ASCII, Modbus RTU ou Modbus TCP. Les deux premiers sont plutôt similaires dans le sens où ils utilisent tous les deux une topologie en ligne ou en guirlande et ont le même support physique. Il est à noter toutefois que le mode RTU est plus performant que le mode ASCII et qu'en conséquence ce dernier tend progressivement à disparaître et n'est plus vraiment d'actualité. En ce qui concerne le mode TCP, il est appelé ainsi car cette version de Modbus est destinée à fonctionner sur un réseau utilisant les protocoles TCP/IP pour assurer les fonctionnalités de la quatrième couche du modèle OSI, aussi connue sous le nom de la couche transport (Zurawski, 2017). C'est un protocole très utilisé, mais c'est bien Modbus RTU qui reste la version la plus dominante à l'heure actuelle, malgré la rapide progression des technologies Ethernet (GM International, 2020). Plusieurs facteurs peuvent expliquer sa forte popularité. Comme nous l'avons mentionné précédemment, il s'agit d'un protocole ouvert qui est compatible avec la vaste majorité des appareils industriels, anciens comme nouveaux. Aucune licence n'est nécessaire pour l'implantation de ce protocole. Non seulement simple à

configurer, Modbus RTU est également simple à prendre en main, à comprendre et à utiliser, même par des personnes non spécialisées. Les données échangées présentent un format rapidement assimilable par l'utilisateur, avec quatre types de variables et des commandes se limitant à la lecture ou à l'écriture. Son mode de fonctionnement suivant le principe du « maître-esclave » limite le nombre d'acteurs entrant en jeu. Le maître envoie une requête, l'esclave lui répond. Nous pouvons également ajouter que ce protocole est très léger, il présente des données simples que n'importe quel appareil, même de très faible capacité, peut traiter. En outre, son support RS485 est lui aussi facile à prendre en main. Pour finir, l'emploi de Modbus RTU permet aussi d'échanger plus facilement des données par le biais de signaux numériques et de s'affranchir des calculs de conversion nécessaires lors de transmissions de signaux analogiques. Cela limite alors les incertitudes qui pourraient en résulter. C'est pourquoi Modbus RTU est le mode que nous avons choisi d'étudier et sur lequel nous nous attarderons en détails. Cette désignation est un diminutif pour *Remote Terminal Unit*.

Lors de l'échange de données par le biais de ce protocole, on peut observer que chaque caractère est codé sur 11 bits :

- un bit d'amorce servant à indiquer le début de la transmission du caractère,
- huit bits de données permettant de coder deux caractères hexadécimaux,
- un bit de parité,
- un ou deux bits de fin servant à indiquer que la transmission du caractère se termine.

Il est à préciser que le bit de parité sert à déceler si des erreurs sont apparues lors de la transmission. Ce bit prend la valeur 0 si la somme des autres bits est paire, et 1 dans le cas inverse. Ce bit est optionnel et n'est pas toujours utilisé. En ce qui concerne les bits de fin, le deuxième bit, optionnel, sert à compenser l'absence éventuelle du bit de parité afin de toujours avoir 11 bits codant un caractère.

Il y a aussi à la fin un champ laissé à la vérification d'erreur via un check cyclique (CRC). Par ailleurs, un laps de temps, qui équivaut à la durée de transmission d'au moins 3,5 caractères, est laissé entre chaque trame transmise afin de repérer facilement le début et la fin du message. Avec le mode RTU, la transmission d'une trame doit être continue, le flux des caractères

contenus dans le message doit ainsi s'enchaîner sans interruption. Si un intervalle de temps correspondant à plus de 1,5 caractère a lieu durant la transmission, la trame sera alors déclarée invalide et ne sera donc pas prise en compte (Zurawski, 2017).

2.6.1.2 Structure des messages

Les messages échangés entre le client et le serveur sont composés de trois éléments :

- un champ d'identification,
- un code de fonction,
- un champ « *Data* » contenant les données à transmettre.

Le champ d'identification mentionné ci-dessus ne dépasse pas la taille d'un octet, et se retrouve dans la littérature anglophone sous le nom d'*Unit ID Field*. Ce nom parle de lui-même, il sert à adresser les différents appareils du réseau. Il est à noter que les ID sont assignés aux serveurs et non aux clients. L'adresse doit être unique, chaque serveur a sa propre adresse d'identification. L'ID peut prendre des valeurs allant de 0 à 255, mais certains intervalles ont une fonction particulière. Ainsi, l'ID 0 correspond au *broadcast*, c'est-à-dire un message qui s'adresse à tout le monde. Toutefois, si un message est envoyé en *broadcast*, tous les serveurs le reçoivent mais aucune réponse n'est apportée. D'une façon générale, l'adresse d'un serveur sera comprise entre 1 et 247, les valeurs supérieures jusqu'à 254 étant réservées. Dans le cas d'un ID spécifique, la réponse contiendra toujours ce même ID. Si l'ID 255 est utilisé, cela veut dire qu'il n'y a pas besoin d'une identification particulière supplémentaire, on sait déjà à quel serveur on s'adresse (Zurawski, 2017).

Pour ce qui est du champ contenant le code de fonction, il sera discuté au paragraphe suivant. La partie *Data*, de son côté, contient des données de taille très variable, pouvant aller de 0 à 252 octets. Pour une requête, ce champ contient des informations complémentaires, pour une réponse il contient la réponse à la requête. Cette limite de 252 octets provient du fait qu'une trame de données RTU est encodée sur 256 octets. On précise qu'une trame est la structure de base d'une donnée informatique. La Figure 2.5 montre cette organisation des bits.

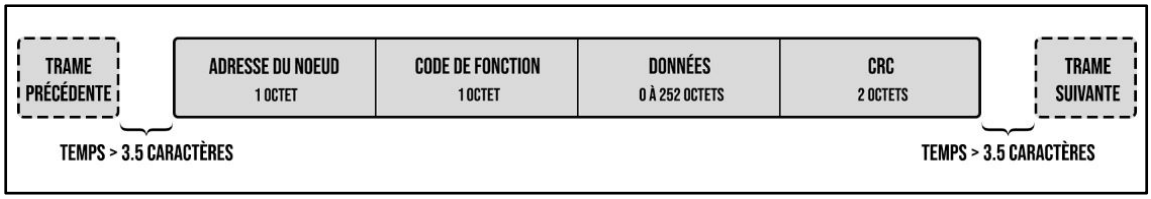


Figure 2.5 Format d'une trame de données avec Modbus RTU

Pour résumer, dans le cas d'une trame RTU, il y a donc un octet réservé pour le champ d'identification, un pour le champ du code, et deux réservés pour les CRC afin de s'assurer que le message n'est pas erroné. Puisque la trame ne peut pas dépasser 256 octets, on comprend ainsi d'où vient la limitation de 252 octets.

Lorsqu'une requête est émise sur le bus, celle-ci contient toujours un code de fonction, d'une taille d'un octet. Ce code va alors donner une indication sur le type d'informations échangées. Il se présente par exemple sous la forme 0x01. L'intervalle des valeurs admises s'étale de 0x01 à 0x7F. Dans le cas d'une réponse, c'est ici que sera indiqué si la requête a été exécutée avec succès ou non. Il est à noter que dans le cas d'une erreur ou si la requête n'a pas abouti, les codes 0x80 ou 0xFF sont retournés pour indiquer que l'action n'a pas pu être traitée (Zurawski, 2017).

Il faut savoir que ces codes sont répartis en trois catégories distinctes :

- Codes assignés de façon publique. Ces codes sont utilisés pour des services normalisés.
- Codes définis par l'utilisateur. Ces codes peuvent être utilisés localement sur des systèmes spécifiques. Cependant, ils ne doivent pas être implantés sur des appareils publics. Les intervalles réservés sont de 65 (0x41) à 72 (0x48) et de 100 (0x64) à 110 (0x6E) inclus.
- Codes réservés. Ces codes de fonction sont utilisés par des entreprises mais ne sont pas ouverts à un usage public.

Par exemple, le code 0x02 indique que la requête a pour fonction de lire une variable de type discrète. Pour savoir quels codes utiliser et à quelles fonctions ils correspondent, il est important de se référer au guide de référence au sein duquel tout est détaillé. Il en va de même pour les codes d'exceptions (Modbus Organization, 2012).

2.6.1.3 Gestion des erreurs

Malgré la robustesse des systèmes industriels, il peut être relativement fréquent de rencontrer des erreurs pouvant survenir lors des interactions entre les différents appareils. La source du problème peut provenir de la transmission en elle-même ou bien du traitement des données effectué par un des appareils impliqués. Modbus fonctionne sous le principe de l'accusé de réception. Pour qu'une transmission soit considérée comme valide et terminée, il faut obligatoirement que le nœud ait renvoyé un message au coordonnateur, ce sans quoi le message est évalué comme en cours de transmission et le client reste en attente d'une réponse, sans pouvoir effectuer d'autres actions. Il est alors aisé de voir la nécessité de tenir compte de ces erreurs de transmission. Une erreur pourrait bloquer et engorger tout le réseau avec la réponse d'une requête qui n'arriverait jamais par exemple. Il est donc essentiel de pallier ce problème. C'est pourquoi les appareils du réseau appliquent le principe du *timeout*, ou délai d'attente. Si aucune réponse n'est reçue au bout d'un certain temps, on considère que le message n'a pas été transmis, il est alors ignoré et les communications reprennent normalement, le client peut amorcer de nouvelles requêtes. Pour un *broadcast*, aucun nœud n'est spécifiquement concerné, c'est pourquoi aucune réponse n'est attendue et il n'y a pas de moyen de contrôler si la demande a été traitée, un *timeout* est donc défini d'office et servira de délai avant que le coordonnateur puisse renvoyer une autre requête (Zurawski, 2017).

2.6.2 Canbus

Canbus, aussi connu sous le nom de « *Controller Area Network Bus* », est un bus de terrain particulièrement prisé dans le milieu de l'automobile. Ce protocole fut développé à partir de 1983 par l'entreprise allemande Bosch puis sa commercialisation débuta trois ans plus tard, en 1986. L'avantage premier de ce genre de technologies était de considérablement réduire le nombre de câbles transportés en voiture et nécessaires à son bon fonctionnement, les longueurs combinées des câbles pouvant préalablement atteindre deux kilomètres pour une seule voiture (Koch & Lueftner, 2019). Les technologies Canbus sont ainsi rapidement devenues très populaires voire même indispensables. Les produits Canbus sont régulés par la norme ISO 11898-1:2015 (International Organization for Standardization, 2015).

2.6.2.1 Versions de Canbus

Dans un premier temps, il est important de noter qu'il existe deux types de Canbus. On peut en effet retrouver le Canbus « *High Speed* » et le Canbus « *Low Speed* » définis respectivement par les norme ISO 11898-2:2003 et ISO 11898-3:2003. Ces deux types ne sont pas compatibles entre eux et leur distinction se fait principalement sur les débits maximums autorisés. Ainsi, le premier aura une vitesse maximale de transfert de 1 Mbit/s alors que celle du second sera seulement de 125 kbit/s. Ce dernier sera cependant insensible aux pannes (*fault tolerant*), il est alors plus robuste aux potentielles erreurs de transmission et parvient plus facilement à les détecter. Les deux versions ne se concentrent donc pas sur les mêmes objectifs et ne sont donc pas à utiliser dans les mêmes contextes. La basse vitesse sera plus adaptée aux longues distances puisqu'un bus à haute vitesse, c'est-à-dire à 1 Mbit/s, ne pourra pas dépasser 40 mètres de longueurs. Il est toutefois à noter que Canbus « *High speed* » est de loin la version la plus utilisée (Turner & Smith, 2014).

Canbus, comme tout bus de terrain, possède un nombre de nœuds adressables limité. Comme chaque nœud correspond à un appareil, il en découle qu'il y a une limite dans la quantité d'appareils qu'il est possible de connecter à un même bus. Ce nombre s'élève à 110. Il peut toutefois être un peu dépassé, et aller jusqu'à 128, mais il faut au préalable prendre ses précautions car au-dessus de 110 appareils des restrictions sont applicables (Koch & Lueftner, 2019).

D'un point de vue de la transmission, le support physique consiste en un double câble torsadé en cuivre. Les deux fils sont alors dénommés « CAN_Low » et « CAN_High ». Il est important de ne pas confondre ces dénominations avec celles des versions de Canbus « *High speed* » et « *Low speed* », il s'agit ici de deux aspects bien distincts. La transmission est par conséquent symétrique, cela a notamment pour effet de rendre les câbles plus robustes aux interférences électromagnétiques qu'ils pourraient être amenés à capter. Un troisième câble, que l'on retrouve dans la littérature sous le nom de « CAN_GND » (pour *ground*, la terre en français), peut être ajouté, et avec ce dernier on pourra éventuellement observer un quatrième câble qui

lui assurera la fonction d'alimentation, avec une tension de 5 Volts. Un avantage indéniable de cette configuration est que, dans le cas où un câble serait défaillant, le système pourra tout de même fonctionner en utilisant seulement le câble restant (Koch & Lueftner, 2019). Ceci est particulièrement intéressant d'un point de vue sécuritaire pour une voiture puisque, on le rappelle, ce protocole est omniprésent dans le milieu automobile.

Dans la grande majorité des cas, la configuration du bus sera en ligne, mais il peut arriver d'avoir une topologie en étoile. Canbus fonctionne avec le principe de l'accès universel, c'est-à-dire que n'importe quel appareil peut décider d'envoyer une requête, il n'y a pas de concept de maître et d'esclaves comme pour Modbus.

2.6.2.2 Structure des messages

Les messages Canbus sont plus complexes dans leur architecture que ceux de Modbus décrits précédemment. A l'instar du protocole Modbus, une trame de données Canbus présentera un champ d'identification. En revanche, cette fois-ci ce champ sert à identifier le message et non le receveur. Les champs « *Data* » et « CRC » sont également présents au sein des messages CAN, toutefois le reste de l'architecture diffère comme nous pouvons l'observer en Figure 2.6.

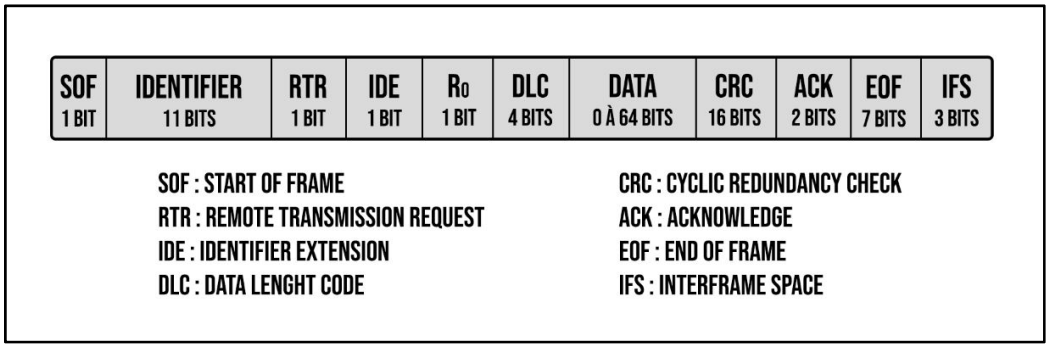


Figure 2.6 Format d'une trame de données standard avec Canbus

Le Tableau 2.3, quant à lui, recense les différents termes présents sur cette figure et esquisse une brève description de ces éléments. La Figure 2.6 et le Tableau 2.3 ont tous deux été adaptés des travaux de Cook et Freudenberg (2008) et de Corrigan (2016). Il est à noter que les

désignations anglaises ont été conservées afin d'assurer une meilleure concordance avec la littérature et avoir des abréviations génériques.

Tableau 2.3 Description des différents champs composant une trame de données Canbus

Champ	Description
SOF	Il s'agit d'un bit dominant permettant d'indiquer le début d'un message et de synchroniser les nœuds du bus.
Identifier	Ce champ permet d'identifier de manière unique un message ainsi que sa priorité.
RTR	Ce champ permet d'indiquer si la trame contient des données ou s'il s'agit d'une trame de requête à distance. Le bit sera respectivement dominant ou récessif.
IDE	Il est utilisé pour indiquer si une extension d'identifiant est à venir. Il est toujours dominant dans le cas des trames standards.
R_0	Ce bit est réservé et doit toujours être dominant.
DLC	Ce champ sert à indiquer le nombre d'octets que forment les données prêtes à être transmises.
Data	C'est dans ce champ que sont transmises les informations du message. Il ne doit pas excéder huit octets et peut être vide dans le cas d'une requête à distance.
CRC	Ce champ est réservé pour le contrôle de redondance cyclique. La somme des bits transmis est effectuée dans le but de déceler une erreur. 15 bits sont alloués à ce calcul et un bit final, récessif, sert de délimitation avec le champ suivant.
ACK	Ce champ sert à vérifier l'intégrité des données. Contenant initialement un bit récessif lors de son émission, si un nœud détecte que ce message ne présente pas d'erreur, il va le réémettre avec un bit dominant, permettant ainsi à l'émetteur de connaitre son état et d'agir en fonction. Le deuxième bit de ce champ est un bit récessif de délimitation.
EOF	Cette section indique la fin de la transmission du message. Ici, la méthode du « *bit stuffing* » n'est plus employée contrairement aux champs précédents. Cette méthode permet d'assurer une bonne synchronisation lors de la transmission en incorporant un bit opposé si cinq bits de même dominance sont émis à la suite.

Tableau 2.4 Description des différents champs composant une trame de données Canbus (suite)

Champ	Description
IFS	Ce champ permet de laisser le temps aux contrôleurs de traiter correctement le message reçu avant d'être en mesure d'effectuer une nouvelle action.

Il est à noter que le format de trame décrit dans ce paragraphe s'applique à la version CAN2.0A, avec un identifiant de 11 bits. Avec cette version, il est possible d'avoir 2048 identifiants uniques. Si cette limite n'est pas suffisante, il est possible de basculer sur la version CAN2.0B. Cette dernière permet d'étendre le nombre de bits alloués à l'identifiant à 29, soit un peu plus de 500 millions d'identifiants uniques.

2.6.2.3 Méthodes de communication

Les bus CAN étant à accès universel, on comprend rapidement que ces derniers doivent mettre en place un moyen pour prévenir d'éventuelles erreurs de transmission, notamment si plusieurs appareils décident d'envoyer une requête au même moment. Il n'y a en effet pas de jeton de priorité avec ce protocole. Ce problème est résolu grâce à la méthode CSMA/CR. Les initiales CSMA signifient *Carrier Sense Multiple Access* et CR signifient *Collision Resolution*. Les protocoles CSMA sont au nombre de trois, parmi eux se trouvent les protocoles CSMA/CD, CSMA/CA et CSMA/CR. Cette famille de protocoles va permettre aux appareils d'écouter le bus sur lequel ils transmettent. Lorsqu'un nœud veut émettre un message, il va d'abord écouter le bus, si ce dernier est vide alors le nœud commence l'émission, sinon il attend. Il peut arriver que deux nœuds commencent l'émission en même temps ou avec un laps de temps tellement court que les signaux n'ont pas eu le temps de se propager, ce qui donne l'impression que le bus est disponible. Dans ce cas, les deux messages sont transmis jusqu'à la détection de cette anomalie. On considère alors qu'il y a eu collision.

Les trois protocoles CSMA diffèrent alors dans leur manière de traiter cette collision. Dans le cas du CSMA/CR, le bus va procéder à une résolution de la collision. Il s'agit de la version la plus complexe mais la plus optimisée des CSMA. Pour ce faire, l'appareil va comparer le signal

émis avec le signal reçu en effectuant une opération sur les bits, un ET logique. Tant qu'il y a égalité, la transmission continue, mais dès qu'une inégalité est détectée la transmission est avortée. Ainsi, le nœud prioritaire transmettra son message sans que celui-ci ne soit détérioré (Koch & Lueftner, 2019). Le fonctionnement de la méthode CSMA/CR est illustré en Figure 2.7. Cela implique que c'est l'identifiant du message qui va dicter sa priorité au sein du bus, les messages ayant les identifiants les plus faibles, les plus proches de 0, seront ainsi les messages prioritaires.

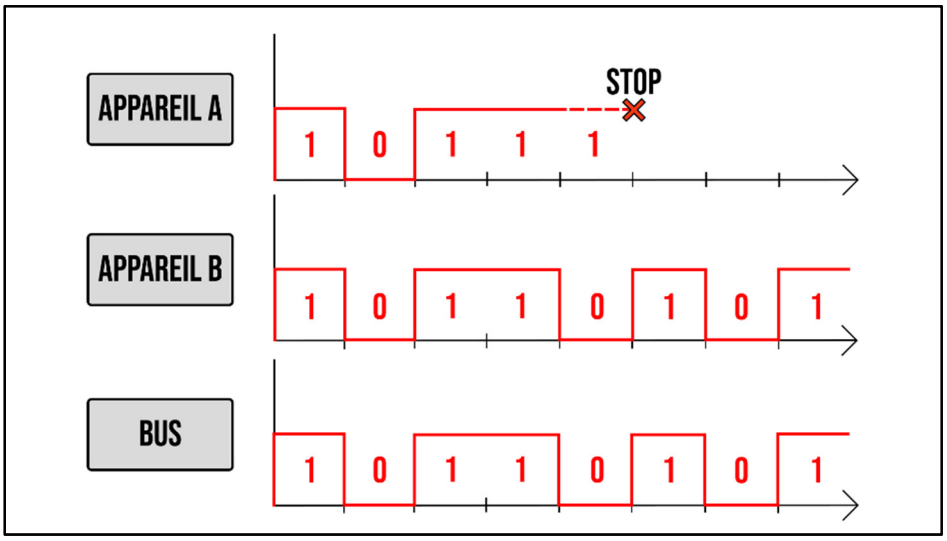


Figure 2.7 Exemple de fonctionnement de la méthode CSMA/CR avec deux appareils transmettant simultanément

2.7 REST via HTTP

REST est l'acronyme de « *Representational State Transfer* » qui est une architecture logicielle permettant de construire des services web évolutifs, le plus souvent (presque tout le temps) via HTTP. Ce terme est défini pour la première fois, dès 2000, par Roy Thomas Fielding, un docteur en informatique diplômé de l'université de Californie (2000). HTTP, ou « *Hypertext Transfer Protocol* », est quant à lui un protocole de communication basé sur le principe du client-serveur. C'est le protocole de base utilisé pour le web, il est donc mondialement répandu. Avec REST, les ressources sont identifiées de manière unique par un URI, mais ce n'est pas suffisant pour que l'architecture d'un système soit qualifiée de la sorte, il faut en effet qu'elle

respecte plusieurs contraintes. Le Tableau 2.5 énumère les différentes contraintes imposées par l'architecture REST.

En plus des cinq contraintes présentées dans le Tableau 2.5, une autre contrainte facultative indique que le serveur peut transmettre du code exécutable au client (Pielli, Zucchetto, Andrea Zanella, Vangelista, & Zorzi, 2015).

Tableau 2.5 Différentes contraintes d'une architecture REST

Contrainte	**Description**
Client-serveur	Adoption d'un environnement dans lequel un ou plusieurs appareils vont envoyer des requêtes à un serveur qui va ensuite leur répondre.
Sans état	Chaque requête est indépendante et aucune information sur la session de communication n'est gardée. C'est donc le client qui doit fournir les différentes informations lors de sa requête ce qui a comme effet de simplifier le fonctionnement du serveur.
En couche	Le client communique directement avec le serveur ou peut aussi avoir recours à des serveurs intermédiaires. Ces intermédiaires permettent, entre autres, d'assurer une meilleure sécurité.
Mise en cache	Les réponses peuvent être mises en cache, ceci permettant d'être plus flexible et d'améliorer les performances.
Interface uniforme	Cette contrainte permet à chaque partie de fonctionner indépendamment. Elle implique que les ressources soient identifiées dans les requêtes et leur manipulation se fait selon le principe CRUD (*Create, Retrieve, Update, Delete*).

2.8 MQTT

Le protocole MQTT, ou « *Message Queuing Telemetry Transport* », est un protocole de communication ne consommant que très peu de ressources. Ce protocole est donc une aubaine pour les appareils limités et les réseaux ayant une faible bande passante et de grosses latences.

MQTT est ainsi particulièrement apprécié dans le domaine de l'IIoT (MQTT.org, 2020). Par le biais de ce protocole, les appareils sont capables de communiquer entre eux à distance, de manière asynchrone via le principe d'évènements et de publication/souscription. Il est basé sur le protocole TCP/IP.

Initialement créé en 1999 par Andy-Stanford-Clark et Arlen Nipper, respectivement employés à IBM et Arcom, MQTT est depuis sa version 3.1.1 normalisé par l'*Organization for the Advancement of Structured Information Standards* (OASIS) ainsi que par l'ISO (International Organization for Standardization, 2016). L'intérêt apporté à ce protocole ne cesse d'augmenter au fil des années. Ainsi, si le nombre d'articles publiés entre 1999 et 2004 au sujet du protocole MQTT était de 87, il est passé à 1530 entre 2010 et 2015 puis à 14600 entre 2015 et 2019 (ces chiffres concernent seulement Google Scholar). On note alors très rapidement une croissance exponentielle. Il est par conséquent très probable que ce protocole soit massivement adopté dans les années futures, d'autant plus qu'il semble être plus performant que AMQP pour les réseaux restreints (Mishra & Kertesz, 2020).

MQTT, à l'instar du protocole AMQP, fonctionne selon le concept du « *publish/subscribe* ». L'architecture de MQTT se base sur le principe du client-serveur. Le rôle du serveur est assuré par un broker, un programme qui réceptionne les différentes informations publiées avec leurs sujets respectifs et qui sert d'intermédiaire en retransmettant ces informations aux clients abonnés si ces derniers sont abonnés à ces mêmes sujets. Il existe de nombreux brokers tels que Mosquitto, HiveMQ ou encore Azure IoT pour n'en citer que quelques-uns (Mishra & Kertesz, 2020).

Les différents sujets, plus couramment désignés par leur équivalence anglaise « *topics* », constituent le point central de ce type de communication. Ce sont des `strings` codés en UTF-8 (*Unicode Transformation Format*, 8 bits), le *topic* est l'identifiant par lequel les messages seront filtrés. Un message ne sera transmis par le broker au client que si ce dernier lui a préalablement indiqué qu'il souhaitait s'abonner à son *topic*. De la même manière, lorsqu'un client veut publier un message, il doit obligatoirement indiquer son *topic*.

MQTT est un protocole très en vogue qui s'est vu déployé sur les langages de programmation les plus populaires (Python, C, Java ou encore php) par le biais de bibliothèques, son implantation en est donc grandement facilitée. Il est également disponible sur les plateformes mobiles iOS et Android (Pielli et al., 2015).

2.9 Conclusion

Dans le cadre de ce chapitre, nous avons discuté des bus de terrain et de leur fonctionnement. Ces protocoles, partie intégrante de l'industrie, peu importe le domaine, ont été développés il y a plus de 40 ans pour certains et reposent sur des technologies anciennes très peu adaptées à la vision proposée par l'industrie du futur. Ils seront toutefois présents pour encore de nombreuses années, il est par conséquent capital de trouver des solutions pour permettre aux diverses technologies, modernes comme anciennes, de cohabiter et d'opérer en harmonie.

Par ailleurs, nous avons abordé les solutions développées pour concilier les bus de terrains avec le monde numérique de l'IIoT. L'intérêt des passerelles IIoT a ainsi été démontré. Nous avons également étudié les différentes possibilités envisageables pour la conception d'une passerelle. Les diverses informations récoltées offrent alors la possibilité de faire des choix réfléchis pour la conception de notre propre passerelle IIoT. Il faut ensuite analyser les avantages et désavantages des types de passerelle (conception fixe, modulaire et sur plateforme libre) et les cas d'usage les plus adaptés avant d'opter pour un choix définitif.

Modbus, Canbus, HTTP et MQTT seront les 4 protocoles inclus dans notre passerelle IIoT. Leur sélection s'est basée sur plusieurs facteurs dont notamment leur popularité et leur accessibilité. Pour que cette association fonctionne, il faut toutefois se procurer un matériel adapté qui supporte ces différents protocoles. Il existe trois versions de Modbus : ASCII, RTU et TCP. La version TCP de Modbus n'est pas un protocole temps réel puisque le nombre de retransmissions TCP est illimité (*unbounded retries*). Canbus, quant à lui, possède deux vitesses de communication, la haute vitesse à 1 Mbit/s étant celle qui est la plus répandue dans

l’industrie. Les protocoles de haut niveau HTTP et MQTT ont des modèles de communication très différents. Dans le cas de HTTP via REST, le modèle est du type requête/réponse et sans mémorisation des échanges antérieurs. Pour MQTT, le modèle est du type publication/souscription et nécessite des brokers comme agents intermédiaires entre les entités de communication.

CHAPITRE 3

CONCEPTION DE LA PASSERELLE

3.1 Introduction

Les précédents chapitres nous ont permis de poser le cadre du sujet de recherche. Ce chapitre va alors présenter notre contribution à l'avancée technique dans le domaine des objets connectés industriels. L'objectif de ce sujet est en effet de concevoir une passerelle IIoT opérationnelle. Dans premier temps, la méthodologie de travail sera abordée. Nous nous attacherons ensuite à présenter les différents composants de notre passerelle IIoT, et expliquer les choix de ces derniers. En outre, nous expliquerons les montages réalisés et leur rôle dans notre processus de développement. Par ailleurs, nous discuterons des différents éléments extérieurs à la passerelle, ces derniers étant essentiels à sa conception. Il sera alors question d'éléments physiques, tels que des appareils assurant des communications via les bus de terrain, comme d'éléments virtuels à l'instar de la plateforme IIoT avec laquelle nous échangerons.

L'aspect logiciel sera également abordé puisque nous étudierons en détail comment se sont déroulés la phase de programmation et le design des fonctionnalités de la passerelle IIoT. Les patrons de conception « Méthode » et « Adaptateur » seront appliqués pour coordonner les activités de la passerelle. L'adoption des patrons de conception fait partie des meilleures pratiques du développement logiciel et ils facilitent la maintenance et l'évolution des logiciels conçus. Enfin, l'entièreté de ce chapitre viendra répondre à la question « Comment concevoir une passerelle IIoT abordable et performante » qui est centrale à cette recherche.

3.2 Méthode de développements

Le travail de recherche et la maîtrise d'un sujet sont un processus long, il est alors essentiel d'adopter une méthode de travail efficace. De fait, il a été décidé d'appliquer la méthode SCRUM. Il s'agit d'une méthode agile de gestion de projet permettant de modifier un travail

très rapidement, dynamiquement. Cette méthode a été jugée pratique dans le cas où certains axes seraient à modifier et si les attentes du sujet étaient amenées à évoluer. Apparu au début des années 2000, il s'agit d'un mode de travail relativement nouveau (Ihenacho, 2014).

SCRUM est une méthode incrémentale et itérative. Plusieurs itérations permettent de construire la solution finale. Le résultat obtenu diffère ainsi souvent de celui imaginé lors de la définition du sujet de recherche. Ces itérations sont dénommées *sprints*. Un *sprint* dure souvent entre deux et quatre semaines. Pendant un *sprint*, il y a une phase de développement et un contrôle qualité. À la fin du *sprint*, un livrable est montré au client pour lui présenter l'avancement et avoir son retour. Dans notre cas, le client sera le directeur de recherche.

Pour avancer dans le projet, on va définir ce qu'on nomme des *user stories*, ces dernières décrivent de manière simple, voire vulgarisée, la marche à suivre et recensent les différents points et fonctionnalités à traiter. À chaque point sont attribuées une description, une importance et une estimation du travail escompté. À partir de ces *user stories*, nous définissons un *product backlog* qui reflète plus précisément les besoins de ces dernières et les points sont triés par ordre de priorité. Enfin, nous piochons dans le *product backlog* les points à traiter durant un *sprint*. Plusieurs *sprints* sont parfois nécessaires pour couvrir une *user story*. Les différents évènements définis par la méthode SCRUM sont rappelés dans le Tableau 3.1.

Tableau 3.1 Évènements constituants la méthode SCRUM

Évènement	**Description**
Epic	Les *epics* sont utilisés pour le regroupement d'un grand nombre de tâches reflétant certaines similitudes. Un *epic* englobe ainsi plusieurs *user stories*.
User story	Les *user stories* sont définies par le client, elles permettent de décrire les différentes tâches escomptées dans un langage simple et non technique.

Tableau 3.2 Évènements constituants la méthode SCRUM (suite)

Évènement	Description
Sprint	Un *sprint* est une période définie, fixe, durant laquelle il est prévu de réaliser une ou plusieurs tâches. À la fin d'un *sprint*, un échange est effectué avec le client afin d'avoir un retour sur le travail fourni.
Task	Les *tasks* sont les évènements élémentaires de la méthode SCRUM. Une tâche représente une action simple et est réalisée durant un *sprint*. Elle peut éventuellement être divisée en sous-tâches.

La Figure 3.1 illustre les principales étapes de notre sujet de recherche. Cette frise chronologique a été conçue à l'aide du logiciel en ligne Jira qui est un logiciel de gestion de projets prenant en charge la méthode SCRUM. Les différents *epics* que l'on peut observer sont chacun composés de plusieurs *user stories*.

Comme nous pouvons l'observer, il a été envisagé d'implanter le protocole Profibus sur notre passerelle IIoT. Cette fonctionnalité a plus tard été abandonnée notamment à cause des problèmes émanant de l'aspect propriétaire de ce protocole. La phase de recherche a indiqué qu'une implantation de ce protocole poserait de nombreuses contraintes, nous avons ainsi adapté notre projet en adéquation avec nos observations.

Au final, nous pouvons remarquer que cette méthode augmente la productivité et permet d'éviter les gâchis, notamment au niveau de l'investissement dans les composants. Elle permet également de simplifier l'organisation du travail en divisant le problème général en sous-problèmes. Au fil de notre recherche, nous avons également partagé de nombreux livrables, à la fin de chaque *sprint*. Cela nous a permis de ne pas nous égarer et de toujours garder le bon objectif en vue. Enfin, cela traduit une bonne transparence dans le travail avec le directeur de recherche.

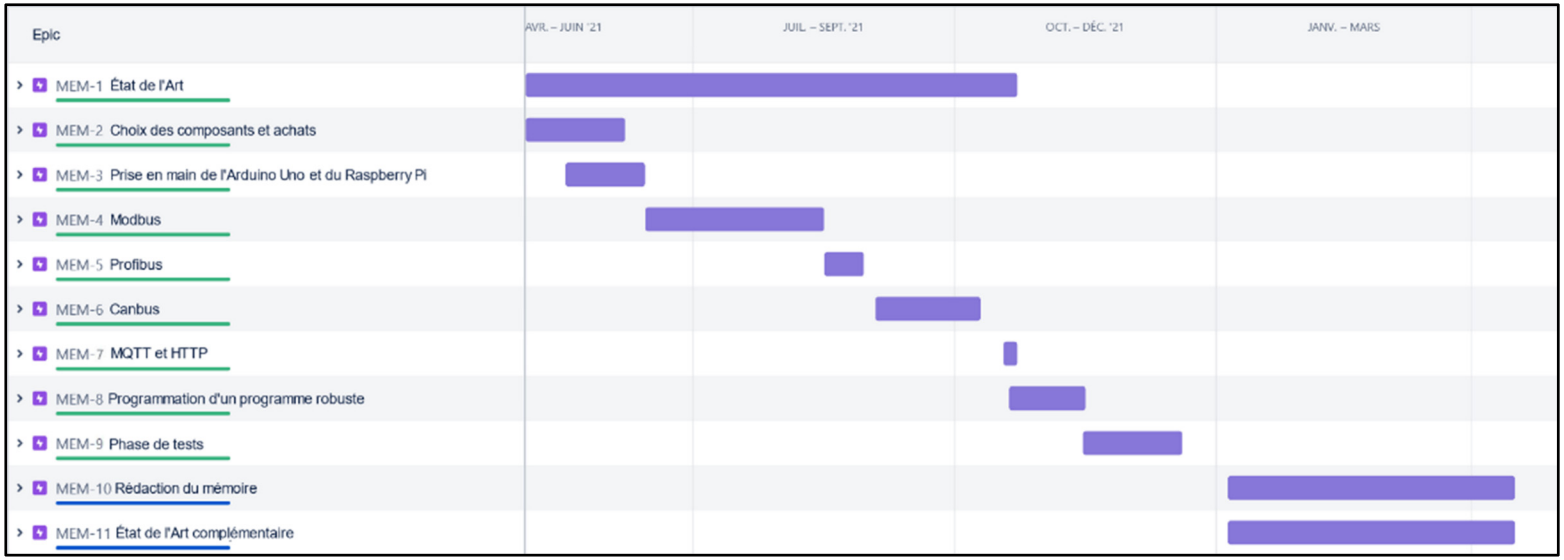


Figure 3.1 Les étapes principales de la conception de la passerelle IIoT

3.3 Conception par intégration des systèmes

La carte contrôleur ou SBC (*Single Board Computer*) est le cœur de la passerelle et le choix de ce composant ne doit pas être pris à la légère. Par le biais de cet élément, nous serons en mesure de traiter les données récoltées, les transcrire dans le bon format et de les envoyer aux services infonuagiques. Comme cela a été discuté au chapitre 2, s'orienter vers des solutions non-propriétaires présente de multiples avantages, c'est donc l'option qui a été envisagée. Dans la gamme des composants open source, deux noms sont incontournables et font office de référence : Raspberry Pi et Arduino.

3.3.1 Carte Raspberry Pi

Le Raspberry Pi est un SoC (*System on a Chip*), un petit ordinateur mono-carte lancé en 2012 par la fondation Raspberry Pi. Ce système est très apprécié par les personnes voulant réaliser des expérimentations d'ordre informatique, électronique ou robotique. Sa popularité peut s'expliquer par son faible coût et la panoplie de fonctionnalités qu'il peut offrir. Plusieurs versions de Raspberry Pi se sont suivies, la version 4, en date de 2019, est la dernière à ce jour.

Le Raspberry Pi 4 Model B se vend avec 1, 2 ou 4 Go de RAM (*Random Access Memory*). Les performances de ce dernier, compte tenu de sa petite taille, sont très honnêtes. Il dispose en effet d'un microprocesseur ARM Cortex-A72 à quatre cœurs, avec une fréquence de calcul de 1,5 GHz. Il fonctionne en 64 bits. Par ailleurs, le Raspberry Pi 4 Model B est un microprocesseur doté de quatre ports USB, dont deux USB 3.0, de cartes réseaux Ethernet et Wifi, ainsi que de la technologie Bluetooth 5.0. Il est alimenté en 5V par micro-USB et fonctionne sous le système d'exploitation Linux (Raspberry Pi (Trading) Ltd., 2019). Il existe d'ailleurs une version de ce système d'exploitation propre à lui, Raspbian. Ce microordinateur présente l'avantage de posséder 28 broches GPIO (*General Purpose Input/Output*) qui supportent des protocoles comme par exemple UART, I2C ou SPI, ce qui offre la possibilité à l'utilisateur de réaliser toutes sortes d'expérimentations.

D’après les caractéristiques affichées du Raspberry Pi, ce dernier semble donc être l’élément idéal pour remplir le rôle de contrôleur de notre passerelle IIoT. En outre, il existe sur Internet quelques tutoriels démontrant qu’il est possible d’employer un Raspberry Pi pour effectuer des communications Modbus, moyennant quelques composants supplémentaires (Zihatec, n.d.). Il en va de même pour Canbus (Youness, 2018). Le Raspberry Pi sera donc un dispositif que nous utiliserons lors de nos expérimentations.

3.3.2 Carte Arduino

Les produits Arduino sont des cartes électroniques faisant partie de la famille des microcontrôleurs. Elles sont très prisées dans le domaine du *Do It Yourself* (DIY) de par leur faible coût et de par la diversité de leurs applications. Elles permettent notamment la réalisation rapide de systèmes électroniques. La programmation sur Arduino est réalisée avec le langage C/C++. La plus populaire d’entre elles se prénomme Uno et est une solution particulièrement privilégiée pour servir d’interface entre capteurs et actionneurs et gérer le contrôle de ces derniers.

D’un point de vue technique, la carte Uno dispose de 2 ko de RAM statique, de 32 ko de mémoire flash, de 1 ko de EEPROM (*Electrically-Erasable Programmable Read-Only Memory*). Elle présente une architecture à 8 bits et sa fréquence de calcul est de 16 MHz. La Arduino Uno dispose de deux microcontrôleurs, l’ATmega328P et l’ATmega16U2, le premier étant le contrôleur principal (Wong & Rousseau, 2018). Elle présente divers ports tels que des ports I2C, SPI et UART et possède également de nombreuses broches entrée-sortie (E/S).

Étant de la famille des microcontrôleurs, cette carte ne requiert aucun système d’exploitation pour fonctionner, seul un code d’exécution est nécessaire. Ce code est développé sur PC via son propre environnement de développement et est transféré sur la carte en USB. Son profil en fait une parfaite candidate pour servir de nœud Canbus ou Modbus, le tutoriel de Zihatec sur l’emploi de Modbus avec une carte Uno en est la parfaite illustration (2018). Nous retiendrons donc cette pièce pour les expérimentations, mais il est important en revanche de noter que

l'accès à internet n'est pas une fonctionnalité accessible sans modules complémentaires et que ses fonctionnalités sont plus limitées qu'un Raspberry Pi, ce dernier étant un ordinateur à part entière. Le rôle de la carte Arduino ne sera pas donc celui d'un contrôleur de passerelle mais simplement celui d'un nœud, d'un appareil traitant des données et pouvant interagir avec la passerelle. Nous investirons alors dans deux cartes Arduino, un bus avec plusieurs appareils étant préférable pour se rapprocher d'une condition industrielle réelle.

3.4 Interfaces HAT et Shields

Nous avons remarqué que la carte Arduino présentait de nombreuses interfaces, tout comme un Raspberry Pi. Cependant aucun de ces deux systèmes ne dispose d'interface compatible avec les spécifications des bus de terrain Canbus et Modbus. Pour corriger ce problème, l'achat d'interfaces supplémentaires est requis.

Un HAT, pour *Hardware Attached on Top*, est une carte d'extension, pouvant être rajoutée au Raspberry Pi dans l'optique d'introduire des fonctionnalités supplémentaires. Les HAT sont connectés au Pi par le biais de ses GPIO et respectent les normes d'utilisation de l'appareil. Leurs intérêts résident notamment dans le fait d'être particulièrement simples à installer, avec une configuration automatique, et d'être amovibles, aucune soudure n'est nécessaire. Un HAT doit disposer obligatoirement d'une EEPROM d'identification valide comportant les informations du vendeur et l'arborescence du dispositif et posséder un connecteur GPIO standard de 40 broches.

Pour la mise en place du protocole Modbus, un HAT intégrant une interface RS485 est de mise. Nous avons effectivement choisi de nous orienter vers la norme RS485 car il s'agit de la plus répandue pour Modbus. Plusieurs modèles sont présents sur le marché. Parmi ceux-ci, le modèle présenté en Figure 3.2 a été retenu pour effectuer les expérimentations.

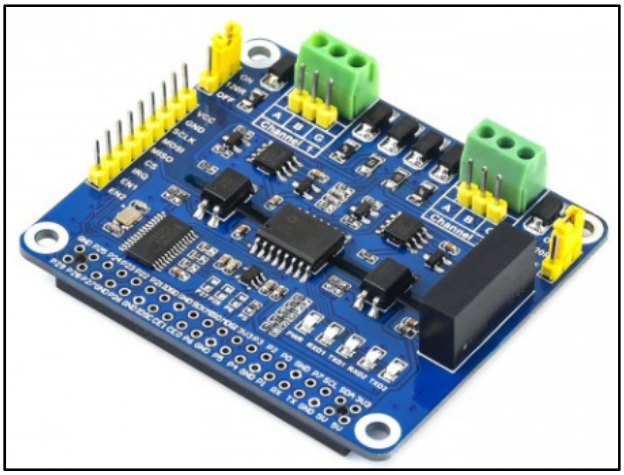


Figure 3.2 Extension Modbus sélectionnée
pour la carte Raspberry Pi

Il est possible d'observer sur la Figure 3.2 le HAT « 2-Channel Isolated RS485 Expansion HAT » développé par l'entreprise Waveshare. Cette extension permet des communications semi-duplex (*half-duplex*) avec une commutation automatique entre les modes de transmission et de réception. Elle intègre le protocole UART avec une puce SC16IS752, tandis que la puce SP3485 gère les transmissions RS485. Les échanges entre le Pi et l'interface se font par le biais du protocole SPI. Les signaux électriques transitant par le HAT sont tous isolés, y compris au niveau de l'alimentation, ce qui permet d'éviter des interférences magnétiques ou électrostatiques pouvant altérer les signaux et corrompre les données échangées. Le HAT intègre également des LED indiquant à l'utilisateur l'état du système et des communications. De plus, on peut noter la présence de résistances de 120 Ω conformément aux spécifications du protocole. Le débit de données transmises autorisé varie de 300 bit/s à 921 600 bit/s, ce qui offre une gamme intéressante à l'utilisateur (Waveshare, 2022). Il peut être intéressant de rajouter que ce HAT dispose de deux connectiques RS485 et permet ainsi d'émuler deux instances Modbus simultanément sur le Pi et peut s'avérer utile lors du développement du programme.

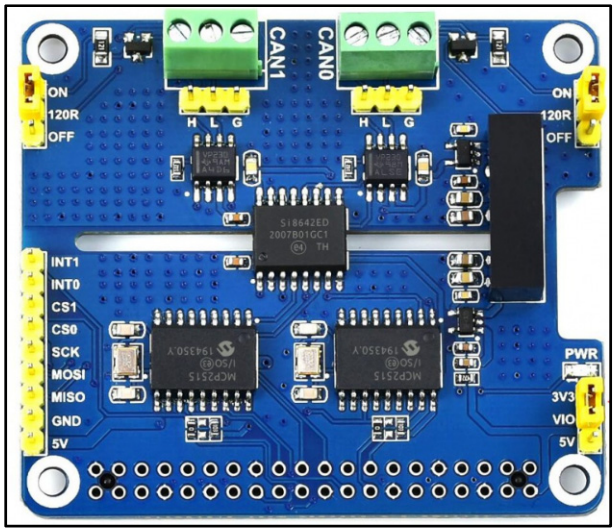


Figure 3.3 Extension Canbus sélectionnée
pour la carte Raspberry Pi

En ce qui concerne l'implantation de Canbus, le choix s'est porté sur un autre produit du constructeur Waveshare, le « 2-CH CAN HAT ». Il s'agit de l'homologue du HAT RS485 présenté précédemment, mais adapté à Canbus, et plus précisément à CAN2.0. Une photographie de ce HAT est montrée sur la Figure 3.3. Il présente ainsi les mêmes caractéristiques, tant au niveau de son mode de fonctionnement qu'au niveau de ses fonctionnalités. On remarquera ainsi la présence d'indicateurs LED, de résistances de 120 Ω et d'une isolation magnétique des composants. Le HAT dispose d'un contrôleur CAN MCP2515 et d'une interface SPI pour pouvoir communiquer avec le Raspberry Pi. Les communications avec le bus sont quant à elles assurées par l'émetteur-récepteur SI65HVD230 (Waveshare, 2021).

Les deux HAT présentés offrent par ailleurs la possibilité d'interchanger le mode de fonctionnement entre 3.3V et 5V. Cette option s'avère utile dans notre cas puisque nous désirons effectuer des communications avec une Arduino Uno, fonctionnant le plus souvent à 5V. Ce détail peut ainsi nous éviter d'endommager les composants en s'assurant d'avoir des niveaux de tensions équivalents.

Au niveau de la compatibilité entre ces deux produits, les informations techniques du constructeur indiquent l'emploi de broches différentes pour chacun des deux HAT, il est ainsi possible de les connecter tous deux au microordinateur.

Les Shields, quant à eux, sont des composants physiques pouvant se greffer directement sur les produits Arduino, à l'instar des HAT pour les Raspberry Pi. Leur nombre est plus que conséquent et les fonctionnalités offertes par ces derniers sont nombreuses. Les Shields auxquels nous prêtons une attention toute particulière sont bien évidemment ceux proposant des interfaces RS485 et CAN.

Nous cherchons ainsi un Shield qui puisse assurer la conversion d'un signal RS485 en un signal TTL (*Transistor-Transistor Logic*). La puce la plus répandue effectuant cette action est la MAX485. Cette dernière présente toutefois un défaut majeur, elle ne dispose pas d'un commutateur automatique assurant les transitions entre phases de transmission et de réception (Maxim Integrated, 2014). Cette considération induit la nécessité pour l'utilisateur de gérer de lui-même les changements d'état du système lors de ses différentes opérations. Nous lui préférons ainsi la MAX13487 qui permet quant à elle la réalisation de cette opération (Maxim Integrated, 2015). Le Shield arborant cette puce, en Figure 3.4, répond au nom de « SH-U12 » et est développé par l'entreprise DSD Tech. Ce composant tolère donc la commutation automatique du signal et est en prime muni d'indicateurs RxD (*Receive Data*) et TxD (*Transmit Data*) permettant d'observer visuellement quel est son état actuel.

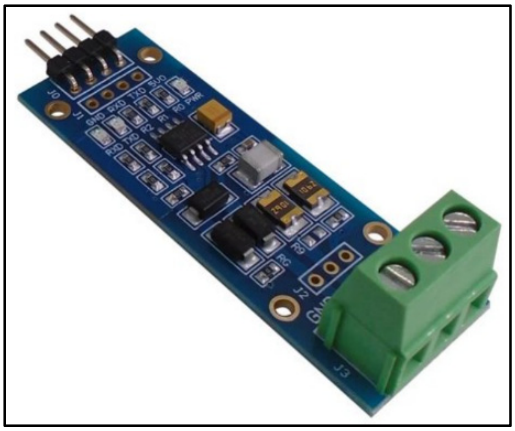


Figure 3.4 Shield DSD Tech SH-U12
pour la carte Arduino

Il reste maintenant à définir quel Shield nous permettra d'assurer les communications avec le bus CAN. Il a été dans un premier temps question de se pencher vers le module « MCP2515 CAN », ce dernier étant un Shield CAN à faible coût. Le contrôleur MCP2515 est accompagné d'un émetteur-récepteur TJA1050 tolérant un débit pouvant aller jusqu'à 1 Mbit/s. Ce Shield possède une interface SPI pour communiquer avec la Uno. En parallèle, nous avons également investi dans un Shield désigné par l'appellation « *Serial CAN-BUS Module based on MCP2551 and MCP2515* » qui utilise le même contrôleur mais dont l'émetteur-récepteur est cette fois ci le MCP2551 (Longan Labs, 2017). Le MCP2551 présente des caractéristiques similaires au TJA1050, nous n'allons donc pas comparer ces deux derniers pour faire notre choix. En revanche il est intéressant de noter que le deuxième Shield utilise une architecture de communication UART. Le Tableau 3.3, traduit et adapté d'un article de Yida (2019), effectue une comparaison rapide entre I2C, SPI et UART.

Tableau 3.3 Comparaison entre les protocoles I2C, SPI et UART

Protocole	**I2C**	**SPI**	**UART**
Complexité	Facile de disposer plusieurs appareils en chaîne	Complexe avec un nombre croissant d'appareils	Simple
Vitesse	Moyenne	Le plus rapide	Le plus lent
Nombre d'appareils	Jusqu'à 127, mais devient vite complexe	Beaucoup mais devient complexe	Jusqu'à 2
Nombre de fils	2	4	2
Duplex	Half-duplex	Full-duplex	Full-duplex

Dans notre cas, la communication n'a lieu qu'entre deux appareils, à savoir le Shield et la carte Arduino. Les trois protocoles mentionnés ci-dessus conviennent donc à notre application, nous avons alors tout intérêt à nous orienter vers la solution la plus simple, c'est-à-dire le protocole UART. Il n'est pas nécessaire de complexifier le système avec des fils supplémentaires, d'autant plus que la vitesse permise par l'architecture UART est suffisante. Par ailleurs, le Shield disposant de cette architecture et également accompagné d'une bibliothèque de programmation constructeur ce qui peut se révéler très pratique. En outre, ce Shield présente des indicateurs LED RxD et TxD qui, comme nous l'avons déjà indiqué, peuvent se révéler salutaires. Nous privilégierons ainsi en priorité le Shield « *Serial CAN-BUS Module based on MCP2551 and MCP2515* », une photographie de ce dernier est disponible en Figure 3.5.

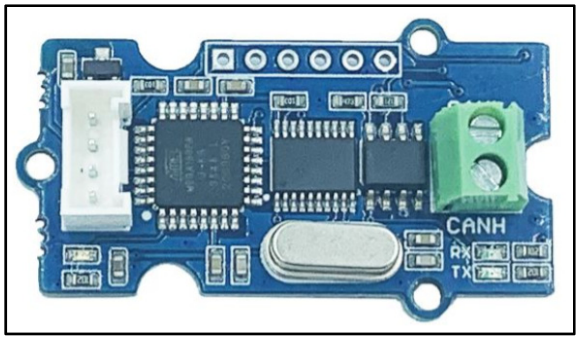


Figure 3.5 Shield Serial CAN-BUS de
Longan Labs pour la carte Arduino

3.5 Ressources logicielles

Dans les sections précédentes de ce chapitre, tous les composants physiques employés pour la conception de notre passerelle IIoT ont été étudiés, il nous reste désormais à nous intéresser à l'aspect logiciel. Les divers composants mentionnés ne sont qu'un support aux communications, il faut établir les instructions qui permettront à la passerelle d'interagir avec les autres éléments, de comprendre et d'assimiler les actions demandées.

Dans un premier temps, pour ce qui est des cartes Uno, nous nous orienterons par défaut vers le langage C/C++ puisqu'il s'agit du langage utilisé par l'environnement de développement officiel d'Arduino.

En revanche, en ce qui concerne le Raspberry Pi, le choix se doit d'être plus murement réfléchi. Les deux candidats principaux sont les langages C et Python. Plusieurs critères ont alors été considérés afin de conserver l'option la plus optimale. Le premier point concerne la complexité du langage. Python est un langage bien plus simple à maîtriser que C, il propose également plus d'options de base, avec peu de lignes de codes, le développement du programme en devient alors plus accessible. D'un point de vue de performance, il est à constater que C est un langage plus puissant que Python, moins gourmand en mémoire et il est également plus rapide puisque c'est un langage compilé (InterviewBit, 2022). Pour notre cas d'usage, les performances des deux langages sont toutefois toutes deux largement suffisantes. Enfin, un

critère important concerne l'adoption et la présence de bibliothèques de programmation. Ces bibliothèques, véritables recueils de fonctions, permettent à l'utilisateur d'avoir des fonctions déjà entièrement codées et ainsi d'économiser énormément d'effort et de temps. Ces bibliothèques vont être particulièrement cruciales pour notre développement car coder in extenso chaque protocole de la passerelle relèverait d'un travail titanesque. Sur ce point, Python est extrêmement plébiscité, son catalogue de bibliothèques est très fourni et nous permettra d'avoir accès à de nombreuses fonctions nécessaires pour notre programme. Au final, notre choix s'est ainsi porté sur le langage Python. Nous détaillerons dans la section suivante les différentes bibliothèques utilisées.

3.5.1 Bibliothèques de développement

Nous rappelons que les cartes Arduino sont utilisées pour simuler des machines industrielles échangeant des données par le biais de Modbus et de Canbus. Il est ainsi nécessaire d'avoir des bibliothèques implantant les fonctions de ces protocoles. Des bibliothèques supplémentaires seront requises pour assurer un fonctionnement sans encombre de la carte avec ses différents modules.

3.5.1.1 Arduino

Pour le protocole Canbus, nous tirerons parti de la bibliothèque « *Serial_CAN_Module* » (Lo, 2022). Il s'agit d'une bibliothèque Arduino élaborée par des développeurs de Longan Labs et conçue pour fonctionner avec leurs Shields, dont celui que nous avons choisi d'utiliser pour supporter les communications CAN.

En ce qui concerne le protocole Modbus, notre choix s'est porté sur la bibliothèque répondant au nom de « *ModbusSlave* » (Zamir et al., 2021). Ce projet, qui regroupe à l'heure actuelle 19 contributeurs, est très complet et offre la prise en charge de la totalité des fonctionnalités principales de Modbus. Ces fonctions, au nombre de huit, sont énumérées dans le Tableau 3.4. Elles permettent à l'utilisateur de manipuler les variables en dissociant d'une part leur type, et d'autre part leur mode d'accès.

Tableau 3.4 Les différentes fonctions Modbus

Code	Nom de la fonction	Description
01 (0x01)	*Read Coil*	Un *coil* est une variable booléenne, il s'agit d'un unique bit. Cette fonction permet la lecture d'une ou plusieurs variables.
02 (0x02)	*Read Discrete Input*	Une entrée discrète est une variable booléenne (1 bit). A la différence d'un *coil*, une entrée discrète n'est accessible uniquement en lecture. Cette fonction permet la lecture d'une ou plusieurs variables.
03 (0x03)	*Read Holding Register*	Cette fonction permet la lecture de registres de 16 bits. Nous traduirons par la suite les *holding registers* par « registres d'exploitation » pour éviter une confusion avec les autres types de registres.
04 (0x04)	*Read Input Register*	Cette fonction est similaire à la fonction 03, la distinction réside dans le type de registres traités. Il n'est possible d'accéder aux registres d'entrées seulement en lecture et non en écriture.
05 (0x05)	*Write Coil*	Cette instruction permet l'écriture d'un *coil*.
06 (0x06)	*Write Holding Register*	Cette instruction permet l'écriture d'un registre d'exploitation.
15 (0x0F)	*Write Multiple Coils*	Par le biais de cette fonction, l'utilisateur peut écrire simultanément plusieurs *coils* ayant des adresses voisines.
16 (0x10)	*Write Multiple Holding Registers*	La fonction 16 est identique à la fonction 15, mais s'adresse à des registres d'exploitation de 16 bits.

La bibliothèque « Servo » sera également utilisée, elle est intégrée de base à l'environnement de développement d'Arduino. Cette bibliothèque permet le contrôle du servomoteur présent dans le montage. De même, nous avons besoin de fonctions pour agir sur l'affichage de l'écran

LCD. Cela est rendu possible par la bibliothèque « *LiquidCrystal_I2C* » (Rickman et al., 2020). La bibliothèque intégrée « *LiquidCrystal* » n'est pas utilisée, car elle n'est pas compatible avec le type d'écran LCD que nous possédons, ce dernier présentant une interface I2C.

En addition, nous utiliserons aussi la bibliothèque « *avr/sleep* » qui est native à l'environnement Arduino. Elle offre la possibilité de placer la carte en mode sommeil et de gérer ce dernier. Cette option participe à la réduction de la consommation d'énergie des appareils embarqués qui se doit d'être la plus faible possible.

Enfin, la dernière bibliothèque Arduino à laquelle nous aurons recours se nomme « *SoftwareSerial* ». Présente nativement sur Arduino, elle permet d'accueillir des communications sérielles sur des broches choisies par l'utilisateur en émulant des ports Rx (*Receive*) et Tx (*Transmit*). Il est à noter que les cartes Uno disposent déjà d'une paire physique (non émulée) de broches Rx et Tx, respectivement numérotées 0 et 1. Ce choix est alors motivé principalement par un aspect pratique. En effet, lorsque les communications sérielles avec le bus sont effectuées via les broches par défaut, il n'est pas possible d'échanger avec le moniteur PC lié à l'environnement de développement. Cela peut se révéler contraignant car le moniteur peut permettre de déceler certaines erreurs en envoyant des requêtes ou bien en affichant les statuts de ces dernières. Nous avons donc fait le choix pour notre phase expérimentale d'utiliser des ports émulés. En outre, il est possible d'utiliser plusieurs ports émulés dans un même programme. Il faut toutefois préciser qu'une seule paire de port *Software Serial* à la fois peut être en écoute. Ainsi, pour faire fonctionner plusieurs ports *Software Serial* dans un programme il faudra répartir le temps d'écoute des différents ports en utilisant par exemple des algorithmes tels que celui du « tourniquet » (round-robin).

3.5.1.2 Raspberry Pi

Mise à part les deux HAT offrant une interface avec les bus de terrain, le Raspberry Pi n'est pas relié à plusieurs composants externes comme les cartes Arduino. Pourtant le nombre de

bibliothèques employées est paradoxalement plus important. La raison réside dans la complexité du programme. Les cartes Arduino ne constituent qu'un moyen permettant d'obtenir des échanges au sein des bus. En dehors des communications, leur programme ne requiert pas d'attention particulière, le code ne nécessite pas d'être perfectionné. En revanche, le Raspberry Pi représente le cœur de notre passerelle IIoT, l'ensemble du code doit ainsi être façonné minutieusement, en adéquation avec sa fonction. Le programme s'en retrouve alors plus riche, avec diverses fonctionnalités supplémentaires, plus complexes, et ne révélant pas nécessairement des communications en série. Par souci de clarté, les différentes bibliothèques requises pour le bon fonctionnement du programme sont alors listées succinctement dans le Tableau 3.5.

Tableau 3.5 Bibliothèques Python utilisées

Bibliothèque	**Fonction**
csv	La bibliothèque csv, intégrée de base à l'environnement Python, rend possible la prise en charge des fichiers au format CSV. Ce format est très utilisé pour le traitement de données.
json	json est une bibliothèque intégrée qui permet de traiter des informations sous le format JSON. L'utilité de cette dernière se révèlera lors des échanges de données entre la passerelle et le Cloud, certains Cloud supportent en effet ce format de données.
keyboard	La bibliothèque *keyboard* permet la création et la gestion d'évènements liés à l'activité des touches du clavier (BoppreH et al., 2020). Ces opérations se révèlerons utiles pour les interactions entre la console et l'utilisateur.
logging	Cette bibliothèque intégrée à Python permet d'obtenir une multitudes informations pratiques sur les opérations du programme et leur état. Elle est compatible avec un très grand nombre d'autres bibliothèques et se révèle essentielle dans une phase de développement pour déceler certaines erreurs.
os	Cette bibliothèque offre accès à certaines fonctions en rapport avec le système d'exploitation et sera ainsi requise à certaines étapes du programme. Elle est incluse de base dans l'environnement Python.
paho-mqtt	paho-mqtt est une bibliothèque Python créée et développée par Eclipse Foundation (Light, 2021). C'est par son biais que nous serons en mesure d'implanter les communications reposant sur le protocole MQTT au sein du programme.

Tableau 3.6 Bibliothèques Python utilisées (suite)

Nom de la bibliothèque	**Fonction**
pymodbus	pymodbus est une bibliothèque apportant l'ensemble des fonctionnalités de Modbus sous Python (Collins et al., 2021). Avec cette dernière, nous serons alors en mesure de recevoir des requêtes émanant du bus, et également d'en initier.
python-can	Cette bibliothèque a été conçue dans l'optique de rendre compatible le protocole Canbus avec Python (Thorne et al., 2022). python-can apporte ainsi les outils nécessaires pour transcrire les informations du bus et les utiliser au sein du programme.
requests	*requests* est une bibliothèque destinée à l'envoi de requêtes HTTP (Reitz et al., 2022). Elle a ainsi été sélectionnée pour l'échange de données entre la passerelle IIoT et le serveur infonuagique via le protocole HTTP.
RPI.GPIO	Cette bibliothèque est normalement présente sur tous les systèmes Linux destinés à fonctionner sur Raspberry Pi. Elle permet le contrôle des broches GPIO de l'appareil (Croston, 2022).
ssl	Cette bibliothèque permet l'implantation du protocole SSL et sera utilisée pour garantir des échanges sécurisés avec le Cloud. ssl est une bibliothèque native.
struct	Cette bibliothèque déjà intégrée à Python permet de manipuler des valeurs sous différentes formes et d'effectuer des conversions entre ces formes, elle sera particulièrement utile pour le traitement des données et l'intercompatibilité des protocoles.
sys	Nous utilisons la bibliothèque sys pour agir sur des variables en rapport avec l'environnement Python. La bibliothèque est incluse de base dans Python.
termios	Le module termios est disponible sur les exploitations Linux et offre une interface de contrôle des entrées et sorties d'un terminal. Nous aurons recours à cette bibliothèque pour la gestion du *buffer* (de la mémoire tampon) des commandes claviers de l'utilisateur.
time	*time* est une bibliothèque présente sur toutes les suites Python. Elle permet de réaliser toutes les opérations en lien avec le temps et l'horloge du système. *time* sera donc indispensable vis-à-vis des différentes sessions de communications effectuées.

3.6 Conception des montages

Les montages présentés dans cette section nous ont permis d'obtenir des bus CAN et Modbus pleinement fonctionnels et ainsi d'observer le comportement de la passerelle IIoT lors de l'utilisation de ces protocoles dans un environnement connu et contrôlé. Il est préférable de réaliser ces tests sur ce genre de montage lors d'une phase de développement. Ainsi, lorsqu'une erreur se produit, les répercussions sont nulles, ce qui n'aurait pas été le cas en conditions réelles dans une entreprise. Par ailleurs, nous pouvons moduler nos montages comme bon nous semble dans le but de les adapter aux fonctionnalités que l'on souhaite tester.

La Figure 3.6 représente la disposition des divers composants et de leurs branchements lors des essais avec le protocole Canbus.

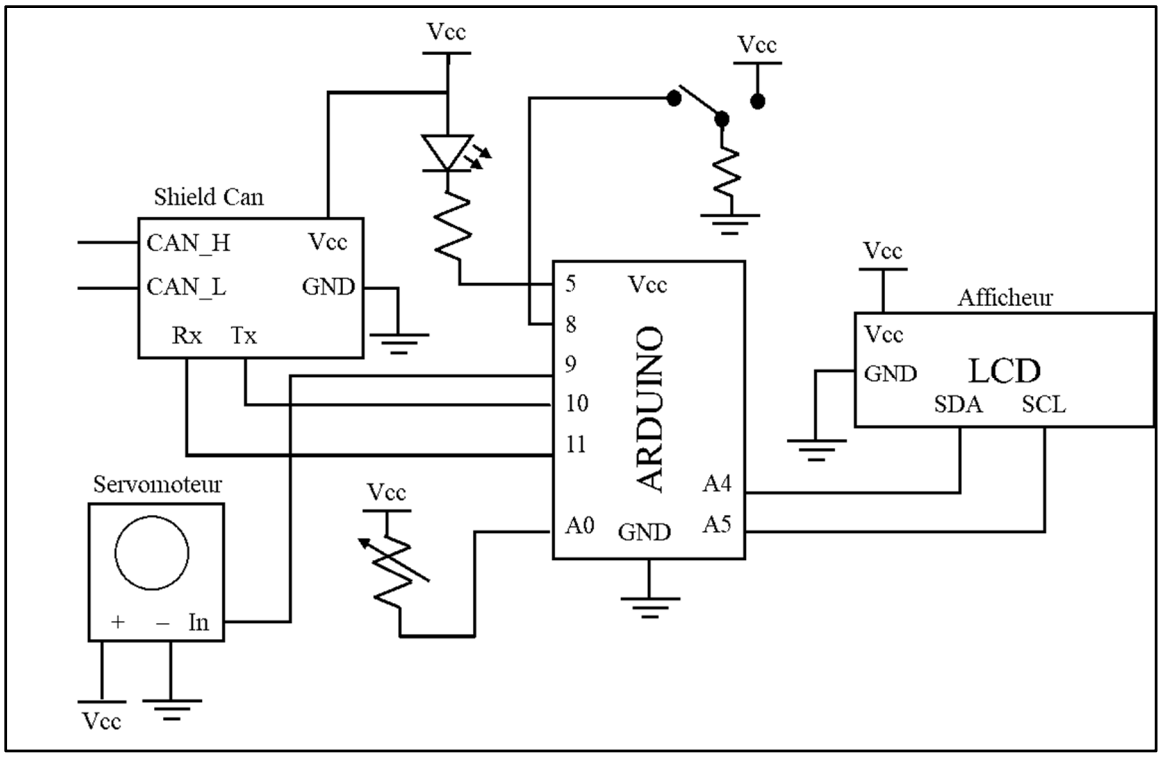


Figure 3.6 Schéma de montage d'un nœud Canbus avec une carte Arduino

Le Tableau 3.7 explique plus en détail les correspondances entre les branchements des différents éléments. Nous identifierons respectivement les signaux d'alimentation et de terre comme « Vcc commun » et « GND commun ».

Tableau 3.7 Branchements des composants pour le montage Canbus

Composant	**Branchements**
CAN Shield	• GND ⇔ GND commun • 5V ⇔ Vcc commun • RX ⇔ Broche 11 Arduino • TX ⇔ Broche 10 Arduino • CAN_H ⇔ CAN_H du bus • CAN_L ⇔ CAN_L du bus
Écran LCD	• GND ⇔ GND commun • VCC ⇔ Vcc commun • SDA ⇔ Broche A4 Arduino • SCL ⇔ Broche A5 Arduino
Interrupteur à bascule	• Broche 1 ⇔ GND commun • Broche 2 ⇔ Vcc commun • Broche 3 ⇔ Broche 8 Arduino
LED	• Broche + ⇔ Broche 5 Arduino • Broche – ⇔ GND commun
Potentiomètre	• Broche 1 ⇔ Vcc commun • Broche 2 ⇔ Broche A0 Arduino • Broche 3 ⇔ GND commun
Servomoteur	• Signal ⇔ Broche 9 Arduino • VCC ⇔ Vcc commun • GND ⇔ GND commun

Il est à préciser que, en plus des branchements indiqués dans le Tableau 3.7, il reste à connecter les broches 5V et GND de la carte Arduino respectivement au Vcc commun et au GND commun.

Ce montage permet d'incorporer des entrées et des sorties physiques à notre carte Arduino. D'un côté nous avons des éléments tels que le servomoteur, l'écran LCD ou bien la LED servant de témoins visuels à l'utilisateur et lui permettant de vérifier de manière concrète la bonne réception des données du bus. De l'autre, nous retrouvons des composants comme le potentiomètre ou l'interrupteur sur lesquels l'opérateur peut agir directement et ainsi contrôler les valeurs de certains registres de la carte Arduino.

Lorsque nous souhaitons utiliser nos cartes Arduino Uno comme nœuds Modbus, le montage reste presque inchangé. Nous utilisons les mêmes composants et ces derniers gardent la même disposition. Il suffit simplement de remplacer le Shield Canbus par le Shield Modbus. Les entrées et les sorties sont semblables, seules celles de l'interface avec le bus diffèrent, elles ne se présentent alors plus sous les noms « CAN_H » et « CAN_L » mais s'appellent « A+ » et « B- ». Les interfaces RS485 dont nous disposons possèdent en plus un signal GND permettant de garder une référence commune entre les nœuds, nous connecterons ainsi ce signal en complément.

Dans le cas de Modbus, nous pouvons rajouter un signal au niveau de la broche 2 de la carte Arduino. Cette broche est l'une des deux *Interrupt Pin* des cartes Uno. Par le biais de ces broches d'interruption nous pouvons contrôler la puce, et ce de manière externe. Cela offre la possibilité de mettre le système en sommeil profond pour un temps indéfini. Il est ensuite possible la faire réagir à un évènement et programmer son réveil à la réception d'un signal. Cette opération n'est possible qu'avec Modbus car il s'agit d'un protocole utilisant le principe du *Polling*, le nœud ne fait alors que réagir et n'a pas besoin d'agir par lui-même. Il est possible de programmer le coordonnateur pour envoyer un signal au nœud quelques millisecondes avant l'envoi d'une requête Modbus, le nœud recevra donc l'ordre de se réveiller et sera en mesure de réceptionner le message. Dans le cas de Canbus, le nœud doit être en mesure d'envoyer une

requête ou d'écouter le bus à n'importe quel moment, sans l'intervention d'un élément extérieur, ce type de mise en sommeil n'est donc pas envisageable. Il est toutefois possible d'effectuer des mises en sommeil, en boucle, avec un réveil programmé au bout d'un laps de temps défini. Enfin, les composants matériels servant aux divers montages de cette recherche sont présentés en ANNEXE III.

3.6.1 Montage de la Passerelle IIoT

Les connexions à faire entre les différents composants de la passerelle IIoT sont très rapides et il n'est pas possible de se tromper dans les branchements puisque les HAT Canbus et Modbus présentent chacun une connectique à 40 broches qui correspondent parfaitement à la tête GPIO de 40 broches du Raspberry Pi. Les broches réservées à chacun de ces deux HAT sont complémentaires, leur fonctionnement peut ainsi être simultané. Ces deux HAT possèdent une tête d'extension GPIO de 40 broches identique à celle du Raspberry Pi afin de permettre l'ajout de composants supplémentaires. Ainsi, dans l'idéal, nous pourrions directement empiler les deux HAT, cependant cette disposition bloque l'accès aux vis permettant de brancher et de débrancher les fils reliant la passerelle aux bus, ainsi que la visibilité des indicateurs LED. Nous réserverons alors cette configuration pour un produit développé et prêt à l'emploi, lors d'une phase de développement cette configuration n'est pas pratique. Pour pallier ce problème, nous nous sommes procurés le « *Stack HAT* », un composant développé par le même constructeur que les deux autres HAT et offrant la possibilité de connecter jusqu'à cinq HAT en série (Waveshare, 2019). La configuration finale de la passerelle IIoT est ainsi visible en Figure 3.7.

Pour que les HAT soient reconnus par le système d'exploitation et que les bonnes broches soient activées, il est nécessaire de rajouter plusieurs lignes de paramètres dans le fichier « /boot/config.txt » et d'effectuer un redémarrage de l'appareil. Pour le HAT RS485, il n'y aura qu'une ligne à rajouter (Waveshare, 2022) :

- `dtoverlay=sc16is752-spi1,int_pin=24`

Cette ligne de commande ajoute le dispositif matériel SC16IS752 en indiquant que le port SPI du circuit SC16IS752 doit échanger avec l'interface SPI1 et que la puce utilisera la broche 24 du GPIO comme broche d'interruption. Ces informations sont lues par le « *boot firmware* » lors du démarrage de la carte Raspberry Pi puis passées comme données au noyau du système d'exploitation pour fin d'initialisation. Malheureusement, l'interface de configuration dans Raspbian est très limitée et ne peut pas remplacer l'utilisation du fichier de configuration « config.txt » qui permet à l'utilisateur un paramétrage plus avancé. Pour le HAT CAN, il y aura trois lignes supplémentaires (Waveshare, 2021) :

- `dtparam=spi=on`
- `dtoverlay=mcp2515-can1,oscillator=16000000,interrupt=25`
- `dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=23`

Dans cette configuration, la première ligne active le port SPI principal (le SPI0), les deux autres lignes associent les interfaces aux vecteurs d'interruption 23 et 25. Elles indiquent aussi la fréquence en Hz de l'oscillateur du circuit MCP2515.

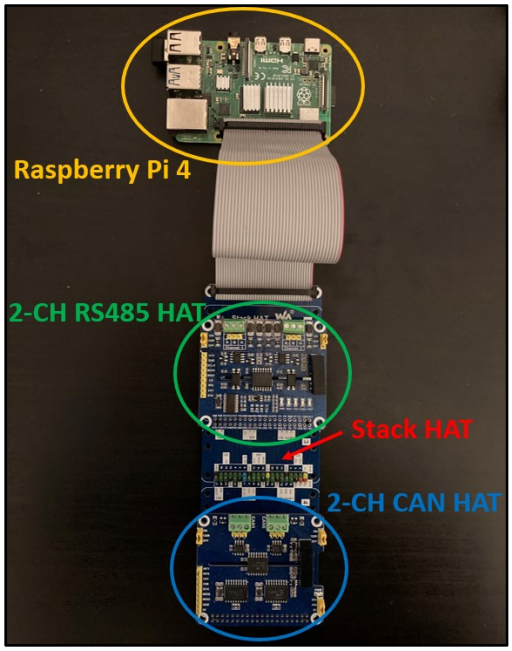


Figure 3.7 Photographie du
Raspberry Pi avec ses différents HAT

3.6.2 Interconnexions du bus de terrain

Comme mentionné dans le chapitre 2, il existe différentes topologies de bus de terrain, la Figure 3.8 illustre les différentes configurations existantes.

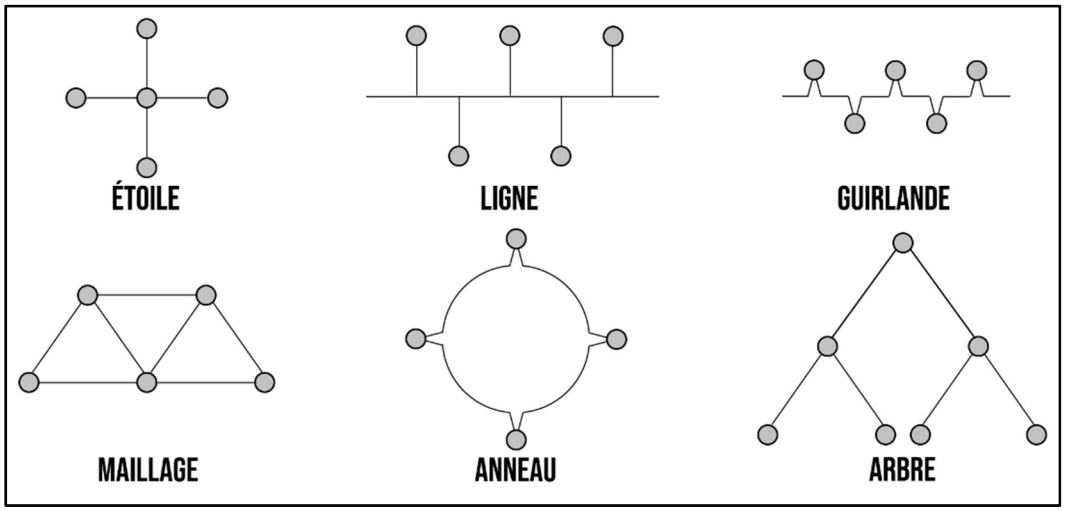


Figure 3.8 Différentes topologies de bus de terrain

Les topologies les plus populaires pour les deux bus de terrain sont en ligne ou en guirlande, néanmoins pour un bus en guirlande les modules doivent présenter deux connecteurs pour que le signal puisse passer séquentiellement dans chacun d'eux. Nos modules ne présentant qu'une interface, la configuration en ligne est sélectionnée.

Un schéma du montage est disponible sur la Figure 3.9. Dans ce montage, seule une carte Arduino a été intégrée par souci de clarté. De plus, seul le protocole Modbus est opérationnel ici pour les mêmes considérations. Il est à noter toutefois que les branchements sont presque identiques pour le protocole Canbus. Nous avons en revanche joint l'adaptateur USB-RS485 « SH_U11 » pour montrer un bus à plus de deux appareils et les branchements correspondants.

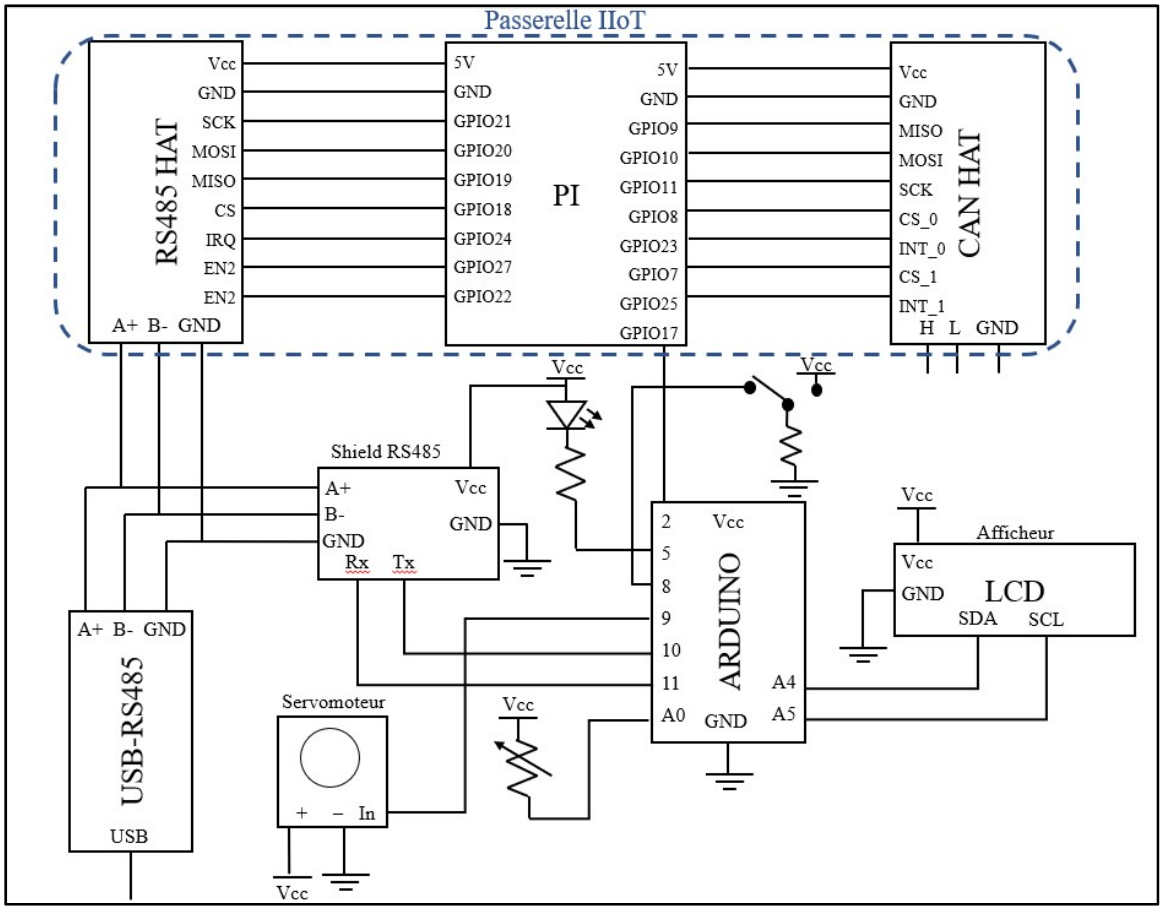


Figure 3.9 Schéma de montage partiel de l'environnement de test de la passerelle IIoT

L'organisation de l'environnement physique d'expérimentation est désormais terminée, il reste à s'atteler aux programmes Python et C qui permettront le fonctionnement des appareils. Nous rappelons que la liste des composants matériels de ce montage est présentée dans l'Annexe III, elle n'inclut pas le Raspberry Pi, les cartes Arduino, les HAT et les Shields.

Enfin, au niveau de la configuration des bus, nous devons prendre en considération les débits admissibles par les différents acteurs. Pour que les appareils appartenant à un même bus puissent se comprendre et communiquer sans encombre, les données échangées doivent être débitées à la même vitesse, on parle alors de *bitrate* ou de *baudrate*. Le *bitrate* se réfère au nombre de bits transférés en une seconde tandis que le b*audrate* est utilisé pour indiquer le nombre de pulsations du signal en une seconde. Il existe ainsi une nuance entre les deux notions

et parfois, dans les systèmes modernes, un baud peut comporter plusieurs bits, toutefois la plupart du temps *bitrate* et *baudrate* ont la même valeur. Si nous présentons les deux terminologies, c'est que chacune d'elle est présente dans la documentation et les bibliothèques utilisées. Ainsi, il sera question du *baudrate* dans les bibliothèques Modbus et de *bitrate* dans les bibliothèques Canbus.

Il faut ainsi définir des variables *baudrate* et *bitrate* que nous utiliserons comme paramètres dans les programmes, en s'assurant bien que les valeurs correspondent entre les différents appareils. Lorsque l'on définit les valeurs, il est également important de vérifier que chacun des composants est en mesure de supporter un tel débit conformément aux indications présentes dans leurs guides techniques.

3.7 Plateforme ThingSpeak et sa configuration

Les plateformes IIoT et leurs rôles ont été présentés en section 1.2.5, mais nous n'avions alors pas encore défini celle avec laquelle nous effectuerons nos tests. Parmi les différentes options ayant préalablement été présentées, notre choix s'est porté sur ThingSpeak.

ThingSpeak est une plateforme IIoT open source initialement disponible sur GitHub. Elle est désormais supportée par MathWorks, une entreprise très connue notamment pour son logiciel MATLAB. Cette version en ligne a le mérite d'être très simple d'utilisation, accessible directement depuis navigateur, et est compatible avec les deux protocoles de communications que nous avons sélectionnés. Elle présente en effet des interfaces pour MQTT et REST.

Outre le fait d'être très rapide à configurer, le service est gratuit. Avec l'offre gratuite, nous avons la possibilité d'échanger jusqu'à trois millions de messages par an. Une limite est également imposée sur la fréquence de réception des messages qui est limitée à 1 toutes les 15 secondes. Nous avons jugé cette limitation suffisante dans un premier temps pour effectuer nos tests. Dans l'optique où il serait nécessaire d'avoir de meilleures performances, il est possible de basculer sur un plan payant ou bien de changer de plateforme.

Enfin, il est important de noter que la plateforme IIoT offre plusieurs configurations de connexion via le protocole MQTT, ainsi on pourra choisir entre une connexion TCP et WebSocket, et avec la possibilité de chiffrer ou non les données aux moyens des protocoles TLS/SSL. Nous avons alors un certificat ThingSpeak dans les fichiers du Raspberry Pi qui permet de vérifier ces connexions et nous assure le droit d'échanger avec le broker MQTT. Ce certificat est un fichier portant l'extension .crt et contient une clé publique pour le chiffrement et une signature numérique certifiant que cette clé publique appartient bel et bien au broker MQTT.

Afin d'avoir une interface REST, il est nécessaire de créer une chaîne. Cette dernière possèdera des champs (ex : Température, humidité) et permettra le recueil et le stockage des données. La transmission de données à cette chaîne s'effectue par le biais de requêtes HTTP. Il est important de noter que la plateforme IIoT supporte quatre types de requêtes HTTP différentes, à savoir *GET*, *POST*, *DELETE* et *PUT*. La méthode *GET* est utilisée pour les requêtes visant à obtenir une information, comme les entrées d'un champ par exemple. La méthode *POST*, quant à elle, offre la possibilité de créer de nouvelles chaînes directement depuis la passerelle ou bien de fournir de nouvelles entrées. L'écriture de données peut être réalisée à l'aide d'un *GET* ou bien d'un *POST*, selon les préférences de l'utilisateur. La méthode *DELETE* est dédiée à la suppression de données, que ça soit les entrées d'une chaîne ou bien la chaîne en elle-même. Enfin, la méthode *PUT* est essentiellement utilisée pour la gestion des paramètres de la chaîne.

Pour ce qui est du protocole MQTT, une configuration préalable est également requise. Nous avons ainsi créé une chaîne supplémentaire. Il est possible de communiquer sur la même chaîne via HTTP et MQTT, mais nous avons choisi de concevoir deux chaînes distinctes pour mieux partitionner le travail et les tests de contrôle.

Que ce soit via REST ou via MQTT, les données échangées entre la passerelle et la plateforme se doivent d'être sécurisées, il est ainsi nécessaire de connaître un certain nombre de paramètres afin de pouvoir les incorporer par la suite à notre programme Python. Les différentes variables,

telles qu'elles sont nommées dans le programme, sont énumérées dans le Tableau 3.8. Peu importe la plateforme IIoT utilisée, ces paramètres seront sensiblement les mêmes.

Tableau 3.8 Liste des différentes variables de connexion à la plateforme IIoT

Protocole	**Variable**	**Description**
REST (HTTP)	rest_url	Il s'agit de l'URL à utiliser pour pouvoir effectuer les requêtes HTTP et communiquer avec l'interface REST de ThingSpeak. Cette variable compose la partie fixe de l'URL qui doit ensuite être concaténée avec les champs concernés par la requête. Sa valeur est « api.thingspeak.com/ ».
	rest_channel	Ce paramètre permet d'indiquer à ThingSpeak à quelle chaîne le message se destine. Sa valeur est « 1495776 ».
	rest_cle_utilisateur	La clé utilisateur est requise pour les requêtes de type *DELETE*. Cette clé permet ainsi de s'assurer que l'appareil est autorisé à effectuer une telle action. Sa valeur est « FUW3PCFBCX7XCBZA ».
	rest_cle_lecture	A l'instar de la clé utilisateur, la clé de lecture est utilisée à des fins de sécurité. Elle intervient alors dans toutes les requêtes destinées à la lecture d'une donnée. Sa valeur est « DU5TQTSCB7DFHR8P ».
	rest_cle_ecriture	La clé d'écriture présente les mêmes caractéristiques que la clé de lecture, mais s'adresse aux requêtes d'écriture. Sa valeur est « O520BJYN9V6HBSRY ».

Tableau 3.9 Liste des différentes variables de connexion à la plateforme IIoT (suite)

Protocole	Variable	Description
MQTT	mqtt_url	Au même titre que rest_url, cette variable permet d'accéder à l'API (*Application Programming Interface*) ThingSpeak du protocole correspondant. Sa valeur est « mqtt3.thingspeak.com ».
	mqtt_channel	La variable mqtt_channel est utilisée pour spécifier à quelle chaîne nous nous adressons. Sa valeur est « 1500138 ».
	mqtt_username	Le nom d'utilisateur, accompagné du mot de passe, permet d'être sûr que l'appareil est accrédité à échanger des données avec la plateforme. Cette combinaison est unique à chaque appareil. Sa valeur est « MiA5JhIGMSU9CCwrBDAcNTg ».
	mqtt_password	Il s'agit du mot de passe associé à l'identifiant ci-dessus. Sa valeur est « 9ygkTsZj1e0pXrwfK0z1S/iw ».

3.8 Conception du programme

De nombreuses fonctions devront être intégrées dans les programmes, pour être certain de n'oublier aucun élément nous avons tout d'abord étudié les différentes fonctions en amont de leur codage. Pour ce faire, nous avons utilisé le logiciel StarUML qui offre la possibilité de produire plusieurs types de diagrammes UML (*Unified Modeling Language*).

Les cartes Arduino sont utilisées comme des nœuds Modbus et non comme des coordonnatrices, cela limite le nombre d'actions différentes à implanter. Ces diverses actions sont présentées en Figure 3.10.

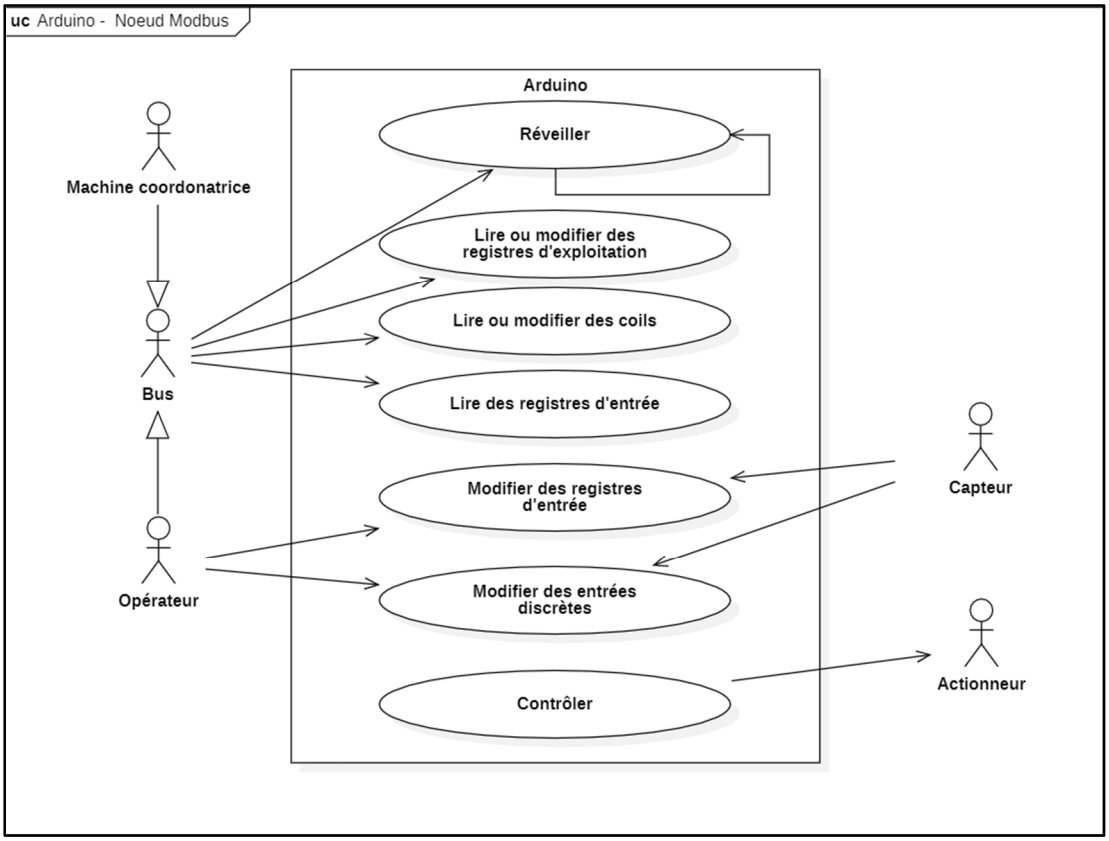


Figure 3.10 Diagramme des cas d'utilisation d'un nœud Modbus

Comme nous pouvons le constater, le microcontrôleur doit réagir aux sollicitations de plusieurs acteurs extérieurs. La fonction réveiller peut-être déclenchée de manière externe, par un signal provenant du coordonnateur, ou bien de manière interne à l'aide d'une boucle infinie et de séquences fixes de sommeil et d'éveil. La plupart des fonctions sont réalisées à la suite d'une requête émise par le bus à l'exception des variables d'entrées qui elles ne sont accessibles qu'en lecture par ce dernier. Ces variables se trouvent alors modifiées directement par le microcontrôleur et ses capteurs, comme pour le cas d'un thermomètre par exemple, ou bien par un opérateur. L'opérateur peut en effet agir sur les variables d'entrées de manière physique en influant sur la valeur du potentiomètre ou en manipulant l'interrupteur. Enfin, nous devons

introduire des fonctions qui se chargeront de contrôler les différents modules du nœud en réaction aux requêtes obtenues. Cela se traduira par exemple par l'allumage d'une LED, par la rotation du servomoteur ou bien par l'affichage d'un texte sur l'écran LCD.

Pour la mise en place de Canbus, l'approche est légèrement différente que pour Modbus puisque le bus est à accès universel. De plus, il n'y a pas de distinction entre les registres, les fonctions requises sont résumées sur la Figure 3.11.

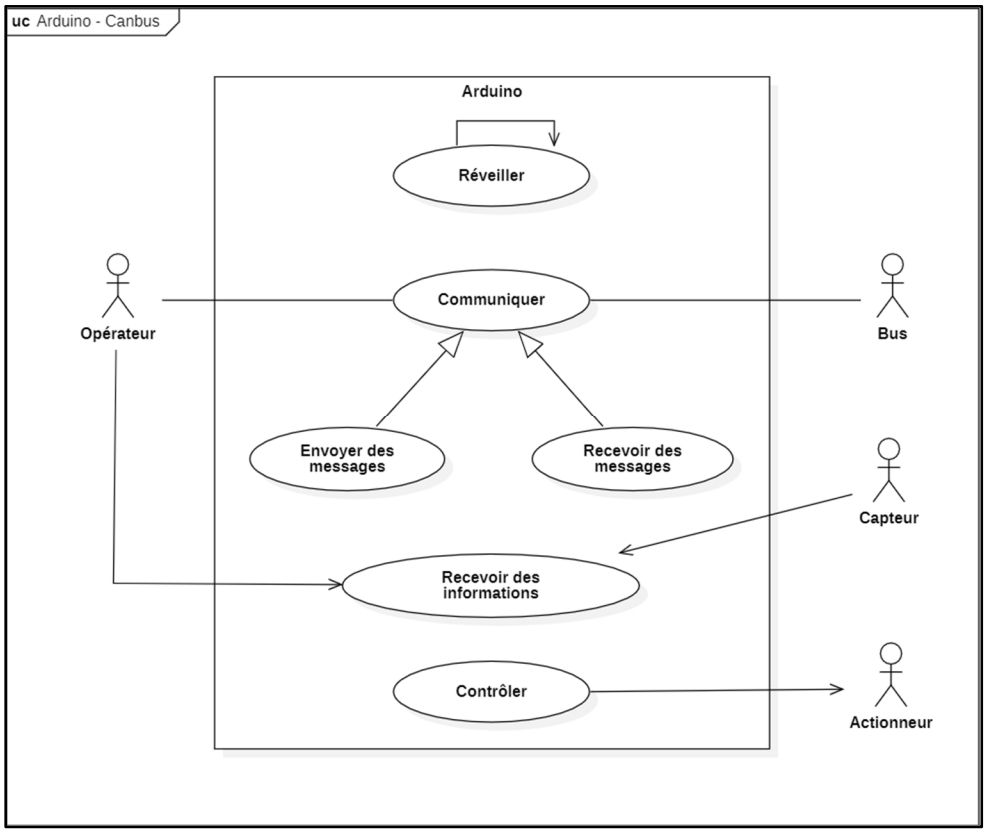


Figure 3.11 Diagramme des cas d'utilisation d'un appareil Canbus

Nous retrouvons ainsi la fonction de réveil, mais son déclenchement ne peut être que d'origine interne. Tout comme précédemment, certaines variables pourront être modifiées via les capteurs du système ou bien par l'opérateur, et la puce Arduino assurera le contrôle des différents modules. Pour ce qui est des interactions avec le bus, la carte Uno pourra envoyer des messages ou bien en recevoir. Il y a une multitude de paramètres différents qui peuvent

être échangés, on dénombre en effet 2048 adresses possibles donc 2048 actions différentes. Afin de garder le programme le plus simple possible, seules quelques-unes seront programmées.

Il a été décidé d'échanger une variable flottante pour vérifier le bon fonctionnement de l'encodage des nombres à virgules et la précision de la conversion. En outre, pour que les appareils soient conscients qu'il s'agit d'un nombre à virgule, nous avons décidé de dédier la plage 5XX à ce type de variables. Plus de précisions sur le traitement des nombres à virgule seront apportées dans la section 3.8.1.6.

Au niveau de la réception des messages, nous avons programmé le microcontrôleur pour ne réagir qu'à certains ID, ceux non définis sont alors ignorés par l'appareil. La distinction entre les ID est effectuée à l'aide d'un *switch case*, il est alors très aisé de rajouter des options au besoin.

3.8.1 Passerelle IIoT

Contrairement aux cartes Arduino ne supportant qu'un seul protocole à la fois, le programme Python de notre passerelle devra supporter d'une part les bus de terrain Modbus et Canbus, et d'autre part les protocoles MQTT et HTTP. Nous souhaitons que la passerelle soit modulable et qu'il soit relativement aisé pour un utilisateur de rajouter de nouveaux protocoles, pour ce faire nous avons opté pour une programmation orientée objet. Si l'utilisateur souhaite ajouter un nouveau protocole, il n'aura alors qu'à recréer une classe avec les fonctions spécifiques à ce dernier en supplément du code déjà existant.

Nous avons défini trois axes majeurs entre les différentes fonctions qu'il est nécessaire de traiter pour posséder une passerelle fonctionnelle. Il y a d'abord toutes les fonctionnalités permettant la prise en charge des différents bus de terrain., il y a ensuite tout ce qui concerne la configuration de la passerelle. Enfin, on retrouve toutes les instructions liées à la gestion des données et la communication avec le service infonuagique.

Nous avons exclu tout ce qui concernait la consultation et le traitement des données en lignes. Ces charges relèvent directement de la plateforme IIoT et non de la passerelle, les options sont en conséquence déjà incluses par le service ThingSpeak. Le diagramme des cas d'utilisation relevant des communications sur le bus est présenté en Figure 3.12. Il recense les différentes actions que la passerelle est en mesure d'assurer. Pour plus d'information sur le fonctionnement de la passerelle, l'ensemble des actions annotées dans cette figure sont détaillées en ANNEXE I.

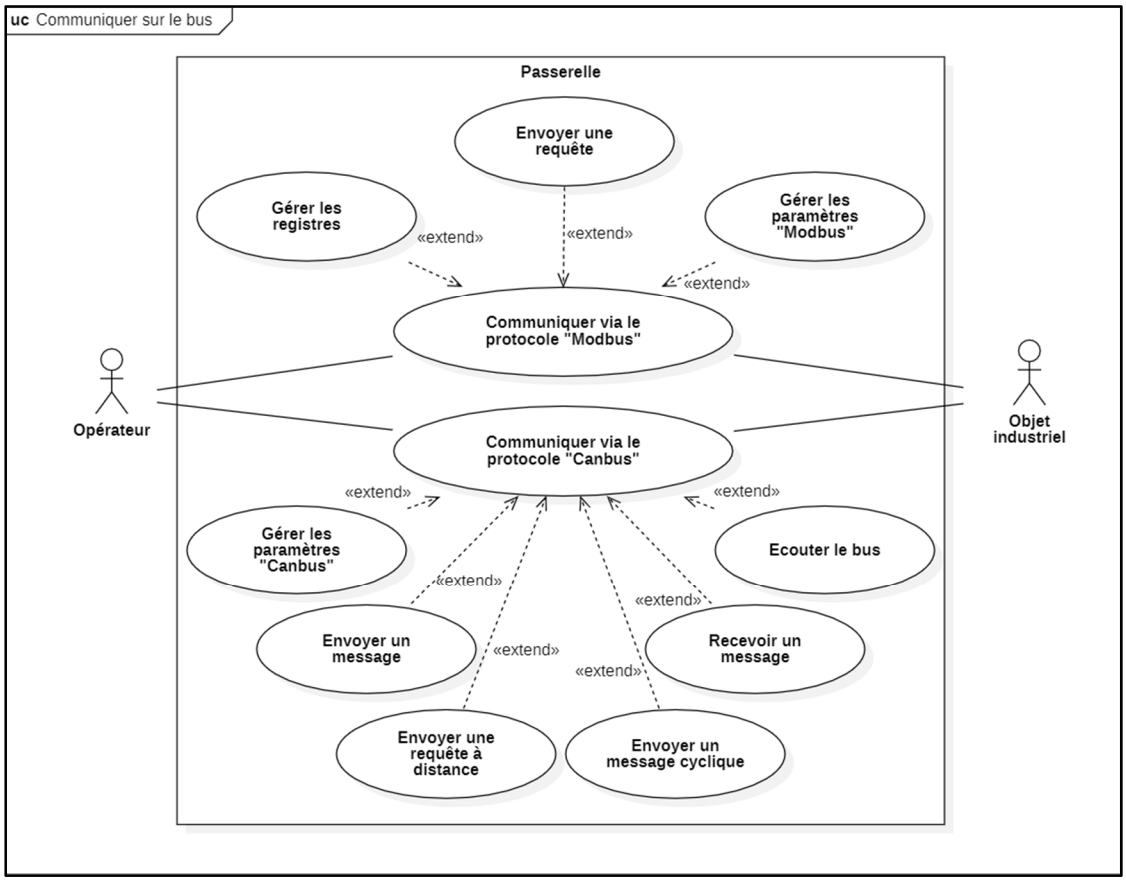


Figure 3.12 Diagramme des cas d'utilisation lors d'une communication avec un bus de terrain

En ce qui concerne le protocole Modbus, la passerelle doit être en mesure d'envoyer une requête afin de communiquer avec les différents nœuds du bus. Pour cela, les huit principales fonctions de Modbus seront intégrées au programme.

Depuis l'interface, l'opérateur doit également être en capacité de modifier les paramètres de communication avec le bus, cela comprend notamment le *baudrate*, l'adresse du nœud (qui par défaut est initialisée à 0x01), le port par lequel sont émis les messages et le mode d'encodage des flottants. Il est toujours possible de changer ces valeurs au sein du code Python, mais nous avons jugé plus pratique le fait de pouvoir réaliser ces changements sans avoir à relancer le programme. Il est à noter qu'il n'y a normalement pas lieu de changer le port puisqu'il dépend directement du HAT, mais cette option offre une meilleure compatibilité du programme qui pourra alors théoriquement fonctionner avec d'autres composants. Dans notre cas, la variable « `PORT` » se verra attribuer la valeur « `/dev/ttySC0` ».

Enfin, nous avons implanté une option permettant de définir le format des registres du nœud. Ce choix permet d'éviter certaines erreurs et de savoir à quelles adresses envoyer une requête. Par exemple, si le nœud ne possède que dix registres et que nous envoyons une requête d'accès au 11^{e} registre, une erreur se produira et la passerelle ne recevra aucune réponse car nous n'avons pas implanté d'instruction permettant la gestion de ce type d'erreur sur les cartes Arduino. Par ailleurs, nous avons choisi de dissocier les registres liés aux nombres entiers, aux nombres à virgules et aux caractères ASCII pour éviter d'avoir des aberrations à la lecture si les adresses des registres choisis ne sont pas conformes. Cela peut se produite lorsqu'on veut lire une chaine de caractères mais que l'adresse pointe vers un nombre flottant. Pour pallier ce problème, la fonctionnalité instaurée permet de définir les adresses réelles du premier registre de chaque type de variables (`int, float, str`) et leur longueur. À partir de ce point le programme réalisera par lui-même les décalages et la vérification de l'existence du registre visé avant l'envoi du message, et un avertissement sera alors transmis à l'utilisateur en cas d'erreur.

Pour ce qui est de Canbus, un panel de configuration est également disponible, il est semblable à celui de Modbus. Les paramètres modifiables dans ce menu sont le *bitrate*, la chaîne, le type de bus et le mode d'encodage des flottants. Par défaut et en accord avec la littérature, le nom de la chaîne est réglé sur « can0 ». Nous avons laissé la variable « `bustype` » car la bibliothèque CAN est compatible avec un bon nombre d'entre eux, notamment « serial », « socketcan », « pcan » ou encore « ixxat ». Nous utiliserons néanmoins « socketcan », cette dernière étant l'interface mentionnée dans la documentation du HAT. SocketCAN est une interface CAN open source destinée aux systèmes Linux.

La passerelle offre en outre la possibilité d'envoyer des messages Canbus, qu'il s'agisse de nombres entiers, flottants ou bien de chaînes de caractères. Il est également possible de recevoir les mêmes types de variables.

En addition à la fonction de réception d'un message qui va simplement retourner le dernier message reçu par la passerelle, une fonction d'écoute du bus est aussi implantée. Cette fonction est destinée à l'écoute du bus, c'est-à-dire la réception ininterrompue d'une quantité indéterminée de messages pour un temps défini par l'utilisateur. Il est aussi possible d'ajouter en paramètre l'ID du message à écouter et ainsi de filtrer les autres messages qui seront alors ignorés.

La passerelle est également en mesure d'envoyer des requêtes à distance. Ce genre de requête se résume à l'envoi d'une trame avec un champ de données nul, seul l'ID est renseigné. À la réception d'une telle trame de données, un appareil peut réagir en conséquence s'il a été programmé pour.

Pour finir, la dernière fonctionnalité développée dans ce programme consiste en l'envoi de messages cycliques. Le principe est donc de programmer la transmission d'une trame de données à intervalles réguliers. Cette fonction est notamment utile si on veut communiquer sur l'état d'une variable de manière régulière.

3.8.1.1 Configuration de la passerelle

Les fonctionnalités précédentes portent toute sur l'implantation et la compatibilité avec les bus de terrain. Les fonctions présentées dans cette section se rapportent plutôt au fonctionnement interne de la passerelle. La Figure 3.13 présente ces différentes fonctionnalités.

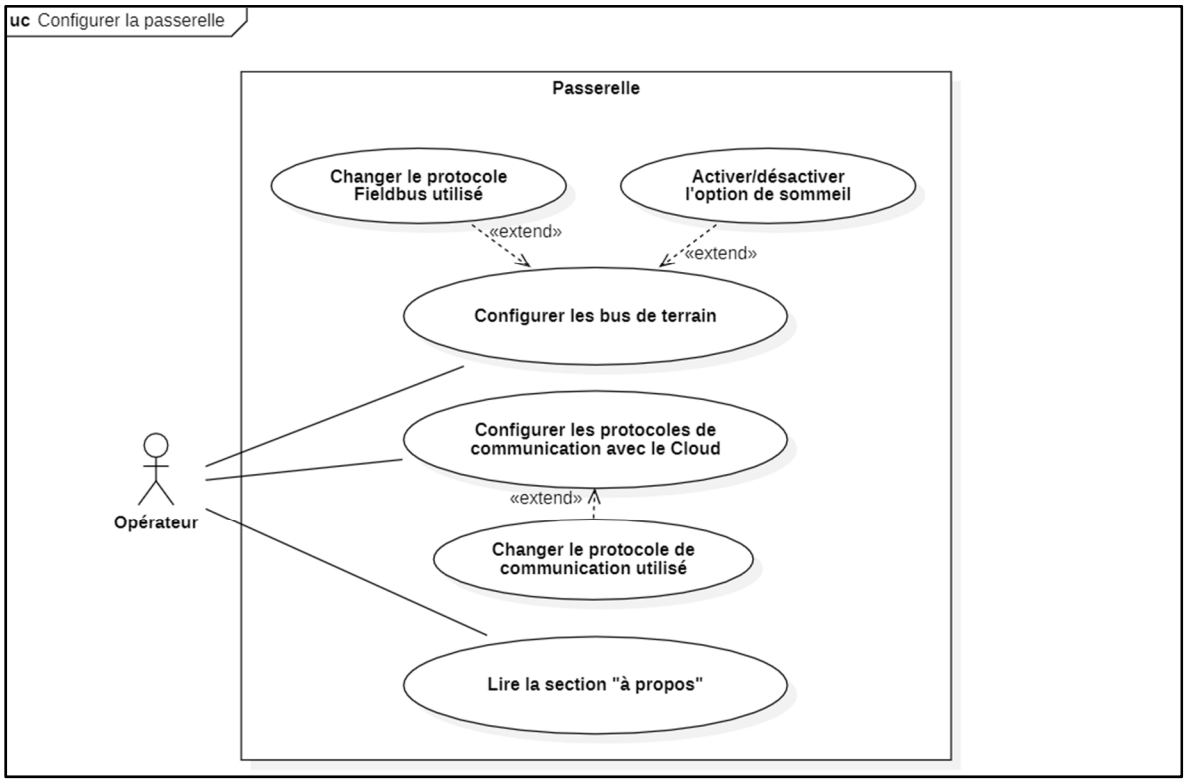


Figure 3.13 Diagramme des cas d'utilisation lors de la configuration de la passerelle

On dénote plusieurs cas d'utilisation associés à la configuration de la passerelle, que ce soit au niveau des bus de terrain ou bien au niveau des communications de plus haut niveau. En premier lieu, la passerelle dispose d'une fonctionnalité permettant de changer le bus de terrain employé en plein fonctionnement, sans avoir à redémarrer le programme, il en va de même pour le protocole de communication. Ainsi, l'utilisateur est libre d'alterner entre Modbus et Canbus, et entre MQTT et HTTP comme bon lui semble.

Nous avons également réalisé une fonction pour activer et désactiver la prise en charge du sommeil des nœuds. En d'autres termes, cette option permet d'activer ou non l'émission d'un signal électrique en amont de l'envoi d'une requête.

3.8.1.2 Base de données et service infonuagique

La Figure 3.14 illustre les diverses actions en rapport avec la gestion des données et leur stockage, que ce soit interne à la passerelle IIoT ou au niveau des communications avec le service infonuagique.

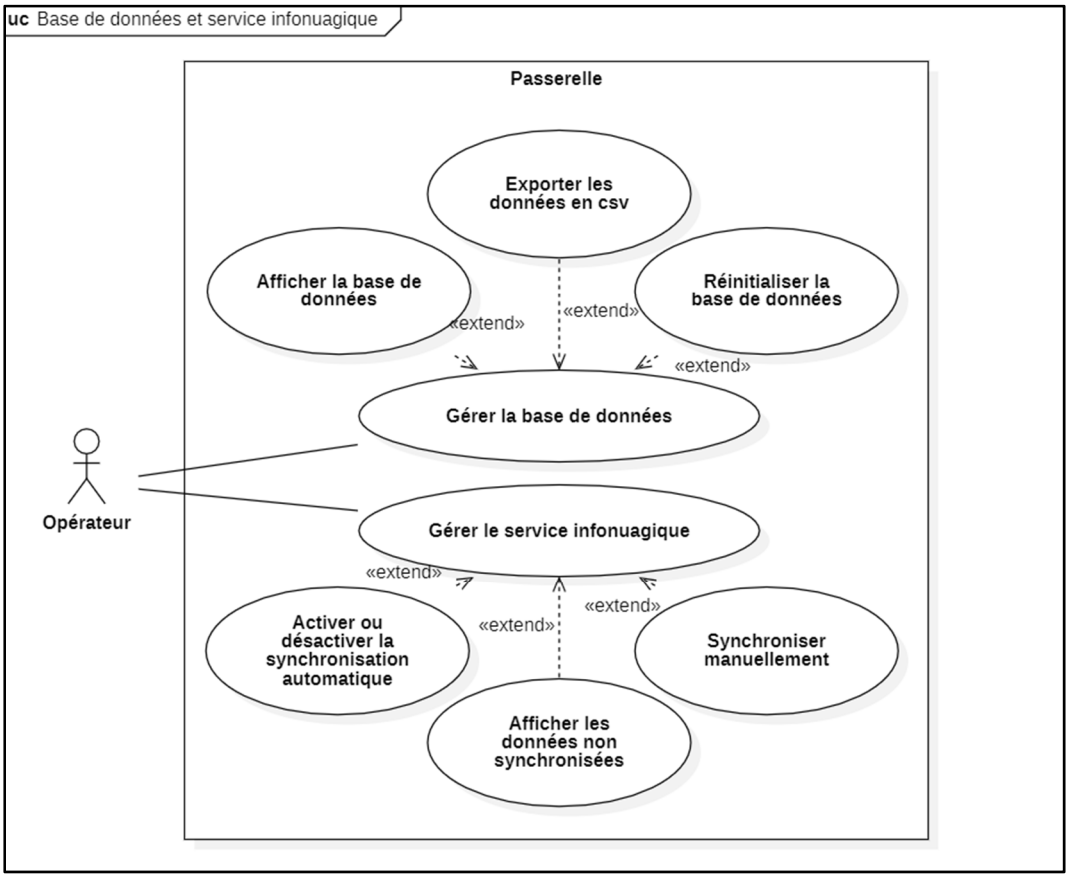


Figure 3.14 Diagramme des cas d'utilisation en rapport avec la gestion des données

Outre le fait que les données soient communiquées au service infonuagique, elles peuvent aussi être stockées au sein de la passerelle. Cette fonctionnalité peut se révéler particulièrement utile en cas de panne de réseau. Nous évitons ainsi d'éventuelles pertes d'informations dans le cas où la connexion avec la plateforme IIoT ne pourrait pas s'effectuer. Il est alors intéressant de pouvoir afficher les données récoltées par la passerelle durant le fonctionnement, par le biais de la console. Il existe aussi un mode permettant d'exporter ces données dans un fichier CSV. Enfin, une option proposant la réinitialisation des données est présente.

Concernant les fonctionnalités en rapport avec le service infonuagique, nous avons réalisé une option permettant de sélectionner si nous souhaitons que les données soient automatiquement transmises au Cloud. Par défaut, cette option est activée. Il est aussi possible d'afficher les données non synchronisées pour pouvoir les synchroniser soi-même.

3.8.1.3 Classes et fonctions

Après avoir identifié les fonctionnalités principales que devait assurer notre passerelle IIoT, il a fallu définir les diverses classes d'objets, les fonctions qu'elles contiennent et le reste des variables dont nous aurons besoin pour avoir un programme opérationnel. Afin de garder une organisation claire, nous avons utilisé des *design patterns*. Ces patrons de conception permettent d'avoir un code structuré, pratique et relativement standard pour faciliter sa compréhension par des personnes tierces. Au nombre de 23, ils ont été définis il y a presque 30 ans par le « *Gang of Four »* et sont toujours employés de nos jours (Gamma, Helm, Johnson, & Vlissides, 1994). En outre, cela accentue les propriétés modulaires de la passerelle IIoT, son adaptabilité s'en voit alors grandement augmentée. L'architecture générale du programme est alors illustrée Figure 3.15. Le diagramme des classes visible sur cette figure a été bâti en partie à l'aide des différents cas d'utilisation énumérés précédemment.

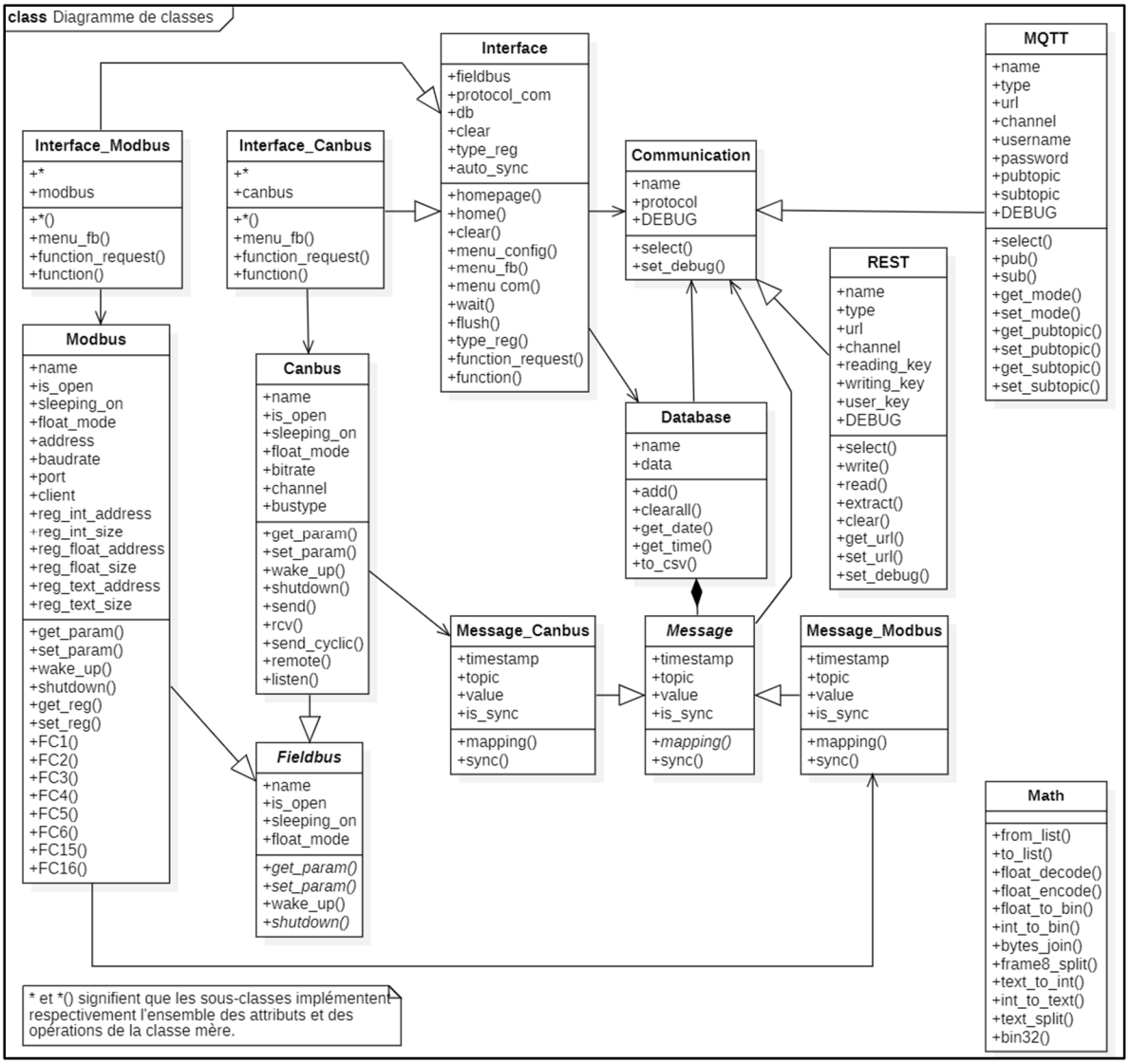


Figure 3.15 Diagramme des classes du programme Python

Il est à noter que nous ne nous sommes pas occupés du caractère public ou privé des fonctions et des variables, ce genre de considérations n'est en effet pas nécessaire avec le langage Python. Par ailleurs, dans chaque classe la fonction constructeur `__init__()` est toujours surchargée, nous ne l'avons cependant pas reportée par soucis de lisibilité. Cette remarque est aussi valable pour les fonctions `__str__()` et `__repr__()`.

Les différentes fonctionnalités prises en charge par la passerelle IIoT ont été réparties en classes distinctes pour avoir un code cohérent. Ainsi, chaque protocole aura sa propre classe et leurs instances respectives seront déclarées au lancement du programme. De ce fait, le fonctionnement de chaque protocole est indépendant des autres. On dénote alors la présence des classes `Modbus`, `Canbus`, `MQTT` et `REST`.

Pour que la passerelle puisse être en mesure d'opérer correctement, une interface de contrôle est nécessaire. Nous voulons pouvoir avoir la capacité d'échanger avec le programme, d'envoyer des instructions et également être en mesure de lire les résultats correspondants. C'est ici qu'intervient la classe « `Interface` », elle assurera le rôle de l'interface homme-machine. C'est par le biais de cette classe que l'utilisateur est en mesure de régler les paramètres, d'afficher les informations souhaitées et de transmettre les requêtes. Nous remarquons également la présence des classes `Interface_Modbus` et `Interface_Canbus`, il s'agit là d'adaptateurs et leur utilité sera exposée en section 3.8.1.5.

L'interface communique ensuite avec les protocoles de terrain, l'utilisateur a en effet renseigné les informations nécessaires pour l'envoi d'une trame de données. La classe `Modbus` ou `Canbus` instancie alors respectivement un objet de la classe `Message_Canbus` ou `Message_Modbus`, en fonction des données reçues par le bus. Ces messages peuvent ensuite être enregistrés et horodatés dans la base de données, une instance de la classe `Database`, pour avoir ensuite accès aux différentes fonctions de cette classe comme par exemple l'exportation des données en CSV. Les données de l'objet message peuvent directement être synchronisées avec le service infonuagique lors de sa génération ou bien ultérieurement, par l'intermédiaire de la base de données, c'est pourquoi les classes *`Message`* et `Database` interagissent avec la classe `Communication` et ses sous-classes `REST` et `MQTT`.

La classe `Communication` est utilisée comme base pour la définition spécifique des protocoles de communication. Elle regroupe toutes les fonctions qui seront communes aux différents protocoles de communication que l'utilisateur voudra implanter et permet ainsi d'avoir des méthodes déjà présentes grâce aux relations d'héritage suivant un principe de généralité-

spécificité. Ces classes réalisent les méthodes permettant d'effectuer des échanges avec les plateformes IIoT. Il n'a pas été possible d'avoir recours à un patron de conception car les divers protocoles sont trop disparates pour avec des fonctions communes. Il en va de même pour la classe *Fieldbus*. Cette classe offre une implantation facilitée de nouveaux protocoles de terrains en décrivant des attributs et des opérations qui se retrouveront et seront utiles pour chaque protocole. À l'image des classes et sous-classes assurant la gestion des protocoles de communications, les classes *Fieldbus*, Canbus et Modbus ne décrivent pas réellement la structure d'un patron de conception, mais leur agencement est toutefois assez intuitif pour l'utilisateur.

Il est à noter que la classe *Fieldbus*, au même titre que la classe *Message*, est représentée en italique, cela signifie qu'il s'agit de classes abstraites. Ce sont des classes n'étant pas destinées à être instanciées, mais qui en revanche contiennent des méthodes servant de modèles pour d'autres classes.

Pour ce qui est de la partie concernant la gestion des messages, et comprenant notamment les classes *Message*, Message_Canbus et Message_Modbus, le patron de méthode a été utilisé et sera discuté plus en détail en section 3.8.1.4. La présence de la classe Math sera quant à elle discutée dans la section 3.8.1.6.

3.8.1.4 Patrons de méthode

Les patrons de méthode sont des patrons de conception comportementaux permettant de définir un modèle pour l'exécution de ses méthodes. Ce modèle est contenu dans une classe mère abstraite et laisse la possibilité aux sous-classes de surcharger les méthodes suivant les besoins. La structure doit cependant rester inchangée par rapport à la classe abstraite. Un exemple de diagramme a été réalisé en Figure 3.16 et permet d'illustrer ce *design pattern*.

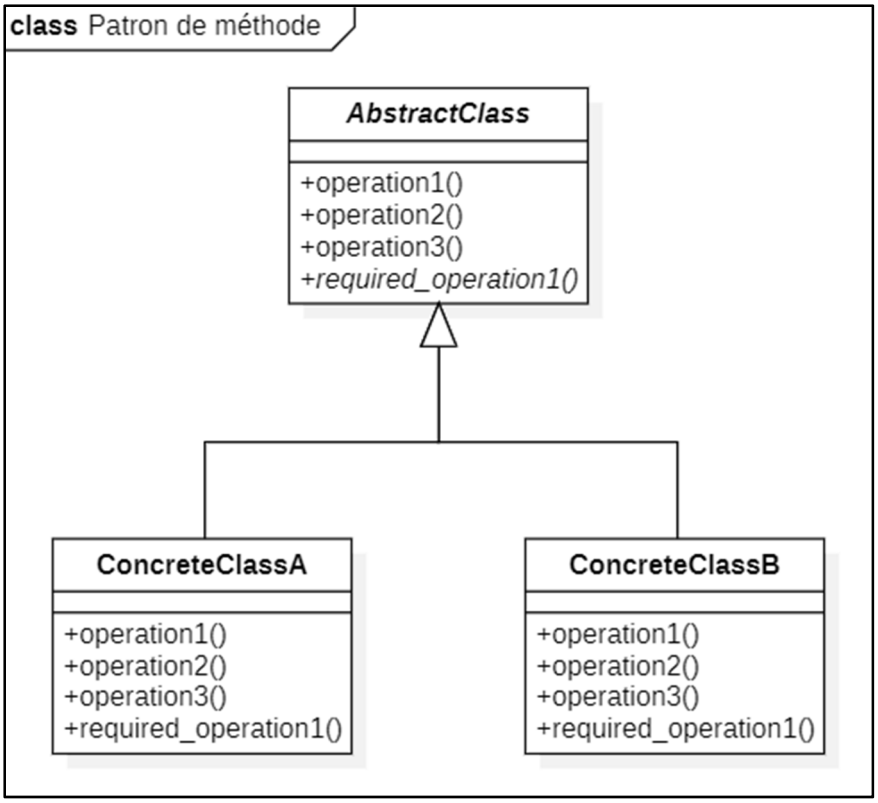


Figure 3.16 Diagramme UML du patron de méthode

Dans cet exemple, on peut apercevoir que la classe mère *`AbstractClass`* est bien une classe abstraite et qu'elle définit quatre fonctions différentes, dont une abstraite. Par conséquent, l'ensemble de ces fonctions doivent être reportées dans les classes concrètes `ConcreteClassA` et `ConcreteClassB`. Lorsqu'une opération est abstraite, comme c'est le cas de la fonction *`required_operation1()`*, cela signifie que cette dernière doit impérativement être définie dans les classes concrètes, en revanche les autres opérations telles que `operation1()` présentent déjà une implantation par défaut et ne requièrent pas nécessairement d'être redéfinies.

L'emploi de ce patron présente alors plusieurs avantages, il permet notamment de rassembler les opérations similaires dans une même classe et d'éviter les redondances dans le code. De plus, si l'on souhaite modifier ces opérations, il est alors possible d'éviter d'avoir à altérer toutes les classes.

En ce qui nous concerne, ce patron de méthode est utilisé pour l'instanciation et la gestion des messages, c'est-à-dire les objets des classes *Message*, Message_Modbus et Message_Canbus. La classe *Message* occupe le rôle de la classe *AbstractClass* et nos deux autres classes sont les classes concrètes. Leur nom est assez explicite quant aux protocoles auxquelles elles se réfèrent. Puisque les trames de données des protocoles Modbus et Canbus ne sont pas identiques et que les bibliothèques ne renvoient pas les données au même format, il est nécessaire de redéfinir les constructeurs __init__(). De plus, nous avons implanté la méthode mapping() dans chacune de ces deux classes, cette dernière étant définie comme une méthode abstraite au sein de la classe *Message*. En revanche la méthode sync() est déjà disponible par défaut et ne devrait à priori pas nécessiter de modification en cas d'ajout d'un nouveau protocole. Si tel était le cas, il est toujours possible de surcharger cette méthode.

Les interactions avec la base de données et les protocoles de communications se font alors sans encombre et font abstraction du type de message puisque chacun d'eux présente une structure identique. L'adoption du patron de méthode permet alors l'instauration d'un code clair qui simplifie les interactions avec les autres instances du programme.

3.8.1.5 Adaptateurs

Les adaptateurs sont des patrons structurels qui sont responsables d'établir une interface entre deux objets qui ne sont initialement pas en mesure de communiquer entre eux. Il existe deux types différents d'adaptateurs, on retrouve les adaptateurs d'objets et les adaptateurs de classes. Nous nous intéresserons uniquement aux adaptateurs d'objets. Un diagramme UML disponible en Figure 3.17 représente la structure générale de ce type de *design pattern*, sans égard pour le type de programmation utilisé.

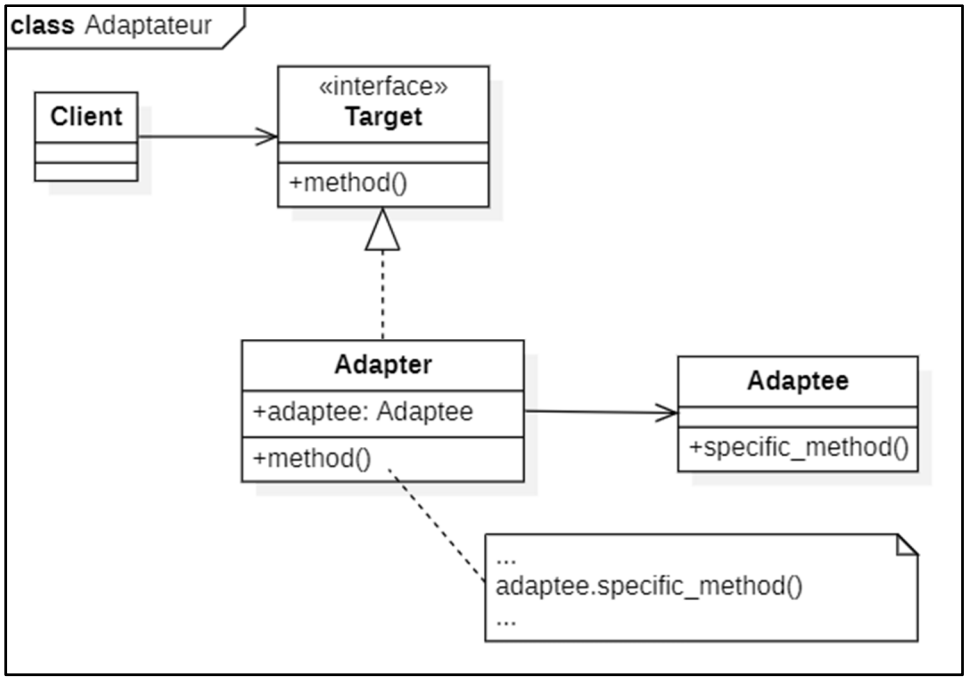


Figure 3.17 Diagramme UML du patron
de conception structurel Adaptateur

Dans la figure précédente, la classe centrale est nommée `Adapter`, d'où le nom du patron. Cette dernière sert à transcrire les informations fournies par les fonctions de la classe `Target` dans un langage compréhensible par la classe `Adaptee`. Sans l'ajout de cette classe, les classes `Target` et `Adaptee` ne pourraient pas interagir entre elles. Pour pallier ce problème, il est donc nécessaire de créer la classe `Adapter` qui sera en mesure de comprendre les deux autres classes. Elle va alors encapsuler l'objet à adapter, ici une instance de la classe `Adaptee`, et surcharger la méthode de la classe `Target` de sorte à modifier les instructions et appeler l'objet « adaptee » dans un format qu'il lui est familier.

Il est à noter qu'il n'y a pas vraiment de notion d'interface dans le langage Python, ainsi dans notre cas il faut considérer les classes `Client` et `Target` comme étant une seule est unique classe. L'adaptateur héritera alors des fonctions de cette classe. Il s'agit de la classe `Interface` dans le diagramme des classes de la Figure 3.15.

Dans notre programme, nous avons alors recours à ce patron à deux occasions, pour les classes `Interface_Modbus` et `Interface_Canbus`. L'objectif est d'établir un lien entre les classes regroupant les fonctions régissant l'usage des bus de terrain (respectivement `Modbus` et `Canbus`) et l'interface utilisateur. Les classes `Interface_Modbus` et `Interface_Canbus` permettent ainsi de traduire les instructions que l'utilisateur renseignent par le biais de la console et de les transmettre à l'unité traitant des bus de terrain.

Par ailleurs, si l'utilisateur souhaite ajouter un nouveau protocole de terrain « X » à la passerelle, il devra simplement créer une classe `X` héritant de la classe abstraite *`Fieldbus`* à l'instar des classes `Modbus` ou `Canbus`. Dans cette classe il sera alors nécessaire d'indiquer toutes les méthodes que le protocole utilise. Il faudra par la suite déclarer un adaptateur `Interface_X`. Par conséquent il n'y aura nul besoin de toucher à la classe interface, seules quelques bribes de cette dernière seront à surcharger via les méthodes de l'adaptateur, et il ne sera pas nécessaire de tout refaire. En outre, l'utilisateur devra également développer une nouvelle classe `Message_X` à l'aide du patron de méthode décrit dans la section précédente.

3.8.1.6 Nombres à virgule flottante et chaînes de caractères

Que ce soit Modbus comme Canbus, il n'est normalement pas possible d'échanger des données à virgule flottante (*floats*). Les trames de données émises sont composées de bits et ces bits permettent l'encodage de nombres entiers (*integers*). Ces nombres sont le seul format de données qu'il est possible d'envoyer en théorie. Pour remédier à ce problème, certains subterfuges ont été développés. Par conséquent, des solutions existent pour transmettre des nombres à virgule à l'aide de nombres entiers.

Nous désignerons la première méthode comme étant la méthode des valeurs fractionnaires. Il s'agit du moyen le plus simple pour obtenir un nombre flottant à partir d'un entier. Pour ce faire, trois paramètres sont requis. Nous avons bien entendu la valeur du nombre, qui correspond à un flottant *f* ou à un entier *i* suivant s'il s'agit de l'encodage ou du décodage de ce dernier. En complément, l'utilisateur doit préciser le rapport d'échelle *s* et la valeur de

décalage *d*. Le décodage du nombre flottant revient alors simplement à l'application d'une fonction affine :

$$f = i * s + d \tag{3.1}$$

De la même manière, pour encoder un nombre à virgule, il suffit simplement de calculer l'entier correspondant à l'aide de la formule inverse :

$$i = \left\lfloor \frac{f - d}{s} \right\rceil \tag{3.2}$$

Cette méthode est implantée dans notre programme, cependant ce n'est pas celle que nous utiliserons par défaut car il existe une méthode, à nos yeux, qui est plus optimale, notamment quant à la précision des valeurs qu'il est possible de transmettre. Cette méthode est présente dans la littérature sous le nom « *binary32* », ou encore « *float32* ». En anglais, le nom complet de la méthode est « *Single-precision floating-point format* », elle requiert 32 bits de données et permet une grande précision, jusqu'à sept chiffres après la virgule pour n'importe quel nombre compris entre -126 et 127. Par ailleurs, cette méthode a été normalisé par l'*Institute of Electrical and Electronics Engineers* sous le nom de « IEEE-754 » (IEEE, 2019). Dans cette trame de 32 bits, 1 bit est réservé au signe du nombre, 8 bits sont dédiés à l'exposant et enfin 23 bits composent la mantisse. L'équation utilisée pour le décodage d'un nombre flottant *f* est alors :

$$f = \text{sign} * m * 2^{e-127} \tag{3.3}$$

où sign est le signe (±), *m* est la mantisse et *e* est l'exposant. Au début de la mantisse, il y a un bit invisible qui vaut pour 1, le bit 23 prendra la valeur 1/2, le suivant 1/4, celui d'après 1/8 et ainsi de suite jusqu'au bit 0 (celui le plus à droite). Ainsi, la mantisse a toujours une valeur comprise entre 1 et 2. Cette règle présente une exception, dans le cas où l'exposant ne présente que des 0, il prend la valeur 2^{-126} et le bit invisible de la mantisse disparait, amenant alors son

intervalle de valeur entre 0 et 1. On précise que cette méthode entraine des petites erreurs d'arrondis, il n'est ainsi pas possible de coder parfaitement certains nombres.

Dans le cas de Modbus, la bibliothèque « pymodbus » dispose directement d'un constructeur intégré qui prend en charge la conversion les nombres flottants selon la même norme. Nous avons toutefois implanté cette fonction, car la bibliothèque Canbus n'inclus pas cette méthode. Il est à noter que la méthode IEEE-754 est également celle employée dans nos programmes Arduino.

Au vu de ce qui a été énoncé précédemment, il ne devrait pas être non plus possible de pouvoir échanger des chaînes de caractères via les bus de terrain. Il faut alors trouver un moyen de transformer les caractères textuels en nombres entiers. Pour cela, nous nous sommes orientés vers la norme Unicode et nous utilisons les représentations décimales des divers caractères. Il en résulte une association unique entre un caractère et un nombre entier.

Pour faire écho aux nombres flottants, la bibliothèque Modbus présente de nouveau une fonctionnalité permettant de prendre directement en charge la conversion des données, mais ce n'est pas le cas pour Canbus, c'est pourquoi nous avons conçu nos propres fonctions. Il est à noter que les plages d'ID 6XX seront attribuées arbitrairement à l'échange de chaînes de caractères.

En ce qui concerne les programmes Arduino, nous nous servons des pointeurs et utilisons la fonction « union » pour accéder à la représentation binaire des caractères transmis et pour les transposer en caractères ASCII.

Sur la Figure 3.15, on peut remarquer la présence d'une classe `Math`. Il ne s'agit en réalité pas vraiment d'une classe Python, mais plutôt d'un agrégat de fonctions rassemblées dans un programme annexe. Elles ont été regroupées ensemble car elles relèvent toutes du même rôle. Ces dernières ne sont pas attachées spécifiquement à une classe mais, au contraire,

interviennent à plusieurs endroits du programme. Ces fonctions apportent des outils essentiels au traitement des données.

Le reste des fonctions présentes dans le programme « Math » permettent d'organiser les données de sorte à ce qu'elles puissent être envoyées sur les bus de terrain. Nous rappelons que le plus grand nombre qu'il est possible de coder sur un octet est 255, pour les grands nombres il est alors intéressant d'être en mesure de décomposer ce nombre en puissances de 256, et ainsi d'avoir une liste étant l'image d'un nombre. Par exemple, le nombre 123 456 789 se présentera sous la forme [0, 0, 0, 0, 7, 91, 205, 21]. Il est alors possible d'envoyer le nombre sous cette forme. On limite la taille de la liste à huit octets car il s'agit de la taille maximale pour une trame Canbus, ce qui permet tout de même de coder des nombres jusqu'à valeur de 18 446 744 073 709 551 615. Nous avons également défini une fonction permettant de réaliser la conversion inverse.

Enfin, le code présente des fonctions destinées à joindre ou à couper des variables, que ce soit des listes, des nombres binaires ou bien des chaînes de caractères. Dans une trame de huit octets, on peut transmettre plusieurs nombres à la fois, ou deux nombres flottants de 32 bits par exemple, il est ainsi nécessaire de les dissocier avant d'effectuer la conversion. En outre, les opérations permettant d'influer sur un nombre binaire comme on le ferait sur une liste se révèlent utile pour réaliser la méthode « *binary32* » puisqu'il faut être en mesure d'accoler un bit invisible à la mantisse. La fonction de séparation des chaînes de caractères permet de s'assurer que chaque morceau ne dépassera pas huit octets. Si tel est le cas, son envoi se fera sur plusieurs trames à la suite.

3.9 Conclusion

Dans ce chapitre, nous avons ainsi pu cerner les objectifs de cette recherche, à savoir être en mesure de concevoir une passerelle IIoT fonctionnant avec les protocoles Modbus, Canbus, MQTT et HTTP.

La méthode de travail flexible SCRUM a été adoptée pour mener à bien ce projet. Cette dernière nous a permis de fractionner intelligemment la charge de travail, avec des échéances fixes et un suivi rigoureux.

Nous avons comparé différents composants se prêtant à la conception d'une passerelle et sélectionné les meilleurs candidats. Ainsi, la passerelle IIoT sera contrôlée par un Raspberry Pi 4 à 4 Go de RAM auquel on est venu greffer deux HAT permettant d'assurer les communications avec les deux bus de terrain. Concernant l'environnement d'expérimentations, nous avons sélectionné deux cartes Arduino pour leur simplicité et leurs performances. À l'instar des HAT pour le Raspberry Pi, nous avons investi dans des Shields Modbus et Canbus afin d'être en possession des interfaces compatibles avec les protocoles utilisés. Nous avons ensuite réalisé un montage avec divers composants externes tels qu'un servomoteur et un écran LCD afin de rendre les appareils plus complexes, plus similaires à leurs comparses industriels et d'avoir un retour visuel sur l'état des échanges. En supplément, un thermomètre Modbus ainsi qu'un adaptateur USB-RS485 ont été acquis pour la réalisation de tests préliminaires. Le montage des pièces a ensuite été réalisé.

Pour que le système soit configuré dans son intégralité, il a dès lors été nécessaire de choisir une plateforme IIoT. Après réflexion, la plateforme ThingSpeak semble être celle qui correspond le mieux à nos attentes. Divers réglages ont alors été effectués afin d'obtenir les informations de connexion aux interfaces MQTT et REST.

Par la suite, il a été question de la programmation des cartes Arduino et du Raspberry Pi, respectivement sous les langages C++ et Python. Nous avons alors étudié les fonctionnalités devant être incluses dans nos programmes et essayé de bâtir une architecture en conséquence. Cette architecture a été étudiée pour être facilement compréhensible pour un utilisateur quelconque et aussi pour permettre l'implantation de nouveaux protocoles dans le futur. Pour ce faire, un code basé sur une approche orientée objet a été adopté. Nous avons aussi utilisé les patrons de conception que sont l'Adaptateur et le Patron de méthode pour avoir un code se rapprochant des standards de programmation.

Nous avons en parallèle sondé Internet dans l'intention de trouver des bibliothèques prenant en charge les fonctions de bases de nos quatre protocoles et ainsi faciliter l'implantation de ces derniers. En conséquence, notre programme intègre, entre autres, les bibliothèques « pymodbus », « python-can », « paho-mqtt » et « *requests* », elles assurent respectivement le support de requêtes Modbus, Canbus, MQTT et HTTP. Pour ce qui est d'Arduino, seuls les protocoles Modbus et Canbus sont supportés, à l'aide des bibliothèques « *ModbusSlave* » et « *Serial_CAN_Module* ». Les autres bibliothèques employées permettent notamment l'apport de nouvelles fonctionnalités plus ou moins complexes et la gestion des modules externes.

Par ailleurs, la passerelle présente d'autres fonctionnalités externes à celles des protocoles. Une interface sous forme de console a été développée pour permettre à l'opérateur de discuter avec le bus, ou la base de données et aussi de modifier divers paramètres de la passerelle. On note également la prise en charge de variables représentant des nombres flottants ou bien des chaînes de caractères qui ne sont pas forcément compatibles avec les bus de terrain dans leur format initial.

La passerelle IIoT est désormais théoriquement prête à l'emploi, il reste maintenant à lancer une batterie d'expérimentations pour observer si elle répond conformément à nos attentes et si l'installation ne présente pas d'erreurs. Les résultats seront par conséquent discutés au chapitre suivant.

CHAPITRE 4

RÉSULTATS ET DISCUSSIONS

4.1 Introduction

La conception de la passerelle IIoT a été présentée dans le chapitre précédent, mais, pour valider les performances et estimer si elle correspond ou non à nos attentes, plusieurs tests ont dû être réalisés. Dans ce chapitre, nous allons par conséquent rapporter les résultats de validations effectuées au cours du travail de recherche. Ce chapitre servira non seulement à évaluer les performances du système, mais il servira également à présenter son fonctionnement en pratique. Nous observerons ainsi ce à quoi ce ressemble l'interface de la machine et comment l'utilisateur peut l'utiliser. Enfin, une attention toute particulière sera apportée à la discussion des résultats.

Nous mentionnerons dans un premier temps les mesures prises en amont des tests réalisés sur la passerelle IIoT. Nous procéderons par la suite à la validation des performances du système, en opérant tout d'abord sur le protocole Modbus. Les protocoles Canbus, MQTT et HTTP subirons ensuite les tests adéquats. L'ensemble des cas d'utilisation seront traités afin de rester le plus exhaustif possible. Les notions de robustesse et de fiabilité seront également abordées.

4.2 Validations préliminaires

Lors du processus de conception de la passerelle, Modbus est le premier protocole sur lequel nous nous sommes penchés, c'est donc par ce protocole que nous avons commencé les tests. Nous avons à notre disposition des composants Modbus qui ne nécessitent aucune configuration, une série de tests a ainsi été réalisée pour valider l'interconnexion physique et le protocole. Par ailleurs, cela été un bon moyen d'éprouver l'intégrité des différentes pièces. En effet, sans configuration requise l'erreur ne peut qu'être liée qu'aux composants ou à leur agencement.

4.2.1 Test du module Modbus sur PC

Le thermomètre Modbus AM2320 est un objet Modbus très simple, il se contente d'envoyer périodiquement des valeurs numériques. Cet objet remplit le rôle d'un nœud, pour lire ses données sur PC nous avons alors besoin d'un coordonnateur, nous utilisons donc le logiciel « *Simply Modbus Master* 8.1.2 ». Ce logiciel permet la communication entre un ordinateur et un dispositif Modbus. Pour ce faire, il suffit de relier le dispositif Modbus au port USB de l'ordinateur via un convertisseur USB-RS485 (voir Figure 4.1).

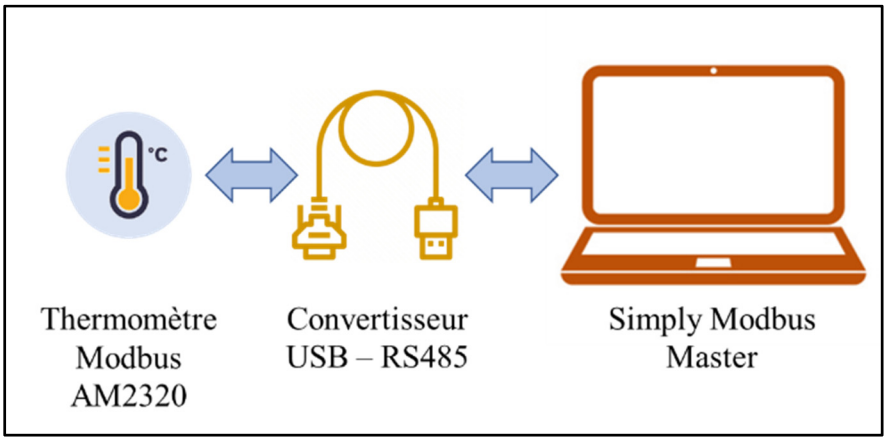


Figure 4.1 Illustration du montage
pour les tests préliminaires

Le résultat de l'échange des données entre le thermomètre AM2320 et le logiciel *Simply Modbus Master* est présenté en Figure 4.2. Cette figure correspond à la capture d'écran du logiciel mentionné précédemment juste après l'émission d'une requête et la réception de sa réponse.

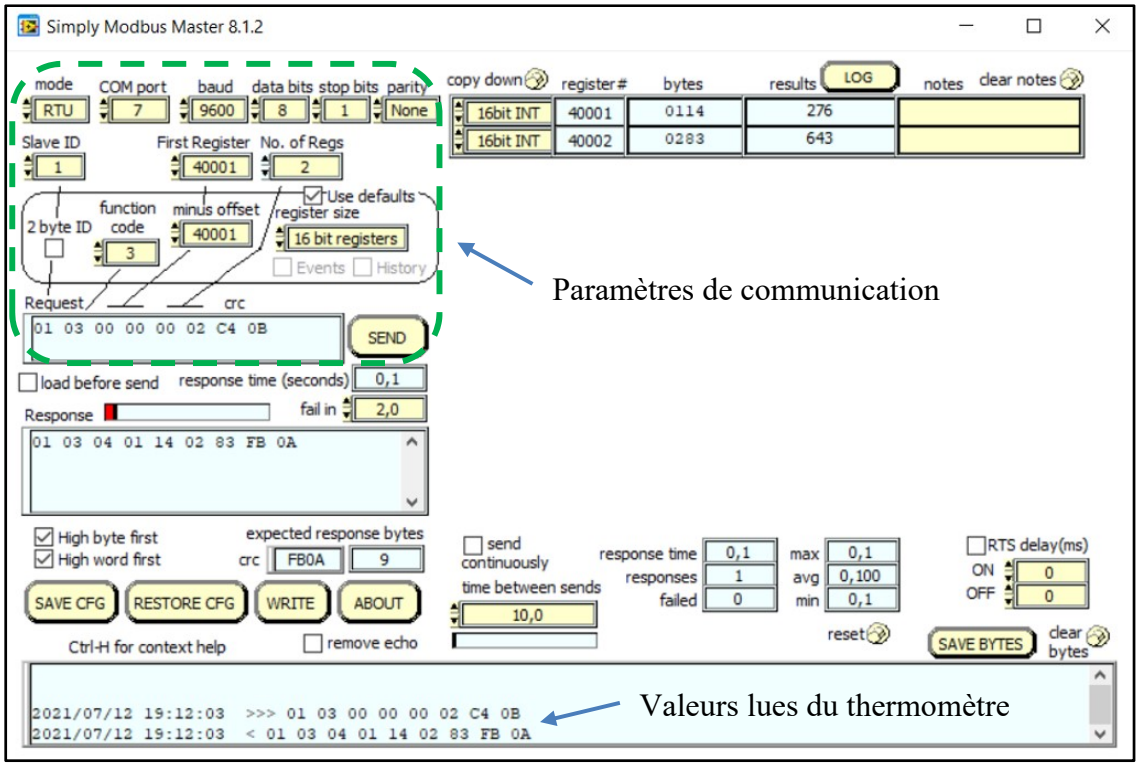


Figure 4.2 Échange de données entre le thermomètre Modbus et le logiciel PC

Nous rappelons que le thermomètre Modbus transmet deux valeurs, à savoir la température et l'humidité. Nous utilisons ici la fonction 03 (*Read Holding Register*) et nous indiquons que nous souhaitons lire deux registres. Comme l'appareil ne possède que deux registres, le choix n'est ici pas laissé, nous lirons les registres 0 et 1. Les spécifications du thermomètre nous imposent un *baudrate* de 9 600 bit/s et aucun bit de parité. Son adresse est 1, on se retrouve alors avec la requête visible sur la Figure 4.2. On souhaite communiquer avec l'appareil ayant pour adresse 01, avec la fonction 03, en commençant par le registre d'adresse 00 et 02 registres seront lus. Il y a ensuite le CRC. De même, en réponse, la trame nous indique qu'elle concerne l'adresse 01, que c'est une réponse à une fonction 03, que 04 octets sont transmis (deux octets par registres), puis outre le CRC final nous avons les valeurs 0114 et 0283. Ce sont les équivalences hexadécimales de 276 et 643. Nous reconnaissons ici les valeurs initialement recherchées, moyennant un facteur 10. La transmission est ainsi un succès, la température se trouve être de 27.6 °C avec une humidité à hauteur de 64.3%. Nous tenons à préciser que la durée d'un échange Modbus (émission et réception d'une trame de données) est de l'ordre

d'une dizaine de millisecondes. La durée de la plupart des tests s'en retrouve alors sensiblement identique, les manipulations humaines étant généralement les parties les plus chronophages.

4.2.2 Test de la carte Arduino

Après avoir pris en main le protocole Modbus, il a été nécessaire de configurer correctement les cartes Arduino pour qu'elles opèrent de façon adéquate à nos attentes. Elles représentent en effet les principaux collaborateurs de notre passerelle IIoT lors des échanges via les protocoles de terrain. Nous avons précédemment expliqué pourquoi nous avions décidé de dissocier les différents registres Modbus et les types de variables qu'ils contenaient. Dans cette sous-section, nous allons définir le nombre de registres que nous souhaitons allouer à chaque type de variable. Les valeurs annoncées seront valables pour toutes les expériences à venir.

Tout d'abord, pour ce qui est des *coils*, nous avons choisi d'allouer cinq adresses. Nous rappelons qu'un coil est un type de registre Modbus d'une taille de 1 bit accessible en lecture comme en écriture. Ces variables booléennes sont utilisées pour le contrôle de sorties discrètes, ces dernières ne possèdent alors que deux états, à savoir « vrai » ou « faux ». Chacune de ces adresses étant associée à une E/S de la carte Arduino. Ainsi, les adresses 0 à 4 seront respectivement associées aux broches 3 à 7. Nous n'avons pas utilisé les broches 0 et 1 puisque nous les gardons de côté dans le cas où nous en aurions besoin pour des communications séries. La broche 2, quant à elle, est dédiée aux actions liées au réveil de l'appareil. La LED présente dans le montage est reliée à l'entrée 5 conformément aux branchements illustrés sur la Figure 3.6.

En ce qui concerne les entrées discrètes, il n'y aura qu'un unique registre. Il sera associé à l'E/S de la broche 8 qui est relié à l'interrupteur à bascule. Nous aurons six registres pour les registres d'entrées de sorte à ce que chacune des E/S analogique (A0, A1, A2, A3, A4, A5) de la carte soient allouée. Il est à noter que le potentiomètre est connecté à la broche A0, soit au registre d'entrées d'adresse 0.

Enfin, les registres d'exploitation sont ceux affectés par les potentiels problèmes liés aux types de variables, ils seront donc plus nombreux. Nous aurons ainsi 18 registres de ce type. Les quatre premiers seront utilisés pour les `ints`, de sorte à pouvoir stocker quatre `ints`. Les six adresses qui suivent sont réservées pour les `floats`. Comme un `float` est codé sur 32 bits, ou sur deux registres, nous pourrons ainsi stocker trois variables de ce type. Pour finir, nous réserverons huit registres pour les `strings`. Une ligne de notre écran LCD est composée de 16 caractères et chaque registre peut renfermer deux caractères, ces registres peuvent alors nous permettre d'écrire sur toute la première ligne de l'écran, la deuxième étant réservée à d'autres fins.

Il est bien entendu possible d'avoir un nombre de registres bien plus conséquent, mais cela n'aurait rien apporté dans notre cas. Toutes les entrées et sorties de la carte peuvent être contrôlées avec les divers registres définis. Nous avons ainsi fait le choix de réduire au maximum le nombre de registres définis par souci de clarté.

Il n'est pas souhaitable de tester le fonctionnement en simultané de notre passerelle IIoT et des nœuds Arduino, d'autant plus que nous avons une alternative. Nous avons alors utilisé le logiciel PC pour agir sur les registres Arduino et voir si la carte réagit correctement aux requêtes envoyées. Le montage se retrouve être le même que celui présenté au chapitre 3 en ignorant le Raspberry Pi. L'objectif est ici de s'assurer que les cartes Arduino fonctionnent sans encombre pour pouvoir par la suite se focaliser sur le fonctionnement de la passerelle IIoT.

Nous avons ainsi testé les différentes fonctions existantes afin d'observer la robustesse de notre programme Arduino. La capture d'écran de la Figure 4.3 a été effectuée lors des essais de la fonction 06 (*Write Holding Register*).

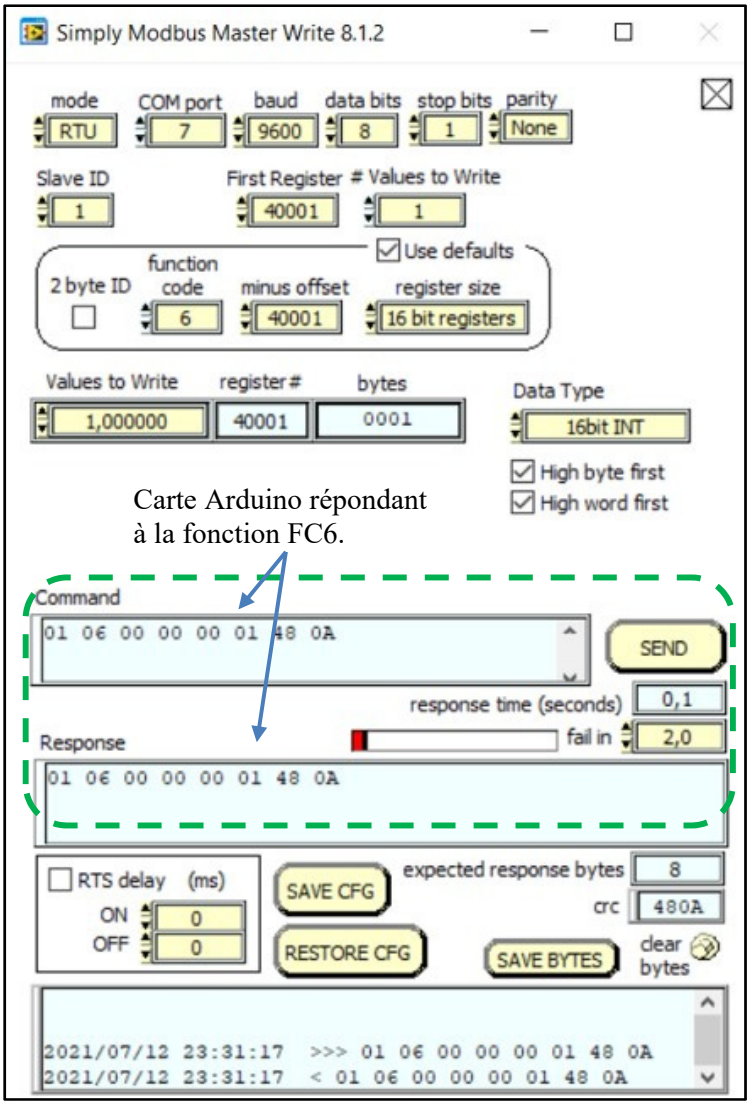


Figure 4.3 Communication Modbus entre le logiciel PC et le nœud Arduino

Il est à noter que contrairement aux requêtes de lecture où des valeurs sont attendues, le nœud n'a ici aucune information à renvoyer. Ainsi, pour signaler au coordonnateur que la requête a bien été reçue et traitée, la même trame de données est renvoyée en écho. La réponse est donc identique à la demande. C'est effectivement ce qu'on peut observer sur la Figure 4.3 avec la trame « 01 06 00 00 00 01 48 0A ». Comme nous pouvons le deviner, il a ici été demandé d'écrire la valeur 1 dans le registre 0.

Ces tests nous ont permis de mettre à l'essai notre programme Modbus pour Arduino et de le perfectionner. En outre, nous avons pu constater que le matériel fonctionnait correctement.

4.3 Validation de la Passerelle IIoT

Dans cette sous-section, nous allons présenter la console via laquelle l'utilisateur est en mesure de communiquer avec la passerelle IIoT. De fait, nous allons présenter les divers menus et les actions proposées par ces derniers. Il est à préciser que tous les tests ont été effectués avec des débits de transmission respectivement initialisés à 9 600 bit/s et 125 000 bit/s pour Modbus et Canbus. Le réglage correspondant à 9 600 bit/s est le plus couramment employé par les appareils Modbus, c'est en quelque sorte une valeur par défaut que tous les appareils supportent. Pour ce qui est de Canbus, nous avons choisi un débit de 125 kbit/s pour les mêmes considérations. Il s'agit d'ailleurs des valeurs renseignées par défaut dans les différentes bibliothèques utilisées.

Le reste de cette sous-section a principalement vocation à présenter l'organisation de l'interface utilisateur et ses menus principaux. L'utilisateur est redirigé vers l'écran d'accueil lorsqu'il démarre le programme. Cet écran d'accueil ressemble à celui de la Figure 4.4.

Au lancement, le programme commence par instancier les différents protocoles pris en charge par la passerelle. Les protocoles utilisés par défaut sont également présentés à l'utilisateur. Dans notre cas, nous avons choisi Modbus et MQTT. Puisqu'il s'agit du protocole Modbus, une fonction permettant le choix du type de variables échangées a été intégrée. Ce choix est toujours modifiable ultérieurement via le sous-menu correspondant.

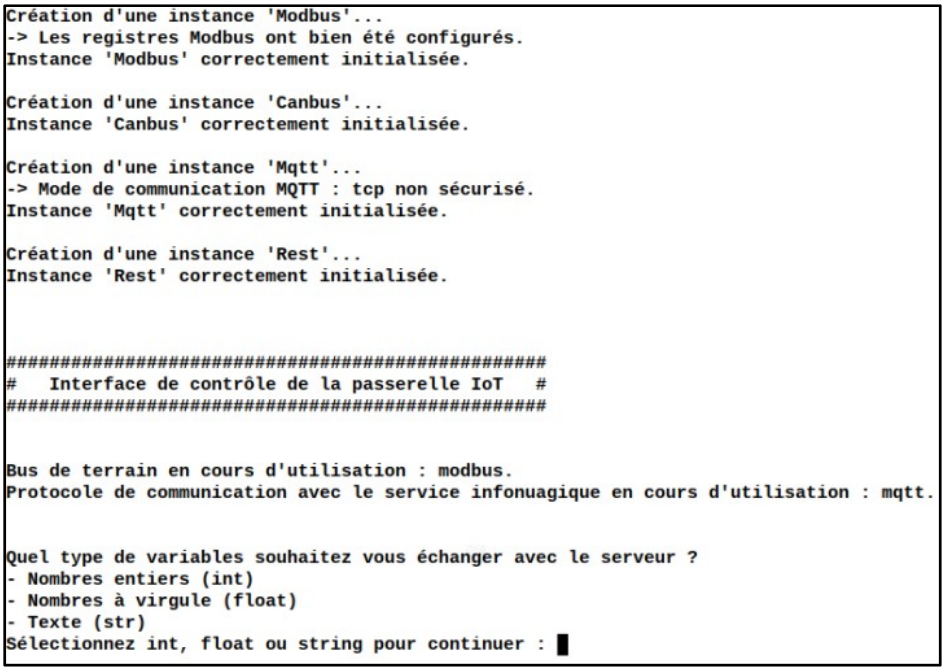


Figure 4.4 Écran de la page d'accueil

Après avoir renseigné, s'il y a lieu, le type de variables à échanger, un menu principal est présenté à l'utilisateur tel qu'on peut le voir sur la Figure 4.5.

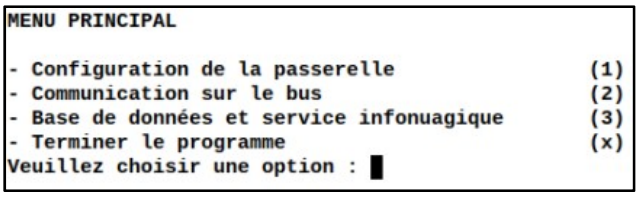


Figure 4.5 Écran du menu principal

Les trois options proposées à l'utilisateur sont celles dont nous avons discuté en section 3.8.1. Parmi ces trois options, nous attarderons en détail sur celle dénommée « Communication sur le bus ». Il s'agit en effet de la plus dense et la plus complexe, elle renferme toutes les fonctions de communication avec les bus de terrain. Nous étudierons par la suite l'option « Base de données et services infonuagiques ». Les fonctions proposées dans le menu « Configuration de la passerelle » reflètent les descriptions faites en sections 3.8.1.1. Ces fonctions ne seront pas mentionnées plus en détails dans ce chapitre car il ne s'agit que de fonctions relatives à la

gestion de variables Python et ne relèvent aucunement de l'un des quatre protocoles implantés. Des captures d'écran de ce menu sont disponibles en ANNEXE II.

4.3.1 Test des protocoles de terrain

L'ensemble des fonctionnalités en lien avec ces protocoles est accessible via l'option 2 du menu principal. Ce menu est également désigné comme étant le « menu opérateur » puisque c'est par son biais que l'utilisateur pourra opérer sur le bus. Les multiples actions proposées par l'intermédiaire de ce menu diffèrent en fonction du bus de terrain utilisé. Ce changement s'effectue automatiquement lorsque l'on décide de modifier le bus cible.

4.3.1.1 Fonctions Modbus

Via le menu opérateur, il est possible d'effectuer l'ensemble des huit fonctions Modbus implantées. L'utilisateur peut également choisir d'afficher les paramètres du bus, de modifier l'adresse du nœud auquel la future requête sera émise, de changer le type de données à transmettre ou encore d'agir sur la configuration des registres. La liste de ces diverses actions telles qu'elles sont présentées à l'opérateur sont illustrées en Figure 4.6.

```
MENU OPERATEUR

Bus en cours d'utilisation : modbus.

Veuillez choisir une action à effectuer :
- Lire des coils                                          (FC1)
- Lire des entrées discrètes                              (FC2)
- Lire des registres d'exploitation                       (FC3)
- Lire des registres d'entrée                             (FC4)
- Ecrire dans un coil                                     (FC5)
- Ecrire dans un registre d'exploitation                  (FC6)
- Ecrire dans plusieurs coils                             (FC15)
- Ecrire dans plusieurs registres d'exploitation          (FC16)
- Afficher les paramètres utilisés                        (param)
- Choisir l'adresse du serveur (par défaut 1)             (adresse)
- Modifier la taille des registres                        (reg)
- Changer le type de données traitées (int, float ou string)  (mode)
- Retour au menu principal.                               (x)
Veuillez choisir une option :
```

Figure 4.6 Menu opérateur avec Modbus

Test Modbus-1

La fonction FC1 est dédiée à la lecture de *coils*. La carte Arduino dispose de cinq registres avec des *coils*, il est ainsi possible de consulter ces registres en une requête tel qu'on peut l'observer en Figure 4.7.

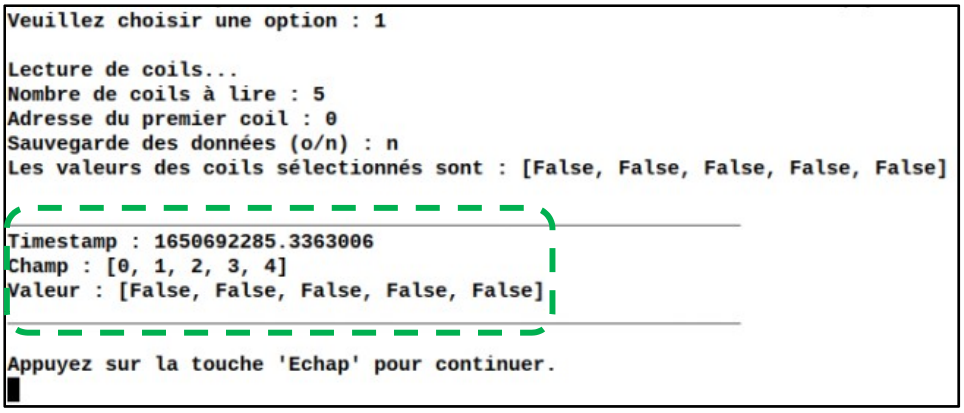


Figure 4.7 Essai de la fonction Modbus FC1

Tous les registres ont été initialisés à 0, il est ainsi normal d'obtenir que des valeurs « *False* ». La valeur que l'on peut voir au niveau du « *Timestamp* » correspond au nombre de secondes écoulées depuis « l'*epoch* ». Pour les systèmes Linux et donc pour notre Raspberry Pi, l'*epoch* est définie au premier janvier 1970, à 0 heure et 0 minute (UTC).

Test Modbus-2

Nous avons ensuite entrepris de tester la deuxième fonction de ce sous-menu, à savoir la fonction FC2. Nous avons ainsi envoyé une première requête. Nous avons ensuite changé l'état de l'interrupteur connecté à l'entrée discrète avant de renvoyer une nouvelle requête. La Figure 4.8 nous illustre les résultats des deux requêtes. À gauche le test a été effectué avec l'interrupteur ouvert et à droite avec l'interrupteur, et donc le circuit, fermé. Nous pouvons ainsi bien voir que le changement d'état de l'interrupteur a entrainé une modification du registre associé entre les deux lectures, en passant de la valeur « *False* » à la valeur « *True* ».

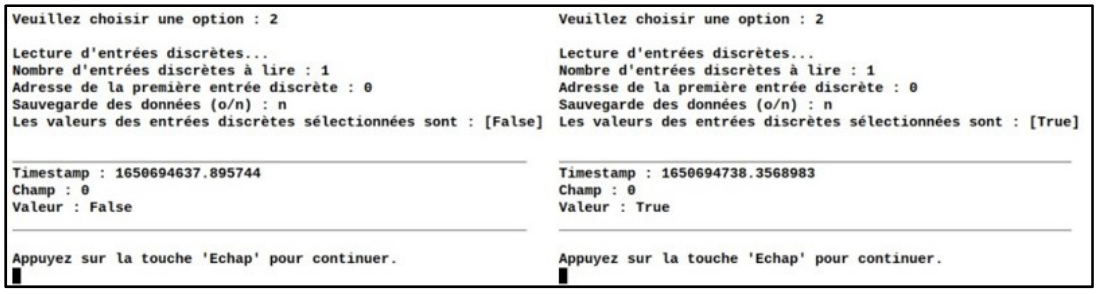


Figure 4.8 Essais de la fonction Modbus FC2

Test Modbus-3

Nous allons maintenant présenter la fonction FC4, la fonction FC3 sera quant à elle éprouvée par la suite. Pour ce test, nous allons lire une première fois le registre 0 des registres d'entrées (broche A0 reliée au potentiomètre). Nous allons ensuite, à l'instar de l'interrupteur, influer sur la valeur du potentiomètre et effectuer une nouvelle requête. Le résultat est visible en Figure 4.9. Nous pouvons une nouvelle fois observer un changement dans les valeurs renvoyées. Si le registre indiquait la valeur 58 avant de manipuler le potentiomètre, la variable a suite à cela augmenté pour atteindre 141.

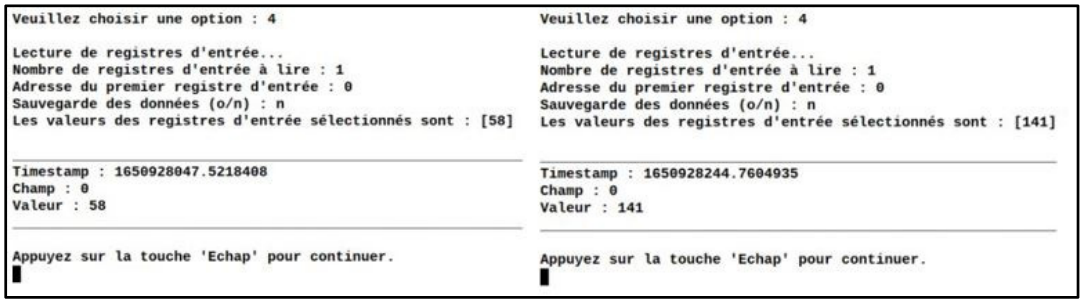


Figure 4.9 Essais de la fonction Modbus FC4

Test Modbus-4

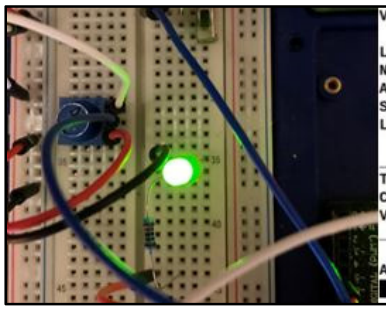
Le programme de la passerelle IIoT permet également l'envoi de *coils* avec la fonction FC15. Nous allons alors écrire sur les cinq adresses de *coils.* Préalablement tous initialisés sur la valeur *False*, nous voulons que les adresses 0, 2 et 4 prennent la valeur *True*, de sorte notamment à allumer la LED associée à l'adresse 2. L'opération est alors présentée en Figure 4.10.

```
Veuillez choisir une option : 15

Ecriture de plusieurs coils...
Nombre de coils à modifier : 5
Adresse du premier coil à modifier : 0
Entrez les valeurs à donner aux coils (0 ou 1) :
1
4 valeurs restantes
0
3 valeurs restantes
1
2 valeurs restantes
0
1 valeurs restantes
1
0 valeurs restantes
Les valeurs [1, 0, 1, 0, 1] ont été écrites dans les coils sélectionnés

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.10 Essai de la fonction Modbus FC15

Test Modbus-5

Les différents *coils* ont alors bien été modifiés. De plus, il est possible de vérifier cette opération en observant la LED s'allumer et en utilisant la fonction FC1 comme précédemment telle qu'on peut le remarquer sur la Figure 4.11. Les valeurs indiquées sur cette figure font écho à celles de la Figure 4.10.



```
Veuillez choisir une option : 1

Lecture de coils...
Nombre de coils à lire : 5
Adresse du premier coil : 0
Sauvegarde des données (o/n) : n
Les valeurs des coils sélectionnés sont : [True, False, True, False, True]

Timestamp : 1648512809.456131
Champ : [0, 1, 2, 3, 4]
Valeur : [True, False, True, False, True]

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.11 Vérification de la bonne modification des coils

Test Modbus-6

Nous allons utiliser la fonction FC16 pour modifier les quatre registres d'exploitation réservés aux `ints`. Il est à noter que l'adresse 0 des `ints` est associée à l'angle du servomoteur. L'envoi d'une requête FC16 est illustré sur la Figure 4.12.

```
Veuillez choisir une option : 16

Ecriture de plusieurs registres d'exploitation...
Nombre de registres d'exploitation à modifier : 4
Adresse du premier registre d'exploitation à modifier : 0
Valeurs à donner aux registres d'exploitations (int) :
4 valeurs restantes
90
3 valeurs restantes
1
2 valeurs restantes
12345
1 valeur restante
003
Les valeurs [90, 1, 12345, 3] ont été écrites dans les registres d'exploitation sélectionnés

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.12 Essai de la fonction Modbus FC16

Suite à cette requête, la carte Arduino a agi en conséquence en modifiant l'angle du servomoteur, ce dernier a alors pris la valeur de 90° comme nous pouvons le remarquer sur la Figure 4.13.

Figure 4.13 Modification de l'angle
du servomoteur

Test Modbus-7

Parmi les fonctions Modbus existantes, il ne reste plus que la fonction FC3 à tester. Nous pouvons en profiter pour observer si les changements ont bien été effectifs sur tous les registres

concernés. L'action demandée en Figure 4.14 consiste à recueillir les valeurs de ces dits registres.

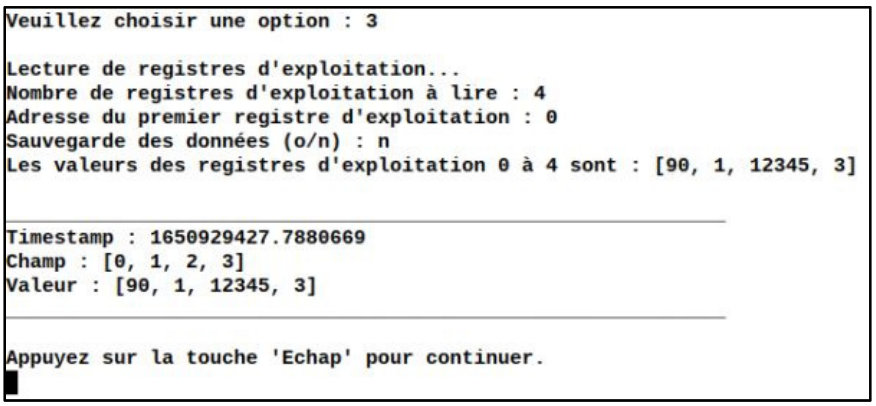


Figure 4.14 Vérification des registres à l'aide de la fonction FC3

Pour ce qui est des actions restantes, l'affichage des paramètres fera simplement apparaitre les quatre paramètres relatifs au bus, à savoir le *baudrate*, le port, l'adresse du nœud de destination et le mode d'encodage des nombres flottants. Plutôt que de simplement afficher les paramètres, il est également possible de les modifier directement via cette section sans avoir à redémarrer le programme. La fonction de changement d'adresse est quant à elle explicite. L'action « mode » offre la possibilité à l'utilisateur de modifier le type de données traitées de la même manière que sur la Figure 4.4. Si une modification a lieu, alors le menu changera pour s'adapter au type de données. Par exemple, les fonctions FC1, FC5 et FC15 sont relatives aux `coils`, leur présence lors de la manipulation de `floats` ou de `strings` n'a donc pas de sens. Ces dernières se verront alors retirer du menu de communication lorsque l'opérateur agit sur des variables de ce type.

Test Modbus-8

L'envoi de `strings` et de `floats` est d'ailleurs la dernière étape à valider avant d'avoir une passerelle totalement opérationnelle avec le protocole Modbus. Après avoir défini le type de variables sur `floats`, nous avons envoyé une requête pour modifier les registres 1 et 2. On

rappelle que la numérotation commence à 0 et qu'il y a trois registres alloués aux nombres à virgule flottante. L'opération décrite ici est visible sur la Figure 4.15.

```
Veuillez choisir une option : 16

Ecriture de plusieurs registres d'exploitation...
Nombre de registres d'exploitation à modifier : 2
Adresse du premier registre d'exploitation à modifier : 1
Valeurs à donner aux registres d'exploitations (float) :
2 valeurs restantes
123.456
1 valeur restante
3.14
Les valeurs [123.456, 3.14] ont été écrites dans les registres d'exploitation sélectionnés

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.15 Essai de la fonction Modbus FC16
avec des nombres à virgule flottante

Pour faire écho à cette requête, nous pouvons utiliser FC3 pour observer les valeurs des trois registres `floats`. Ceci est reflété par la Figure 4.16 sur laquelle nous pouvons effectivement noter la présence des valeurs définies en Figure 4.15.

```
Veuillez choisir une option : 3

Lecture de registres d'exploitation...
Nombre de registres d'exploitation à lire : 3
Adresse du premier registre d'exploitation : 0
Sauvegarde des données (o/n) : n
Les valeur stockées dans les registres d'exploitation 0 à 3 sont [0.0, 123.45600128173828, 3.140000104904175]

_____________________________________________________
Timestamp : 1650930881.771366
Champ : [0, 1, 2]
Valeur : [0.0, 123.45600128173828, 3.140000104904175]
_____________________________________________________

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.16 Essai de la fonction Modbus FC3 avec des nombres à virgule

Nous pouvons apercevoir que les nombres affichés ne sont pas exactement les mêmes que ceux précédemment envoyés. Cela peut s'expliquer par les incertitudes liées au système de codage des nombres flottants. Le problème peut être résolu en effectuant une troncature des valeurs par exemple, mais cette fonctionnalité n'a pas encore été mise en place au moment de la rédaction de ce mémoire.

Test Modbus-9

Enfin, pour ce qui est des chaînes de caractères, il est seulement possible d'envoyer ou de recevoir une chaine. Nous avons jugé que la distinction entre les fonctions FC6 et FC16 n'était alors pas de mise. L'utilisateur rentre une chaine de caractères et le programme s'occupe lui-même de la découpe des caractères. Nous rappelons qu'un registre contient deux caractères. Le test illustré sur la Figure 4.17 consiste à envoyer la chaîne de caractères « Modbus RTU ». Ces caractères sont ensuite affichés sur la première ligne de l'écran LCD.

```
Veuillez choisir une option : 6

Ecriture d'un registre d'exploitation...
Attention : La passerelle est actuellement configurée pour envoyer des données de type str.
Adresse du registre d'exploitation à modifier : 0
Nouvelle valeur : Modbus RTU
Le texte 'Modbus RTU' a été écrit dans les registres d'exploitation

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.17 Essai de la fonction Modbus FC6 avec des chaînes de caractères

L'écran LCD après l'action décrite ci-dessus est illustré en Figure 4.18. On peut ainsi voir que l'encodage mais aussi le décodage de la chaine « Modbus RTU » se sont effectués avec succès.

Figure 4.18 Photographie de l'écran LCD
après réception d'une chaîne de caractères

La fonction FC3 fonctionne avec les `strings` de la même manière qu'avec les `ints` ou les `floats`, comme il est possible d'observer sur la Figure 4.14 et sur la Figure 4.16. Les différentes

fonctions Modbus ont donc été testées avec succès. Nous allons effectuer des tests similaires afin de montrer que notre passerelle incorpore aussi une version fonctionnelle de Canbus.

4.3.1.2 Fonctions Canbus

Contrairement au protocole Modbus, Canbus ne reconnait aucune distinction dans ses registres, le nombre de choix présents dans le menu s'en retrouve alors amenuit tel qu'on peut le voir sur la Figure 4.19.

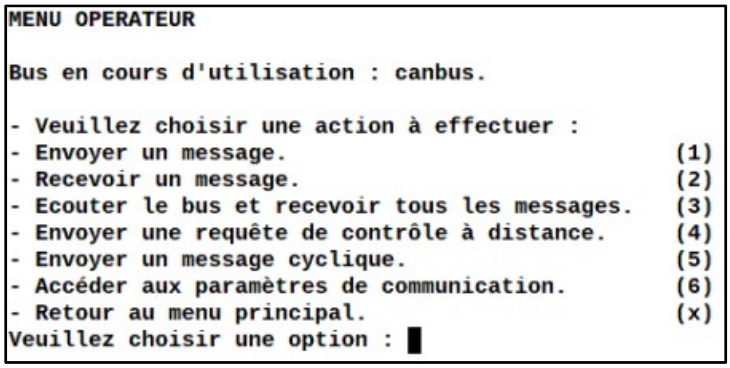


Figure 4.19 Menu opérateur avec Canbus

La première option disponible consiste à envoyer des messages. Il est alors nécessaire de renseigner l'ID du message, le type de variable à transmettre, et sa valeur. Peu importe son type de données, la trame transmise renferme une liste d'octets, équivalent chacun à un `int` d'une valeur comprise entre 0 et 255. L'information concernant le type de données ne concerne alors que la passerelle IIoT, cela permet à l'utilisateur de rentrer ce qu'il souhaite et d'effectuer la correction avant l'envoi de la requête. Le bon décodage de la variable dépendra alors uniquement de l'appareil qui le reçoit. Il est donc important que les différents appareils opérant sur le bus soient en adéquation avec le type de la variable échangée. C'est pour cela que nous avons défini précédemment des plages d'ID réservées à certaines variables. Par exemple nous avons désigné les ID 5XX comme étant associés aux nombres à virgule et les ID 6XX comme étant réservés pour les chaînes de caractères. Ces valeurs sont modifiables dans les codes des programmes Raspberry Pi et Arduino.

Test Canbus-1

Pour valider les performances de la première fonction, nous avons alors envoyé une requête pour quatre formats de variables différents, à savoir : i) une liste d'octets; ii) un nombre entier; iii) un nombre à virgule flottante; iv) une chaîne de caractères. Canbus ne fonctionnant pas sur le principe du *Polling*¸ nous ne recevons pas de trame pour confirmer que les données ont bien été reçues. C'est pourquoi nous avons joint à chaque expérience une photographie de l'écran LCD, ce dernier servant de témoin.

Pour ce test, nous envoyons un message avec l'ID 1 qui correspond à l'identifiant de message associé à la valeur de l'angle du servomoteur. Il est à noter que la valeur de l'angle est stockée sur l'octet 0 du champ de données de la trame. Les résultats sont montrés sur la Figure 4.20. Nous pouvons observer grâce au servomoteur et à l'écran LCD que la trame a bien été reçue et traitée. À l'image des tests Modbus décrits précédemment, le temps de réponse des appareils lors d'un échange de donnée via le bus CAN est de l'ordre des millisecondes.

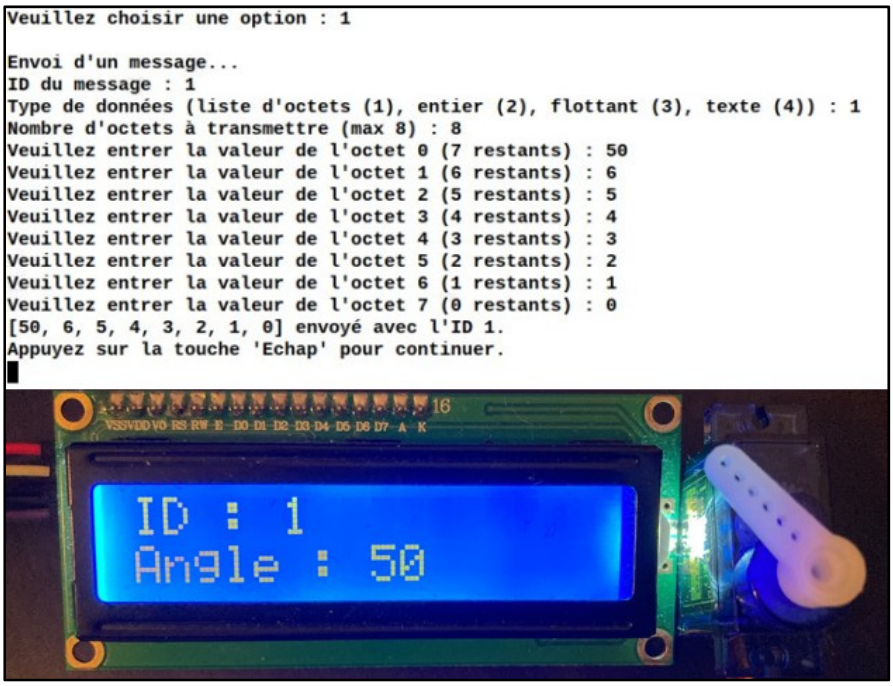


Figure 4.20 Envoi d'une liste d'octets avec Canbus

Test Canbus-2

La Figure 4.21 reprend le même test, mais en fournissant en entrée un nombre entier. Le Raspberry Pi est techniquement en mesure de transmettre des nombres entiers de 64 bits, cependant les bibliothèques Arduino utilisées ne supportent pas les formats supérieurs à 32 bits, nous prendrons ainsi soin de ne pas communiquer de nombres codés sur plus de 32 bits. Il faut toutefois préciser qu'il est possible de coder avec 32 bits des entiers non signés pouvant atteindre la valeur de « 4 294 967 295 ».

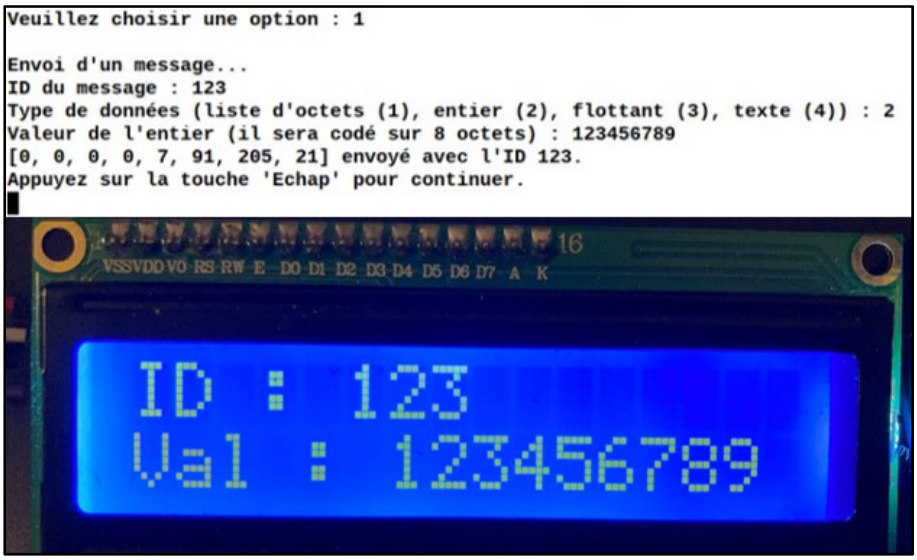


Figure 4.21 Envoi d'un entier avec Canbus

Test Canbus-3

La procédure est donc une nouvelle fois répétée, avec cette fois ci un nombre à virgule flottante en entrée. L'opération est attestée par la Figure 4.22.

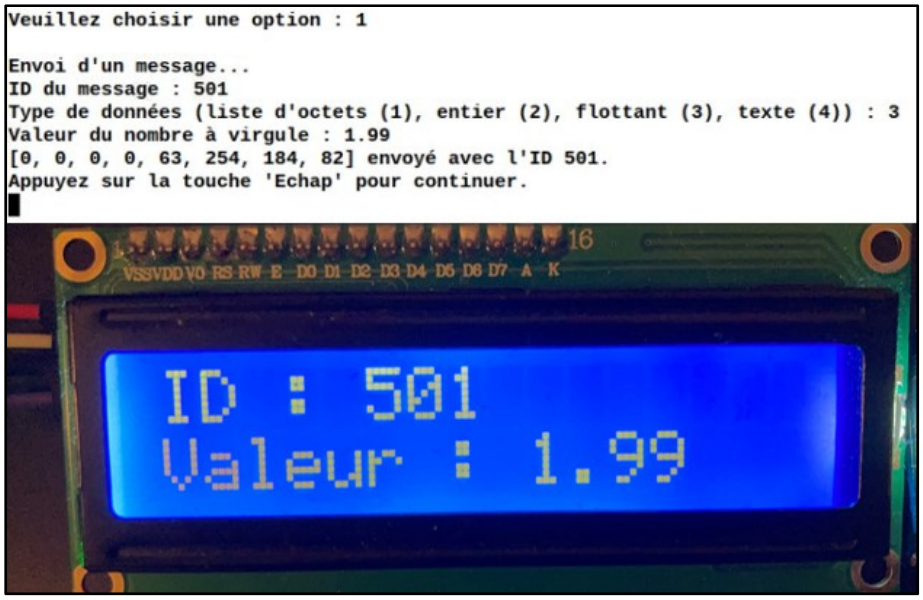


Figure 4.22 Envoi d'un nombre à virgule avec Canbus

Test Canbus-4

Pour finir avec la fonction d'envoi, la passerelle IIoT implante également l'envoi de chaînes de caractères via Canbus. Un essai de la sorte a été effectué sur la Figure 4.23.

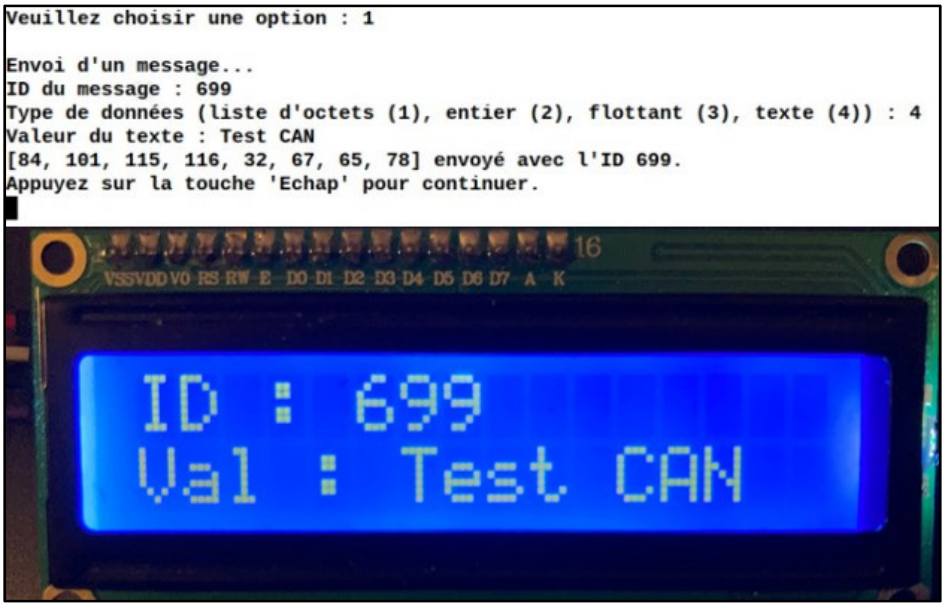


Figure 4.23 Envoi d'une chaîne de caractères avec Canbus

Test Canbus-5

La deuxième fonction du menu est destinée à la réception de messages. L'utilisateur peut choisir de placer la passerelle en attente pour un temps défini, en indiquant un nombre de secondes. Si au bout de ce délai aucun message n'a été reçu, alors l'action se termine d'elle-même. Il est également possible de laisser un temps indéfini, c'est-à-dire que la passerelle n'effectuera pas d'autre action tant qu'aucun message ne sera reçu, peu importe le temps écoulé. Il est bien évidemment toujours possible pour l'utilisateur d'avorter l'action à n'importe quel moment. En complément, nous avons instauré une option qui offre la possibilité de filtrer l'ID choisi. Ainsi, si l'utilisateur ne s'intéresse qu'à un ID particulier, il lui suffit de le renseigner dans le champ approprié.

Un exemple est alors proposé en Figure 4.24. Nous choisissons un *timeout* de 10 secondes pour la réception du message et nous n'indiquons pas d'adresse spécifique. Nous utilisons ensuite le moniteur Arduino pour envoyer une trame d'ID 508 et de valeur 58,641. Nous pouvons alors observer sur la Figure 4.24 que le message a correctement été réceptionné. La très légère divergence de valeur s'explique comme nous l'avons décrit précédemment par les incertitudes liées à la méthode d'encodage.

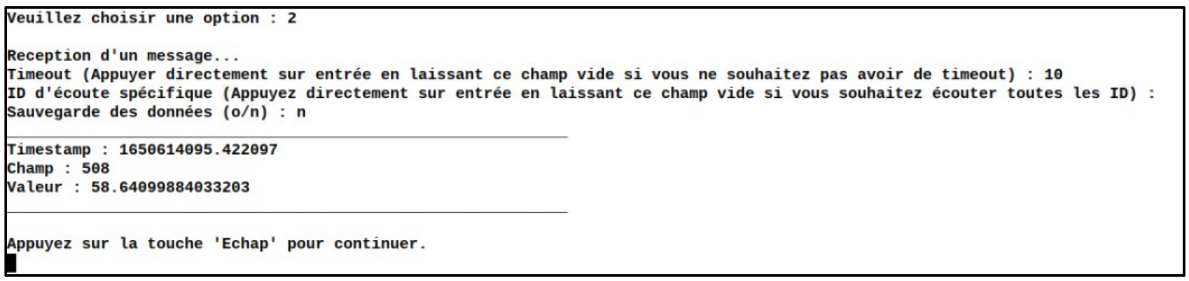


Figure 4.24 Réception d'un message avec Canbus

Test Canbus-6

La fonction d'écoute du bus est très similaire à celle de réception, elle permet d'emmagasiner plusieurs messages à la suite. Pour tester cette fonction, nous allons nous servir du potentiomètre, la carte Arduino avec Canbus étant programmée pour émettre un message à

chaque changement de valeur. Nous l'avons programmée pour envoyer les messages sous l'ID 5, nous avons de fait ajouté ce paramètre en entrée lors de l'écoute du bus visible en Figure 4.25.

```
Veuillez choisir une option : 3

Ecoute du bus...
ID d'écoute spécifique (Appuyez directement sur entrée en laissant ce champ vide si vous souhaitez écouter toutes les ID) : 5
Sauvegarde des données (o/n) : n
Début de l'écoute (Vous pouvez arrêter l'écoute à tout moment en appuyant sur 'Echap').
_____________________________________________
Timestamp : 1650614333.250899
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 15]
_____________________________________________

_____________________________________________
Timestamp : 1650614334.425947
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 26]
_____________________________________________

_____________________________________________
Timestamp : 1650614334.635201
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 37]
_____________________________________________

_____________________________________________
Timestamp : 1650614334.792344
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 48]
_____________________________________________

_____________________________________________
Timestamp : 1650614335.006131
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 59]
_____________________________________________

_____________________________________________
Timestamp : 1650614335.033922
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 70]
_____________________________________________

_____________________________________________
Timestamp : 1650614335.108453
Champ : 5
Valeur : [0, 0, 0, 0, 0, 0, 0, 91]
_____________________________________________
```

Figure 4.25 Écoute de l'ID 5 avec Canbus

Il est à noter que la bibliothèque Canbus d'Arduino est en capacité d'envoyer uniquement des *frames* de huit octets. C'est pour cela que nous recevons une liste de huit octets, la valeur du potentiomètre est visible sur l'octet le plus à droite.

Test Canbus-7

Nous nous devons également de tester et de valider le bon fonctionnement de l'option d'envoi de requêtes à distance. Ce type de trames de données n'est pas supporté pas les bibliothèques Arduino existantes. Pour remédier à ce problème, nous allons plutôt utiliser la deuxième

interface CAN offerte par le HAT en la reliant également au bus. En outre, nous ouvrons un deuxième terminal via le bureau Raspbian. Ces opérations nous permettent d'obtenir deux instances Canbus sur le même système. Nous avons développé un petit programme de quelques lignes qui permet d'indiquer si des trames « *remote »* ont été reçues et d'afficher leurs valeurs. Le résultat obtenu est illustré en Figure 4.26.

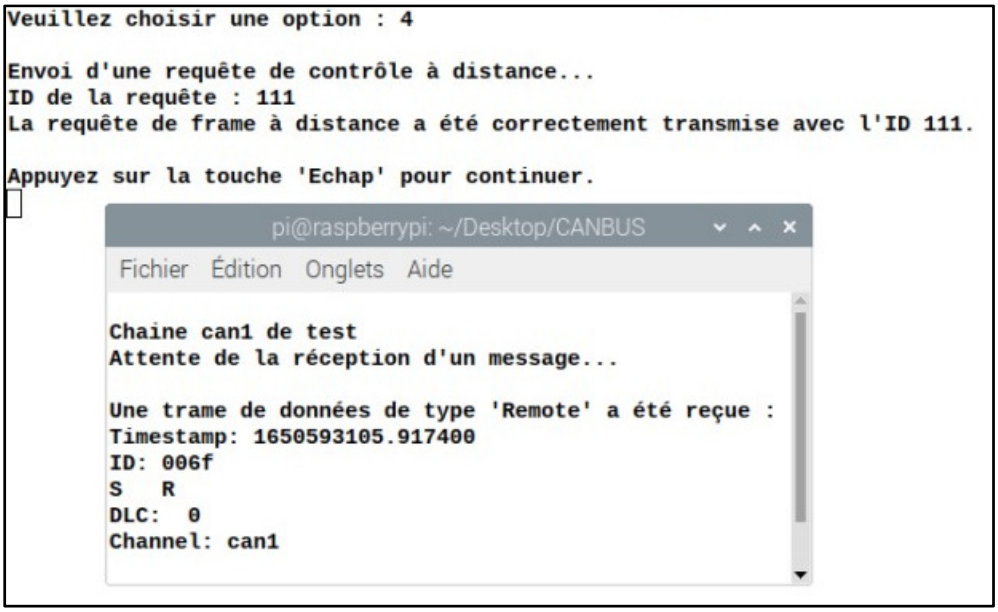


Figure 4.26 Envoi et réception d'une trame à distance avec Canbus

Il est possible d'observer en arrière-plan la console de la passerelle IIoT, on envoie une trame de contrôle à distance avec l'ID 111. Cette trame est reçue par la chaîne « *can1* » qui correspond à notre deuxième interface CAN. L'affichage des données de la trame est celui par défaut proposé par la bibliothèque « python-can ». 006f est l'affichage hexadécimal du nombre 111, il s'agit donc bien de la trame que nous avons émise. La valeur S est utilisée pour indiquer qu'il s'agit d'un ID standard de 11 bits, tandis que la valeur R indique que la trame est une « *remote frame »*. Le champ des données est composé de 0 bit comme indiqué par le DLC. Ainsi, nous pouvons conclure quant au fait que la fonction d'envoi de trame de contrôle à distance est pleinement fonctionnelle.

Test Canbus-8

Une autre fonction existante consiste en l'envoi cyclique de données ayant le même ID. Il est alors possible de choisir l'ID, la période entre chaque émission de messages et le temps maximal total de l'opération, mais ce dernier paramètre demeure optionnel. Bien que l'ID doive rester constante, l'utilisateur a toutefois la possibilité de modifier les valeurs des données transmises comme nous pouvons le voir en Figure 4.27. Durant ce test, nous pouvons observer les LED Rx et Tx clignoter toutes les trois secondes, et de surcroît voir l'angle du servomoteur s'incliner de 10°. Après modification de la *frame*, l'angle prend la valeur de 90°.

```
Veuillez choisir une option : 5

Envoi d'un message cyclique
Pour le moment, les seules variables compatibles avec cette fonction sont les listes d'octets.
ID du message : 1
Nombre d'octets à transmettre (max 8) : 8
Veuillez entrer la valeur de l'octet 0 (0 à 255) : 10
Veuillez entrer la valeur de l'octet 1 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 2 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 3 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 4 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 5 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 6 (0 à 255) : 0
Veuillez entrer la valeur de l'octet 7 (0 à 255) : 0
Délai entre chaque message (en secondes) : 3
Temps maximal avant l'arrêt de l'émission (laissez ce champ vide si vous ne voulez pas de temps maximal) :
La trame de données est actuellement envoyée de manière cyclique...
Veuillez choisir l'action à effectuer en saisissant le numéro correspondant :
Interrompre la communication cyclique.        (1)
Modifier les données de la trame transmise.   (2)
2
Veuillez indiquer le nombre d'octets à transmettre (max 8) : 8
Valeur de l'octet 0 (0 à 255) : 90
Valeur de l'octet 1 (0 à 255) : 0
Valeur de l'octet 2 (0 à 255) : 0
Valeur de l'octet 3 (0 à 255) : 0
Valeur de l'octet 4 (0 à 255) : 0
Valeur de l'octet 5 (0 à 255) : 0
Valeur de l'octet 6 (0 à 255) : 0
Valeur de l'octet 7 (0 à 255) : 0
Modification de la trame...
Veuillez choisir l'action à effectuer en saisissant le numéro correspondant :
Interrompre la communication cyclique.        (1)
Modifier les données de la trame transmise.   (2)
1
La transmission cyclique s'est correctement achevée.

Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.27 Envoi cyclique de messages avec Canbus

Enfin, il est possible d'accéder aux paramètres de communication afin de les consulter et de les modifier. Cette fonction est très proche de celle décrite dans le menu Modbus, mais les variables concernées diffèrent. Il est ainsi possible d'influer sur le *bitrate*, le nom de la chaîne, le type de bus et la méthode d'encodage des nombres flottants.

Nous avons ainsi présenté dans la section 4.3.1 les différentes options que proposait notre passerelle IIoT quant aux communications avec les bus de terrain. À chaque fois que l'option « Sauvegarde des données » nous était proposée, nous la réfutions afin de nous concentrer uniquement sur les protocoles Modbus et Canbus. Lorsque cette option est activée, ces données sont enregistrées et une synchronisation de ces dernières est effectuée par défaut avec ThingSpeak via le protocole de communication actif à l'instant de la requête. Nous allons donc désormais effectuer de nouvelles requêtes en activant cette fonctionnalité afin d'observer les performances des protocoles MQTT et HTTP.

4.3.2 Validation de la communication avec un service infonuagique

Il reste maintenant à s'assurer que les protocoles MQTT et HTTP ont correctement été codés et configurés. Il est à noter que ThingSpeak n'accepte que l'envoi de nombres entiers ou décimaux, les chaînes de caractères ne sont pas prises en charge. Nous allons envoyer à plusieurs reprises des valeurs à la plateforme IIoT. L'emploi du potentiomètre peut alors se révéler fort intéressant dans ce cas de figure. Nous avons simplement à le tourner pour modifier sa valeur, puis effectuer des requêtes de lecture, avec Modbus ou Canbus.

Les nombres décimaux étant des cas plus complexes que les entiers, une opération concluante sur ces derniers entrainera nécessairement une opération concluante sur des entiers. Pour cette raison, nous allons travailler directement avec des nombres à virgule, et modifier sur quelques points le programme Arduino afin de permettre une telle action. Nous allons diviser les valeurs du potentiomètre retournées par 10 (et donc obtenir des flottants), enregistrer ces dernières dans les registres `floats` de Modbus et les transmettre sous l'ID 555 lors de l'usage du protocole Canbus. On rappelle que les ID 5XX sont utilisés pour les flottants dans notre cas.

Par ailleurs, il est bon de savoir que la plateforme ThingSpeak ne permet que d'avoir dix champs de données par chaîne. Ces champs sont nommés de 1 à 10 et peuvent se voir allouer à chacun un graphique sur lequel sont affichées leurs données. Nous n'avons pas possibilité de

renommer ces champs, c'est pour cela qu'il est nécessaire de modifier le champ « *topic* » des données avant leur envoi. En effet, par défaut le *topic* d'un message Modbus sera l'adresse du registre visé, et celui d'un message Canbus sera son ID. Dans notre cas, l'ID 555 devra alors être reconvertie en un nombre compris entre 1 et 10. C'est le rôle de la fonction « *mapping* ». Cette fonction est ainsi vouée à effectuer les bonnes associations entre l'identification d'un message du point de vue « bus de terrain » et son identification au niveau de la plateforme IIoT.

4.3.2.1 Envoi de données avec MQTT

L'objectif de cette section est de tester les capacités de la passerelle à employer le protocole MQTT pour acheminer ses données au service infonuagique. Nous avons ainsi collecté à plusieurs reprises les valeurs du potentiomètre et, contrairement aux précédentes expérimentations, nous avons utilisé l'option de sauvegarde des données. Cette option aura pour effet d'enregistrer les valeurs dans la base de données interne à la passerelle IIoT et de les transférer à ThingSpeak si la synchronisation est activée. Le résultat de la dernière itération est présenté sur la Figure 4.28.

```
Reception d'un message...
Timeout (Appuyer directement sur entrée en laissant ce champ vide si vous ne souhaitez pas avoir de timeout) :
ID d'écoute spécifique (Appuyez directement sur entrée en laissant ce champ vide si vous souhaitez écouter toutes les ID) :
Sauvegarde des données (o/n) : o
_____________________________________________
Timestamp : 1648853994.460547
Champ : 555
Valeur : 27.799999237060547
_____________________________________________

Données synchronisées via mqtt.
Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.28 Collecte et sauvegarde de données via Canbus et MQTT

La dernière valeur fournie par le potentiomètre est donc 27.8. La Figure 4.29 nous montre les données collectées sur ThingSpeak. Nous pouvons observer les multiples valeurs transmises, y compris la dernière, correspondant à celle décrite ci-dessus.

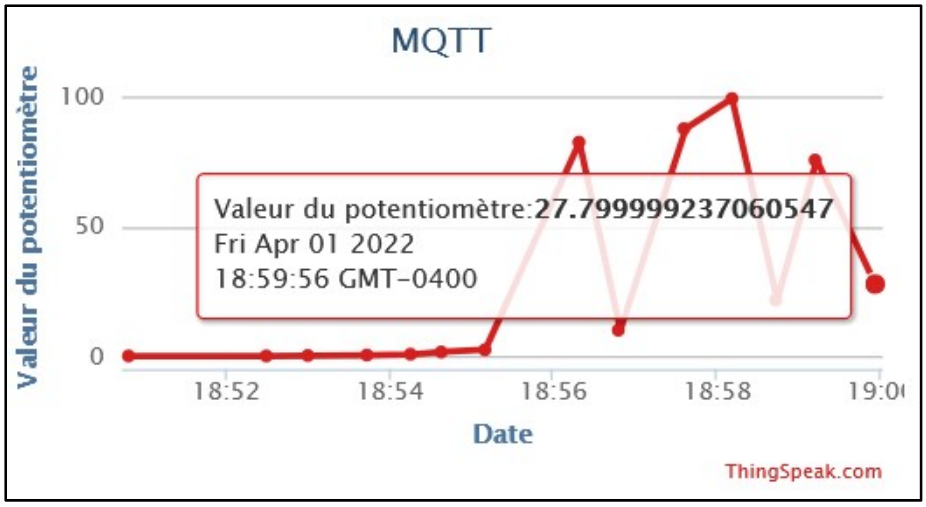


Figure 4.29 Graphique des différentes valeurs récoltées
par le service infonuagique via MQTT

Par ailleurs, nous avons programmé la passerelle pour détecter et indiquer si un quelconque problème s'est produit lors de l'échange et a compromis ce dernier. Par exemple, si nous modifions le mot de passe servant à s'identifier auprès de la plateforme IIoT, il sera considéré comme erroné et l'accès nous sera alors refusé. Si un tel scénario se produit, l'utilisateur sera informé comme nous pouvons l'observer en Figure 4.30.

```
Reception d'un message...
Timeout (Appuyer directement sur entrée en laissant ce champ vide si vous ne souhaitez pas avoir de timeout) :
ID d'écoute spécifique (Appuyez directement sur entrée en laissant ce champ vide si vous souhaitez écouter toutes les ID) :
Sauvegarde des données (o/n) : o
_____
Timestamp : 1650667497.327909
Champ : 555
Valeur : 84.19999694824219
_____

La requête n'a pu aboutir.
Code d'état : 4 - Connexion refusée, identifiant ou mot de passe invalide.

Erreur lors de la synchronisation des données.
Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.30 Gestion des exceptions avec MQTT

Nous précisons que pour obtenir l'affichage du code d'état, nous avons préalablement activé l'option de *debug* via la fonction « *set_debug* ». Cette fonction est implantée dans le code Python. Le code d'erreur 4 visible en Figure 4.30 n'est pas l'unique code existant que nous

avons rapporté dans notre programme. Le Tableau 4.1 présente les différents codes d'état utilisés au sein du programme.

Tableau 4.1 Codes d'état du protocole MQTT

Code d'état	Description
0	Connexion réussie.
1	Connexion refusée, version du protocole incorrecte.
2	Connexion refusée, identifiant du client invalide.
3	Connexion refusée, serveur non disponible.
4	Connexion refusée, identifiant ou mot de passe invalide.
5	Connexion refusée, accès non autorisé.

Si les données devaient être extraites de communications avec un appareil Modbus, le résultat de la collecte sur ThingSpeak en serait identique. La Figure 4.31 témoigne de l'acheminement d'une valeur de l'appareil Modbus jusqu'au service infonuagique.

Figure 4.31 Collecte et sauvegarde de données via Modbus et MQTT

Nous avons ainsi prouvé l'efficience de la passerelle IIoT quant à l'emploi du protocole MQTT, dans la section suivante nous allons effectuer les mêmes opérations avec REST.

4.3.2.2 Envoi de données avec REST

Nous effectuons ici la même expérimentation que précédemment, en privilégiant cette fois l'architecture REST et le protocole HTTP. Une nouvelle fois, nous utilisons le protocole Canbus. Une capture d'écran de la dernière prise de mesures est disponible en Figure 4.32.

```
Reception d'un message...
Timeout (Appuyer directement sur entrée en laissant ce champ vide si vous ne souhaitez pas avoir de timeout) :
ID d'écoute spécifique (Appuyez directement sur entrée en laissant ce champ vide si vous souhaitez écouter toutes les ID) :
Sauvegarde des données (o/n) : o
_____________________________________________________
Timestamp : 1648854636.871303
Champ : 555
Valeur : 78.80000305175781
_____________________________________________________

La requête a été effectuée avec succès.
Code d'état : 200

Données synchronisées via rest.
Appuyez sur la touche 'Echap' pour continuer.
```

Figure 4.32 Collecte et sauvegarde de données via Canbus et REST

De la même manière que précédemment, il est possible de voir le résultat en temps réel en se connectant à l'espace utilisateur de ThingSpeak. La Figure 4.33 illustre les différentes valeurs recueillies.

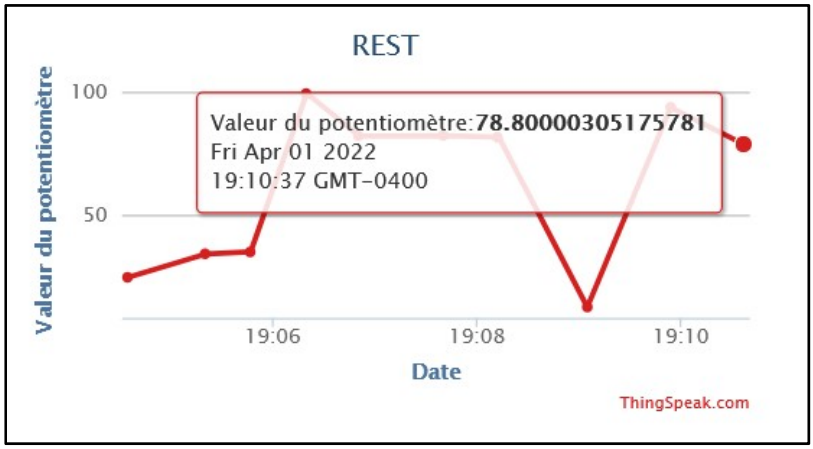


Figure 4.33 Graphique des différentes valeurs récoltées par le service infonuagique via REST

Nous avons activé la fonction « *debug* » du programme qui permet d'obtenir l'état de la communication et de s'assurer qu'il s'agit d'un succès. Cette option peut, au même titre que pour le protocole MQTT, être activée ou désactivée à souhait. Le code 200, et les codes 2XX d'une manière générale, indiquent une communication réussie. En cas d'erreur du côté du client, nous aurions un code 4XX, et un code 5XX serait retourné en cas d'erreur du côté serveur conformément aux recommandations de l'*Internet Assigned Numbers Authority* (Internet Assigned Numbers Authority, 2021).

Nous avons par exemple altéré la valeur de la clé d'écriture REST afin d'illustrer ce qu'il se passe lors de l'envoi d'une requête invalide. Le résultat est présenté en Figure 4.34.

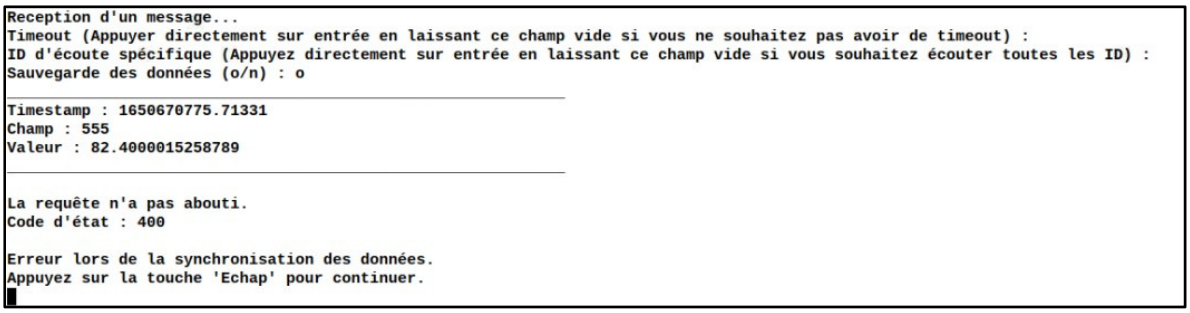


Figure 4.34 Gestion des exceptions avec REST

Nous pouvons alors observer que les données n'ont pas été synchronisées. Le code renvoyé à l'utilisateur est le code 400, qui est le code employé pour indiquer une mauvaise requête (*Bad Request*).

Les tests précédents ont été réalisés en utilisant le protocole Canbus, mais notre passerelle est capable d'effectuer les mêmes opérations lorsque le protocole Modbus est sélectionné tel qu'on peut le voir en Figure 4.35.

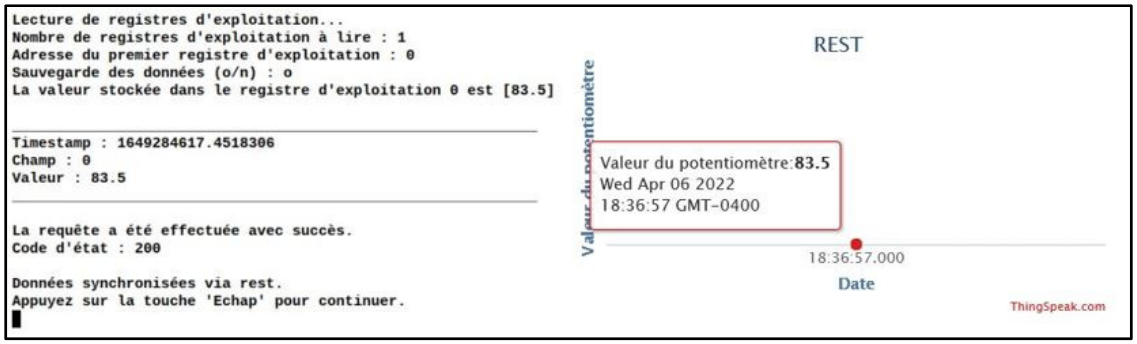


Figure 4.35 Collecte et sauvegarde de données via Modbus et REST

4.3.3 Base de données et services infonuagiques

C'est au travers de ce menu que l'utilisateur est en capacité d'effectuer les diverses actions en rapport avec la gestion des données collectées et leur synchronisation avec les services infonuagiques. Une fois sélectionné, ce menu offre l'accès à deux nouveaux sous-menus, l'un destiné spécifiquement à la base de données interne au programme et l'autre simplement destiné aux options de synchronisation des données avec le Cloud.

Une capture d'écran du premier sous-menu est proposée en Figure 4.36. Nous pouvons alors observer les trois options mises à disposition de l'utilisateur.

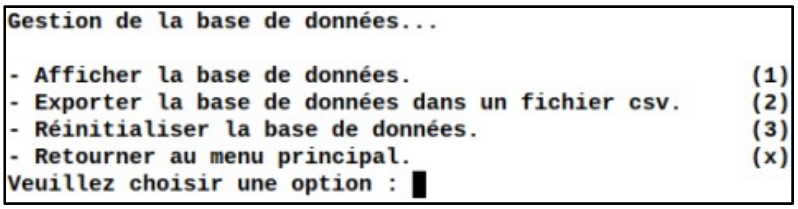


Figure 4.36 Menu de gestion de la base de données

Sur la Figure 4.37, nous avons effectué une capture d'écran de la console après avoir sélectionné l'option 1 ainsi qu'une capture d'écran du logiciel LibreOffice Calc après avoir ouvert le CSV généré par l'action 2. L'action 3 n'a pas été rapportée dans ce mémoire, elle supprime simplement toutes les entrées de la base de données.

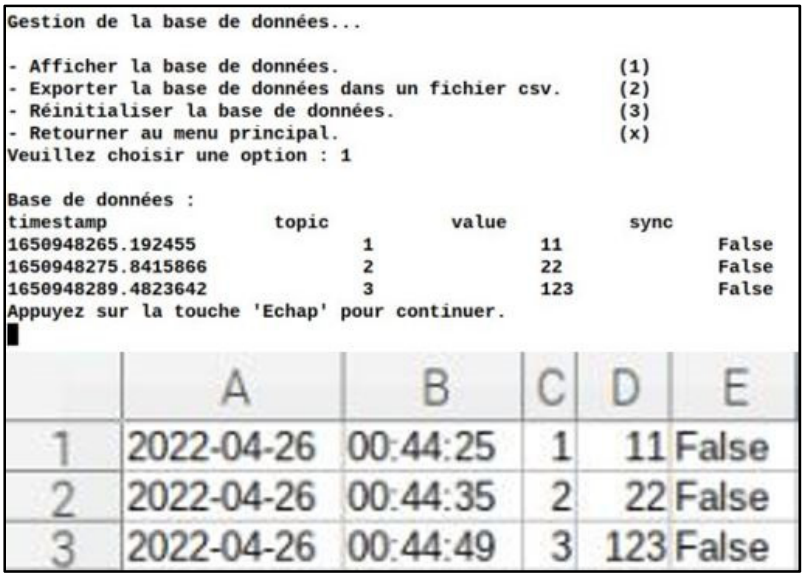


Figure 4.37 Affichage de la base de données
et exportation au format CSV

Le deuxième sous-menu, qui se prédestine quant à lui aux opérations de synchronisation, est présenté sur la Figure 4.38. La première action propose à l'opérateur d'activer ou de désactiver la synchronisation automatique des données. Par défaut, cette synchronisation est activée. Lorsque c'est le cas, les données sont automatiquement transmises au Cloud dès l'instant que l'utilisateur opte pour la sauvegarde de données au moment où l'option « Sauvegarde des données (o/n) » apparaît. Si la synchronisation automatique est désactivée et que l'utilisateur décide de sauvegarder tout de même les données, elles seront sauvegardées dans la base de données interne du programme avec un attribut « *is_sync* » possédant la valeur « *False* ». Par exemple, les données présentent sur la Figure 4.37 n'ont pas été synchronisées, la synchronisation automatique ayant préalablement été désactivée pour les besoins du test.

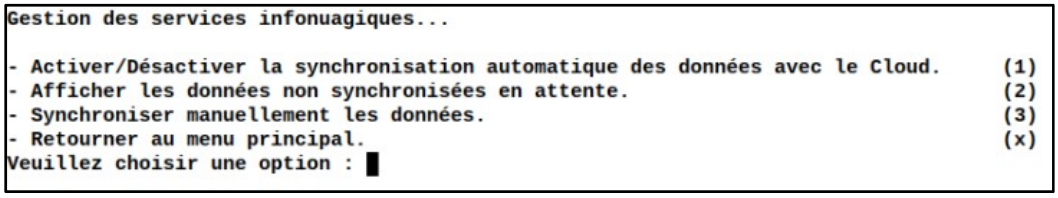


Figure 4.38 Menu de gestion des services infonuagiques

L'option 2 est identique à la fonction d'affichage de la base de données, au détail près que seules les données non synchronisées seront renvoyées. Pour finir, il est possible de synchroniser rétroactivement les données via l'option 3. La passerelle propose alors à l'utilisateur de synchroniser uniquement les données spécifiques à un champ, ou bien toutes les données en attente de synchronisation. Dans le test précédent, nous avions trois valeurs non synchronisées, de champs respectifs 1, 2 et 3. Nous choisissons alors sur la Figure 4.39 de ne synchroniser que les données liées au champ 2.

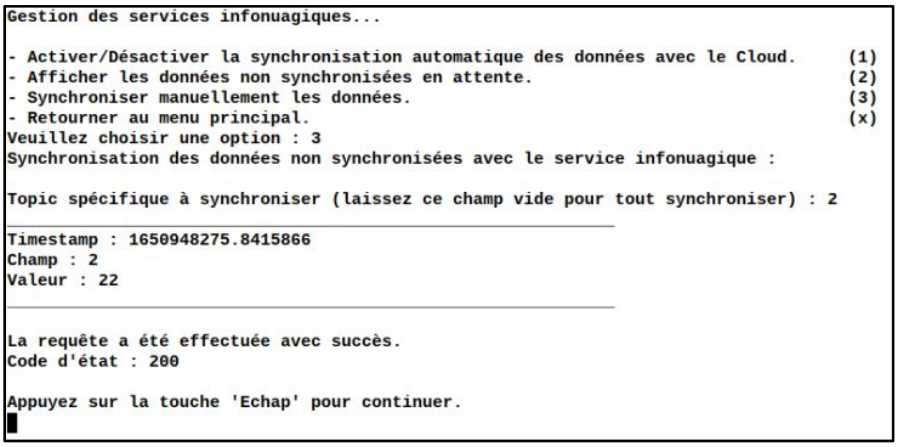


Figure 4.39 Synchronisation manuelle de données

Pour pouvoir transmettre des données à ThingSpeak possédant une date antérieure à la date d'envoi, il est nécessaire de les envoyer sous le format JSON avec un attribut « *created_at* » auquel on associe une chaîne de caractères représentant la date suivant la norme ISO 8601 (International Organization for Standardization, 2019). A l'heure actuelle, la passerelle n'est capable d'envoyer des données au format JSON qu'avec REST.

Nous avons ainsi présenté l'ensemble des fonctions offertes par la passerelle IIoT. Nous n'avons cependant pas encore observé comment cette dernière réagit lorsqu'une erreur se produit.

4.4 Gestion des erreurs et des exceptions

Les erreurs peuvent être multiples dans un système de ce genre. Leurs origines aussi sont variées, elles peuvent provenir d'un défaut matériel, logiciel ou bien humain. Nous avons fait en sorte que notre passerelle soit la plus robuste possible.

Si aucune action n'est prise, le programme s'arrêterait à la première erreur rencontrée, et il faudrait alors le redémarrer à chaque fois. Cela peut se révéler très préjudiciable, notamment si la passerelle IIoT est vouée à fonctionner avec une certaine autonomie.

4.4.1 Erreurs humaines

Il n'est pas pensable d'espérer que l'opérateur ne soit pas à l'origine d'erreur. Dès lors qu'il doit entrer une information dans la console, il faut s'assurer que cette dernière soit cohérente et ait un format adapté à l'information désirée. Par exemple, lorsqu'on demande une adresse, le programme s'attend à recevoir un nombre entier, si une chaîne de caractères est reçue en entrée il ne sera normalement pas en mesure de bien l'interpréter et une erreur se produira, pouvant entrainer un arrêt prématuré du programme.

Nous nous sommes alors assurés de contrôler chaque donnée fournie par l'utilisateur. Ces données sont ainsi testées, et si elles répondent correctement aux critères définis alors elles sont acceptées et l'action poursuit son cours. À l'inverse, si les données sont jugées incohérentes, alors l'action prévue est annulée, un message est affiché pour prévenir l'utilisateur de l'erreur et le programme retourne au point initial, offrant la possibilité à l'utilisateur de recommencer l'opération. Cette façon de fonctionner permet de prévenir d'une quelconque erreur et d'obtenir un programme robuste qui, quoi qu'il arrive, ne s'arrêtera pas d'opérer. Un exemple est apporté en Figure 4.40.

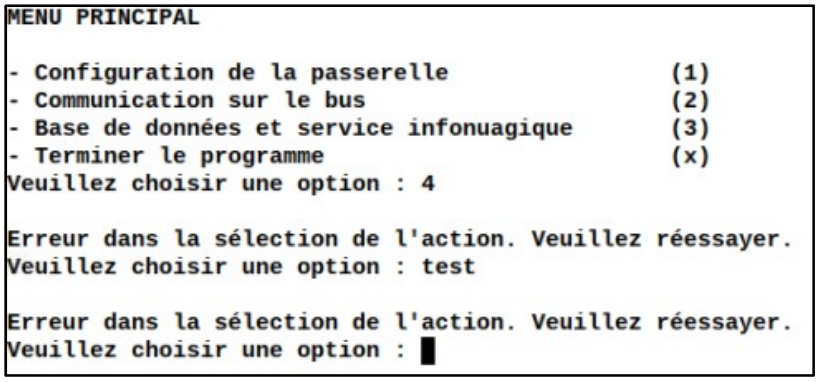


Figure 4.40 Traitement des erreurs lorsque l'utilisateur fournit des informations invalides

Il est important de noter que cette sécurité est présente partout dans le programme, à chaque menu, dès lors que l'utilisateur est en position de rentrer une donnée. En instaurant une telle fonctionnalité, nous nous assurons ainsi que le programme ne puisse pas être interrompu abruptement par une action de l'utilisateur. Lors du développement, nous avons également prêté une attention toute particulière aux exceptions qui pouvaient être retournées par les différentes bibliothèques en cas d'erreur matérielle ou logicielle.

4.4.2 Erreurs et renvois d'exception

Même si le programme est, en théorie, dépourvu de toute erreur d'exécution, nous ne sommes jamais à l'abri d'une erreur externe. Il se peut par exemple qu'un branchement soit mal effectué, et que l'appareil ne soit pas correctement relié au bus. Dans le cas du protocole Modbus, cette situation peut se révéler très néfaste car la passerelle IIoT s'attend à recevoir un accusé de réception après l'envoi d'une requête. De base, avec les fonctions intégrées de la bibliothèque pymodbus, si l'accusé n'est pas réceptionné, alors l'instance déclenche une exception et arrête le programme. Nous avons alors fait en sorte de récupérer les exceptions levées par la bibliothèque et d'afficher le message associé dans la console, mais sans interrompre le fonctionnement du programme pour autant comme nous pouvons l'observer sur la Figure 4.41. Cela permet alors à la personne de vérifier l'erreur et de continuer ses opérations sans qu'un redémarrage du programme soit nécessaire.

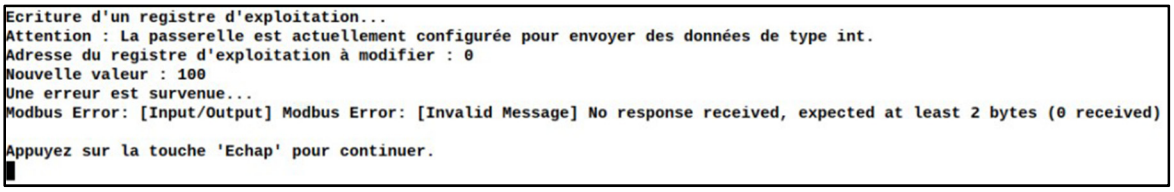


Figure 4.41 Gestion d'une exception soulevée suite à l'envoi d'une requête Modbus

Nous n'avons pas jugé nécessaire de traduire les messages d'exception émis par les bibliothèques Python. Les informations sont facilement compréhensibles en anglais et le nombre d'exceptions différentes est relativement conséquent.

Les messages d'exceptions de toutes les bibliothèques inclues dans le code sont relayés à l'utilisateur via la console. Ces bibliothèques sont très vastes et disposent de nombreuses fonctionnalités, cela nous offre alors la possibilité de traiter un grand nombre d'exceptions. Par exemple, la bibliothèque « pymodbus » ne présente pas moins de neuf types d'exceptions différentes.

La passerelle IIoT offre ainsi de nombreuses fonctionnalités à l'utilisateur qui peut alors manipuler les protocoles Modbus, Canbus, MQTT et HTTP selon sa convenance. Lors de son utilisation, la passerelle va se révéler particulièrement robuste à tout type d'erreurs qui pourraient survenir, qu'elles soient humaines ou matérielles. Nous pouvons maintenant nous demander si les communications avec les bus en elles-mêmes sont également dépourvues d'erreurs.

4.5 Contrôle de performances

Nous avons vu précédemment que l'ensemble des fonctions offertes par la passerelle s'exécute correctement. Il nous reste désormais à vérifier que ce niveau de performance est stable en ce qui concerne les bus de terrain.

Les communications ne sont en théorie pas exemptes de toute erreur, il peut arriver pour certaines raisons que des bits soient corrompus, entrainant une perte de l'intégrité de la trame de données. Les bus de terrain ont une vocation industrielle, dans un milieu où les erreurs ne sont guère permises, ils ont ainsi été pensé pour être robustes. C'est pour remédier à ce genre de problèmes que des moyens de vérification tels que les bits d'erreur ou bien les CRC ont été mis en place. Une tolérance proche de 0 est alors de mise. Par exemple, les communications au sein d'une voiture sont régies par le protocole Canbus, une erreur de transmission lors d'un déplacement pourrait alors potentiellement nuire à la sécurité de la personne.

L'objectif est ainsi de s'assurer que notre passerelle réponde à ces critères des performances en arborant un taux d'erreur nul. Cette vérification est d'autant plus importante que les composants que nous utilisons tels que les interfaces RS485 et CAN ont été obtenus pour une somme modique, nous ne connaissons pas réellement leurs performances, surtout à débits de transmissions élevés. Jusqu'à présent seuls les débits de 9 600 bit/s pour Modbus et de 125 kbit/s pour Canbus ont été traités.

Pour ce faire, nous avons amorcé 1 000 requêtes pour chaque débit différent. Il a été décidé que ce nombre de répétitions offrait un bon compromis entre échantillon représentatif et durée des tests. Les trames reçues ont alors été analysées et leurs valeurs comparées à celles désirées. Les résultats des essais sur le protocole Modbus sont présentés dans le Tableau 4.2.

Tableau 4.2 Taux d'erreurs lors de l'échange de trames Modbus

Baudrate (bit/s)	**Requêtes initiées**	**Trames corrompues**	**Erreur (%)**
1 200	1 000	0	0
2 400	1 000	0	0
9 600	1 000	0	0
19 200	1 000	0	0
38 400	1 000	0	0
57 600	1 000	0	0
115 200	1 000	0	0

Nous pouvons alors observer que l'intégralité des requêtes a bien été traitée, sans qu'aucune d'elle ne soit corrompue. Il est à noter que pour un *baudrate* de 115 200 bit/s, nous obtenions lors de nos premiers tests un taux d'erreur de l'ordre de 12%. Il s'avère en fait que ces erreurs provenaient des ports séries émulés de la carte Arduino, la bibliothèque « *SoftwareSerial* » ne supporte en effet que des *baudrates* inférieurs ou égaux à 57 600 bit/s. Pour réaliser le test au débit de 115 200 bit/s, nous nous sommes alors servis des ports séries physiques, à savoir les broches 0 et 1. En reproduisant l'essai avec 1 000 nouvelles requêtes, nous notons un taux d'erreur de 0% conformément à nos attentes. Les débits présents dans le Tableau 4.2 constituent l'ensemble des *baudrates* usuels recommandés pour Modbus. Les performances attendues de la passerelle IIoT concernant le protocole Modbus sont donc validées. Les résultats de ces expériences illustrent en effet une bonne fiabilité du dispositif.

Nous effectuons alors la même batterie de tests pour valider les performances du protocole Canbus. Les débits de fonctionnement de ce protocole sont de manière générale plus élevés que pour Modbus. Leurs valeurs ne sont pas forcément toutes atteignables par les ports série et le protocole UART effectuant la jonction entre le Shield et la carte Arduino. C'est pourquoi nous devons définir dans le programme Arduino un débit sériel et un débit Canbus. Nous avons sélectionné pour les communications UART un débit de 57 600 bit/s puisqu'il s'agit de la plus grande valeur atteignable pour un fonctionnement optimal avec la bibliothèque « *SoftwareSerial* ». Les valeurs des débits utilisés pour le protocole Canbus sont énumérés dans le Tableau 4.3. Il s'agit des valeurs usuelles définies pour ce protocole et étant prises en charge par la bibliothèque Arduino.

Tableau 4.3 Taux d'erreurs lors de l'échange de trames Canbus

Bitrate (kbit/s)	Requêtes initiées	Trames corrompues	Erreur (%)
5	1 000	0	0
10	1 000	0	0
20	1 000	0	0
25	1 000	0	0
31,2	1 000	0	0
33	1 000	0	0
40	1 000	0	0
50	1 000	0	0
80	1 000	0	0
83,3	1 000	0	0
100	1 000	0	0
125	1 000	0	0
200	1 000	0	0
250	1 000	0	0
500	1 000	0	0
666	1 000	0	0
1 000	1 000	0	0

Nous pouvons alors nous apercevoir une nouvelle fois que la totalité des échanges s'est réalisée avec succès. Aucune erreur n'est à déplorer. Les performances de la passerelle sont donc également validées vis-à-vis du protocole Canbus ainsi donc que pour l'ensemble des bus de terrain actuellement disponibles dans la passerelle IIoT.

4.6 Conclusion

La passerelle IIoT développée propose ainsi à l'utilisateur une grande variété d'actions différentes via la console du Raspberry Pi. L'intégralité de ses fonctionnalités a été éprouvée avec succès.

Nous nous sommes assurés dans un premier temps du bon fonctionnement de l'environnement d'expérimentation en réalisant plusieurs tests préliminaires. Ces tests ont permis de démontrer que l'environnement d'expérimentation était valide et propice à accueillir de futurs tests visant

à observer les performances du système. En addition, la configuration des registres Modbus a été effectuée.

Par la suite, nous nous sommes attachés à présenter chaque option proposée à l'opérateur, permettant simultanément d'expliquer la marche à suivre pour opérer avec la passerelle et d'attester de ses performances. L'utilisateur peut sélectionner les actions voulues par le biais de la console et de menus à choix multiples. L'ensemble des fonctions associées à chaque protocole a ainsi été passé en revue.

À posteriori, l'implantation de chacun des quatre protocoles à savoir Modbus, Canbus, MQTT et HTTP a bien été validée. Nous avons en effet été capables de recueillir des données stockées dans des registres de la carte Arduino et de les transmettre sur ThingSpeak. Les tests ont d'abord été réalisés de manière indépendante, en ne se concentrant que sur un protocole à la fois, puis des actions de plus ample envergure ont par la suite été effectuées afin d'observer l'ensemble de la chaine de transmission des données. Le système mis en place a permis ainsi d'effectuer une liaison concluante entre un appareil de terrain et un service infonuagique, deux mondes totalement distincts.

Par ailleurs, la présence de fonctionnalités annexes a également été abordée. La passerelle IIoT dispose d'une gamme d'options configurables conséquente, que ce soit au niveau des débits d'échanges, des registres utilisés, de la synchronisation des données ou encore de la gestion de la base de données interne. Il est à noter également que les tests se sont révélés concluant pour l'ensemble des données supportées, que ce soit des nombres entiers, des nombres décimaux ou encore des chaînes de caractères.

La robustesse du système a également été étudiée. L'interface utilisateur a été conçue de telle sorte à ce qu'une mauvaise manipulation de l'opérateur n'entraine aucune répercussion sur son fonctionnement. Nous avons en outre mis en place une gestion des exceptions efficace permettant d'assurer un fonctionnement fluide du programme. Enfin, des expériences de

répétabilité ont été entreprises, démontrant des performances optimales avec l'ensemble des débits disponibles.

CONCLUSION ET RECOMMANDATIONS

Notre travail de recherche nous a ainsi amené à la conception d'une passerelle IIoT permettant d'une part l'échange de données avec des machines industrielles et leurs moyens de communication traditionnels, et d'autre part leur intégration dans le monde numérique. Avec l'arrivée des nouvelles technologies et de l'IoT, une fracture se fait ressentir entre les anciennes et nouvelles méthodes de gestion de données. L'intégration de ces nouvelles technologies est pourtant essentielle pour gagner en productivité et la maintenir. L'emploi de dispositifs tels que ces passerelles est alors nécessaire pour faire le pont entre ces deux mondes. Il apparaît néanmoins que les systèmes existants sont fréquemment limités par le nombre de protocoles qu'ils tolèrent. Les problèmes d'interopérabilité sont récurrents, le nombre de bus de terrain présents dans l'industrie est considérable et chacun apporte son lot de spécificités. Les architectures de ces passerelles IIoT se retrouvent alors être variées afin de complaire au plus grand nombre, voire modulaires pour permettre l'intégration de protocoles se rapprochant le plus des besoins de l'utilisateur.

Il est toutefois possible d'observer que l'accès à ces dispositifs n'est pas acquis pour tous, leur prix conséquent peut se révéler être un frein, notamment auprès des PME ou de toute entreprise ayant des moyens limités. Une partie de ces passerelles sont propriétaires et fonctionnent comme des boîtes noires, leur utilité s'en voit par conséquent réduite pour certains cas d'usage comme lors de formations ou d'entrainements du personnel. Les projets de recherche pour assurer l'intégration des bus de terrain avec les nouvelles technologies sont de facto apparus et se sont multipliés ces dernières années. La revue de littérature a mis en exergue une préférence pour les systèmes open source. Le plus souvent, la passerelle développée par les chercheurs met en relation un protocole comme un bus de terrain, destiné à l'échange avec les machines, et un protocole de communication haut-niveau chargé des échanges avec les plateformes IIoT. Notre sujet de recherche s'inscrit donc dans cette dynamique. L'objectif est en effet la conception d'une passerelle IIoT multi-protocole apte à relier objets industriels et services infonuagiques.

Tout au long de notre recherche, nous nous sommes appuyés sur la méthode Agile SCRUM afin d'organiser correctement notre avancée. L'agencement du travail sous forme de *sprints* nous a procuré une ligne de conduite et des objectifs clairs. Nous avons ainsi développé un système capable de communiquer avec deux bus de terrain très répandus dans notre société, à savoir Modbus et Canbus. En complément, la passerelle est en mesure de converser avec n'importe quelle plateforme IIoT via les protocoles de communication haut-niveau MQTT et HTTP, au moyen de l'architecture REST pour ce dernier. Le fonctionnement de ces quatre protocoles a ainsi été minutieusement étudié. Une attention toute particulière a été prêtée à la sélection des ressources matérielles comme logicielles. Ainsi, nous avons choisi de concevoir notre passerelle IIoT sur un Raspberry Pi et nous avons acquis deux cartes Arduino pour pouvoir communiquer avec cette dernière. L'achat d'interfaces CAN et RS485 s'est alors révélé nécessaire pour relier les divers composants au bus associé. Au niveau de la programmation, nous nous sommes orientés vers le langage Python pour sa popularité, sa simplicité et son catalogue de bibliothèques de programmation abondamment fourni.

Le système développé présente ainsi la possibilité d'utiliser plusieurs protocoles différents, et dispose d'une interface utilisateur qui permet son emploi et l'accès à de nombreuses fonctionnalités sans pour autant nécessiter de qualification particulière. La simplicité de cette interface couplée au coût d'achat très modeste des différents composants rend cette passerelle IIoT très accessible. Pour des personnes plus qualifiées, l'architecture de la passerelle IIoT et de son code offre la possibilité d'implanter une multitude d'autres protocoles, les seules limitations étant physiques. Cette considération procure à notre système une grande flexibilité. Cette flexibilité et les perspectives d'amélioration qu'elle implique lui permettent de se démarquer d'une grande partie des autres projets de recherche constituant ainsi une contribution à l'avancée technique. La pluralité des protocoles implantés et la possibilité d'en intégrer de nouveau offrent une meilleure interopérabilité.

De nombreux tests ont été effectués dans le but de valider le fonctionnement et les performances de la passerelle IIoT tout en couvrant l'ensemble des cas d'utilisation définis. Les microcontrôleurs Arduino se sont alors révélés essentiels afin de reproduire le

fonctionnement des bus industriels et de permettre des communications avec la passerelle via ces derniers. Par ailleurs, il est à noter que les performances de la passerelle IIoT sont compatibles avec un usage industriel, le taux d'erreur apparent étant nul pour l'ensemble des débits usuels.

Le système présenté dans ce mémoire n'en est toutefois qu'à un stade de développement et plusieurs points peuvent encore être étudiés. Dans un premier temps, la majeure partie des considérations relevant de la cybersécurité ont été omises, cet aspect est pourtant essentiel pour qu'une passerelle IIoT soit vouée à un usage industriel. Ensuite, l'environnement industriel peut présenter des fluctuations de tension générées par des machines de production. Il serait intéressant d'étudier l'impact du bruit électrique et l'interférence électromagnétique sur le fonctionnement et les performances de la passerelle. Par ailleurs, à l'exception des tests de répétabilité, les divers tests effectués étaient ponctuels et de courte durée. Des essais supplémentaires devront alors être apportés pour s'assurer d'un fonctionnement sans encombre dans la durée. Cela peut se faire en effectuant un *burn-in test*, c'est-à-dire en exécutant le programme durant plusieurs heures.

L'interface utilisateur actuellement en place utilise la console du système d'exploitation, il pourrait être intéressant d'adopter une interface graphique. Outre le côté esthétique, l'approche graphique serait plus ergonomique et mènerait à une interface plus lisible, plus efficace et plus intuitive. Il est également toujours possible d'ajouter de nouvelles fonctionnalités. Par exemple, nous n'avons pas incorporé de fonction de correction des incertitudes liées à la conversion *binary32*, cela peut cependant se révéler utile. L'inclusion d'une base de données telle que MongoDB ou Cassandra permettrait de décupler les possibilités quant à la gestion des données en amont du service infonuagique, notamment au niveau du tri et de la sélection de filtres. In fine, certaines améliorations peuvent aussi être réalisées au niveau matériel. Les divers composants de la passerelle IIoT sont actuellement simplement assemblés entre eux. Il serait intéressant de concevoir un boitier permettant leur rangement, en ayant par exemple recours à l'impression 3D. Ce boitier présenterait plusieurs avantages, comme de protéger les composants de leur environnement, des chocs et de la poussière. Il permettrait aussi de

présenter le système comme une unique entité et non un assemblage de composants. En outre, sa transportabilité n'en serait que meilleure. Nous sommes ainsi persuadés que ce sujet a des perspectives prometteuses.

ANNEXE I

DIAGRAMMES DE SEQUENCE DES ACTIONS DE COMMUNICATION AVEC LES BUS

Lorsque l'utilisateur souhaite utiliser la passerelle IIoT afin de communiquer avec le bus, diverses actions s'offrent à lui. Ces actions sont alors décrites dans cette annexe par le biais de diagrammes de séquence présentant pour chacune d'elles les flux d'échanges et les interactions entre les différents acteurs.

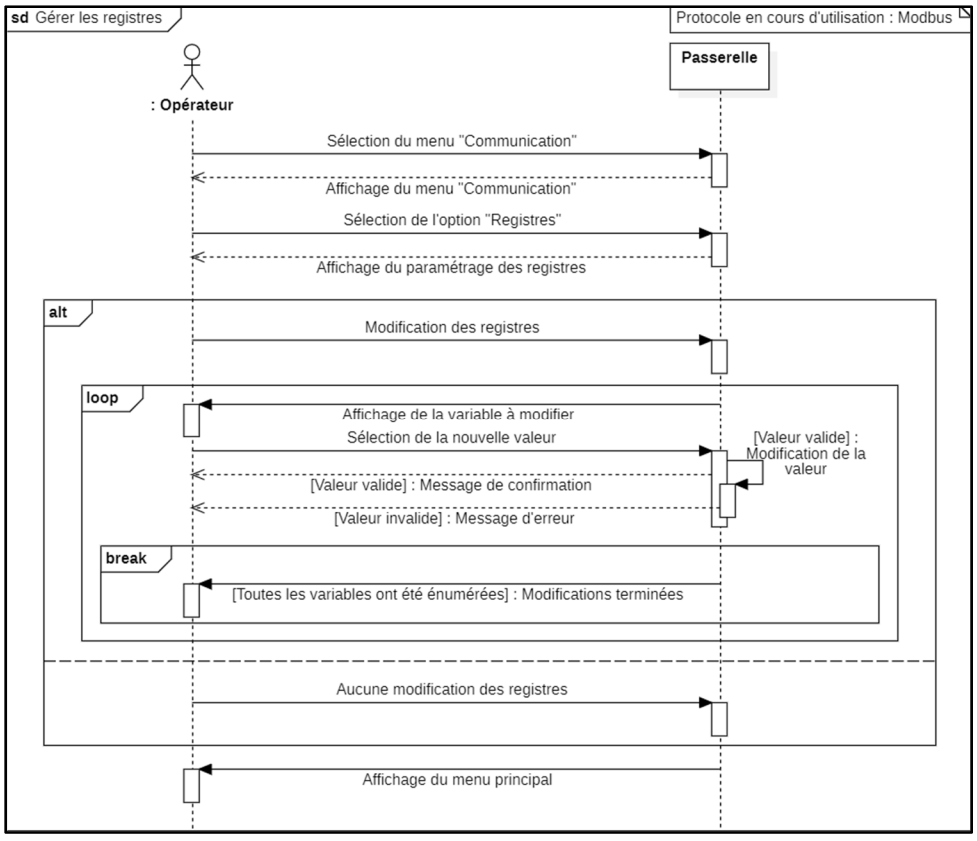


Figure-A I-1 Diagramme de séquence "Gérer les registres Modbus"

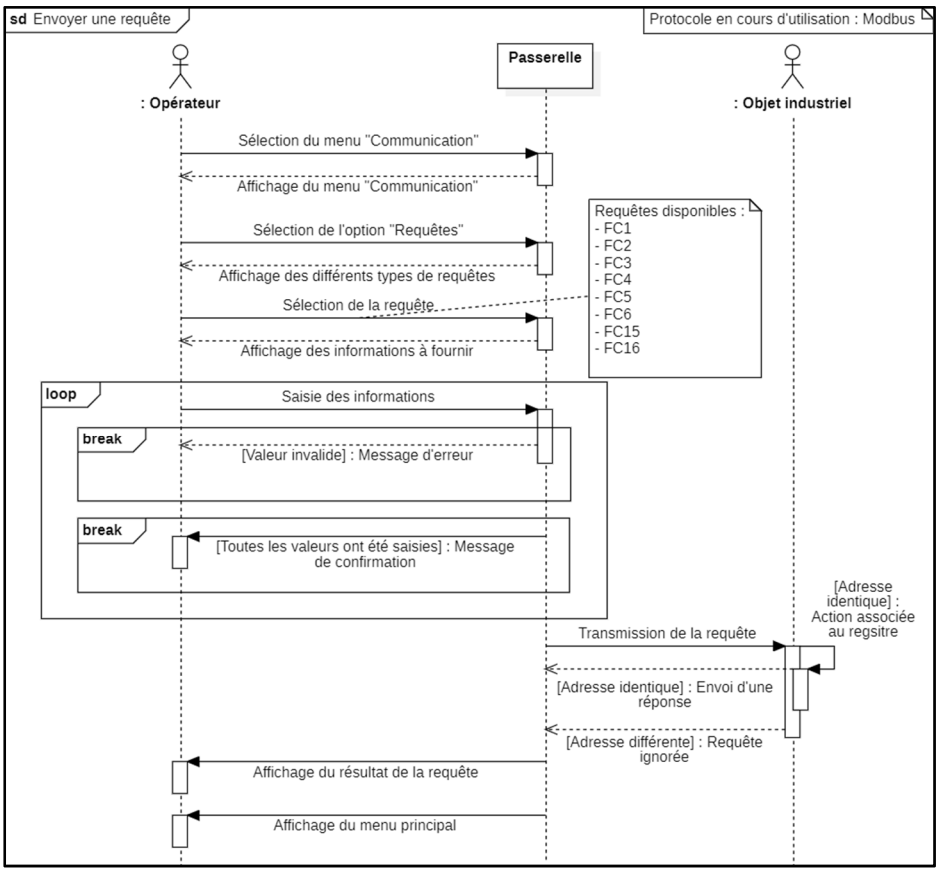


Figure-A I-2 Diagramme de séquence "Envoyer une requête Modbus"

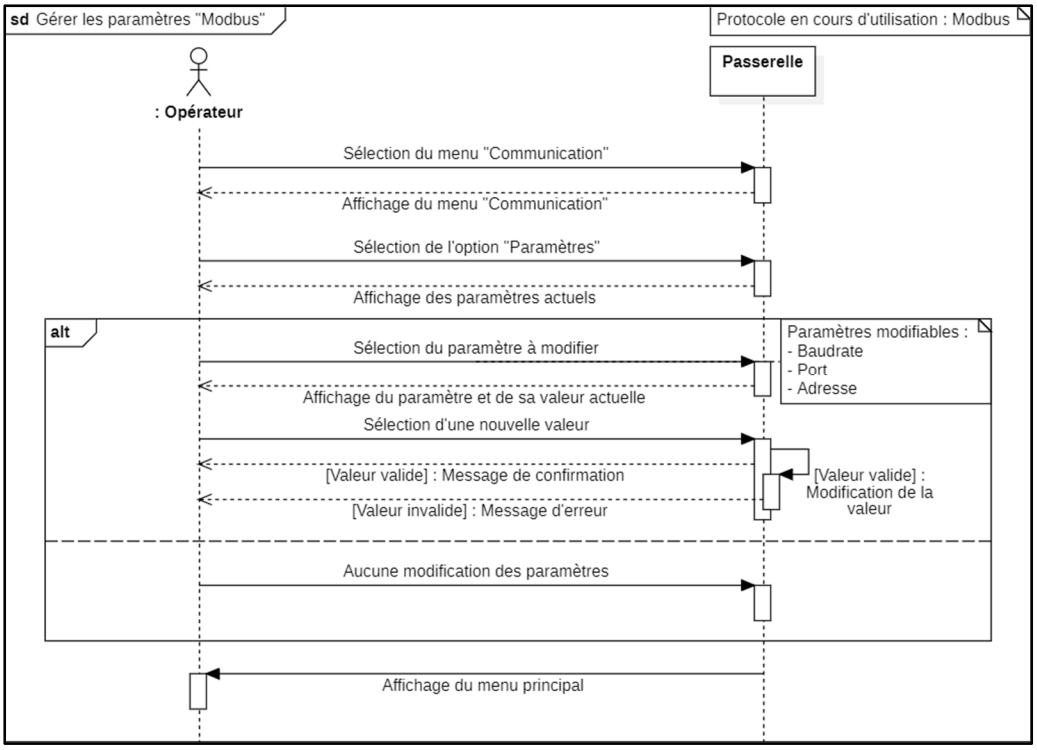


Figure-A I-3 Diagramme de séquence "Gérer les paramètres Modbus"

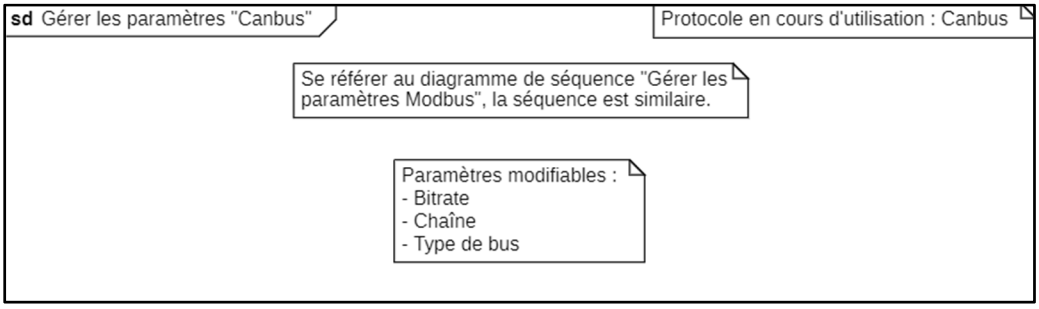


Figure-A I-4 Diagramme de séquence "Gérer les paramètres Canbus"

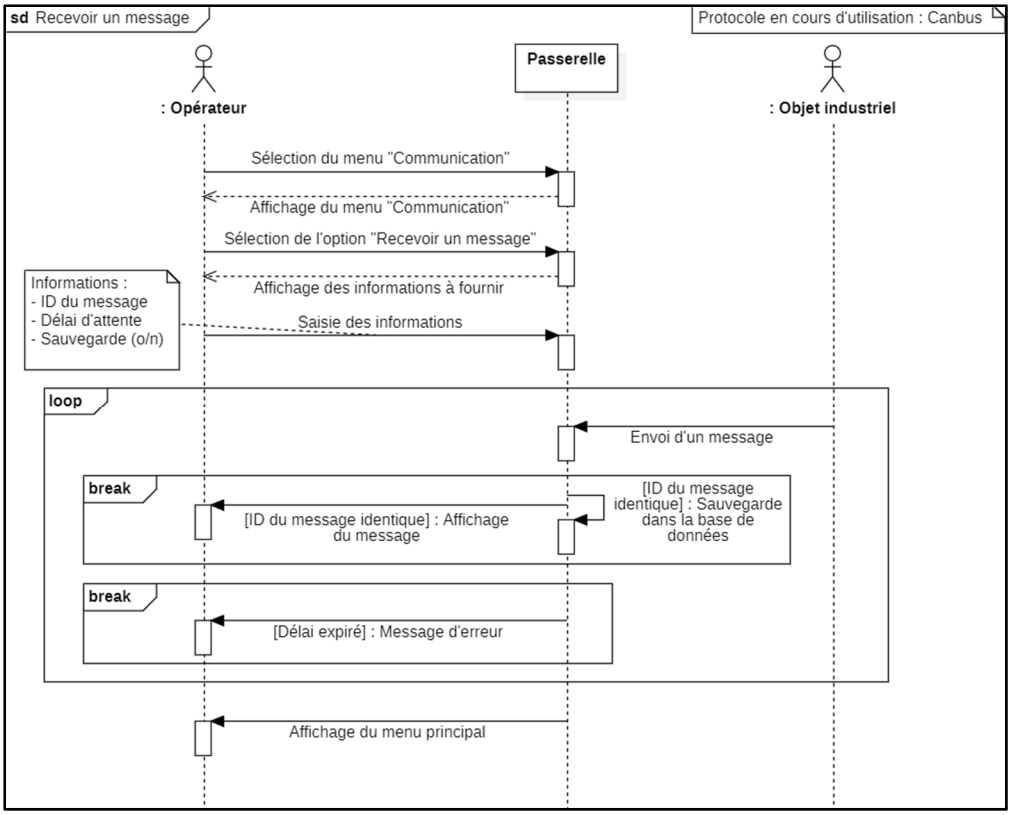


Figure-A I-5 Diagramme de séquence "Recevoir un message Canbus"

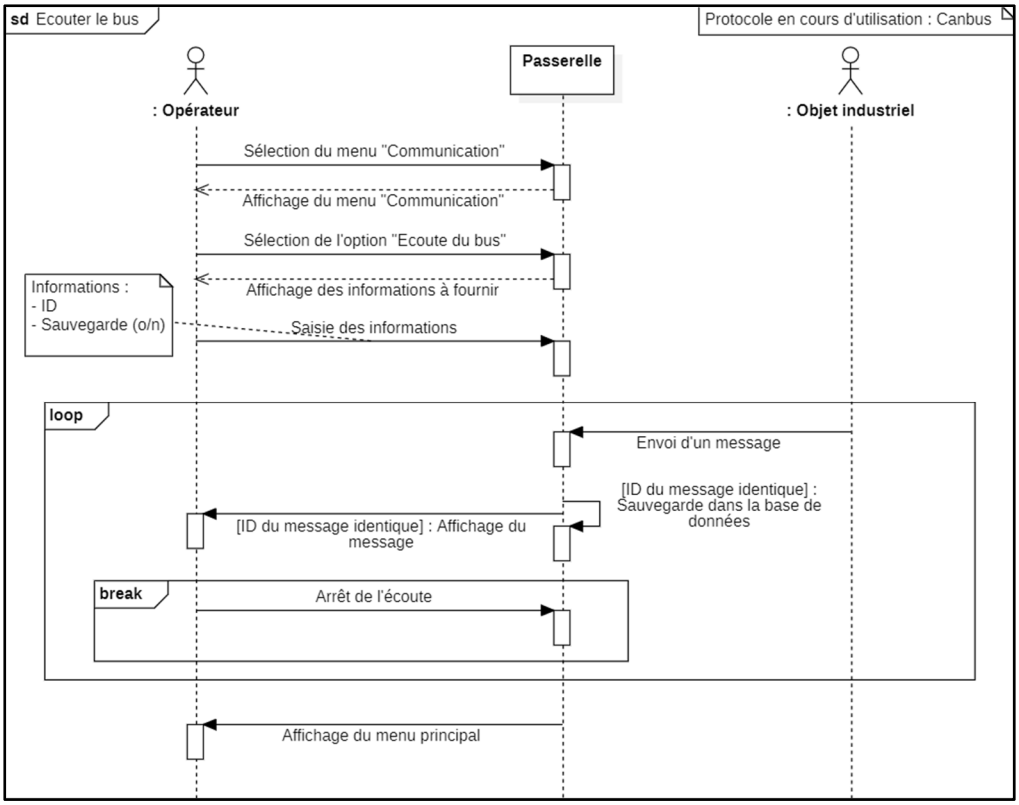


Figure-A I-6 Diagramme de séquence "Écouter le bus Canbus"

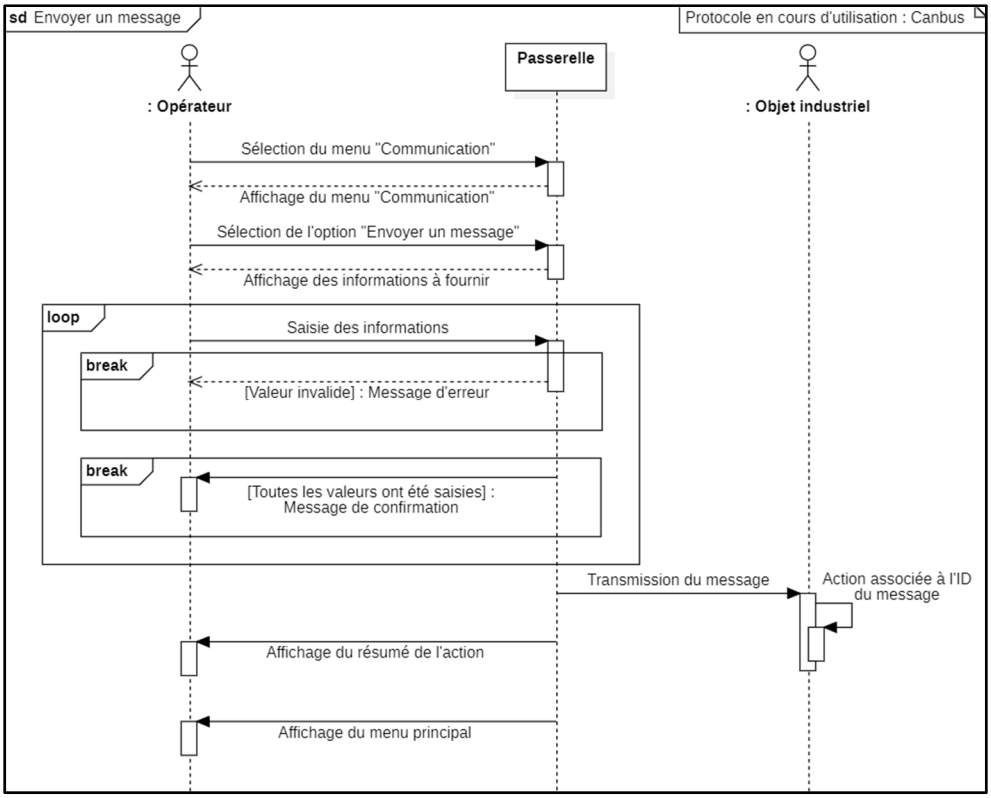


Figure-A I-7 Diagramme de séquence "Envoyer un message Canbus"

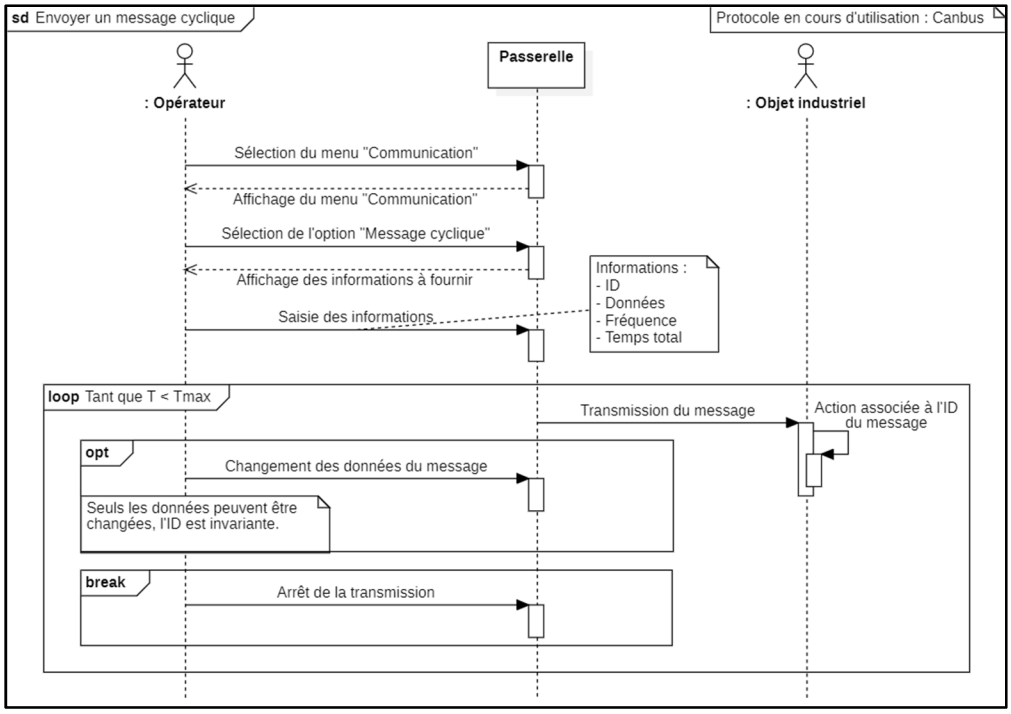


Figure-A I-8 Diagramme de séquence "Envoyer un message cyclique Canbus"

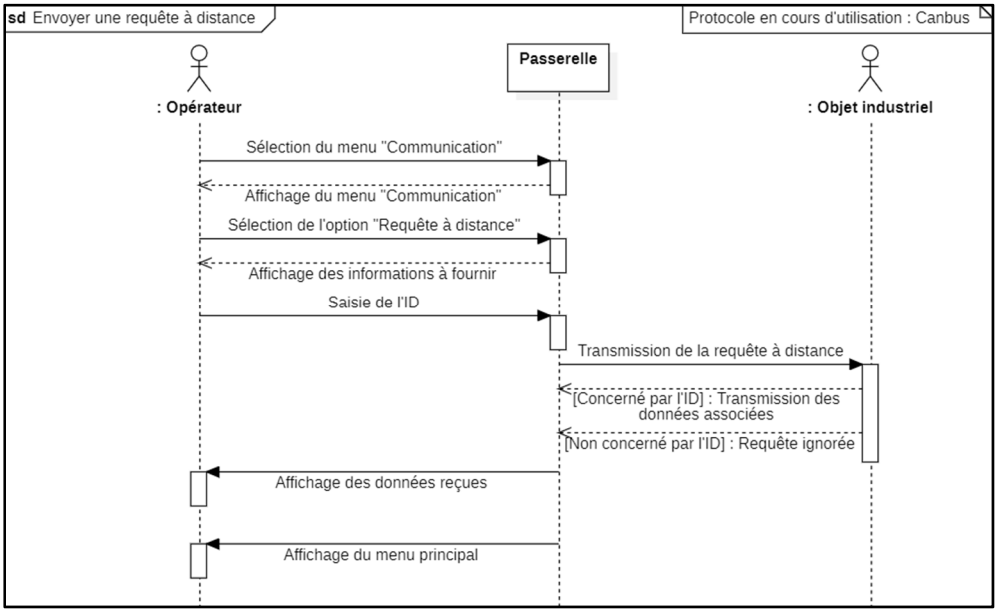


Figure-A I-9 Diagramme de séquence "Envoyer une requête à distance Canbus"

ANNEXE II

MENU CONFIGURATION

La passerelle IIoT permet à l'opérateur de réaliser plusieurs réglages via son menu de configuration. Ci-dessous sont illustrées les trois principales actions accessibles par ce menu.

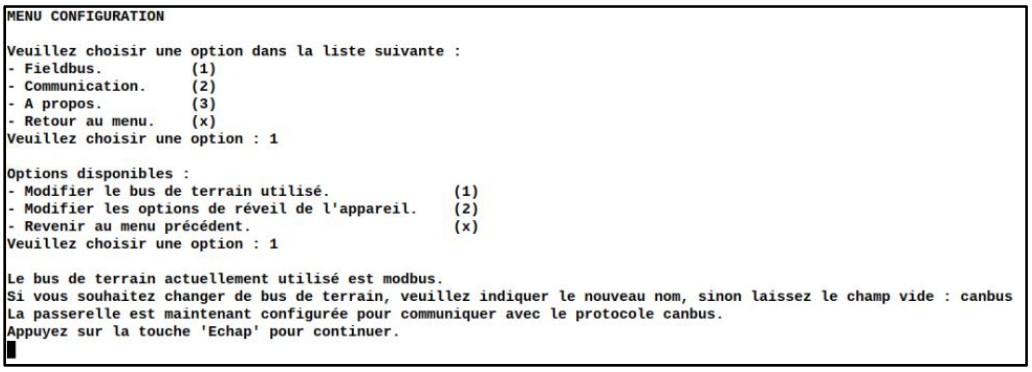


Figure-A II-1 Modification du bus de terrain utilisé pour les communications

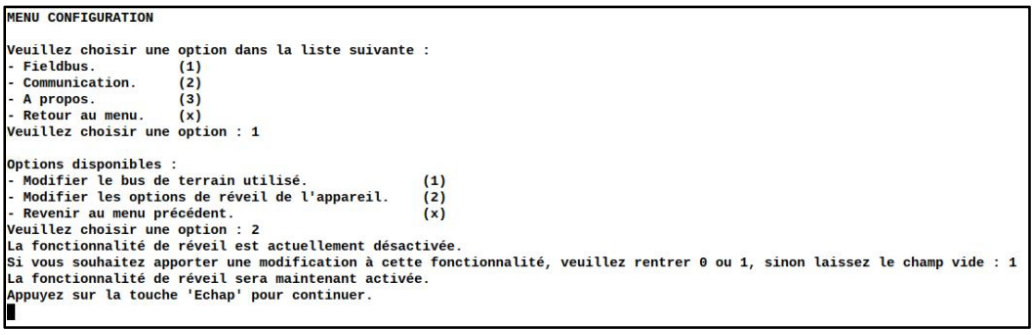


Figure-A II-2 Activation de la fonctionnalité de réveil pour le protocole Modbus

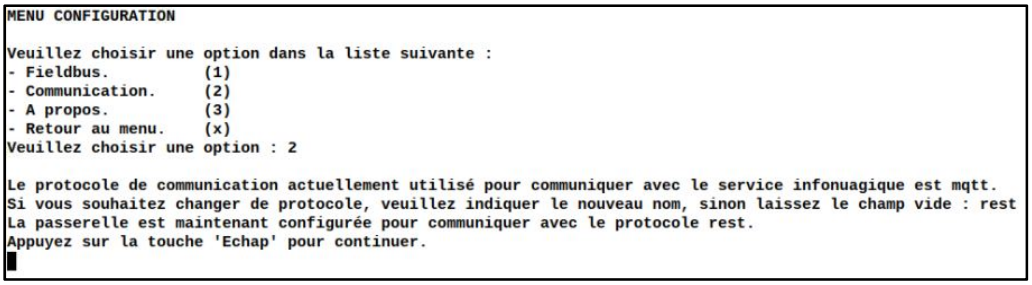


Figure-A II-3 Modification du protocole de communication utilisé

ANNEXE III

LISTE DES COMPOSANTS MATÉRIELS

Plusieurs montages électroniques ont été réalisés pour valider le fonctionnement et évaluer les performances de la passerelle multi-protocole de ce projet de recherche. Voici la liste des composants matériels utilisés :

- Écran LCD

L'écran LCD utilisé est reconnu sous la désignation « I2C LCD1602 ». Il permet l'affichage de caractères. Grâce à cet élément, il nous est possible de vérifier si les transferts de données textuelles s'effectuent sans encombre. Cette vérification est capitale, car les bus de terrain ne supportent pas les caractères ASCII et il n'est pas possible d'envoyer directement un caractère. Les données sont transmises sous leur forme la plus basique, à savoir sous forme de bits, une série de conversions en amont et en aval du bus sont alors de mise.

- Servomoteur

Il s'agit d'un actionneur, d'un moteur asservi en position. Ce composant, le SG90, offre une possibilité simple de tester l'envoi et la réception de nombres entiers et d'avoir un témoin physique et visuel.

- LED

L'emploi de LED peut s'avérer efficace pour rapidement déceler l'état d'un registre ou d'un bit. Il est par exemple possible d'associer l'état éteint d'une LED avec la valeur 0 d'un bit, et similairement déclencher l'allumage lorsque cette valeur passe à 1.

- Potentiomètre

L'utilisation d'un potentiomètre est, dans notre cas, principalement dirigée vers la génération d'une valeur variable et contrôlée par l'utilisateur. Cette valeur est par la suite destinée à être transmise au bus et à la passerelle IIoT.

- Interrupteur

La maquette disposera d'un interrupteur à bascule. Son intérêt est similaire à celui du potentiomètre, nous avons en effet recours à un interrupteur pour contrôler la valeur d'un registre. Une différence réside néanmoins dans le fait que cette valeur est binaire (0 ou 1) contrairement au potentiomètre, le type des données n'est ainsi pas le même.

- Résistances

La solution mise en place nécessite plusieurs résistances pour le bon fonctionnement des composants et le respect des exigences électriques. Pour ce faire, nous emploierons des résistances de 220 Ω, 1 kΩ et 10 kΩ.

- Fils électriques

La mise en relation des différents composants énoncés dans cette liste est rendue possible par l'emploi de fils électriques. Une attention toute particulière sera alors accordée aux branchements afin de réaliser un circuit propre et pertinent, respectant l'intégrité des composants.

- Câbles

Les bus de terrain reposent sur des supports physiques devant être conformes à certaines normes. On ne peut donc pas utiliser de simples fils électriques. La norme RS485 recommande l'usage d'une paire de câbles torsadés pour la transmission du signal, accompagnée d'un câble de terre. Le calibre de ces câbles doit de préférence correspondre à 22 ou 24 AWG. L'unité AWG, pour *American Wire Gauge*, est une unité de mesure américaine permettant de quantifier le diamètre d'un câble électrique. L'aspect torsadé de la paire de câbles et la symétrie de leurs signaux permettent de réduire grandement les interférences magnétiques qu'ils pourraient subir de la part d'éléments environnants. Cela permet de se soustraire au besoin d'un blindage assurant l'isolation de la ligne. En addition, les câbles se doivent d'avoir une impédance comprise aux alentours de 100 à 120 Ω (Kugelstadt, 2021). Il est important de préciser que l'emploi d'un blindage peut s'avérer utile et est même recommandé lorsque les

distances deviennent importantes ou bien que l'environnement est particulièrement saturé par des ondes électromagnétiques (Measurlogic, 2020).

En ce qui concerne les réseaux CAN, il est recommandé de disposer d'une paire de câbles torsadés ayant une impédance proche de 120 Ω. Une certaine tolérance est admise quant à cette valeur, elle peut en effet osciller entre 95 et 140 Ω (Elektromotus, n. d.). Le calibre du câble doit de préférence se trouver proche d'une valeur de 24 AWG (Corrigan, 2008).

Les câbles conformes à la norme EIA/TIA-568-B sont tout à fait adaptés aux critères imposés par la norme RS485 avec une impédance de 100 Ω et un diamètre de 24 AWG (Anixter Inc, 2006). Ces câbles répondent également aux exigences des bus CAN. Ce sont donc ces câbles dont nous servirons pour l'ensemble de nos tests.

- Adaptateur USB

Il s'agit ici d'un adaptateur RS485/RS422 de référence SH-U11. Cet objet nous permet de relier notre bus à n'importe quel PC et ainsi émuler une nouvelle machine industrielle. Il existe en effet des logiciels qui permettent de créer un coordonnateur ou un nœud, de façon virtuelle, directement depuis Windows, Linux ou Mac et ainsi d'interagir avec les autres nœuds. On peut alors modifier les registres de l'appareil ou choisir des requêtes à envoyer. L'intérêt de ce type de logiciel est surtout de déboguer un système, cela se révèle donc utile, notamment en phase de développement. Parmi ces logiciels, notre choix s'est orienté vers « *SimplyModbus* », une suite de logiciels gratuite, basée sur les dons. Deux logiciels nous seront utiles, la version « *Simply Modbus Master 8.1.2* » permettant de simuler un coordonnateur et « *Simply Modbus Slave 8.1.3* » offrant la possibilité de simuler un nœud (SimplyModbus, 2020).

- Capteur Modbus

L'achat d'un capteur de température et d'humidité AM2320 ayant une interface RS485 déjà greffée est un moyen simple d'être en possession d'un esclave Modbus qui ne requiert aucun réglage de notre part. Cette pièce, identifiée par le numéro « Walfrontui392yr1gv » est déjà entièrement configurée et ne nécessite qu'une alimentation de 5V. Fonctionnant avec Modbus

RTU, elle envoie de manière cyclique par le biais de la fonction « 06 – *Write* » les valeurs de température, en degrés Celsius, et d'humidité, en pourcentage. Ces valeurs sont transmises avec un facteur multiplicatif de 10 pour contourner les problèmes de nombres flottants, son usage est donc extrêmement simple. Après correction, la précision des valeurs correspond à un dixième d'unité.

BIBLIOGRAPHIE

Agarwal, P., & Alam, M. (2020). Investigating IoT Middleware Platforms for Smart Application Development. Dans *Smart Cities—Opportunities and Challenges* (pp. 231-244). doi: 10.1007/978-981-15-2545-2_21

Anixter Inc. (2006). Standards Reference Guide.

Bassi, L. (2017). Industry 4.0: Hope, hype or revolution? Dans *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)* (pp. 1-6). doi: 10.1109/RTSI.2017.8065927

Bera, S. (2021). Do-It-Yourself: CAN Bus to Control Appliances on Any Floor of a Building. *Electronics for You*.

Bertrand-Martinez, E., Dias Feio, P., Brito Nascimento, V. d., Kon, F., & Abelém, A. (2020). Classification and evaluation of IoT brokers: A methodology. *International Journal of Network Management, 31*(3). doi: 10.1002/nem.2115

BoppreH et al. (2020). keyboard 0.13.5. Repéré à https://pypi.org/project/keyboard/

Calderón Godoy, A., & González Pérez, I. (2018). Integration of Sensor and Actuator Networks and the SCADA System to Promote the Migration of the Legacy Flexible Manufacturing System towards the Industry 4.0 Concept. *Journal of Sensor and Actuator Networks, 7*(2). doi: 10.3390/jsan7020023

Cao, K., Liu, Y., Meng, G., & Sun, Q. (2020). An Overview on Edge Computing Research. *IEEE Access, 8*, 85714-85728. doi: 10.1109/ACCESS.2020.2991734

Chen, C. H., Lin, M. Y., & Liu, C. C. (2018). Edge Computing Gateway of the Industrial Internet of Things Using Multiple Collaborative Microcontrollers. *IEEE Network, 32*(1), 24-32. doi: 10.1109/Mnet.2018.1700146

Chui, M., Collins, M., & Patel, M. (2021). *The Internet of Things: Catching up to an accelerating opportunity*. McKinsey & Company.

Cisco. (2020). *Cisco Annual Internet Report (2018–2023)*.

Collins et al. (2021). pymodbus 2.5.3. Repéré à https://pypi.org/project/pymodbus/

Cook, J. A., & Freudenberg, J. S. (2008). Controller Area Network (CAN).

Cordeiro, A., Costa, P., Pires, V., & Foito, D. (2018). Modbus Protocol as Gateway Between Different Fieldbus Devices - a Didactic Approach. Dans *Teaching and Learning in a Digital World* (pp. 860-871). doi: 10.1007/978-3-319-73210-7_99

Corotinschi, G., & Gaitan, V. G. (2018). Enabling IoT connectivity for Modbus networks by using IoT edge gateways (pp. 175-179). Piscataway: The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

Corrigan, S. (2008). Controller Area Network Physical Layer Requirements.

Corrigan, S. (2016). *Introduction to the Controller Area Network (CAN)*. Texas Instruments Incorporated.

Croston, B. (2022). RPi.GPIO 0.7.1. Repéré à https://pypi.org/project/RPi.GPIO/

Cupek, R., Ziebinski, A., & Drewniak, M. (2017). An OPC UA server as a gateway that shares CAN network data and engineering knowledge. Dans *2017 IEEE International Conference on Industrial Technology (ICIT)* (pp. 1424-1429). doi: 10.1109/ICIT.2017.7915574

Dell Inc. (2017). *Dell Edge Gateway 5000 Series*. Repéré à https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/us/Spec_Sheet_Dell_Edge_Gateway_5000_Series.pdf

Dietrich, S., May, G., Hoyningen-Huene, J. v., Mueller, A., & Fohler, G. (2017). Frame Conversion Schemes for Cascaded Wired / Wireless Communication Networks of Factory Automation. *Mobile Networks and Applications, 23*(4), 817-827. doi: 10.1007/s11036-017-0881-2

Dirgantoro, K. P., Nwadiugwu, W.-P., Lee, J. M., & Kim, D.-S. (2020). Dual fieldbus industrial IoT networks using edge server architecture. *Manufacturing Letters, 24*, 108-112. doi: 10.1016/j.mfglet.2020.04.006

Divyashree, M., & Rangaraju, H. G. (2018). Internet of Things (IoT): A Survey. Dans *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)* (pp. 1-6). doi: 10.1109/ICNEWS.2018.8903919

Elektromotus. (n. d.). *Elektromotus CAN bus topology recommendations v0.2 rc2*.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures"* (Doctoral dissertation, University of California, Irvine).

Galloway, B., & Hancke, G. P. (2013). Introduction to Industrial Control Networks. *IEEE Communications Surveys & Tutorials, 15*(2), 860-880. doi: 10.1109/SURV.2012.071812.00124

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education. Repéré à https://books.google.ca/books?id=6oHuKQe3TjQC

Gang, H., Zhilou, Y., Naikuo, C., Guirong, F., Lei, Z., Boyi, H., & Shiming, Q. (2020). Development of OPC UA based centralized server Fieldbus data high efficiency transmit architecture. Dans *2020 39th Chinese Control Conference (CCC)* (pp. 4580-4585). doi: 10.23919/CCC50068.2020.9188765

Gemirter, C. B., Şenturca, Ç., & Baydere, Ş. (2021). A Comparative Evaluation of AMQP, MQTT and HTTP Protocols Using Real-Time Public Smart City Data. Dans *2021 6th International Conference on Computer Science and Engineering (UBMK)* (pp. 542-547). doi: 10.1109/UBMK52708.2021.9559032

GM International. (2020). Why Industry Will Continue to Use Modbus. Repéré à https://news.gminternational.com/why-industry-will-continue-use-modbus

Guth, J., Breitenbücher, U., Falkenthal, M., Leymann, F., & Reinfurt, L. (2016). Comparison of IoT platform architectures: A field study based on a reference architecture. Dans *2016 Cloudification of the Internet of Things (CIoT)* (pp. 1-6). doi: 10.1109/CIOT.2016.7872918

Hassanpour, V., Rajabi, S., Shayan, Z., Hafezi, Z., & Arefi, M. M. (2017). Low-cost home automation using Arduino and Modbus protocol (pp. 284-289). Piscataway: The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

Hemmatpour, M., Ghazivakili, M., Montrucchio, B., & Rebaudengo, M. (2017). DIIG: A Distributed Industrial IoT Gateway. Dans *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 1, pp. 755-759). doi: 10.1109/COMPSAC.2017.110

IEEE. (2019). *IEEE Standard for Floating-Point Arithmetic*. doi: 10.1109/IEEESTD.2019.8766229

Ihenacho, T. (2014). Agile SCRUM Methodology - A Project Management Framework. *Agile SCRUM Methodology - A Project Management Framework*.

Industrial IoT Gateways SIMATIC CloudConnect 7. (n. d.). Repéré à https://new.siemens.com/global/en/products/automation/industrial-communication/industrial-ethernet/industrial-iot-gateway-simatic-cloudconnect-7.html

Integrate Revolution Pi into an industrial network. (n. d.). Repéré le 12 janvier 2022 à https://revolutionpi.com/gateways/

International Electrotechnical Commission. (2003). *Digital data communications for measurement and control - Fieldbus for use in industrial control systems - Part 2: Physical layer specification and service definition*.

International Electrotechnical Commission. (2019). *Industrial communication networks - Fieldbus specifications - Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series (IEC 61158-1:2019)*.

International Organization for Standardization. (2015). *Road vehicles - Controller area network (CAN) - Part 1 : Data link layer and physical signalling (ISO 11898-1:2015)*.

International Organization for Standardization. (2016). *Information technology - Message Queuing Telemetry Transport (MQTT) v3.1.1 (ISO/IEC 20922:2016)*. Switzerland.

International Organization for Standardization. (2019). *Date and time format (ISO 8601-1:2019)*.

Internet Assigned Numbers Authority. (2021). Hypertext Transfer Protocol (HTTP) Status Code Registry. Repéré à https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml#http-status-codes-1

InterviewBit. (2022). Difference Between C and Python. Repéré à https://www.interviewbit.com/blog/difference-between-c-and-python/

Jaloudi, S. (2019). Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study. *Future Internet, 11*(3), 66. Repéré à https://www.mdpi.com/1999-5903/11/3/66

Khanchuea, K., & Siripokarpirom, R. (2019). A Multi-Protocol IoT Gateway and WiFi/BLE Sensor Nodes for Smart Home and Building Automation: Design and Implementation. Dans *2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)* (pp. 1-6). doi: 10.1109/ICTEmSys.2019.8695968

Kim, T.-W., Kim, J., Lee, D., Moon, J., & Jeon, J. (2016). *Development of gateway based on BroadR-reach for application in automation network*. doi: 10.1109/CCAA.2016.7813805

Koch, R., & Lueftner, R. (2019). *Communication Networks in Automation*.

Krishnasamy, E., Varrette, S., & Mucciardi, M. (2020). Edge Computing: An Overview of Framework and Applications. *Zenodo*. doi: 10.5281/zenodo.5717280

Kugelstadt, T. (2021). *The RS-485 Design Guide*. Texas Instruments Incorporated.

Light, R. (2021). paho-mqtt 1.6.1. Repéré à https://pypi.org/project/paho-mqtt/

Lima, F., Massote, A. A., & Maia, R. F. (2019). IoT Energy Retrofit and the Connection of Legacy Machines Inside the Industry 4.0 Concept. Dans *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society* (Vol. 1, pp. 5499-5504). doi: 10.1109/IECON.2019.8927799

Lo, S. (2022). Longan Labs Serial CAN Bus Module Library. Repéré à https://github.com/Longan-Labs/Serial_CAN_Arduino

Longan Labs. (2017). SERIAL CAN BUS MODULE. Repéré à https://docs.longan-labs.cc/1030001/

Lüder, A., Schleipen, M., Schmidt, N., Pfrommer, J., & Henßen, R. (2017). One step towards an industry 4.0 component. Dans *2017 13th IEEE Conference on Automation Science and Engineering (CASE)* (pp. 1268-1273). doi: 10.1109/COASE.2017.8256275

Luis Bustamante, A., Patricio, M. A., & Molina, J. M. (2019). Thinger.io: An Open Source Platform for Deploying Data Fusion Applications in IoT Environments. *Sensors (Basel), 19*(5). doi: 10.3390/s19051044. Repéré à https://www.ncbi.nlm.nih.gov/pubmed/30823643

Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications, 03*(05), 164-173. doi: 10.4236/jcc.2015.35021

Mamo, F. T., Sikora, A., & Rathfelder, C. (2017). Legacy to Industry 4.0: A Profibus Sniffer. *Journal of Physics: Conference Series, 870*. doi: 10.1088/1742-6596/870/1/012002

Mangkalajan, S., Koodtalang, W., Sangsuwan, T., & Pudchuen, N. (2019). Virtual Process Using LabVIEW in Combination with Modbus TCP for Fieldbus Control System (pp. 21-24). Piscataway: The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

Martinez, B., Vilajosana, X., Kim, I. H., Zhou, J., Tuset-Peiro, P., Xhafa, A., . . . Lu, X. (2017). I3Mote: An Open Development Platform for the Intelligent Industrial Internet. *Sensors (Basel), 17*(5). doi: 10.3390/s17050986. Repéré à https://www.ncbi.nlm.nih.gov/pubmed/28452945

Matt, D., & Rauch, E. (2020). SME 4.0: The Role of Small- and Medium-Sized Enterprises in the Digital Transformation. Dans (pp. 3-36). doi: 10.1007/978-3-030-25425-4_1

Maxim Integrated. (2014). MAX481/MAX483/MAX485/MAX487–MAX491/MAX1487 Low-Power, Slew-Rate-Limited RS-485/RS-422 Transceivers.

Maxim Integrated. (2015). MAX13487E/MAX13488E Half-Duplex RS-485-/RS-422 Compatible Transceiver with AutoDirection Control.

Measurlogic. (2020). RS-485 Cabling Requirements for Modbus RTU and BACnet MSTP Revision R20A.

Miao, W., Ting-Jie, L., Fei-Yang, L., Jing, S., & Hui-Ying, D. (2010). *Research on the architecture of Internet of Things* présentée à 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE). doi: 10.1109/icacte.2010.5579493

Mishra, B., & Kertesz, A. (2020). The Use of MQTT in M2M and IoT Systems: A Survey. *IEEE Access, 8*, 201071-201086. doi: 10.1109/access.2020.3035849

Modbus Organization. (2012). Modbus Application Protocol V1_1b3.

Modular IoT Gateway OEM IoT Gateway for Zigbee, Thread & Bluetooth based Enterprise Solutions. (n. d.). Repéré le 12 janvier 2022 à https://volansys.com/modular-iot-gateway/

Moeuf, A., Pellerin, R., Lamouri, S., Tamayo-Giraldo, S., & Barbaray, R. (2017). The industrial management of SMEs in the era of Industry 4.0. *International Journal of Production Research, 56*(3), 1118-1136. doi: 10.1080/00207543.2017.1372647

Mossin, E. A., Pantoni, R. P., & Brandao, D. (2009). A fieldbus simulator for training purposes. *ISA Trans, 48*(1), 132-141. doi: 10.1016/j.isatra.2008.07.005. Repéré à https://www.ncbi.nlm.nih.gov/pubmed/18757053

MQTT.org. (2020). FAQ - What is MQTT. Repéré à https://mqtt.org/faq/

Nguyen-Hoang, P., & Vo-Tan, P. (2019). *Development An Open-Source Industrial IoT Gateway* présentée à 2019 19th International Symposium on Communications and Information Technologies (ISCIT). doi: 10.1109/iscit.2019.8905157

Nikolov, N. (2020). Research of MQTT, CoAP, HTTP and XMPP IoT Communication protocols for Embedded Systems. Dans *2020 XXIX International Scientific Conference Electronics (ET)* (pp. 1-4). doi: 10.1109/ET50336.2020.9238208

Parikh, D. (2018). Selecting the right IoT gateway. Repéré à https://www.newelectronics.co.uk/content/features/selecting-the-right-iot-gateway

Pielli, C., Zucchetto, D., Andrea Zanella, A., Vangelista, L., & Zorzi, M. (2015). Platforms and Protocols for the Internet of Things. *EAI Endorsed Transactions on Internet of Things, 1*(1). doi: 10.4108/eai.26-10-2015.150599

Qaisar, E. J. (2012). Introduction to cloud computing for developers: Key concepts, the players and their offerings. Dans *2012 IEEE TCF Information Technology Professional Conference* (pp. 1-6). doi: 10.1109/TCFProIT.2012.6221131

Raspberry Pi (Trading) Ltd. (2019). *Raspberry Pi 4 Model B Datasheet*. Repéré à https://www.raspberrypi.org

Ravulavaru, A. (2018). *Enterprise Internet of Things Handbook*. Packt Publishing. Repéré à https://app.knovel.com/hotlink/toc/id:kpEITH0001/enterprise-internet-things/enterprise-internet-things

Reitz et al. (2022). requests 2.27.1. Repéré à https://pypi.org/project/requests/

Rickman et al. (2020). LiquidCrystal_I2C. Repéré à https://github.com/johnrickman/LiquidCrystal_I2C

Rojas, R., & Garcia, M. (2020). Implementation of Industrial Internet of Things and Cyber-Physical Systems in SMEs for Distributed and Service-Oriented Control. Dans (pp. 73-103). doi: 10.1007/978-3-030-25425-4_3

Rüßmann, M., Lorenz, M., Gerbert, P. d. S., Waldner, M., Justus, J., Engel, P., & Harnisch, M. J. (2015). *Industry 4.0 : The Future of Productivity and Growth in Manufacturing Industries*.

Sharma, C., & Gondhi, N. K. (2018). Communication Protocol Stack for Constrained IoT Systems. Dans *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)* (pp. 1-6). doi: 10.1109/IoT-SIU.2018.8519904

SimplyModbus. (2020). Data Communication Test Software. Repéré à https://www.simplymodbus.ca/index.html

Stouffer, K., Pillitteri, V., Lightman, S., Abrams, M., & Hahn, A. (2015). Guide to Industrial Control Systems (ICS) Security. doi: 10.6028/NIST.SP.800-82r2

Sun, C., Guo, K., Xu, Z., Ma, J., & Hu, D. (2019). Design and Development of Modbus/MQTT Gateway for Industrial IoT Cloud Applications Using Raspberry Pi (pp. 2267-2271). Piscataway: The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

Thorne et al. (2022). python-can 4.0.0. Repéré à https://pypi.org/project/python-can/

Toc, S., & Korodi, A. (2018). Modbus-OPC UA Wrapper Using Node-RED and IoT-2040 with Application in the Water Industry. Dans *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)* (pp. 000099-000104). doi: 10.1109/SISY.2018.8524749

Turner, J., & Smith, C. (2014). Common types of CAN. *Automotive Electrical & Air Conditioning News*.

Veneri, G., & Capasso, A. (2018). *Hands On Industrial Internet of Things*.

Viegas, V., Postolache, O. A., Girao, P. M. B. S., & Pereira, J. M. D. (2016). Quimera: The Easy Way to Simulate Foundation Fieldbus Applications. *Computer Applications in Engineering Education, 24*(6), 914-925. doi: 10.1002/cae.21761. Repéré à <Go to ISI>://WOS:000396370700009

Waveshare. (2019). Stack HAT. Repéré à https://www.waveshare.com/wiki/Stack_HAT

Waveshare. (2021). 2-CH CAN HAT. Repéré à https://www.waveshare.com/wiki/2-CH_CAN_HAT

Waveshare. (2022). 2-CH RS485 HAT. Repéré à https://www.waveshare.com/wiki/2-CH_RS485_HAT

Wei, C., Xijun, W., Wenxia, S., & Ruitao, Y. (2016). *The design of PROFINET-MODBUS protocol conversion gateway based on the ERTEC 200P*. doi: 10.1109/SKIMA.2016.7916202

Wong, T., & Rousseau, F. (2018). GPA788 Conception et intégration des objets connectés.

Xu, S., Fei, M., & Wang, H. (2016). Design of Hybrid Wired/Wireless Fieldbus Network for Turbine Power Generation System. *Information, 7*, 37. doi: 10.3390/info7030037

Ye, Y., & Lei, H. (2016). Wireless industrial communication system based on Profibus-DP and ZigBee. Dans *2016 11th International Conference on Computer Science & Education (ICCSE)* (pp. 666-669). doi: 10.1109/ICCSE.2016.7581659

Yida. (2019). UART vs I2C vs SPI – Communication Protocols and Uses. Repéré à https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/

You, W., & Ge, H. (2019). Design and Implementation of Modbus Protocol for Intelligent Building Security. Dans *2019 IEEE 19th International Conference on Communication Technology (ICCT)* (pp. 420-423). doi: 10.1109/ICCT46805.2019.8946996

Youness. (2018). How to Connect Raspberry Pi to CAN Bus. Repéré à https://www.hackster.io/youness/how-to-connect-raspberry-pi-to-can-bus-b60235

Zamir et al. (2021). ModbusSlave. Repéré à https://github.com/yaacov/ArduinoModbusSlave

Zawra, L. M., Mansour, H. A., & Messiha, N. W. (2019). Migration of Legacy Industrial Automation Systems in the Context of Industry 4.0- A Comparative Study. Dans *2019 International Conference on Fourth Industrial Revolution (ICFIR)* (pp. 1-7). doi: 10.1109/ICFIR.2019.8894776

Zhang, H., Li, Y., & Zhu, H. (2011). Development for Protocol Conversion Gateway of Profibus and Modbus. *Procedia Engineering, 15*, 767-771. doi: 10.1016/j.proeng.2011.08.143

Zhou, Y., Li, X., Liu, M., & Lin, F. (2016). Research and development of android client for PROFIBUS-DP based on Wi-Fi. Dans *2016 2nd International Conference on Control, Automation and Robotics (ICCAR)* (pp. 370-373). doi: 10.1109/ICCAR.2016.7486757

Zihatec. (2018). How to Use Modbus with Arduino. Repéré à https://create.arduino.cc/projecthub/hwhardsoft/how-to-use-modbus-with-arduino-6f434b

Zihatec. (n.d.). How to Use Modbus With Raspberry Pi. Repéré à https://www.instructables.com/How-to-Use-Modbus-With-Raspberry-Pi/

Zimmermann, H. (1980). OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications, 28*(4), 425-432. doi: 10.1109/TCOM.1980.1094702

Zurawski, R. (2017). *Industrial Communication Technology Handbook*. CRC Press. Repéré à https://books.google.ca/books?id=ppzNBQAAQBAJ