Université de Lille

cnrs

École Doctorale régionale Sciences pour l'Ingénieur Lille Nord-de-France

Mémoire pour l'obtention du titre de

DOCTEUR DE L'UNIVERSITÉ DE LILLE

Discipline : Informatique et applications

présentée et soutenue publiquement par

Mahieddine YAKER

le 19 décembre 2019

Conception et réalisation d'un écosystème basé sur des propriétés formelles d'isolation mémoire pour l'Internet des objets

Directeur de thèse : **Gilles Grimaud**
Co-encadrant de thèse : **Julien Cartigny**

Jury

Samia Bouzefrane,	Professeur	Rapporteure
Didier Donsez,	Professeur	Rapporteur
Nathalie Rolland,	Professeur	Examinatrice
Kevin Marquet,	Maître de conférences	Examinateur
Chrystel Gaber,	Docteur R&D	Invitée

THESE

Remerciements

J'aimerais tout d'abord remercier mes encadrants de thèse, Gilles Grimaud et Julien Cartigny de la confiance et du soutien qu'ils m'ont accordé durant ces trois années de thèse. Je les remercie de m'avoir encouragé à me lancer dans cette aventure et pour tous les conseils humains, techniques et scientifiques qu'ils m'ont prodigué. Comme je leur ai toujours dit, même si cette aventure fut parfois difficile, je n'ai aucun regret de l'avoir partagé avec eux.

Je remercie chaleureusement Madame Samia Bouzefrane et Monsieur Didier Donsez pour avoir accepté d'être les rapporteurs de cette thèse. Je suis naturellement honoré et fier de votre présence au sein du jury, ainsi que pour l'intérêt que vous avez porté à mon travail. Je suis également reconnaissant envers Madame Nathalie Haese Rolland et Monsieur Kevin Marquet d'avoir accepté de faire partie du jury de cette thèse en tant qu'examinateurs.

Je tiens à exprimer toute ma gratitude à Chrystel Gaber pour son expérience partagée et la collaboration fructueuse à la réalisation de mes travaux.

Sur un ton plus personnel, j'aimerais remercier Narjes, Quentin, Nadir et Thomas pour tous les moments inoubliables que l'on a passés, à répéter, pour ironiser sur les difficultés rencontrées, en plaisantant : "Faites une thèse, qu'ils disaient". J'aimerais remercier aussi les membres de l'équipe 2XS, avec qui j'ai passé plusieurs années dans une ambiance unique au monde et avec qui j'ai beaucoup appris. Je cite : Mickey, Thomas, Samuel, David, François, Florian et Etienne. Sans oublier Alex, avec qui c'était toujours un plaisir de déjeuner et de papoter.

Je tiens à remercier ma famille, plus particulièrement mes parents. Ils ont tout sacrifié, y compris des carrières florissantes, pour donner à mon frère et moi-même la chance de réaliser de grandes choses ici, en France. Je leur dédie cette réussite.

Je voulais aussi remercier une amie proche, Véronique, Avec qui la vie se transforme en aventure. Qui m'a soutenu durant ces trois années de thèse, qui m'a vu rire et douter, mais qui a toujours eu les bons réflexes pour me redonner du courage et le sourire pour faire face aux problèmes. Elle n'a jamais douté de moi, de mes capacités, de ce que je suis. À qui je dis simplement merci.

Évidemment, je ne peux pas oublier Dylan, Nicolas et Mickael. Une amitié qui dure déjà depuis plusieurs années.

Je remercie aussi Stéphane et Oliver pour nos escapades à moto et les bons moments partagés.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction Générale

Je ne blâme pas les gens pour les erreurs qu'ils commettent, mais pour qu'ils en assument les conséquences!

Jurassic Park, John Hammond

Sommaire

1.1 Contexte des objets connectés

Les premiers systèmes embarqués ont été développés dans les années 60, et la préoccupation principale de leurs concepteurs était la fiabilité et la sûreté de fonctionnement. Il était essentiel que ces systèmes garantissent le retour sain et sauf des astronautes. Les passagers ne devaient pas avoir peur de prendre l'avion, car celui-ci était sûr.

Avec l'avènement d'Internet et sa démocratisation, les systèmes embarqués sont devenus des objets connectés. Durant les années 2000 et 2010, leur nombre a explosé. En 2020, on en comptait déjà des milliards. Cette croissance s'explique par l'apparition de nouveaux besoins, mais aussi par la simplicité de développement.

Cette explosion d'objets connectés ou IoT (Internet of Things) ne s'est pas limitée aux environnements non critiques. De nos jours, ce type de composants s'est répandu au sein d'environnements exigeants comme l'automobile. Ils permettent de rendre le véhicule plus intelligent dans son fonctionnement, comme l'optimisation de la consommation de carburant avec des injecteurs contrôlés par ordinateur, remplaçant le carburateur qui débitait toujours la même quantité d'essence quelles que soient les conditions atmosphériques. Ils ont aussi fait leur apparition dans l'habitacle afin de proposer de nouveaux services aux passagers. Wi-Fi, streaming de musique ou de films, GPS intelligent, tous ces services impliquent des objets connectés. Malheureusement, une voiture de la marque Jeep présentait une faille dans le programme gérant la connectivité entre le smartphone et la voiture. Cette faille a permis à des attaquants de prendre le contrôle du véhicule à 15 km de distance. En 2015, un spécialiste de la sécurité informatique a réussi à accéder à des composants sensibles d'un avion depuis le siège passager. Ces deux exemples montrent que même dans des environnements critiques comme les transports, des failles peuvent exister et remettre en question toutes les problématiques de fiabilité et de sûreté.

Ces objets connectés intègrent des services de plus en plus complets. Ils permettent même à leurs utilisateurs d'installer de nouveaux services durant la durée de vie de l'objet. Certains de ces services peuvent provenir de nombreuses entreprises, dont certaines conçoivent des produits concurrents. Il existe de nombreux services de streaming musical, de VOD, d'assistants sportifs. Si deux services concurrents fonctionnent au sein d'un même objet dépourvu de mécanisme de sécurité, il n'est pas impossible que l'un des deux services cherche à nuire à son concurrent afin de gagner la confiance du client final, au détriment de l'autre.

Un autre avantage des objets connectés est leur capacité à travailler en groupe. Ce réseau d'objets permet de répondre à de nombreuses problématiques. Par exemple, on peut utiliser un réseau d'objets connectés entre eux au sein d'un champ agricole afin de mettre en place une surveillance intelligente (pluie, vent, lumière...) et, ainsi, optimiser la production et la gestion du champ. Toutefois, si l'un de ces objets est défaillant ou présente un comportement inattendu, sa capacité d'interaction avec les autres objets peut permettre la propagation de cette défaillance et entraîner une dégradation de service ou un déni de service total dans tout le groupe.

La problématique de fiabilité ainsi que de sûreté de fonctionnement est étroitement liée au concept de temps réel. Cependant, dans un contexte connecté, les objets dotés de propriétés temps réel peuvent être sujets à des détournements ou des attaques qui remettent en question toutes les garanties mises en place.

Tous ces cas d'usage démontrent un manque de sécurité au sein de ces objets, mais aussi un manque de confiance entre les acteurs impliqués dans ces objets.

1.2 Problématique

Dans cette section, nous mettons en avant les différentes problématiques auxquelles nous souhaitons apporter une réponse.

Diversité et responsabilité des acteurs au sein d'un objet connecté Un des problèmes que nous soulevons dans cette thèse est la diversité grandissante des acteurs participant à la fabrication et à la gestion d'un objet connecté. À l'origine, les systèmes embarqués étaient développés par une seule entité, une même entreprise. Cela créait un climat de confiance optimal. Cependant, avec leur connectivité et leur complexification, motivées aussi par une réduction des coûts de fabrication, de nombreux acteurs et entreprises peuvent désormais contribuer à la mise en œuvre de l'objet. Cela a érodé la confiance initiale quant aux résultats du développement et au fonctionnement. En cas de défaillance, certains objets ne comportent aucun mécanisme permettant d'identifier la source du problème. Est-ce le composant gérant la carte réseau ? La faille provient-elle de l'application musicale ? Il devient difficile de répondre à de telles questions, et bien souvent, le responsable désigné est celui qui est à l'origine de l'objet.

Dans cette thèse, nous souhaitons proposer une hiérarchie logicielle qui serait le reflet de la hiérarchie contractuelle mise en place pour le développement d'un objet connecté. Le but de cette hiérarchie est de spécifier explicitement les rôles de chaque acteur au sein de l'objet, mais aussi ses droits afin de permettre leur mise en responsabilité en cas de défaillance.

Confiance entre acteurs Une autre problématique des objets connectés est la confiance entre des acteurs proposant des services au sein d'un même objet. Nous souhaitons proposer à ces acteurs des environnements permettant à chacun de fonctionner sans risque de subir un piratage industriel de la part des concurrents qui pourrait mettre à mal le service proposé.

Problématique de sécurité des systèmes temps réel Un des points qui nous intéresse également concerne les objets connectés temps réel. Nous constatons de nombreux problèmes de sécurité au sein de ces objets. Nous souhaitons donc proposer une construction de ces systèmes à travers un descriptif détaillé d'une méthode permettant de garantir les propriétés d'isolation temporelle avec un mécanisme d'isolation spatiale formellement prouvé.

1.3 Contexte technologique

Les travaux de cette thèse ont été réalisés au sein de l'équipe [2xs] du laboratoire Cristal de l'université de Lille. La sécurité dans les systèmes et environnements embarqués contraints constitue le principal axe de recherche de l'équipe. La thèse est principalement financée par le projet européen CELTIC+/ODSI [ODS] (On Demand Secure Isolation). L'objectif de ce projet est de concevoir et de développer une nouvelle plateforme avec de fortes garanties de sécurité. Cette plateforme est destinée à un contexte de systèmes embarqués et d'objets connectés, impliquant divers acteurs, de sa conception à sa gestion.

En plus de l'université de Lille, plusieurs partenaires industriels et académiques interviennent au sein de ce projet : Orange SA, Prove and Run, Internet Of Trust pour la France,

Nextel pour l'Espagne, Resonate MP4 pour la Roumanie, etc. L'équipe 2XS a travaillé sur la conception, le développement et la vérification d'un noyau d'isolation mémoire utilisé par la plateforme. La conception de l'architecture présentée dans cette thèse a bénéficié de l'expérience et de la collaboration avec des ingénieurs d'Orange SA, entre autres.

1.4 Présentation du manuscrit

Le document de thèse est organisé comme suit :

État de l'art Dans ce chapitre, nous proposons une vision des systèmes embarqués et des objets connectés tout au long des 60 dernières années, avec l'identification des évolutions et des nouvelles problématiques qui en découlent.

Problématique de la sécurité des systèmes embarqués Après avoir établi le contexte des systèmes embarqués, nous identifions les problématiques concrètes qui nous intéressent et auxquelles nous souhaitons apporter une réponse.

Architecture et implémentation de l'écosystème embarqué et connecté Dans ce chapitre, nous proposons des réponses aux problématiques identifiées à travers une architecture que nous présentons. Nous proposons également une méthode illustrant le maintien des garanties temporelles au sein d'un système à criticité mixte.

Mise à l'épreuve Dans ce chapitre, nous réalisons des mesures de performances sur l'implémentation des mécanismes nécessaires pour notre architecture à travers des micro- et des macro-tests. Nous examinons aussi l'impact de l'isolation mémoire sur un système temps réel.

Conclusion La conclusion propose un bilan du travail réalisé durant la thèse, les perspectives d'évolution de l'architecture, mais aussi un avis plus personnel sur les objets connectés, leur développement et leur future orientation.

1.5 Références

[2xs] Equipe 2XS - laboratoire CRIStAL de l'université de Lille. 4

[ODS] Project ODSI – Celtic+. 4

Chapitre 2

État de l'art

Faut voir grand dans la vie, quitte à voyager à travers le temps au volant d'une voiture, autant en choisir une qui ait d'la gueule !

Retour vers le futur, Dr. Emmett Brown.

Sommaire

2.1 Introduction

Les systèmes embarqués modernes ont grandement évolué durant les 60 dernières années. Ils se sont aussi répandus massivement dans nos environnements. Dans ce chapitre, dans un premier temps, nous allons voir l'évolution matérielle et logicielle des systèmes embarqués. Puis, nous allons identifier de la manière la plus exhaustive les acteurs et leurs enjeux au sein de ces systèmes. Pour finir, nous verrons quels mécanismes peuvent être mis en pratique afin de leur apporter des garanties de sécurité et de sûreté.

2.2 Historique des systèmes embarqués

Dans cette section, nous allons dépeindre l'évolution des systèmes embarqués, depuis les systèmes conçus pour la conquête spatiale, jusqu'aux objets connectés dans nos domiciles.

2.2.1 Les systèmes monolithiques 3 pages

Logiciels embarqués

Un système embarqué est défini non pas par la question "qu'est-ce qu'un système embarqué?" mais plutôt par la question "qu'est-ce qui fait qu'un système est embarqué?".

L'environnement est la principale caractéristique qui définit qu'un système est embarqué [Kün87]. L'environnement de déploiement impose des contraintes sur le développement, le coût et la taille. Il est cependant possible de déterminer des dénominateurs communs aux systèmes embarqués. La criticité de l'environnement de déploiement influe sur le niveau de fiabilité et de sûreté. De plus, lorsque le système sera dans un environnement non maîtrisé totalement, la sécurité devient l'un des objectifs du système.

Il est évident qu'après l'apparition des premiers systèmes informatisés, le besoin d'intégrer des composants de calcul, de contrôle et d'automatisation dans des environnements contraignants est devenu évident. Les systèmes embarqués tels que définis plus haut sont apparus avec les premières capsules spatiales de la NASA ou encore dans le cadre du programme Apollo[TH66]. Ces systèmes avaient des contraintes environnementales. Étant donné qu'ils devaient être emportés au sein des capsules, quatre grands axes de conception ont guidé les développeurs[LM66] :

— La masse, chaque kilogramme supplémentaire complexifie le décollage et la navigation ;

— Le volume, étant donné la petitesse des capsules ;

— La consommation d'énergie, les capsules fonctionnant sur batterie la plupart du temps ;

— La fiabilité, chaque erreur ou problème à distance peut être insoluble, mettant en danger la vie des astronautes. Avec de fortes contraintes temporelles.

De plus, les composants logiciels sont intimement liés au matériel. Chaque composant répond à la gestion d'un composant matériel. Ils sont définis et implémentés sur Terre dans un laboratoire. Puis, dans les années 70, sont apparus les premiers microprocesseurs [KA03] chez Intel, utilisés très vite dans le domaine de l'embarqué, notamment dans l'aviation.

Le secteur du spatial, exploration ou satellite, en pointe au niveau de la technologie de l'époque, a vu ses innovations être reprises dans d'autres domaines plus terrestres. Dans

les années 1960 et surtout 1970, le département de la défense américaine [MO78] [Fis78] [BDC84] [Ili81] a dépensé énormément dans l'informatisation de l'armement. Ils ont distingué deux types de systèmes, les systèmes embarqués et les systèmes de traitement de données. Dans le cadre des systèmes embarqués (guidage, missiles, surveillance, communication), la question du cycle de vie et des processus de développement étaient dans les préoccupations. Ce processus fait intervenir plusieurs acteurs, techniques, administratifs ou politiques. La fiabilité et la sûreté de fonctionnement sont au cœur du développement de ces systèmes liés à l'armement.

Le développement des systèmes embarqués comprend autant le développement de la partie matérielle, logicielle, mais aussi la maintenance[DGS81]. Dans leurs prémices, la majorité des ressources de développement sont allouées à la partie matérielle. Avec les années, les ressources se sont petit à petit transmises aux composants logiciels et à sa maintenance.

Plus tard, dans les années 80[PT89], les systèmes embarqués sont arrivés dans le civil et le grand public avec leur propagation dans différents domaines, comme l'aviation civile ou l'automobile avec l'ABS électronique.

Dans le cadre du développement de systèmes embarqués, nous pouvons distinguer deux grandes situations, déterminées par l'objectif même du système.

Dans le premier cas, le système sera utilisé dans un environnement critique, ce qui implique d'allouer plus de ressources budgétaires, humaines et techniques, avant et après le déploiement. Ce type de système est complexe et coûteux en matériel et logiciel, de ce fait, il ne sera pas produit en grande quantité avec un effort de maintenance post-déploiement.

Le second cas concerne les systèmes destinés à des environnements aucunement critiques. Ce type de développement et d'investissement est minimal. Peu de ressources sont allouées, ce système sera produit en grande quantité. De ce fait, un pourcentage de défaillances logicielles ou matérielles est toléré.

Quelle que soit la criticité, la destination du système ou son coût, le point commun entre ces systèmes est le fait qu'ils soient fermés. Une fois conçu, fabriqué, déployé, et même durant sa maintenance, le système continue de suivre la spécification mise en place au départ ou celle mise à jour suite à la maintenance.

Concernant leurs structures, les systèmes embarqués ont évolué, quel que soit leur domaine d'application. Comme dit précédemment, les systèmes étaient très étroitement liés au matériel. Dans un premier temps, ces environnements embarqués comportent des composants conçus et fabriqués spécifiquement pour un contexte précis. Nous sommes en présence de logiciels embarqués.

L'utilisation de microprocesseurs génériques a fait passer le travail des ingénieurs du développement de logiciels pour l'embarqué au développement de systèmes plus complexes et plus complets. Le matériel devenant plus générique, les fonctionnalités liées à la fiabilité, à la sûreté ou à la disponibilité ont été transférées au niveau logiciel.

De ce fait, au lieu de simples composants logiciels interagissant avec le matériel, des systèmes d'exploitation sont mis en place.

Dans des environnements critiques, avec de forts besoins de fiabilité, des systèmes temps-réel sont utilisés. Les systèmes embarqués critiques deviennent à partir de ces moments-là intimement liés aux systèmes temps-réel. Les premiers, comme vus précédemment, faisaient partie des missions spatiales[SH68].

Les systèmes temps-réels embarqués suivent le même paradigme [CMMS79] [PCPC75]. Qu'ils soient de type système temps-réel strict à fort niveau de criticité (système de guidage aéronautique) ou à faible niveau (système de gestion d'un lave-linge), ils sont en

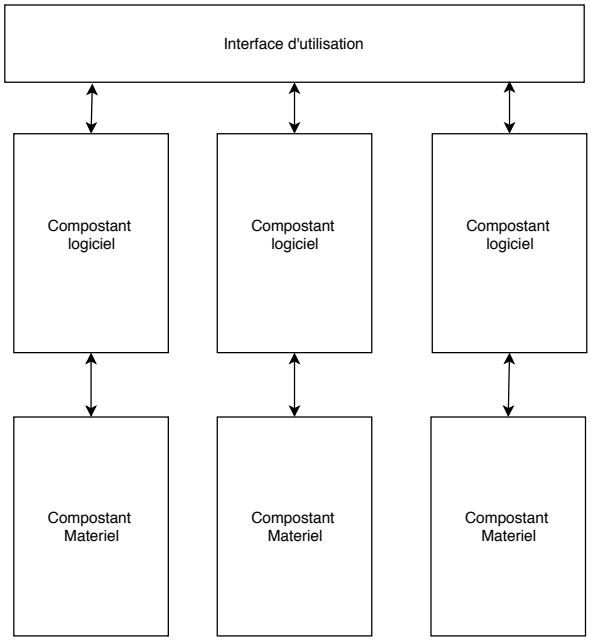

FIGURE 2.1 – Architecture primaire d'un système embarqué

charge de garantir à différents niveaux les propriétés suivantes[PD93] :

— Tolérance aux fautes : Suivant la criticité de l'environnement de déploiement, il est acceptable d'avoir un certain niveau d'erreurs ;

— Sûreté de fonctionnement ;

— Fiabilité.

Cependant, peu importe l'environnement et sa criticité, la structure de ce type de système embarqué reste identique et suit le schéma décrit dans la figure 2.2

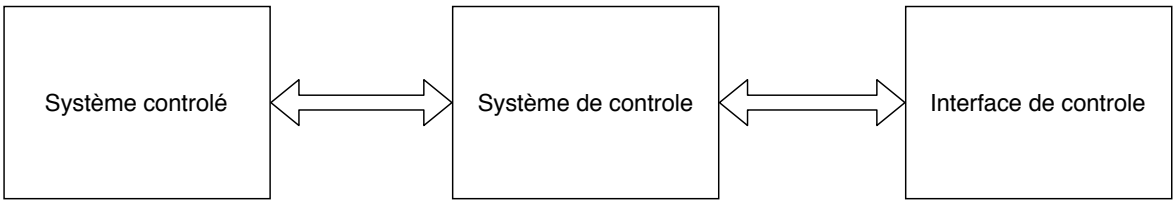

FIGURE 2.2 – Architecture d'un système temps-réel

Le système de contrôle contient des tâches applicatives destinées à gérer les ressources (périphériques, ressources logicielles). L'interface de contrôle est utilisée pour monitorer et gérer le système temps-réel.

Au sein de la partie système de contrôle, les tâches applicatives qui sont développées sont réalisées suivant une spécification déterminée en amont du déploiement [Sta96], suivant un objectif précis. Chaque ajout ou modification apporté au système ou ses tâches peut compromettre les objectifs de fiabilité et de sûreté de fonctionnement du système.

Fournisseur, intégrateur et cycle de vie

Quand nous nous intéressons plus particulièrement au cycle de développement et aux cycles de vie des systèmes embarqués[Koo] [DMG97] [KKH09] [SST89], nous distinguons un schéma redondant dans le processus.

Le développement se découpe en plusieurs phases :

1. Émergence d'une problématique : Un nouveau besoin est né et un système embarqué est nécessaire afin de résoudre cette problématique ;
2. Concept et spécification de la solution : Ingénieurs et clients définissent ensemble le cadre global et le cahier des charges du système embarqué ;
3. Codesign matériel/logiciel du système embarqué : À l'aide des ressources mises à disposition par la spécification précédente, une architecture logicielle et matérielle est mise en place ;
4. Développement du système embarqué : Cette phase consiste à créer et à mettre en place les différents composants logiciels et matériels ;
5. Production d'un prototype : Un prototype du système embarqué dans un environnement similaire à une situation réelle est mis en place ;
6. Test et validation : Une série de tests et de validations du système embarqué sont exécutés ;
7. Déploiement : Le système embarqué est déployé dans l'environnement de destination ;
8. Mise à jour et maintenance : Suivant une mise à jour de la spécification ou la détection de défaillance, des actions sont menées si le système ou l'environnement le permettent ;
9. Fin de vie du système : Le système a terminé la mission pour laquelle il a été conçu.

Chacune des étapes n'est pas unique, elles peuvent être réitérées autant de fois que nécessaire pour atteindre l'objectif escompté (p. ex. boucle sur le développement, la production ou les tests, jusqu'à la validation du système).

Spécification et responsabilité

Quand nous parlons de contrat de commande d'un système ou de cahier des charges, nous nous positionnons clairement dans une collaboration entre une entité qui a un besoin et au moins une autre qui détient des compétences. Cette collaboration peut prendre plusieurs formes. La principale propriété qui caractérise cette collaboration est la localisation géographique.

Dans le premier cas, toutes les entités font partie de la même entreprise. C'est principalement le cas de gros groupes qui ont un panel de compétences suffisant pour répondre à leurs besoins.

Le second cas concerne des entités n'ayant pas toutes les compétences pour répondre à leur problématique. De ce fait, deux situations peuvent survenir. La première fait appel à une tierce entité qui détient toutes les compétences nécessaires pour répondre à la problématique. La seconde situation fait appel à un groupe d'entités pour répondre à différents sous-problèmes, puis agglomérer les résultats pour finaliser la solution recherchée.

Cependant, dans une collaboration entre plusieurs entités, différents problèmes peuvent être rencontrés [HM01] :

— Partage des tâches ;
— Méconnaissance des acteurs entre eux ;
— Problèmes de communication entre acteurs durant le développement ;
— Mauvais partage de connaissance et de documentation sur les composants du développement ;

— Gestion globale du projet et synchronisation ;

— Différence de niveau technique entre les entités.

Pour pouvoir combler ces difficultés, il existe des méthodes permettant d'y répondre ou d'atténuer leur impact. Le modèle de développement à l'aide du modèle en V [Deu13] ou des techniques agiles [RMOT12].

Cependant, malgré ces solutions de gestion, dans un but de réduction de coût ou de vitesse de production, des difficultés, parfois contractuelles ou légales, peuvent survenir. Ces risques peuvent prendre différentes formes [EMJ08] [EKP16], allant de l'échec complet de production du produit au développement ne répondant pas aux qualités escomptées.

2.2.2 Systèmes ouverts 3 pages

Apparition de nouveaux besoins

Les systèmes embarqués comme vus précédemment étaient conçus et développés pour répondre à une problématique dans un contexte assez précis et un environnement connu en amont du développement. Cette problématique était résolue avec une association logicielle et matérielle. Et très souvent le contexte devait comporter des propriétés temps-réel. Ces systèmes embarqués étaient des boîtes fermées[Lee08].

Cependant, avec l'évolution des environnements et des besoins, la possibilité de modifier les systèmes embarqués après leur déploiement, sans accès physique direct, mais via un réseau, est devenue nécessaire (Internet ou autre) [KNP97]. Les systèmes sont passés d'environnements clos et centralisés à un contexte ouvert et distribué.

On peut résumer les buts de ce type de systèmes en points :

— Extensibilité : Ajouter ou supprimer des composants logiciels pour répondre à des nouveaux besoins ;

— Réutilisabilité : Le système n'est plus dédié à un contexte précis ou doit être déployé dans des environnements quasi similaires ;

— Adaptabilité : Le système s'adapte aux contraintes du contexte de manière la plus précise possible ;

— Qualité de service : une évolutivité en matière de logiciel ne doit pas impliquer une baisse de qualité de service.

Quand le système comporte des propriétés temps-réel, de sécurité ou de fiabilité, son ouverture et son adaptation à ce mode de fonctionnement doivent toujours garantir les propriétés inhérentes (garanties temporelles, disponibilité, mécanismes de sécurité…) malgré l'ajout durant leur fonctionnement de nouveaux composants logiciels [Kop00].

L'ouverture des systèmes n'impacte pas seulement tel ou tel secteur d'activité. Cette volonté d'ouverture se retrouve dans les transports (automobile, aéronautique…), mais aussi dans des environnements industriels. Souvent, l'envie de les rendre modifiables et à distance concerne des environnements de systèmes embarqués distribués [CCM+07] (p. ex. capteurs, systèmes de communication). Cependant, durant les années 1990 et surtout dans les années 2000, l'utilisation de systèmes embarqués a explosé [MY11], avec leur utilisation dans un contexte grand public avec de la domotique notamment, en utilisant les connexions Internet ou tout autre (GSM, Bluetooth…) [YKT+06] [AB05].

Depuis plusieurs années déjà, les systèmes (serveurs, ordinateur de bureau) sont connectés en réseau pour fournir des services (Web, serveur mail…). La continuité était aussi de mettre les systèmes embarqués en réseau.

Les systèmes génériques

Nous avons vu précédemment que la mise en place de systèmes embarqués distribués et/ou ouverts fut motivée par l'apparition de nouveaux besoins. Ces nouveaux besoins ont impliqué l'arrivée de nouveaux acteurs sur le marché. L'ensemble logiciel-matériel d'un système embarqué devenant plus complexe, leurs développements ne sont plus le fait d'une seule entreprise, mais regroupent plusieurs acteurs, chacun proposant une partie du produit final. [GLT03]

Le fait de voir le système embarqué comme un produit commercial implique toute la mécanique connue comme la réduction de coûts de production et l'augmentation de marges. Pour comprendre la généricité des systèmes embarqués, comparons-la à la vente d'une voiture.

Au début de l'automobile, il existait un constructeur qui fabriquait la voiture et tous ses composants. Aujourd'hui, il est devenu inconcevable, étant donné leur complexité, que seul le constructeur fabrique la voiture. Il fait appel à des sous-traitants. Ces sous-traitants fabriquent des composants qui sont revendus à différents constructeurs. Le même système de freinage peut alors se retrouver au sein de différentes marques de véhicules.

Pour les systèmes embarqués, tous les composants (matériels ou logiciels) peuvent être conçus par différents sous-traitants. Chacun proposant une fonctionnalité. De ce fait, le système embarqué final n'est qu'une composition de systèmes génériques.

D'un point de vue purement technique, les systèmes embarqués qui se veulent génériques adoptent les mêmes philosophies que les ordinateurs traditionnels.

Un matériel générique Très tôt, les développeurs, délaissant les circuits "faits maison", ont eu recours à l'utilisation de processeurs (et plus généralement, de composants matériels) issus d'entités externes [PCL+96]. Des composants qui deviennent des briques pour le produit final. Les mêmes composants ou familles de composants utilisés se retrouvent dans différents domaines [SBH16].

Avec des FPGA reconfigurables[HVJ14] ou des processeurs plus complexes à usages multiples (RISC-V, ARM notamment), il devient plus facile de proposer un matériel générique, mais adapté à toutes les situations.

Systèmes d'exploitation génériques Un système embarqué ne nécessite pas nécessairement un système d'exploitation pour fonctionner. Cependant, comme expliqué par les motivations précédemment, il est devenu nécessaire de passer par l'utilisation d'un système. S'il comporte des propriétés très précises (disponibilité, temps-réel, sûreté...), le système d'exploitation peut offrir les mêmes garanties.

Que ce soit dans le domaine de la recherche académique liée aux systèmes embarqués ou dans l'industrie, les mêmes systèmes sont récurrents. Cependant, suivant la criticité de l'environnement de déploiement, certains acteurs ont une expertise et proposent de ce fait des logiciels adaptés aux prérequis des normes[MT03][HVJ14].

De plus, avec l'évolution du matériel utilisé, des méthodes de virtualisation de ressources sont utilisées pour faciliter le développement et le déploiement d'applications au sein du système.

On retrouve dans le schéma 2.3 une liste non exhaustive de systèmes utilisés dans le domaine embarqué[HBPT16][GZ12].

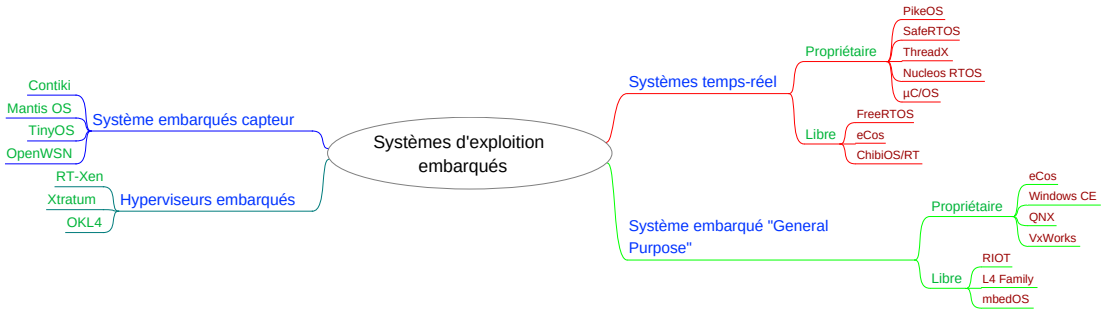

FIGURE 2.3 – Vue non exhaustive de système d'exploitation embarqués

Groupement de systèmes embarqués connectés

Très vite, avec le développement de systèmes embarqués qui sont connectés, l'idée de coopération entre ces systèmes permet de développer de nouveaux contextes d'utilisation, comme le *Fog Computing*[DB16]. On peut voir deux cas d'utilisation de systèmes embarqués en coopération.

Architecture hiérarchique Dans ce cas, un ensemble de systèmes embarqués (capteurs, caméras de surveillance, système de communication) est mis en place et est géré depuis une unité qui centralise les décisions et les ressources.

Architecture distribuée Nous pouvons aussi mettre en place une coopération directe entre des systèmes embarqués. Cette coopération peut se faire avec des systèmes embarqués de la même famille et de la même manufacture, ou alors à travers des protocoles standardisés.

2.3 Écosystème dans un système générique 4 pages

Dans la section précédente, nous avons pu entrevoir un historique et une vue générale des systèmes embarqués, depuis leurs débuts dans les années 60 jusqu'à leur démocratisation et leur utilisation dans des contextes et des environnements très hétérogènes. Nous avons aussi vu la démultiplication des acteurs collaborant au sein de ces systèmes.

2.3.1 Les acteurs de l'écosystème

Dans l'analyse que nous avons faite précédemment, nous avons constaté que les systèmes embarqués sont passés d'un développement interne aux structures à une association de plusieurs acteurs ayant chacun des compétences différentes. De plus, leur multiplication et leur usage dans différents contextes rendent les environnements développés très hétérogènes, ayant un fort niveau de concurrence dans chaque domaine de développement.

Dans cette section, nous allons identifier clairement des rôles[MEL10] trouvés au sein de l'historique. Nous pouvons distinguer l'intégrateur, les fournisseurs de matériels, de logiciels pré et post-émission, le déployeur ainsi que l'utilisateur. Chaque rôle possède un certain nombre de tâches. Chaque acteur peut prétendre à plusieurs rôles.

Intégrateur ou initiateur de l'objet

Est désigné comme intégrateur le rôle d'identifier les besoins liés à un contexte de fonctionnement et de mettre en place un cahier des charges pour le développement du système embarqué. De plus, il doit trouver des acteurs pour la conception, la fabrication et le déploiement de l'objet.

Étant à l'initiative du projet, il détient toutes les capacités possibles au sein du système jusqu'à son déploiement au moins.

Il a surtout une tâche de supervision durant la phase de développement. À l'issue du développement, il doit vérifier que les spécifications de départ et le cahier des charges ont été respectés dans l'objet développé.

Il est l'interface entre tous les acteurs travaillant sur l'objet en amont du déploiement, mais aussi en aval, car il peut être tenu comme responsable de l'objet par les utilisateurs.

Fournisseur de matériel

Ce rôle consiste à proposer et à fabriquer une solution matérielle répondant aux besoins définis par l'intégrateur.

Suivant la complexité matérielle du système embarqué, le fournisseur de matériel peut être un ensemble de fabricants. L'intégrateur désigne le ou les fabricants suivant leur expertise et le coût de production des composants matériels.

De plus, le fournisseur de matériel doit aussi travailler en collaboration avec le fournisseur du logiciel interne ou du système d'exploitation pour être en accord avec les besoins logiciels. Encore une fois, cette collaboration est sous la gestion de l'intégrateur.

Le ou les fabricants de matériels ont un rôle assez particulier. Ils supervisent la couche la plus basse du système embarqué. Cependant, intervenant au tout début du développement, leur impact sur l'objet est assez limité sachant qu'une fois le produit physique terminé et que la spécification a été respectée, ils ne doivent plus intervenir.

S'ils doivent intervenir plus tard dans le cycle de vie de l'objet, cela doit se faire toujours suivant les spécifications fournies par l'intégrateur.

Fournisseur de logiciel interne

Le fournisseur de logiciel désigne le ou les acteurs en charge de concevoir le programme ou le système d'exploitation du système embarqué.

Toujours suivant les spécifications de l'intégrateur, ils doivent fournir des services logiciels afin d'interagir avec le matériel, l'environnement du système ou avec d'autres composants logiciels.

À l'instar des fabricants de matériels, ils interviennent durant la phase de développement, en amont du déploiement utilisateur, mais aussi durant le cycle d'utilisation pour diverses actions de maintenance ou de mise à jour.

Le déployeur

Une fois tous les composants logiciels et matériels conçus et fabriqués, le rôle du déployeur est d'agréger tous les composants logiciels en un produit fini. Il supervise l'émission de l'objet, sa mise à disposition pour l'utilisateur.

Son intervention au sein du cycle de vie de l'objet est assez courte. Il n'intervient qu'une seule fois et représente l'intermédiaire entre la phase de développement et la phase d'utilisation.

De plus, l'objet doit être dans un état de fonctionnement minimal, permettant son utilisation par l'utilisateur final ou son extension au besoin si celui-ci le permet.

Jusqu'à la mise à disposition pour l'utilisateur, le déployeur a tous les droits au sein du système. Cependant, il doit fournir à l'utilisateur le moyen de supprimer les droits du déployeur.

Fournisseur de fonctionnalités

Le fournisseur de fonctionnalités intervient après l'émission de l'objet, durant la phase d'utilisation du système. Il produit des composants logiciels qui peuvent être installés par l'utilisateur.

Il connaît uniquement les spécifications techniques et environnementales de l'objet afin de proposer des fonctionnalités adaptées. Il intervient de manière aléatoire durant la vie de l'objet. Il n'a aucune action possible directement au sein de l'objet. Il intervient toujours sous l'égide de l'utilisateur ou du déployeur. Il n'a de ce fait aucun droit spécifique au sein de l'objet. Il ne fait que proposer une fonctionnalité qui sera ajoutée au sein du système via des mécanismes internes.

Cependant, il peut garder des droits d'accès et de modification à l'intérieur des composants logiciels qu'il fournit.

L'utilisateur

Nous pouvons distinguer plusieurs types d'utilisateurs d'un système embarqué. L'utilisateur final de l'objet peut être un utilisateur humain, un ensemble d'utilisateurs humains, une autre machine ou un ensemble de machines.

Quels que soient leurs types, les utilisateurs profitent des fonctionnalités du système embarqué pour obtenir des résultats. Et si le système le permet, ils peuvent y intégrer ou supprimer des fonctionnalités.

On peut aussi distinguer deux sous-genres d'utilisateur : un administrateur et un utilisateur simple.

L'administrateur doit gérer l'architecture logicielle ou matérielle du système, mettre à jour l'ensemble et ajouter ou supprimer des fonctionnalités. Il se doit alors de posséder tous les droits nécessaires à ces actions au sein du système.

L'utilisateur simple ne fait que profiter des fonctionnalités proposées par le système embarqué. Et ses droits se limitent à l'espace des fonctionnalités uniquement.

2.4 Sécurité des systèmes embarqués

2.4.1 Objet de la sécurité 3 pages

Quelle que soit l'époque à laquelle le système embarqué fut développé, il était nécessaire d'obtenir et de garantir au cours du temps diverses propriétés, suivant le contexte de fonctionnement du système. Ces propriétés incluent la sûreté de fonctionnement, la disponibilité, la fiabilité, ainsi que la sécurité du système et des données.

Il est à noter que ces différentes propriétés sont parfois mélangées ou confondues. Nous allons donc nous efforcer de donner une définition claire de ces propriétés qui nous intéressent.[Ste91].

Sûreté de fonctionnement

La sûreté de fonctionnement est la propriété la plus difficile à cerner au sein d'un système. Ce concept est intimement lié à des environnements de fonctionnement avec un haut niveau de criticité (aviation, automobile...). Nous parlons de sûreté quand il y a une probabilité non négligeable de risques, d'accidents[Bow93][LW09].

Les risques peuvent prendre plusieurs formes (ISO 31004). Ces risques peuvent provenir d'une défaillance d'un composant matériel ou logiciel. Une défaillance interne au système peut amener ce dernier à ne plus fonctionner ou à fournir un traitement erroné. Le lieu de fonctionnement du système peut être un facteur de risques. Ils peuvent survenir à cause des conditions atmosphériques, thermiques ou autres. S'il y a eu une méconnaissance de l'environnement de déploiement lors du développement, cela peut provoquer un comportement inattendu du système durant son fonctionnement. Si nous nous intéressons au groupement de systèmes embarqués (instruments de mesure, sondes...), le risque est d'autant plus grand, car en plus d'une sûreté de fonctionnement interne aux objets, il est nécessaire de prendre en considération une sûreté de fonctionnement globale à ce groupement. Chaque incident dans un système peut impacter tout l'ensemble.

Il est cependant possible de pallier ces difficultés, de proposer une gestion de risques. Cette gestion dépend de l'impact et des pertes engendrés par ces risques, humains ou matériels. La sûreté pouvant être vue comme un problème de fiabilité.

La question de la sûreté de fonctionnement doit être traitée dès la spécification de l'objet embarqué. Il faut que lors du développement du système, les ingénieurs aient une entière connaissance des conditions sources de défaillance. De plus, les ingénieurs peuvent mettre en place des mécanismes de confinement d'erreur et de redondance afin de diminuer l'impact de ces risques. Il est aussi possible de réaliser des tests ou des vérifications du code des systèmes pour certifier du bon fonctionnement des composants du système.

Si le système embarqué développé n'a pas de mécanismes de sécurité en son sein, il peut s'avérer aisé de mettre à mal cette sûreté de fonctionnement, surtout si le système embarqué est connecté[KPCBH15]. Par conséquent, les systèmes embarqués doivent intégrer durant leur développement les deux concepts[BVMG12].

Disponibilité

C'est lors d'un partage de ressources au cours du temps qu'on parle de disponibilité. Ces ressources peuvent prendre plusieurs formes (temps de calcul, mémoire, accès réseau...). La difficulté de gérer le multiplexage de ressources est d'autant plus importante si le système est critique, notamment dans le domaine des transports. Une mauvaise gestion du partage pouvant entraîner des conséquences graves.

On retrouve aussi la problématique de disponibilité dans les systèmes embarqués temps-réel. La notion de pire temps d'exécution et ses méthodes d'estimation sont les préoccupations du développement de ces systèmes[AHQ+15]. En plus de subir des évolutions matérielles (cœurs hétérogènes ou multicœurs) qui complexifient le travail de l'ingénierie, la sûreté de fonctionnement et la fiabilité du système sont liées à une bonne estimation du temps d'exécution et au respect des plans d'ordonnancement mis en place. Dans les systèmes ouverts ou extensibles après émission, des problématiques de sécurité peuvent aussi compromettre ces fonctionnalités[Jon06].

Isolation

L'isolation consiste à partitionner une ressource entre différents composants logiciels ou matériels. L'isolation peut prendre deux formes : l'isolation spatiale et l'isolation temporelle [Rus00].

— Isolation spatiale : L'isolation ou le partitionnement spatial est défini comme le partage de la mémoire et le respect des droits d'accès.
— Isolation temporelle : L'isolation temporelle est définie comme le partage du temps d'exécution suivant un plan d'ordonnancement établi.

L'isolation et le partitionnement ont comme objectif de maintenir des propriétés au système (sûreté de fonctionnement, intégrité de données, disponibilité et fiabilité...) [PDK+15].

Dans les premiers systèmes embarqués qui n'étaient pas extensibles après émission, il n'était pas nécessaire de penser à mettre en place de l'isolation. Cependant, l'ouverture de l'objet ou l'apport de composants post-émission peuvent être sources de défauts de sûreté ou de sécurité. Apporter des mécanismes d'isolation permet de maintenir les propriétés énoncées par la spécification de l'objet.

Parmi ces mécanismes, nous trouvons :

Virtualisation Le terme de virtualisation peut être utilisé pour définir différents types de solutions logicielles ou matérielles permettant de fournir une instance virtuelle d'un composant, sans nécessairement avoir une existence physique. Appelés hyperviseurs ou gestionnaires de machines virtuelles, ces logiciels sont en charge de fournir à des systèmes ou des logiciels des environnements virtuels[PZP15][CLS+08]. Il faut distinguer deux types de virtualisation avec des hyperviseurs : une virtualisation totale ("Full-virtualization" en anglais) et la paravirtualisation. Dans le cadre de la virtualisation totale, le système ou le logiciel invité est dans un environnement totalement virtuel. Pour la paravirtualisation, il s'agit de virtualiser certains composants du système invité. Initialement utilisés dans des environnements dotés de ressources importantes comme les serveurs, les hyperviseurs sont aussi présents dans le domaine embarqué[Hei08][SVLN13].

Protection mémoire Dans un système, il peut être désirable de pouvoir isoler les espaces d'adressage de chaque processus ou application. Pour ce faire, il existe deux solutions majeures : les MMU (*Memory Management Unit*) et les MPU (*Memory Protection Unit*)[HRK12].

Dans le premier cas, chaque processus a une vision sur un espace d'adressage virtuel qui lui est propre, basé sur un espace d'adressage physique. Ceci étant facilité par un composant matériel en charge de traduire les adresses virtuelles en adresses physiques. Chaque adresse physique correspond au plus à une adresse virtuelle.

Dans le second cas, il n'y a que de la mémoire physique. Il s'agit de gérer l'accès aux pages via des flags. Chaque processus non privilégié ne peut accéder qu'à une région de la mémoire qui lui est dédiée. Ce type de matériel est souvent présent dans des systèmes embarqués avec très peu de ressources.

Isolation des périphériques Dans des systèmes d'exploitation, il est important pour un système d'isoler le matériel et son accès. Il existe deux types d'accès au matériel : via les mécanismes spécifiques au processeur (I/O port chez Intel par exemple) et via la MMIO.

Si nous souhaitons accéder au matériel via la première méthode, nous pouvons utiliser des appels à des fonctions fournies par le noyau du système. Ce qui permet de restreindre l'accès.

Dans le second cas, le matériel est adressé au sein d'un espace mémoire qui lui est dédié. Pour écrire ou lire depuis un matériel, il faut lire et écrire dans une adresse mémoire spécifique. Le contrôle se fait via des flags d'accès des adresses spécifiques au matériel.

Cependant, pour des raisons de performance, les composants matériels ont un accès direct à la mémoire physique sans restriction. C'est ce qu'on appelle la DMA (*Direct Memory Access*). Dans certaines failles de sécurité, il est possible de contourner les mécanismes de protection pour écrire ou lire dans des espaces d'adressages [PDK+15], normalement inaccessibles. Pour contrer ce type d'attaque, certains composants matériels proposent ce qu'on appelle une I/O MMU. Ce mécanisme permet de restreindre les accès DMA à certains espaces d'adressage uniquement.

Isolation et Multiplexage temporel Dans un système temps-réel[HKM+12][BD17], composé de tâches, le partitionnement temporel désigne l'utilisation exclusive d'une ressource par une seule tâche et uniquement une tâche à un instant t de la durée de vie du système.

Il existe deux types de systèmes temps-réel :

Système temps-réel souple : La fiabilité du système embarqué n'est pas remise en question si une tâche ne se termine pas à son échéance.

Système temps-réel dur : La fiabilité du système embarqué est remise en question si une tâche ne se termine pas à son échéance.

Chaque tâche est définie par :

— Une période d'exécution

— Une échéance

— Un pire temps d'exécution (*Worst Case Execution Time ou WCET*)

— Un niveau de criticité

Deux situations peuvent se présenter dans le cadre des systèmes embarqués, lorsqu'il est ouvert ou fermé.

Dans le cadre des systèmes fermés, il faut estimer la durée maximale d'exécution de chaque tâche du système avant l'émission et un plan d'ordonnancement est mis en place. L'ordonnanceur du système est en charge uniquement de faire les changements de contexte selon le plan suivant les caractéristiques de la tâche. Dans ce type de système, l'isolation temporelle est garantie en amont de la période de fonctionnement.

Dans le cadre d'un système ouvert, la définition des tâches varie :

— Une période d'exécution

— Une échéance

— Un budget de temps nécessaire estimé pour l'exécution de la tâche

— Un niveau de criticité

Avec ces caractéristiques, les tâches peuvent être intégrées au système après émission. De ce fait, il n'est plus question d'estimation et de plan d'ordonnancement avant l'émission. Lorsqu'une tâche arrive dans un système, elle est intégrée avec un budget temps estimé, et son intégration est conditionnée au temps disponible au sein du système.

Si la tâche n'est pas rejetée (budget pas trop grand), l'ordonnanceur l'intègre au plan déjà existant. Si le budget alloué à la tâche est insuffisant, suivant l'implémentation de l'ordonnanceur, soit la tâche est arrêtée avant la fin de son processus, soit elle continue de s'exécuter, mais son dépassement sera déduit du budget de la prochaine période d'exécution. L'isolation des tâches déjà présentes est maintenue en empêchant toute nouvelle tâche d'accaparer trop de temps d'exécution.

2.4.2 Les moyens de la sécurité

Lors du développement de systèmes embarqués dans des environnements critiques, il est nécessaire de mettre en place des mécanismes de vérification, de validation et de certification pour garantir à leur utilisateur des propriétés affichées par le fournisseur.

L'importance des audits indépendants

La confiance à donner à un logiciel ou à un matériel dépend du nombre de garanties affichées. Ces garanties peuvent être produites de plusieurs manières : via des tests de fonctionnalités, d'intégration ou audits de sécurité ou encore des tests avec panel d'utilisateurs ou les *Test compliance Kit* (TCK).

Déjà, lors du développement du produit, il existe tout un ensemble de mécanismes pour tester les propriétés de sécurité ou de sûreté[HGZG12][Fra16]. Ces tests sont réalisés par les équipes de développements, et le produit n'est mis en production que si les bancs de tests sont concluants.

Mais la confiance peut être plus importante si ces tests et audits sont réalisés par d'autres entités que le fabricant ou l'utilisateur[CMFE09][AIH15]. La confiance est donnée à l'organisme de tests et d'audit. À l'image des autorités de certifications pour la sécurisation de communications, l'utilisateur accorde sa confiance à des entités ayant eu auparavant à faire leurs preuves. Les tests et audits suivent des techniques et des méthodes connues et sont reproductibles par d'autres au besoin.

Législation et certification des systèmes embarqués

Dans le domaine des systèmes embarqués critiques (automobile, étatique...), un fournisseur se doit d'obtenir des certifications sur ses produits. Il en existe aux niveaux matériel et logiciel. Ces certifications et cette réglementation proviennent de la pression des clients sur leurs fournisseurs, qui ont des impératifs de sûreté et de sécurité liés aux environnements de déploiement. Il existe une multitude de certifications et de normes pour les systèmes embarqués et le temps-réel[KZ09].

Les normes et certifications dépendent du pays dans lequel le système est développé ou déployé. Il n'y a pas la même culture de la sûreté et de la sécurité en France qu'aux États-Unis d'Amérique par exemple[NG10]. Cela provient des agences en charge de certifier ces logiciels et matériels qui sont souvent étatiques, comme la *Federal Aviation Administration* aux États-Unis, l'Agence Européenne de la Sécurité Aérienne ou la Direction Générale de l'Aviation Civile (la DGAC) en France pour l'aviation. Même si les émetteurs sont divers, leurs certifications sont reconnues dans leurs domaines.

Cependant, pour garantir une impartialité lors de l'évaluation des systèmes embarqués, il est nécessaire de mettre en place des normes et des certifications indépendantes des états et reconnues par un grand nombre.

Parmi elles, nous trouvons la norme ISO/IEC 15408 ou Critères Communs[MFMP07] (*Common Criteria*). Cette certification est reconnue par 31 pays à ce jour[CCR]. Cette certification permet aux utilisateurs finaux de déterminer le niveau de confiance d'un système, aux développeurs d'avoir un cadre de développement avec toutes les exigences de sûreté et de sécurité et aux évaluateurs d'avoir une grille précise pour déterminer si un système est conforme à la cible choisie. Les niveaux d'assurance (EAL *Evaluation Assurance Level*) de cette certification sont divisés en 7 parties (EAL1 à EAL7), décomposés en tests fonctionnels, structurels, méthodes de conception et développement, mais aussi l'utilisation de méthodes formelles pour les plus hauts niveaux d'assurance.

Les rôles des méthodes formelles

Le but de l'utilisation des méthodes formelles dans le développement de systèmes est de pouvoir vérifier mathématiquement des propriétés, vérifier qu'il n'y a pas de bug ou encore vérifier que le système respecte la spécification de départ.

Malheureusement, l'établissement d'une preuve sur un logiciel est un travail fastidieux. Il est difficile à mettre en œuvre dans tous les développements de logiciels et de systèmes.

Il existe différentes méthodes et outils permettant l'intégration de méthodes formelles au sein de projets. La preuve par le modèle est une méthode de travail sur une abstraction des problématiques logicielles et matérielles rencontrées. Une fois la preuve établie, le modèle est compilé pour être exécuté. Pour éviter la génération de problèmes de sécurité lors de la compilation[KMG+11], des compilateurs vérifiés ont vu le jour tel que CompCert[KLW14] pour le langage C.

Une autre méthode de formalisation d'un projet est de vérifier le code binaire généré après compilation. Cependant, la difficulté de cette vérification réside dans la multitude de matériels et nécessite donc une nouvelle vérification à chaque fois.

2.5 Résumé et conclusion

Nous avons vu dans cette analyse l'évolution des systèmes embarqués qui sont passés de domaines précis et spécifiques à des utilisations variées, en plus de leur ubiquité. Cela a amené le marché des fournisseurs de systèmes embarqués à se diversifier tant au niveau des compétences, mais aussi au niveau de la complexité du produit fourni. Ces systèmes embarqués se sont complexifiés et ont intégré de nouveaux mécanismes et des systèmes de communication.

La diversification des acteurs implique de mettre en place des procédures de collaboration, pour la conception, la fabrication, le déploiement et la maintenance des systèmes embarqués. Tout cela fait appel à différents acteurs qui deviennent des partenaires économiques, liés par contrat. Ces acteurs ont un rôle et des droits au sein de l'objet et de son cycle de vie.

Dans ce chapitre, nous nous sommes intéressés aux propriétés de sécurité et de sûreté liées aux systèmes embarqués. Nous avons vu qu'il était possible de mettre en place des mécanismes garantissant des propriétés des objets. Ces garanties peuvent faire l'objet de certifications et de travaux de formalisation afin de prouver le bien-fondé des affirmations des fabricants de systèmes embarqués. Le but de ces méthodes utilisées est de fournir aux utilisateurs le moyen d'évaluer les systèmes embarqués.

2.6 Références

[AB05] A.Z. Alkar and U. Buhur. An internet based wireless home automation system for multifunctional devices. *IEEE Transactions on Consumer Electronics*, 51(4) :1169–1174, nov 2005. 13

[AHQ+15] Jaume Abella, Carles Hernandez, Eduardo Quinones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods : Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, jun 2015. 18

[AIH15] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A Survey on Testing for Cyber Physical System. pages 194–207. Springer, Cham, nov 2015. 21

[BD17] Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Computing Surveys*, 50(6) :1–37, nov 2017. 20

[BDC84] Donald B. Brick, James Stark Draper, and H. J. Caulfield. Computers in the Military an Space Sciences. *Computer*, 17(10) :250–262, oct 1984. 10

[Bow93] Jonathan Bowen. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8 :189–209(20), July 1993. 18

[BVMG12] Ayan Banerjee, Krishna K. Venkatasubramanian, Tridib Mukherjee, and Sandeep Kumar S. Gupta. Ensuring Safety, Security, and Sustainability of Mission-Critical Cyber–Physical Systems. *Proceedings of the IEEE*, 100(1) :283–299, jan 2012. 18

[CCM+07] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2) :149–162, jun 2007. 13

[CCR] Members of the CCRA : New CC Portal. 21

[CLS+08] Wei Chen, Hongyi Lu, Li Shen, Zhiying Wang, Nong Xiao, and Dan Chen. A novel hardware assisted full virualization technique. In *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008*, pages 1292–1297. IEEE, nov 2008. 19

[CMFE09] Fernando Carvalho, Silvio R.L. Meira, Bruno Freitas, and João Eulino. Embedded software component quality and certification. In *Conference Proceedings of the EUROMICRO*, pages 420–427. IEEE, 2009. 21

[CMMS79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2) :105–115, feb 1979. 10

[DB16] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog Computing : Helping the Internet of Things Realize Its Potential. *Computer*, 49(8) :112–116, aug 2016. 15

[Deu13] Andreas Deuter. Slicing the V-model - Reduced effort, higher flexibility. In *Proceedings - IEEE 8th International Conference on Global Software Engineering, ICGSE 2013*, pages 1–10. IEEE, aug 2013. 13

[DGS81] Malcolm R Davis, Steven Glaseman, and William L Stanley. Acquisition and Support of Embedded Computer System Software, 1981. 10

[DMG97] J.A. Debardelaben, V.K. Madisetti, and A.J. Gadient. Incorporating cost modeling in embedded-system design. *IEEE Design & Test of Computers*, 14(3) :24–35, 1997. 11

[EKP16] Christof Ebert, Marco Kuhrmann, and Rafael Prikladnicki. Global Software Engineering : Evolution and Trends. In *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, pages 144–153. IEEE, aug 2016. 13

[EMJ08] Christof Ebert, Bvs Krishna Murthy, and Namo Narayan Jha. Managing risks in global software engineering : Principles and practices. In *Proceedings - 2008 3rd IEEE International Conference Global Software Engineering, ICGSE 2008*, pages 131–140. IEEE, aug 2008. 13

[Fis78] David A. Fisher. The interaction between the preliminary designs and the technical requirements for the dod common high order language. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 82–83, Piscataway, NJ, USA, 1978. IEEE Press. 10

[Fra16] A. Francillon. Trust, but verify : Why and how to establish trust in embedded devices. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1178–1182, March 2016. 21

[GLT03] B. Graaf, M. Lormans, and H. Toetenel. Embedded software engineering : The state of the practice. *IEEE Software*, 20(6) :61–69, nov 2003. 14

[GZ12] Zonghua Gu and Qingling Zhao. A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. *Journal of Software Engineering and Applications*, 05(04) :277–290, 2012. 14

[HBPT16] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating Systems for Low-End Devices in the Internet of Things : A Survey. *IEEE Internet of Things Journal*, 3(5) :720–734, oct 2016. 14

[Hei08] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems - IIES '08*, pages 11–16, New York, New York, USA, 2008. ACM Press. 19

[HGZG12] Brahim Hamid, Jacob Geisel, Adel Ziani, and David Gonzalez. Safety Lifecycle Development Process Modeling for Embedded Systems - Example of Railway Domain. pages 63–75. Springer, Berlin, Heidelberg, 2012. 21

[HKM+12] Jonathan L. Herman, Christopher J. Kenna, Malcolm S. Mollison, James H. Anderson, and Daniel M. Johnson. RTOS support for multicore mixed-criticality systems. In *Real-Time Technology and Applications - Proceedings*, pages 197–208. IEEE, apr 2012. 20

[HM01] J.D. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2) :16–20, 2001. 12

[HRK12] Anton Hattendorf, Andreas Raabe, and Alois Knoll. Shared memory protection for spatial separation in multicore architectures. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012 - Conference Proceedings*, pages 299–302. IEEE, jun 2012. 19

[HVJ14] Prasanna Hambarde, Rachit Varma, and Shivani Jha. The Survey of Real Time Operating System : RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39. IEEE, jan 2014. 14

[Ili81] R.J. Iliff. Avionics Integration Support Facility Developments Are Becoking Big Business. In *Conference Proceedings Southeastcon '81.*, pages 887–891. IEEE, 1981. 10

[Jon06] E. Jonsson. Towards an integrated conceptual model of security and dependability. In *First International Conference on Availability, Reliability and Security (ARES'06)*, pages 8 pp.–653. IEEE, 2006. 18

[KA03] D. Kalinsky and A. Avnur. Computer-aided real-time design. In *[1988] Proceedings. The Third Israel Conference on Computer Systems and Software Engineering*, pages 4–13. IEEE, 2003. 9

[KKH09] Doo-Hwan Kim, Jong-Phil Kim, and Jang-Eui Hong. Practice Patterns to Improve the Quality of Design Model in Embedded Software Development. In *2009 Ninth International Conference on Quality Software*, pages 179–184. IEEE, aug 2009. 11

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C Semantics : CompCert and the C Standard. pages 543–548. Springer, Cham, 2014. 22

[KMG+11] Gerwin Klein, Toby Murray, Peter Gammie, Thomas Sewell, and Simon Winwood. Provable security : How feasible is it? In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 28–28, Berkeley, CA, USA, 2011. USENIX Association. 22

[KNP97] J. Kalaoja, E. Niemela, and H. Perunka. Feature modelling of component-based embedded software. In *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, pages 444–451. IEEE Comput. Soc, 1997. 13

[Koo] P. Koopman. Embedded system design issues (the rest of the story). In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 310–317. IEEE Comput. Soc. Press. 11

[Kop00] Hermann Kopetz. Software engineering for real-time : A roadmap. In *Proceedings of the 22nd International conference on Future of Software Engineering (FoSE)*, page #, 2000. Vortrag : International Conference on Future of Software Engineering, Limerick, Ireland ; 2000-06-04 – 2000-06-11. 13

[KPCBH15] Siwar Kriaa, Ludovic Pietre-Cambacedes, Marc Bouissou, and Yoran Halgand. A survey of approaches combining safety and security for industrial control systems, jul 2015. 18

[Kün87] Albert T. Kündig. A note on the meaning of "Embedded systems". In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 284 LNCS, pages 1–5, Berlin/Heidelberg, 1987. Springer-Verlag. 9

[KZ09] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems : State of the art, jun 2009. 21

[Lee08] Edward A. Lee. Cyber physical systems : Design challenges. In *Proceedings - 11th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2008*, pages 363–369. IEEE, may 2008. 13

[LM66] T. J. Lawton and C. A. Muntz. Organization of computation and control in the apollo guidance computer. In *Peaceful Uses of Automation in Outer Space*, pages 398–408. Springer US, 1966. 9

[LW09] Nancy G. Leveson and Kathryn Anne Weiss. Software System Safety. *Safety Design for Space Systems*, pages 475–505, jan 2009. 18

[MEL10] Ryan A. McGee, Ulrik Eklund, and Mats Lundin. Stakeholder identification and quality attribute prioritization for a global vehicle control system. In *ACM International Conference Proceeding Series*, pages 43–48, New York, New York, USA, 2010. ACM Press. 15

[MFMP07] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. A common criteria based security requirements engineering process for the development of secure information systems. *Computer Standards & Interfaces*, 29(2) :244–253, feb 2007. 21

[MO78] C.E. Martin and R.F. O'Bleness. Life cycle management concepts for air force computer resources. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.* IEEE, 1978. 10

[MT03] Philip Melanson and Siamak Tafazoli. A Selection Methodology for the RTOS Market. *European Space Agency, (Special Publication) ESA SP*, 532(532) :461–472, 2003. 14

[MY11] Aleksander Malinowski and Hao Yu. Comparison of Embedded System Design for Industrial Applications. *IEEE Transactions on Industrial Informatics*, 7(2) :244–254, may 2011. 13

[NG10] Florence Nnanga-Grisoni. Systèmes embarqués, aéronautique et droit. *Cahiers Droit, Sciences & Technologies*, (3) :307–315, may 2010. 21

[PCL+96] Pierre G. Paulin, Marco Cornero, Clifford Liem, François Naçabal, Chris Donawa, Shailesh Sutarwala, Trevor May, and Carlos Valderrama. Trends In Embedded Systems Technology. In *Hardware/Software Co-Design*, pages 311–337. Springer Netherlands, Dordrecht, 1996. 14

[PCPC75] J. L. Pruitt, W. W. Case, J. L. Pruitt, and W. W. Case. Architecture of a real time operating system. In *Proceedings of the fifth symposium on Operating systems principles - SOSP '75*, volume 9, pages 51–59, New York, New York, USA, 1975. ACM Press. 10

[PD93] Fabio Panzieri and Renzo Davoli. Real time systems : A tutorial. In *Performance Evaluation of Computer and Communication Systems*, pages 435–462. Springer-Verlag, Berlin/Heidelberg, 1993. 11

[PDK+15] Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Muench, and Jan Nowotsch. Mixed-criticality embedded systems-A balance ensuring partitioning and performance. In *Proceedings - 18th Euromicro Conference on Digital System Design, DSD 2015*, pages 453–461. IEEE, aug 2015. 19, 20

[PT89] R.E. Phillips and M.J. Thullen. Embedded Computer System Integration Support. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 616–622. IEEE, 1989. 10

[PZP15] Abhinand Palicherla, Tao Zhang, and Donald E. Porter. Teaching Virtualization by Building a Hypervisor. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*, pages 424–429. ACM Press, 2015. 19

[RMOT12] Pilar Rodríguez, Jouni Markkula, Markku Oivo, and Kimmo Turula. Survey on agile and lean usage in finnish software industry. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, page 139, New York, New York, USA, 2012. ACM Press. 13

[Rus00] John Rushby. Partitioning in avionics architectures : requirements, mechanisms and assurance. *Work*, (March) :67, 2000. 19

[SBH16] Farzad Samie, Lars Bauer, and Jörg Henkel. IoT technologies for embedded computing. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES '16*, pages 1–10, New York, New York, USA, 2016. ACM Press. 14

[SH68] W. I. Stanley and H. F. Hertel. Statistics gathering and simulation for the Apollo real-time operating system. *IBM Systems Journal*, 7(2) :85–102, 1968. 10

[SST89] Juha Pekka Soininen, Matti Sipola, and Kari Tiensyrjä. SW/HW-partitioning of real-time embedded systems. *Microprocessing and Microprogramming*, 27(1-5) :239–244, aug 1989. 11

[Sta96] John A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4) :751–763, dec 1996. 11

[Ste91] D.F. Sterne. On the buzzword 'security policy'. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 219–230. IEEE Comput. Soc. Press, 1991. 17

[SVLN13] Kristian Sandstr??m, Aneta Vulgarakis, Markus Lindgren, and Thomas Nolte. Virtualization technologies in embedded real-time systems. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pages 1–8. IEEE, sep 2013. 19

[TH66] Milton B. Trageser and David G. Hoag. Apollo spacecraft guidance system. In *Peaceful Uses of Automation in Outer Space*, pages 435–463. Springer US, 1966. 9

[YKT+06] B. Yuksekkaya, A.A. Kayalar, M.B. Tosun, M.K. Ozcan, and A.Z. Alkar. A GSM, internet and speech controlled wireless interactive home automation system. *IEEE Transactions on Consumer Electronics*, 52(3) :837–843, aug 2006. 13

Chapitre 3

Problématique de la sécurité des systèmes embarqués

> Un problème créé ne peut être résolu en réfléchissant de la même manière qu'il a été créé.
>
> Albert Einstein

Sommaire

3.1 Introduction

Dans notre observation des systèmes embarqués réalisée dans le chapitre précédent, nous avons constaté que leur développement et leurs fonctionnalités se sont complexifiés au cours des années, passant de systèmes développés en interne à une collaboration regroupant plusieurs équipes ou entreprises.

Cependant, malgré les outils et méthodes que nous avons pu rencontrer pour garantir des propriétés, le cycle de vie d'un système embarqué ou d'un objet connecté comporte quelques problématiques de sécurité ou de sûreté que nous allons nous efforcer d'analyser dans ce chapitre.

3.2 Conception des systèmes embarqués critiques : l'ouverture

3.2.1 Fondement de sécurité dans un système

Intégrité, confidentialité et disponibilité des données sont les premiers éléments auxquels nous nous intéressons quand nous parlons de sécurité des données au sein d'un système.

Les données propres à un composant du système peuvent prendre deux formes :

Code source : Le code source du composant qui définit son comportement.

Informations et ressources : Les informations qui servent au processus et à leur traitement.

Chaque composant logiciel ou matériel possède ces deux types de données. Pour garantir le comportement escompté de ce composant, il est nécessaire d'assurer la préservation de ces données.

Intégrité et Confidentialité Que ce soit le code source ou les ressources utilisées, nous pouvons vouloir maintenir deux propriétés : leur intégrité et leur confidentialité.

L'intégrité est la capacité du système à maintenir les données intactes et à empêcher toute altération non désirée.

La confidentialité est la capacité du système à empêcher tout accès en lecture non autorisée de données.

Les deux propriétés sont liées, mais l'une n'implique pas l'autre. En effet, il est possible d'altérer des données sans forcément avoir un accès en lecture[SM11]. De même, il est possible de voler des données sans forcément les altérer[DCY17].

La menace qui pèse sur les données n'est pas seulement le fruit d'attaques malveillantes. Cela peut aussi provenir d'une mauvaise implémentation de la part de l'entité propriétaire des données, mais aussi des autres entités présentes au sein du système.

Disponibilité et contraintes temps-réel En plus de la protection des données et des informations des composants, une autre garantie sur leur comportement logiciel est nécessaire. Le comportement d'une application ou d'une tâche est défini par son code source, qui doit rester intègre et confidentiel. Cependant, il est aussi nécessaire de maintenir le comportement au cours du temps.

La disponibilité est la capacité d'assurer qu'un code source peut être exécuté, tout en fournissant les ressources nécessaires à son fonctionnement (temps CPU, accès matériel).

Cette problématique de disponibilité est au cœur des systèmes temps-réel comme nous l'avons vu.

Isolation spatiale vs Isolation temporelle Cependant, ces systèmes temps-réel se focalisent surtout sur la fiabilité et le principe "une tâche doit être exécutée dans ses délais et sur un temps raisonnable". Pour cela, comme vu, il existe des mécanismes de garantie temporelle. Cependant, nous devons aussi nous poser la question du maintien de ces mécanismes. Si le code source ou les ressources utilisées par ces mécanismes sont compromis (intégrité et confidentialité), il devient alors difficile de fournir de la disponibilité aux applications.

La confidentialité et l'intégrité sont assurées par l'isolation spatiale, l'isolation mémoire. La disponibilité est une propriété de l'ordre de l'isolation temporelle. Il existe donc une relation entre ces types d'isolation, une relation hiérarchique.

Si l'isolation temporelle n'est plus garantie par le système, il n'est pas nécessaire pour autant que les données et l'isolation mémoire soient compromises. Dans le cas inverse, si l'isolation spatiale est atteinte, nous avons de ce fait accès à des pans de la mémoire, qui peuvent contenir les mécanismes d'isolation temporelle. De ce fait, cette dernière est compromise.

Il est nécessaire que, pour garantir l'isolation temporelle, il faille fournir des garanties fortes sur l'isolation spatiale et la mémoire (figure 3.1). Nous propagerions donc l'isolation spatiale dans toute l'architecture pour maintenir les propriétés de sécurité.

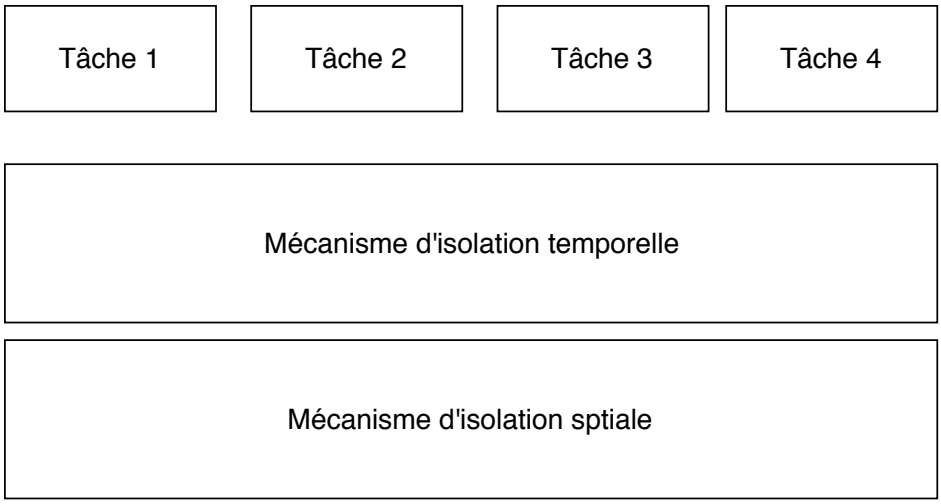

FIGURE 3.1 – Superposition Isolation spatiale et Isolation temporelle

3.2.2 Cohabitation et criticité

Dans le cas d'un système fermé, produit par une même entité, il est plus aisé de vérifier et de respecter les contraintes liées à ces deux propriétés. Cependant, dans le cas de systèmes ouverts et extensibles durant leur durée de vie, il est possible que le système, à un moment donné, intègre des composants malveillants ou mal implémentés.

Cette problématique est d'autant plus importante dans la situation d'une cohabitation entre composants critiques et non critiques.

De nos jours, les véhicules deviennent autonomes et intelligents. En plus de proposer des services de sécurité (aide au freinage, maintien de trajectoire...), ils proposent désormais des services aux usagers, multimédias notamment. Il est possible maintenant dans

un véhicule de diffuser de la musique issue de services en ligne directement depuis les interfaces du véhicule ou d'avoir une connexion Internet pour tous les usagers par wifi.

On se retrouve donc dans une situation de cohabitation entre applications critiques (éléments de sécurité) et des applications non critiques (service de musique).

On peut penser qu'il existe une barrière physique entre ces éléments, qu'ils ne sont pas connectés matériellement ou logiciellement.

Dans les dernières années, plusieurs personnes[Tak18] ont montré que les véhicules modernes et connectés comportaient des failles de sécurité depuis les services proposés, qui impliquent la sécurité même des passagers.

Ces attaques au sein de véhicules montrent clairement une liaison (via le bus CAN notamment) entre la partie critique (gérée par le BSI du véhicule) et la partie multimédia, accessible directement depuis les interfaces utilisateurs.

Ces attaques sont le résultat d'une mauvaise implémentation des couches utilisateurs ou d'une méconnaissance des prérequis de sécurité ou de sûreté dans des véhicules. Les interfaces ne feront qu'évoluer et proposer de plus en plus d'applications aux utilisateurs[She15] (environnement Android, Linux...).

Dans les objets connectés fournissant des services, ces services peuvent être issus de différents fournisseurs. Si deux services proposés au sein de ces objets sont issus de deux fournisseurs distincts, mais concurrents commerciaux, il se peut que l'un d'eux veuille nuire à son adversaire économique pour le discréditer aux yeux des clients et récupérer des parts de marché. Il peut aussi chercher à espionner son concurrent, voler des fonctionnalités ou autre.

Cet environnement dans lequel se développent les acteurs de l'IoT n'est pas un climat de confiance. Il ne permet pas à chaque acteur de se développer sereinement dans un cadre garanti.

3.2.3 La généricité source de faille globale

Très souvent, des failles de sécurité apparaissent dans les systèmes d'exploitation. Linux, Windows, OSX, Android... aucun n'est épargné. Les systèmes embarqués et les objets connectés peuvent aussi comporter des failles. Les systèmes embarqués sont les nouvelles cibles[CWSJ16]. Avec plus de 50 milliards d'objets connectés estimés pour 2020, il est utopique de penser qu'ils seront épargnés par les attaquants malveillants. De plus, ces objets connectés sont pour la plupart dépourvus de réels mécanismes de sécurité et de sûreté efficaces [Eri][AA18].

Les systèmes et environnements utilisés dans l'IoT sont souvent génériques et répandus. Comme vu dans l'état de l'art, nous trouvons souvent les mêmes systèmes d'exploitation ou les mêmes familles. De même pour les composants matériels.

Une réflexion évidente que l'on pourrait se faire concernant ces composants, c'est la possibilité qu'une faille trouvée dans un seul objet puisse être propagée dans tous les objets connectés qui comportent le même composant faillible.

Dans le cas de démarches malveillantes et distantes, nous pouvons imaginer au moins trois types d'attaques sur les objets connectés. Dans chaque scénario, nous avons un objet connecté sans mécanismes de sécurité, comportant au moins une application faillible.

Faille dans un système central Si un attaquant trouve une faille au sein du serveur, il serait en capacité de prendre le contrôle de l'application au sein de l'objet et, en profitant de la faible sécurisation de l'objet, porter atteinte à l'intégralité de l'objet ou des applications (figure 3.2 puis figure 3.3).

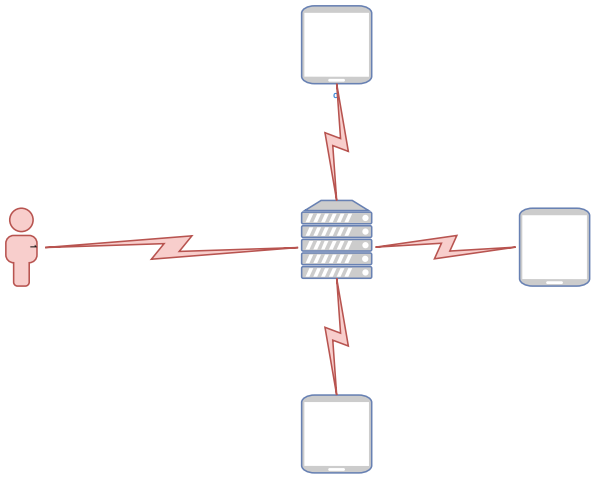

FIGURE 3.2 – Attaque des objets via un point central

Faille directement au sein de l'objet Une application accessible directement via une interface (connexion wifi, bluetooth...) peut être faillible. De ce fait, un individu non autorisé malveillant peut réussir une attaque et s'introduire au sein de l'objet, et peut ainsi accéder à l'intégralité de l'objet (figure 3.3).

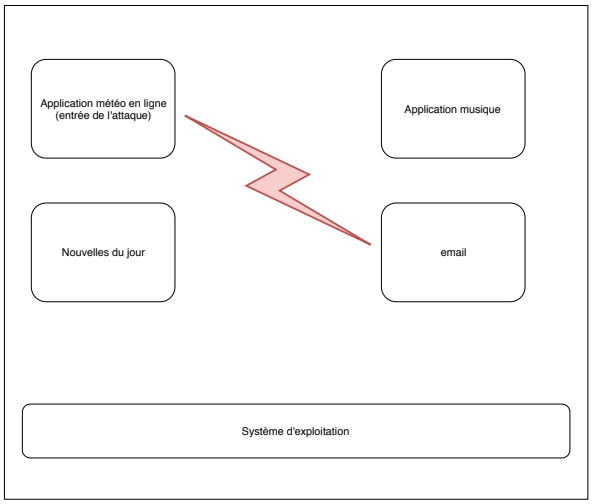

FIGURE 3.3 – Exemple d'objets connectés et propagation d'attaques ou de failles

Déni de service de l'objet Une application peut comporter des erreurs d'implémentation provoquant des comportements indéfinis selon certaines situations de fonctionnement. Dans ce scénario, si cette application se retrouve dans une de ces situations, elle peut mettre hors service tout l'objet et, de ce fait, rendre aussi les autres applications inaccessibles à leurs utilisateurs (figure 3.3).

Ces trois scénarios ne sont que quelques exemples. Mais ce qui paraît évident, c'est qu'il y a potentiellement dans les objets connectés un manque de confinement de failles ou d'attaques. Que le périmètre d'attaque soit interne ou externe à l'objet, ce type de failles peut devenir systémique pour tout un ensemble d'objets ou de produits qui partagent les mêmes bases logicielles ou matérielles.

Requêtes intra-objet Un dernier point que l'on peut soulever au sein des objets concerne la capacité des applications à requérir les services des autres applications. Pour illustrer

ce genre de situation, nous pouvons imaginer un assistant personnel virtuel dans une maison. L'application qui répond aux demandes des utilisateurs doit pouvoir lancer l'application de musique ou encore donner la météo. Il est cependant nécessaire de réguler l'accès des applications, depuis d'autres applications, au sein des objets pour éviter toute utilisation frauduleuse. Dans le cas d'applications neutres comme la météo, ce n'est pas nécessairement pertinent. Cependant, si l'assistant personnel est aussi en capacité d'accéder à des courriers électroniques ou des messages privés, là, nous pouvons nous poser la question du droit d'accès.

3.2.4 Problématique au sein de groupes d'objets

Comme vu dans l'état de l'art, une autre façon d'utiliser les objets connectés consiste à les grouper pour créer de nouveaux services (réseau de capteurs dans un champ agricole pour sa surveillance) ou dans un cadre domotique (figure 3.4), avec un assistant personnel virtuel qui gère un environnement. La distribution des tâches et des fonctionnalités permet en effet de réduire les coûts au sein de chaque objet (coût de production, énergétique...).

Quand nous nous intéressons aux volets sécurité et sûreté de fonctionnement de ce type d'architecture [RZL13], nous pouvons constater différentes lacunes dans le développement et la conception. Cependant, celles qui nous intéressent sont les problématiques autour des communications et des requêtes.

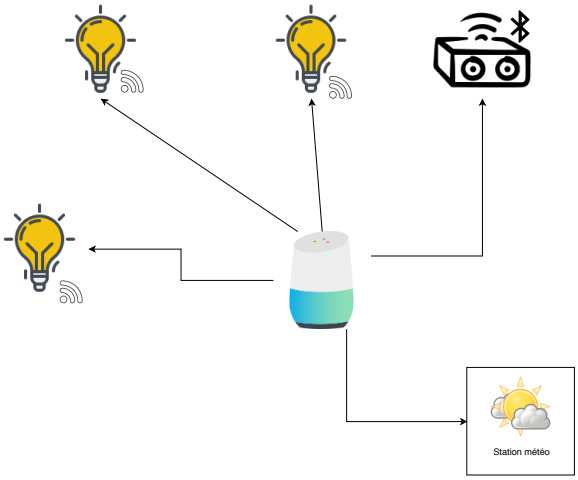

FIGURE 3.4 – Exemple d'une architecture distribué dans un environnement domotique

Préoccupation 1 : Gestion à distance des objets et applications Le premier point concerne la gestion des objets à distance. Qu'elle soit faite par une machine ou par une personne, elle peut prendre différentes formes.

Gestion des objets : Mise à jour de périphériques, mise à jour du système d'exploitation ou installation d'applications, la gestion de l'objet en lui-même est à faire par le propriétaire de l'objet ou une entité mandatée par ce dernier. Il est donc important d'avoir une maîtrise sur les actions au sein et sur l'objet.

Gestion des applications : La gestion des applications (mise à jour, réponse à une requête...) est uniquement le fait du propriétaire de l'application ou d'une entité mandatée par ce dernier. S'il veut garder un certain niveau de sécurité sur son application, il se doit de mettre en place des mécanismes appropriés.

Préoccupation 2 : Requête inter-objet À l'image des requêtes intra-objets, nous pouvons avoir une situation où les applications ont besoin d'accéder à des applications dans d'autres objets. Si nous reprenons le cas de l'assistant personnel, il est devenu courant d'y associer des ampoules connectées, et l'assistant personnel est en capacité de les allumer et de les éteindre à distance. Ceci étant rendu possible car les ampoules intègrent des applications répondant à des requêtes via un réseau (Wifi, Bluetooth, Zigbee...). Les problématiques de sécurité de ce genre de communications existent surtout dans la domotique [SGV+15][ZMR17].

3.3 Acteurs, droits et hiérarchie

3.3.1 Exhaustivité des acteurs de l'IoT

À travers la vision que nous avons eue du cycle de développement et de vie d'un système embarqué, nous avons constaté qu'il existe une multitude d'acteurs intervenant à différents moments du cycle. Qu'ils interviennent avant ou après émission de l'objet, ils ont des tâches et des rôles bien précis. Cependant, la vision que nous avons constatée est-elle complète ? Insuffisante ? Ou comporte-t-elle des rôles inutiles ou redondants dans la vie de l'objet ?

Énumération des acteurs Le nombre d'acteurs participant à l'élaboration de l'objet dépend de la charge de travail de développement. Dans un temps donné, plus un projet est conséquent, plus la charge de travail augmente. De plus, le nombre de personnes travaillant sur le projet augmente aussi. C'est le principe d'homme-mois [Kid05]. En raison d'un manque de compétence ou de ressources en interne, l'initiateur peut aussi chercher à externaliser certaines tâches, voire faire réaliser tous les composants de son objet par des sous-traitants. Le nombre d'acteurs dépend donc du nombre de composants et de tâches à produire pour la fabrication de l'objet.

Pertinence des rôles Ce que nous savons, c'est que dans un projet moderne, de nombreux acteurs (internes ou externes) interviennent. Il est nécessaire de les regrouper dans un ensemble de rôles. Ce regroupement se fait suivant l'expertise de l'acteur.

Toutefois, il se peut aussi qu'un acteur ait plusieurs compétences, ou que des tâches soient internalisées. Par conséquent, le champ de responsabilités de cet acteur augmente proportionnellement au nombre de tâches qu'il traite.

3.3.2 Hiérarchisation des acteurs

Après avoir identifié qu'il existe une multitude de rôles pour la réalisation d'un objet, la question de leur hiérarchisation se pose. Quel est le niveau hiérarchique de chaque acteur ?

L'initiateur du projet est-il nécessairement en haut de la hiérarchie ? Ou est-ce le développeur du système de base ou du matériel ?

On peut penser à reprendre la hiérarchie des composants matériels et logiciels d'un objet. Les matériels et ses composants (processeur, carte wifi...) sont en haut et les composants utilisateurs et applications tout en bas de l'échelle (figure 3.5). Par exemple, dans un smartphone, nous retrouvons cette architecture :

— Un ou des fabricants de composants matériels (p. ex. Broadcom)

— Un concepteur d'OS (Google Android)
— Des fournisseurs de pilotes matériels (Broadcom...)
— Des logiciels intégrés à l'OS privilégiés (couche Google)
— Un initiateur (p. ex. Samsung...)
— Un utilisateur humain
— Des fournisseurs d'applications (Spotify, YouTube...)

Dans cette hiérarchie, l'initiateur n'est pas en haut de la hiérarchie ; plusieurs acteurs détiennent autant voire plus de capacités dans le système. L'initiateur subit le comportement des autres acteurs.

Cependant, si le chemin dessiné par le cahier des charges est suivi, l'initiateur est à la tête et mandate un fabricant de matériel, un concepteur d'OS et tout autre acteur nécessaire pour concevoir le produit. Une fois le produit terminé et déployé, l'utilisateur final, en cas de difficulté avec l'objet, va se rapprocher de l'initiateur, et non pas des fabricants. D'ailleurs, souvent l'utilisateur ne connaît pas les composants de son objet. Pour l'utilisateur, le seul responsable en cas de difficulté est l'initiateur.

Cependant, l'initiateur n'est pas au cœur du système. Il n'a pas spécialement un contrôle logiciel/matériel sur les autres entités.

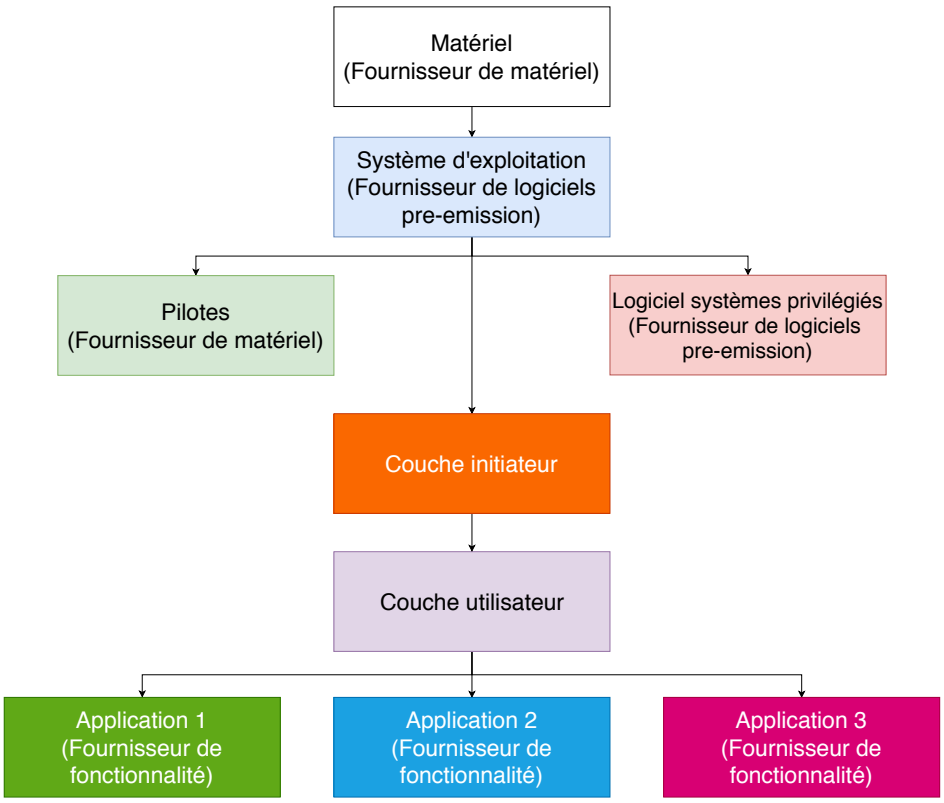

FIGURE 3.5 – Premier point de vue de la hiérarchie des acteurs

3.3.3 Droit et responsabilité des acteurs durant la durée de vie de l'objet

Un grand pouvoir impliquant de grandes responsabilités, les droits de chaque acteur dépendent évidemment de son rôle dans le développement. Pourtant, nous avons vu que

l'initiateur du projet, celui au centre du projet de l'objet, n'est pas nécessairement en haut de la hiérarchie logicielle/matérielle de l'objet, mais c'est lui le responsable de l'objet d'un point de vue légal.

D'un point de vue légal, en cas de difficulté, l'utilisateur se tourne en effet vers l'initiateur. Celui-ci va chercher la source du problème. Si la cause est externe, il doit donc reporter le problème auprès de l'acteur concerné. Si le problème est matériel (défaillance matérielle), l'identification de l'auteur de la faute est facilitée. Dans le cas logiciel, cela peut être plus délicat.

Comme il n'est pas en haut de la hiérarchie du système, il n'est pas en capacité d'identifier l'auteur du problème. Si le système ne comporte pas de mécanismes de sécurité ou de sûreté forts, le problème peut avoir été dilué dans tout le système. Et de ce fait, il y a une dilution des responsabilités de chaque acteur, aux dépens de l'initiateur.

Dans ce type d'architecture, l'initiateur est en haut de la hiérarchie légale, mais pas de celle logicielle et matérielle.

3.3.4 Question de la délégation de tâches

Que devons-nous déléguer quand un système embarqué est exécuté ? Nous avons vu que l'initiateur peut être amené à déléguer par manque de compétences et de ressources. Nous pouvons distinguer plusieurs familles de tâches :

Conception/Fabrication : concevoir et fabriquer un composant logiciel ou matériel suivant un cahier des charges

Gestion : gérer différentes propriétés au sein de l'objet

Mise à jour : mettre à jour les différents composants de l'objet

Les différents acteurs sous-traitants vont se regrouper au sein de ces familles. De surcroît, nous avons aussi vu que le nombre de tâches influe sur la responsabilité de l'acteur.

On souhaiterait que l'initiateur, après la conception de l'objet et son déploiement, ait une vision claire des rôles de chacun et de ses capacités.

Jusqu'à l'émission, l'initiateur supervise et a continuellement un contrôle sur ce que réalisent les autres acteurs. L'objet émis est configuré pour être dans un état d'utilisation normal. Cette configuration de l'objet est une tâche qui n'est pas nécessairement du ressort de l'initiateur, mais suit une politique définie par ce dernier. Si la configuration de l'objet est incorrecte et ne respecte pas les spécifications décidées par l'initiateur, cela met l'objet dans un état indéfini.

Après émission de l'objet, l'initiateur perd la vision sur les tâches de certains acteurs, comme les fournisseurs de services. Ces derniers développent et voient leurs applications être déployées souvent sans que l'initiateur ait connaissance de ces actions. Si la gestion de l'objet est incorrecte, ou que les applications ont un comportement contraire au fonctionnement défini par l'initiateur, ou encore si elles sont malveillantes, cela peut engendrer une défaillance de l'objet. Tout cela est dû à des lacunes de sécurité au sein de l'objet.

La politique de gestion et de sécurité que définissent les fournisseurs de services pour leurs applications ne doit pas interférer ou empêcher la bonne utilisation de l'objet par les autres acteurs. Cependant, l'initiateur ne doit pas interférer non plus avec le fonctionnement interne de ces applications. La seule action qu'il peut entreprendre à travers sa politique d'administration est d'empêcher toute action malveillante sur le reste du système.

La mise à jour des pilotes ou des composants logiciels est aussi une tâche que délègue l'initiateur. Avant leur déploiement, elles peuvent être testées au sein d'objets témoins. Cependant, il n'est pas rare d'avoir aussi des défaillances une fois déployées.

Durant la durée de vie de l'objet, sa politique d'administration et de sécurité dépend du contexte dans lequel l'objet est déployé. Cette politique doit pouvoir être définie par l'utilisateur ; l'initiateur fournit à l'utilisateur la capacité de modifier et de gérer cette politique d'administration. Cependant, il faut aussi responsabiliser les actions de l'utilisateur, afin d'identifier que la source d'une défaillance est située dans les mécanismes d'administration.

3.3.5 Représentation logicielle des acteurs

"Comment transférer une vision contractuelle de l'objet à un environnement logiciel et matériel?" La hiérarchie et les contraintes définies par le contrat commercial ne sont pas toujours respectées au sein de l'objet comme vu dans les sections précédentes. Ceci peut être dû par exemple à des systèmes génériques et réadaptés de manière substantielle pour être utilisés dans le contexte défini par l'initiateur.

Une autre problématique rencontrée dans la représentation logicielle de certains objets est l'absence de l'initiateur au centre des communications entre composants logiciels. Dans la figure 3.6, nous avons une vision simplifiée de la hiérarchie durant le développement de l'objet d'un point de vue contractuel. Si un acteur souhaite communiquer avec un autre, l'initiateur met en place ce canal de communication ou sert d'intermédiaire durant les échanges. De ce fait, dans l'implémentation du système embarqué, si l'initiateur est au centre des communications au niveau contractuel, il faut aussi qu'il ait un certain contrôle au sein de l'objet.

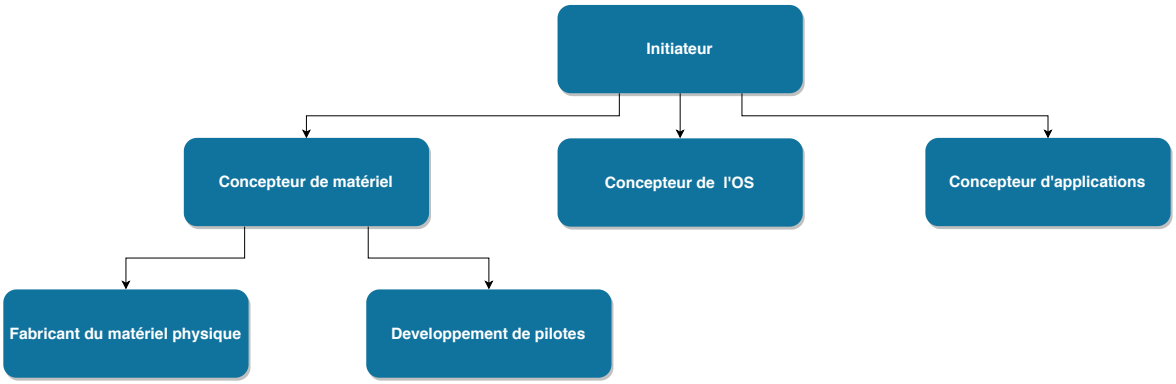

FIGURE 3.6 – Hiérarchie d'un point de vue contractuel

Nous devons aussi garder à l'esprit la volonté de responsabiliser chaque acteur. Au sein d'un objet, la responsabilité d'un acteur, comme vu précédemment, est déterminée par le nombre de tâches déléguées par l'initiateur. Plus l'acteur a de tâches, plus son impact au sein de l'objet est grand. Il est nécessaire dans un premier temps que l'initiateur ait la capacité de mettre en place cette délégation, mais aussi d'y mettre fin au besoin. De plus, nous avons vu qu'il existe une propagation de fautes ou d'attaques entre composants de l'objet, nécessitant de ce fait un mécanisme de confinement entre les composants logiciels.

3.3.6 Précédent de la sécurité par rôle

Le point de vue que nous essayons d'avoir des systèmes embarqués et des objets connectés de nos jours nous a amenés à penser des rôles et des tâches dans le développement et la gestion de ces objets. Cette problématique de rôles, de droits et de responsabilités n'est pas quelque chose de nouveau. En effet, quand nous nous intéressons aux secteurs sensibles nécessitant une structure et une hiérarchie similaire, cela nous amène au domaine des cartes à puce.

Apparues dans les années 1950 [RE03], le but des cartes à puce fut rapidement de proposer un moyen de paiement facile à transporter. Avec l'arrivée des acteurs que sont Visa et MasterCard, les cartes à puce se sont rapidement démocratisées. Leurs usages se sont diversifiés avec les années; nous les trouvons aujourd'hui dans différents secteurs d'activité :

— Communication avec la téléphonie mobile (GSM, UMTS), cartes prépayées

— La santé avec l'identification des individus et de leur dossier

— Systèmes de paiements électroniques

— Sécurité, cryptographie et contrôles d'accès

Au cours des années de leur développement, les cartes à puce sont devenues de plus en plus complexes. Au départ, c'était de simples circuits fabriqués à un usage précis. Puis au cours des années 90, des cartes à puce plus sophistiquées sont apparues. Dotées d'unités de calcul et de plus de mémoire, elles ont commencé à comporter des systèmes et des logiciels tout aussi sophistiqués.

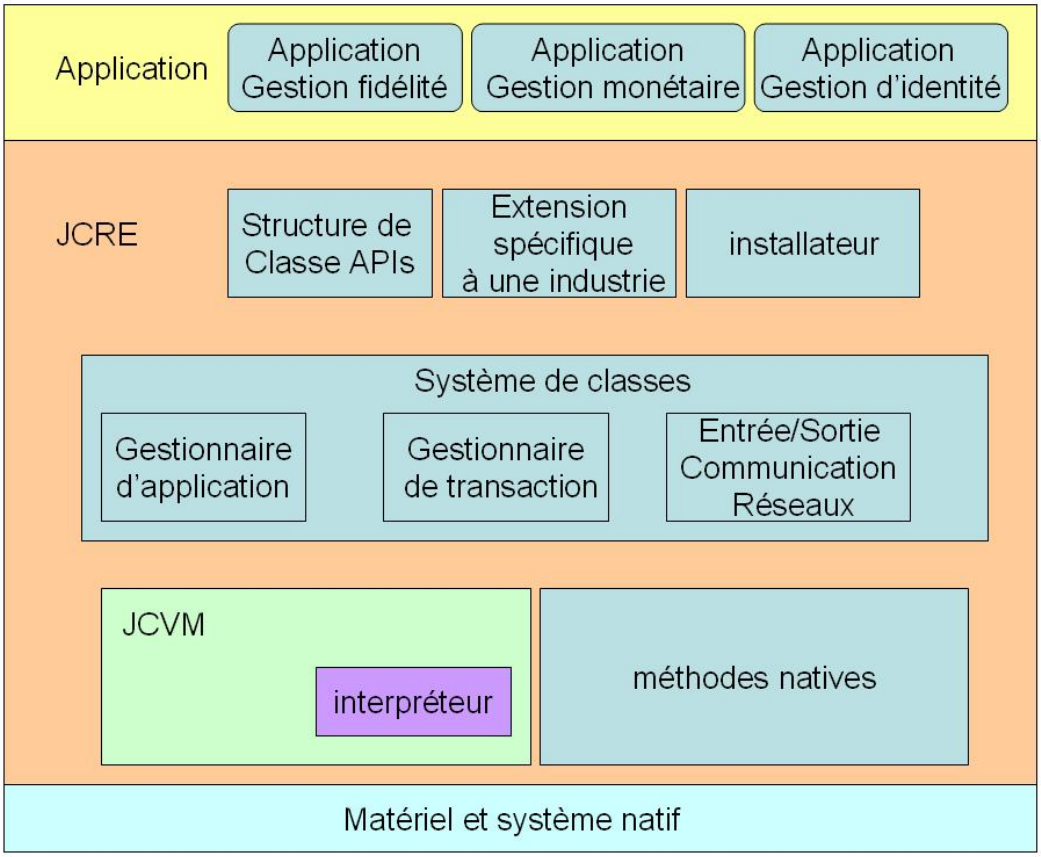

FIGURE 3.7 – Architecture logicielle d'une carte à puce avec plusieurs services

Au début, les cartes à puce étaient monolithiques (une carte à puce pour une tâche/application) puis, dotées de ressources plus importantes, ces nouvelles cartes à puce intègrent des systèmes d'exploitation de plus en plus complexes puis des mécanismes de virtualisation pour intégrer des applications issues de fournisseurs divers [Mar06] [Hus01]. Parmi ces systèmes complexes, nous retrouvons Java Card, Multos et bien d'autres [Sau09].

Avec ce développement de plus en plus complexe des cartes à puce, il est devenu difficile pour une même entité de les développer et de les maintenir [DGGJ03]. De ce fait, durant le cycle de fabrication et de mise à disposition de l'utilisateur de la carte à puce, nous distinguons plusieurs étapes et plusieurs acteurs (figure 3.8).

FIGURE 3.8 – Cycle de production d'une carte à puce et acteurs

Cette complexification et les contraintes fortes de sécurité ont amené les fabricants à mettre en place des mécanismes d'isolation et de protection permettant de fournir un environnement de confiance à toutes les applications au sein de la carte.

D'autres secteurs aussi ont connu cette hiérarchisation pour plus de sécurité et il paraît évident, au vu des problématiques de sécurité importantes, que le domaine de l'embarqué et des objets connectés nécessite d'être repensé pour plus de sécurité.

3.4 Confiance et sécurité dans les systèmes embarqués

3.4.1 L'apport de sécurité dans les systèmes

Nous avons vu dans l'état de l'art qu'il existe de nombreuses méthodes et solutions pour apporter de la sécurité, de la sûreté et de la confiance au sein des systèmes. Toutefois, au vu des problèmes de sécurité connus et rencontrés à ce jour, nous pouvons nous demander quelles sont les difficultés qui empêchent de mettre en place les mécanismes de sécurité contre ces attaques.

La sécurité et la sûreté dans les systèmes embarqués et les objets connectés, comme vu, consistent à maintenir les propriétés du système définies par ses auteurs. Dans des systèmes fermés, la sûreté de fonctionnement et la fiabilité du système sont vérifiées, et les risques possibles durant la durée de vie de l'objet sont connus à l'avance et des solutions sont mises en place pour gérer ces difficultés.

Mais dans le cadre de systèmes ouverts et extensibles, ces propriétés sont remises en question avec des problématiques de sécurité. Nous avons vu que les objets connectés intègrent très peu ou pas de mécanismes de sécurité.

Dans l'état de l'art, nous avons eu une vision non exhaustive montrant qu'il existe une multitude de composants matériels et de solutions logicielles permettant de garantir les propriétés du système. Mais au vu des exemples de problèmes de sécurité dans les objets, nous pouvons constater qu'ils ne comportent que très peu ou aucune garantie de sécurité.

Le premier argument que nous pouvons imaginer pour expliquer ces lacunes est la réduction du coût de développement de l'objet. Les mécanismes de virtualisation ne sont

pas disponibles sur tous les systèmes physiques. Mettre en place un hyperviseur n'est pas possible sur certains modèles de processeurs ou encore les MMU physiques ne sont pas toujours présentes. Étant donné que les systèmes seront développés en grande quantité, leurs initiateurs chercheront à réduire au maximum les coûts de production.

Un autre argument plus subtil est d'ordre culturel. La vision de la sécurité des systèmes peut prendre différentes formes au sein de la communauté de développeurs. Certains développeurs omettent même les principes de sécurité lors du développement, privilégiant l'investissement sur d'autres points (ergonomie, efficacité énergétique...). Cette difficulté peut provenir de l'initiateur. N'étant pas sensible aux problématiques de sécurité, il peut ne pas faire le choix d'intégrer un acteur spécialisé ou sensibilisé.

Ce problème d'intégration de la sécurité via des acteurs sensibilisés vient impacter directement la spécification de l'objet. De ce fait, même s'il y a une phase de test et de validation de fonctionnalités, l'initiateur ne fera pas appel à un ou des acteurs spécialisés dans les audits de sécurité pour vérifier l'objet.

3.4.2 Validité des audits

Les audits de sécurité sont réalisés par des acteurs indépendants, spécialisés dans ce domaine. Les auditeurs effectuent des tests et suivent des méthodes reproductibles pour valider des propriétés du système. Ces méthodes sont appliquées à la fin de la phase de développement, à l'initiative de l'initiateur ou de manière indépendante par d'autres entités (institution, utilisateurs...).

Tous les systèmes embarqués ne font pas l'objet d'audits. Quand un audit est effectué, très souvent, c'est en lien avec des certifications ou des autorisations de mises sur le marché. C'est notamment le cas dans les transports, les environnements étatiques ou militaires. Une nouvelle fois, le contexte de déploiement influence aussi la vérification de l'objet.

Les audits ne sont pas des tests ou des procédés de vérification de sécurité exhaustifs. Ils vérifient beaucoup de propriétés des systèmes ; néanmoins, à l'instar des tests de fonctionnalités, ils ne vérifient pas toutes les propriétés du système. Même dans des environnements critiques comme les transports, nous avons vu qu'il existait des failles de sécurité en dépit de toutes les vérifications effectuées.

De plus, à ce manque d'exhaustivité s'ajoute une grande variété d'auditeurs. Les acteurs spécialisés dans l'audit sont publics comme privés. Ces derniers peuvent aussi employer des méthodes d'audits confidentielles pour une question de savoir-faire. Et la garantie de ces audits repose sur la réputation de l'auditeur.

Les auditeurs deviennent donc acteurs dans le développement de l'objet. S'ils sont liés contractuellement avec l'initiateur, leur responsabilité est impliquée en cas de défaillance de sécurité au sein de l'objet. En outre, les auditeurs et leur réputation représentent une caution sécurité pour les initiateurs.

Un autre point important concernant les audits de sécurité : s'ils ne sont pas demandés par l'initiateur et s'ils ne sont pas satisfaisants, cela peut grandement influencer la réputation de l'objet et de ce fait les ventes.

3.4.3 Difficulté de mise en place des preuves formelles

Une solution apparue depuis quelques années dans le développement de systèmes est l'emploi des méthodes formelles. Nous avons vu dans l'état de l'art que cette méthode permet d'apporter des garanties basées sur des preuves mathématiques.

Ce travail est effectué en amont et au cours du développement, afin de vérifier des propriétés de sécurité précises et importantes du système. De plus, l'avantage de ces preuves est qu'elles sont la plus forte garantie du point de vue des certifications, comme les critères communs.

La première interrogation que nous pouvons émettre concernant ces méthodes est leur continuelle absence de la très grande majorité des projets de développement de systèmes embarqués. Une première piste de réponse nous amène à penser que ce genre de développement nécessite des compétences accrues en développement et en preuve formelle. De plus, les ressources qui doivent être allouées ne sont pas négligeables. Par exemple, le développement et la réalisation de la preuve du micronoyau seL4[KNS+10] est un travail fastidieux nécessitant un nombre non négligeable de développeurs.

Un autre inconvénient avec la réalisation de preuves formelles d'un système est le manque de flexibilité de ces preuves. Une preuve émise est liée à une spécification et une implémentation. Le moindre changement nécessite de revoir la preuve en partie, voire dans toute son intégralité. Ce qui rend les mises à jour des systèmes et l'ajout de fonctionnalités difficiles et coûteuses.

Une autre interrogation sur les preuves formelles concerne la façon de les appliquer aux systèmes embarqués. Au vu de leur coût sur le développement, il faut concentrer les ressources sur les propriétés les plus importantes du système. Il s'agit alors d'identifier la plus petite base de code à prouver formellement pour garantir les propriétés de sécurité du système embarqué.

La preuve formelle d'un système est une garantie forte de sécurité qui peut être donnée aux différents acteurs du système. À tous les niveaux de l'objet, la preuve formelle doit permettre de créer de la confiance entre les différents composants logiciels. Ses propriétés doivent être propagées dans tout le système.

3.4.4 Validation de commandes et sécurité des communications dans les objets connectés

Nous avons vu dans les sections précédentes des exemples d'attaques et de propagation d'attaques au sein d'objets connectés. Ces attaques se propagent via des lacunes du système global. Au sein d'un objet, un autre moyen de nuire à une application serait de détourner ses fonctionnalités sans forcément porter atteinte à son intégrité.

Un objet connecté peut être muni de mécanismes cryptographiques et de vérification de requêtes depuis son extérieur. Cependant, il se peut qu'en interne, les commandes légitimes entre applications ne soient pas validées par une autorité. Deux applications peuvent communiquer légitimement l'une avec l'autre et effectuer des commandes. Si les applications n'ont pas de mécanismes appropriés, un attaquant peut profiter de cette lacune pour exécuter des commandes non attendues dans les applications et obtenir un avantage en leur sein (figure 3.9).

Sécuriser les communications et la question de la cryptographie au sein de l'objet nous amènent à réfléchir sur la gestion des clés de chiffrement utilisées par les différents composants logiciels de l'objet. La première solution serait de penser que les clés et le mécanisme de chiffrement doivent être gérés au sein même de l'application. Cependant, nous avons vu que plus le code source de l'application est conséquent, plus la surface d'attaque et les sources de failles sont importantes.

De ce fait, si nous souhaitons décomposer les éléments critiques d'une application, ceci nous amène donc à repenser l'architecture logicielle d'un objet connecté. Habituellement, nous voyons les applications comme un bloc unique appartenant en effet à une

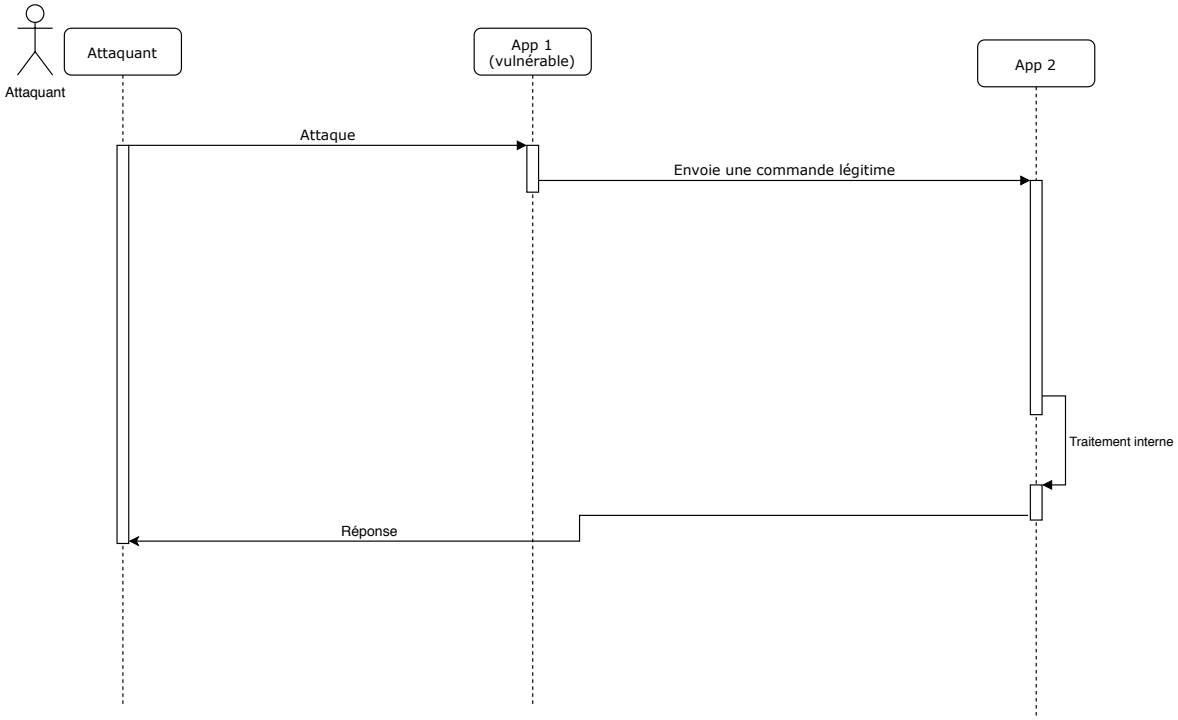

FIGURE 3.9 – Détournement de commandes depuis l'intérieur de l'objet

entité. Mais en ayant une vision plus précise, cette application est composée de plusieurs éléments critiques : banque de clés, mécanisme de chiffrement/déchiffrement ou encore administration de l'application.

Les ressources d'un objet connecté sont limitées (capacité mémoire, CPU, énergie...), ce qui rend la capacité de traitement au sein de l'objet tout aussi limitée. De plus, il peut être facile d'avoir un accès physique à l'objet. Les applications ne profitent pas de la capacité communicante de l'objet pour décentraliser le traitement de requêtes et la validation de commandes alors qu'il est plus aisé de le faire à distance au sein de serveurs plus faciles à protéger.

3.4.5 Cohabitation de systèmes temps-réel et généralistes

Nous avons vu dans les sections précédentes que nous pouvions superposer l'isolation spatiale et l'isolation temporelle pour maintenir les propriétés temps-réel. Cependant, nous avons appliqué cette technique sur une architecture simple à un seul étage de tâches, correspondant à un objet assez simple. Dans le cas où l'architecture de l'objet serait beaucoup plus complexe, les mécanismes temps-réel peuvent être grandement affectés étant donné que le temps d'exécution de chaque tâche est fortement divisé.

Une architecture plus complexe de système embarqué connecté peut intégrer différents types de systèmes d'exploitation. Si nous reprenons le cadre automobile, nous avons de nos jours des systèmes d'exploitation Android utilisés pour l'aspect multimédia du véhicule, cohabitant avec des systèmes temps-réel et critiques (Figure 3.10). Nous pouvons penser que ces deux systèmes fonctionnent au-dessus de la même architecture matérielle. Ces deux systèmes sont tout aussi complexes. Cette cohabitation implique de pouvoir fournir des ressources aux deux systèmes. Dans des situations favorables, les deux systèmes doivent pouvoir fonctionner de manière optimale.

Cette architecture hiérarchique avec différents paradigmes d'ordonnancement amène à revoir les mécanismes temps-réel afin d'y intégrer cette particularité. Pour rappel, un

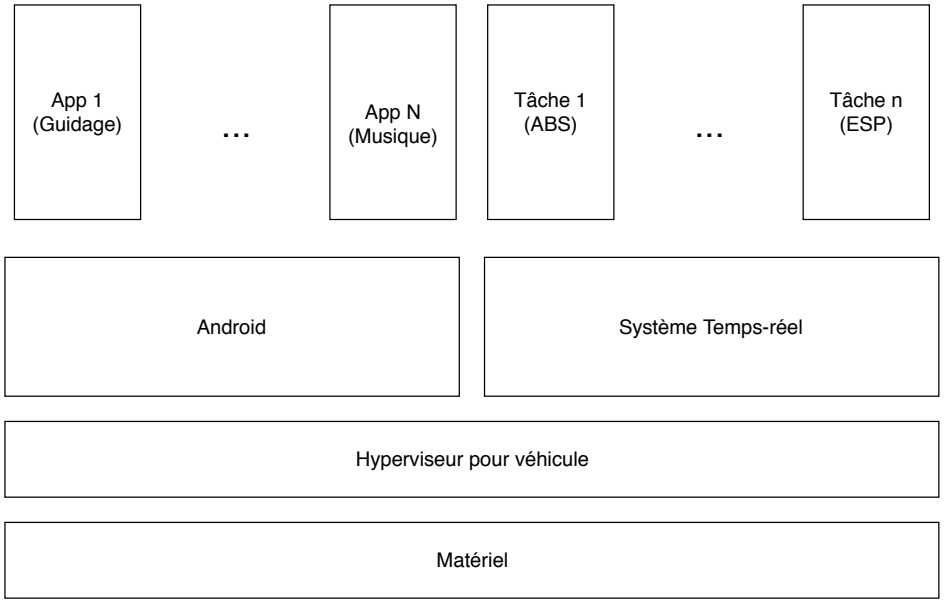

FIGURE 3.10 – Cohabitation d'un système temps-réel critique et d'un système généraliste non-critique

ordonnancement temps-réel de tâches nécessite différentes informations :

— Une échéance
— Une période
— Un pire temps d'exécution (WCET)
— Un niveau de criticité

Dans une architecture complexe comme vu précédemment, si nous souhaitons que la tâche puisse être exécutée dans les conditions attendues, il faut prendre en compte sa position au sein de l'architecture.

3.5 Contribution et conclusion

3.5.1 Axes de contribution

Dans ce chapitre, nous avons essayé d'identifier des problématiques au sein des systèmes embarqués et des objets connectés. Dans cette analyse, nous avons identifié plusieurs types de problématiques.

Positionnement de l'initiateur La première problématique qui nous intéresse est de pouvoir replacer l'initiateur à la base de l'architecture logicielle. Ce dernier est tout en haut de la hiérarchie contractuelle mais, au sein de l'objet, il n'est qu'un utilisateur peu privilégié du système. Toutefois, en cas de problème, il est le principal responsable contractuel. Il s'agira donc de transcrire la vision contractuelle en une vision logicielle de manière à maintenir les mêmes liens hiérarchiques.

Le contrat qui régit l'objet fait intervenir un certain nombre d'acteurs comme nous l'avons vu. Nous allons dans les contributions nous efforcer de placer chaque acteur au sein de l'objet, mais aussi durant le cycle de vie et de production.

Nous définirons son rôle dans l'objet, sa position dans le cycle de production et de vie de l'objet et son degré de responsabilité en cas de défaillance. Tout ceci nous conduira

à mettre en place un écosystème qui aura pour objectif de créer un environnement de confiance pour tous les acteurs.

Spécification et délégation de tâches L'identification exhaustive des acteurs nous amène à les hiérarchiser. Une hiérarchie est définie par un lien d'autorité entre les acteurs mais aussi des droits et tâches possibles pour chacun. Nous voulons que l'initiateur soit toujours à la base de la hiérarchie, mais nous souhaitons aussi qu'il puisse déléguer des tâches afin de réduire la base de confiance. Cette réduction passe d'abord par l'identification, de la manière la plus exhaustive possible, du plus petit ensemble de tâches possibles au sein d'un objet, et qui peuvent influencer les mécanismes de sécurité et de sûreté du système.

Un autre point important consistera à identifier les prérequis minimaux pour maintenir les rôles, les droits et les propriétés de sécurité et de sûreté du système.

Décomposition logicielle des acteurs Nous avons vu qu'un acteur peut être un composant logiciel plus complexe qu'un simple processus système. Nous allons nous efforcer de définir ce qu'est un acteur d'un point de vue logiciel. Nous allons identifier les composants logiciels nécessaires qui le caractérisent et qui permettent de maintenir l'ensemble des propriétés qui lui sont inhérentes. Cette vision nous conduira à avoir un découpage qui permettra de lier le contrat au système final de manière plus précise. Nous allons aussi définir la façon dont se met en place un acteur au sein de l'écosystème et de l'architecture logicielle tout au long de sa durée de vie.

Construction de l'architecture à l'aide d'un mécanisme d'isolation mémoire prouvé Un des points importants que nous avons relevé au cours de l'état de l'art et de son analyse, c'est le besoin constant d'une isolation spatiale. Il s'agira donc de construire une architecture logicielle la plus complète possible à l'aide de mécanismes d'isolation mémoire. De plus, ce mécanisme comporte une preuve formelle sur certaines propriétés de sécurité. Il s'agira aussi de montrer comment ces propriétés formellement prouvées peuvent être propagées entre tous les acteurs et au sein de l'architecture afin de maintenir la confiance entre les acteurs, la sécurité du système et ses composants mais aussi la sûreté de fonctionnement et la disponibilité.

Mécanisme d'autorisation d'actions au sein du système Une problématique de sécurité que nous avons rencontrée aussi concerne la validation des actions au sein du système. Il a paru évident que des actions entre acteurs se doivent d'être validées afin d'éviter tout détournement non envisagé durant les tests et audits. Ce mécanisme de validation profite de la connectivité de l'objet afin d'authentifier les actions auprès d'un tiers en dehors de l'objet. Le but étant de réduire la base de confiance afin de valider le plus aisément possible les propriétés de sécurité et de réduire la surface d'attaque au sein de l'objet.

3.5.2 Conclusion

Tout au long de l'état de l'art et de son analyse, nous nous sommes efforcés d'avoir la vision la plus objective et la plus exhaustive possible sur les systèmes embarqués et les objets connectés. Nous avons essayé d'identifier la manière dont ils sont construits et développés de nos jours. Nous avons et nous allons essayer de répondre à des problématiques les concernant sous l'angle de la sécurité. Pour ce faire, nous allons présenter une

architecture logicielle permettant la conception et la mise en place d'un écosystème embarqué et connecté pour permettre le déploiement d'applications concurrentielles. Cet écosystème se base sur des propriétés fortes de sécurité et de sûreté afin de proposer un environnement de confiance où chaque acteur peut se déployer et fonctionner en ayant des garanties fortes sur ses attentes.

3.6 Références

[AA18] Bako Ali and Ali Awad. Cyber and Physical Security Vulnerability Assessment for IoT-Based Smart Homes. *Sensors*, 18(3) :817, mar 2018. 33

[CWSJ16] Nancy Cam-Winget, Ahmad-Reza Sadeghi, and Yier Jin. Invited - Can IoT be secured. In *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, pages 1–6, New York, New York, USA, 2016. ACM Press. 33

[DCY17] Christian J. D'Orazio, Kim Kwang Raymond Choo, and Laurence T. Yang. Data Exfiltration from Internet of Things Devices : IOS Devices as Case Studies. *IEEE Internet of Things Journal*, 4(2) :524–535, apr 2017. 31

[DGGJ03] Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart Card operating systems : Past Present and Future. In *5th USENIX/NordU Conference (NordU2003)*, Västerås, Sweden, February 2003. 41

[Eri] Eric Hazane. Sécurité numérique des objets connectés, l'heure des choix : : Note de la FRS : : Fondation pour la Recherche Stratégique : : FRS. 33

[Hus01] Dirk Husemann. Standards in the smart card world. *Computer Networks*, 36(4) :473–487, jul 2001. 41

[Kid05] P. Kidwell. The mythical man-month : Essays on software engineering. *IEEE Annals of the History of Computing*, 18(4) :71, 2005. 36

[KNS+10] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4. In *Communications of the ACM*, volume 53, page 107. ACM Press, 2010. 43

[Mar06] Constantinos Markantonakis. The case for a secure multi-application smart card operating system. pages 188–197. Springer, Berlin, Heidelberg, 2006. 41

[RE03] W. (Wolfgang) Rankl and W. (Wolfgang) Effing. *Smart card handbook*. Wiley, 2003. 40

[RZL13] Rodrigo Roman, Jianying Zhou, and Javier Lopez. On the features and challenges of security and privacy in distributed internet of things. *Computer Networks*, 57(10) :2266–2279, jul 2013. 35

[Sau09] Damien Sauveron. Multiapplication smart card : Towards an open smart card? *Information Security Technical Report*, 14(2) :70–78, may 2009. 41

[SGV+15] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. Network-level security and privacy control for smart-home IoT devices. In *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 163–167. IEEE, oct 2015. 36

[She15] Patrick Shelly. Addressing challenges in automotive connectivity : Mobile devices, technologies, and the connected car. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 8(1) :161–169, apr 2015. 33

[SM11] Siddharth Sridhar and G. Manimaran. Data integrity attack and its impacts on voltage control loop in power grid. In *IEEE Power and Energy Society General Meeting*, pages 1–6. IEEE, jul 2011. 31

[Tak18] Yoshiyasu Takefuji. Connected Vehicle Security Vulnerabilities [Commentary]. *IEEE Technology and Society Magazine*, 37(1) :15–18, mar 2018. 33

[ZMR17] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 65–80, Santa Clara, CA, July 2017. USENIX Association. 36

Chapitre 4

Architecture et implémentation de l'écosystème embarqué et connecté

> L'expérience prouve que celui qui n'a jamais confiance en personne ne sera jamais déçu.
>
> Léonard De Vinci

Sommaire

Dans ce chapitre, nous allons présenter l'écosystème des systèmes embarqués et connectés conçus pour assurer des propriétés de sécurité et de sûreté dans un environnement industriel[YGG+18]. Cette architecture provient d'une lecture rigoureuse de l'écosystème de l'Internet des Objets. Ceci nous a permis d'en tirer une classification basée sur des rôles définis et une classification des responsabilités de chaque rôle. De ce travail découlent des droits que nous formalisons à travers une matrice de contrôle d'accès. Nous proposons une implémentation de cette architecture basée sur des mécanismes forts d'isolation mémoire et temporelle.

4.1 Matrice de responsabilité, définition des rôles et des droits

4.1.1 Écosystème industriel de l'IoT

Tout au long de l'état de l'art et de son analyse, nous avons constaté que pour concevoir, fabriquer, gérer et utiliser un système embarqué, cela faisait appel à de nombreuses entités. Elles interviennent à différents moments du cycle de vie de l'objet, en fonction de leurs compétences et de leur apport. Toutefois, en analysant plus précisément les objets, nous avons constaté que lorsque des problèmes de sécurité ou de défaillance surviennent, il devient parfois impossible d'identifier clairement la source. La responsabilité de chaque acteur est diluée, et par défaut, c'est la responsabilité de l'initiateur de l'objet qui est considérée.

Dans un premier temps, si nous résumons tous les acteurs existants et intervenant dans un objet, nous pouvons les regrouper au sein de sept rôles précis. Nous allons donner la définition de chacun de ces rôles. Il faut cependant comprendre qu'un rôle n'est pas nécessairement associé à un seul acteur. Tout comme un seul acteur n'est pas nécessairement présent dans un seul rôle.

Émetteur (Owner en anglais) Nous définissons l'*Émetteur* comme étant l'entité ou les entités qui sont à l'initiative de l'objet. Elles ont la propriété intellectuelle de l'objet. L'Émetteur définit la politique de sécurité et de gestion de l'objet. Pour ce faire, il désigne des entités en charge de produire les différents composants du système. Il est le plus privilégié, il peut effectuer différentes actions au sein du système. Cependant, il doit garder une neutralité dans la gestion de l'objet.

Cette neutralité est définie comme la capacité de ne pas privilégier un acteur en particulier et de maintenir les propriétés de sécurité et de sûreté définies dans la spécification de l'objet.

Assembleur (Packager en anglais) Le rôle de l'*Assembleur* est donné à un seul acteur. Il est en charge d'assembler l'objet. Il est chargé de regrouper tous les composants logiciels et matériels conçus afin de former l'objet final qui sera fourni. De ce fait, il se doit aussi de le mettre dans un état utilisable. Pour maintenir les propriétés de sûreté et de sécurité, il est en charge d'initialiser l'objet avec des paramètres initiaux pour permettre à l'Émetteur d'accéder à l'objet et de déployer sa propre politique de sécurité.

Mainteneur (Maintainer en anglais) Le rôle de *Mainteneur* est donné à au moins un acteur. Ce rôle doit surveiller le fonctionnement des composants matériels et déployer les mises à jour des pilotes fournis par le Producteur.

Producteur (Manufacturer en anglais) Le rôle de *Producteur* est donné à au moins un acteur. Il doit fournir au moins un composant du système. Ce composant peut être matériel (composant WiFi, carte graphique...). Cela peut aussi être un composant logiciel, comme un système d'exploitation (Android, Linux...) ou une solution d'isolation (mémoire ou temporelle). Le Producteur doit fournir les mises à jour des composants dont il est en charge au sein du système. Il est en relation avec le ou les Mainteneurs ainsi que l'Assembleur.

Administrateur (Administrator en anglais) Le rôle d'*Administrateur* de base dans le système est donné par l'Émetteur. Il doit mettre en place la politique de sécurité définie par l'Émetteur au sein de l'objet. Au sein de l'objet, si une modification, comme l'ajout d'une nouvelle entité ou la modification des permissions accordées, est nécessaire, il demande une autorisation à l'Émetteur. Ce rôle peut être assumé ou délégué à un tiers par l'Émetteur.

Un autre point concernant ce rôle. L'architecture étant arborescente, il est possible que chaque entité mette en place un Administrateur pour gérer la politique de sécurité de cette entité.

Fournisseur de services (Service provider en anglais) Un *Fournisseur de services* fournit des services au sein de l'objet ou de la flotte d'objets en utilisant les ressources fournies par l'Émetteur. Il installe, active et met à jour les services au sein de l'objet. Plusieurs Fournisseurs de services peuvent cohabiter au sein de l'objet.

Utilisateur (User en anglais) L'*Utilisateur* utilise les services proposés ci-dessous. Nous pouvons distinguer deux types d'utilisations : un *Utilisateur de la plate-forme* et un *Utilisateur technique*. Un utilisateur de la plate-forme utilise les services proposés par l'objet. Il peut être humain ou une autre machine. L'Utilisateur Technique est un utilisateur issu de l'Émetteur, du Producteur ou du Mainteneur. Il est en charge d'effectuer des actions d'administration telles que la création de domaines (voir 4.1.3), donner des droits à un nouvel utilisateur, charger un service au sein de l'objet.

Dans la figure 4.1, nous avons un exemple d'un cycle de vie d'un objet avec l'intervention de chaque acteur. Certains ont une intervention ponctuelle voire unique (Assembleur), d'autres reviennent dans le cycle avec lequel ils sont constamment présents (Émetteur).

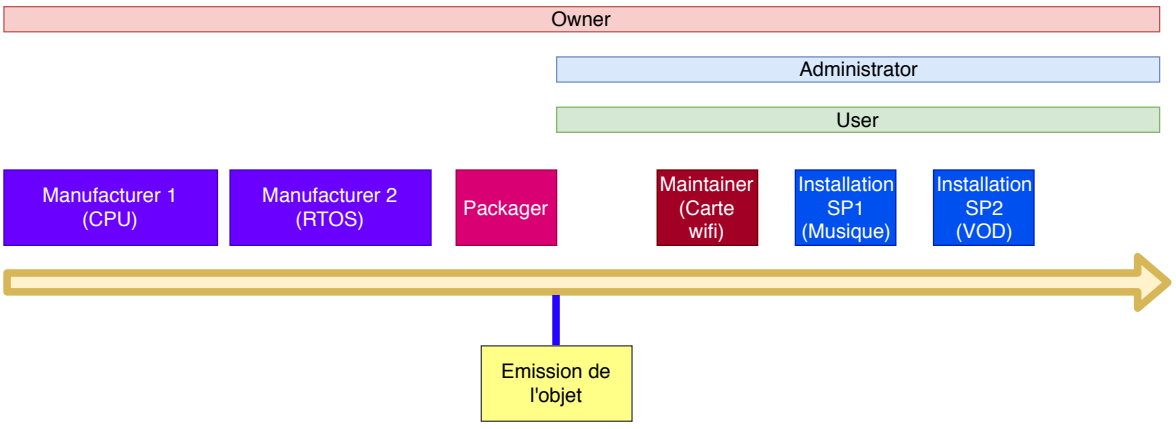

FIGURE 4.1 – Exemple d'un cycle de vie de l'objet avec le détail de l'intervention de chaque acteur

Maintenant que nous avons identifié de manière très exhaustive chaque rôle au sein d'un objet connecté, il s'agira désormais d'identifier, de la même façon, l'ensemble le plus petit de tâches critiques qui peuvent mettre en péril les propriétés de sûreté et de sécurité du système.

4.1.2 Matrice de responsabilité

À travers cette architecture, nous avons comme principal objectif de définir clairement la responsabilité de chaque acteur tout au long de la durée de vie de l'objet. Le but étant de pouvoir retrouver l'auteur de chaque action. Toutefois, le nombre d'actions possibles au sein d'un système est conséquent. Il faut donc dans un premier temps identifier le plus petit ensemble d'actions critiques. Puis dans un second temps, les attribuer à chaque rôle au sein de l'architecture.

Identification exhaustive des responsabilités du système. Pour construire cette liste de responsabilités, nous allons passer d'une vision macroscopique de l'objet et son contexte à une vision plus détaillée. Au vu des rôles déterminés précédemment, nous pouvons les regrouper dans différentes catégories :

— Conception et fabrication de composants
— Administration et configuration
— Mise à jour et maintenance
— Service

Pour rappel, l'objectif de ces responsabilités est de ne pas nuire aux propriétés de sécurité et de sûreté.

Conception et fabrication. Les acteurs qui doivent concevoir et fabriquer les différents composants du système doivent respecter certaines contraintes. Cela implique, en plus de fournir les composants (matériel, pilotes, OS...), qu'ils sont dans l'obligation de ne pas y intégrer des failles volontaires, des *backdoors* notamment. De plus, ayant une parfaite connaissance de leurs produits, ils doivent aussi fournir les mises à jour et correctifs pour le bon fonctionnement du système.

Administration et configuration. La première configuration de l'objet doit intervenir juste avant son émission et sa livraison à l'utilisateur. Pour cela, l'acteur en charge doit avoir réuni tous les composants, doit avoir configuré l'objet dans un état de fonctionnement et doit fournir un moyen de personnaliser l'objet, notamment modifier les accès.

Durant la durée de vie de l'objet, l'administrer et le configurer implique de pouvoir gérer et appliquer une politique de sécurité. Créer, supprimer des domaines ainsi que gérer les accès aux ressources.

Mise à jour et maintenance. Cette responsabilité oblige les acteurs à maintenir le système à jour pour tous ses composants. Ils ont des mécanismes au sein de l'objet qui doivent récupérer et appliquer tous les correctifs fournis par les différents Producteurs. De plus, ils se doivent de maintenir et de garantir des accès aux ressources qu'ils gèrent.

Service. Un service a pour responsabilité d'être à jour pour le bon fonctionnement du système, gérer sa politique de sécurité et les accès à ses ressources.

Attribution des responsabilités. Nous avons nos différentes catégories d'acteurs. Nous avons mis en place une série de responsabilités, et nous allons nous efforcer de les distribuer entre tous les acteurs. Dans le tableau 4.1, nous présentons la responsabilité de chaque rôle au sein de l'objet.

Responsabilité	Manufacturer	Packager	Maintainer	Owner	Administrator	Service Provider
Fournit les différents composants (Matériel, pilotes, OS...)	X					
Regroupe les différents composants en un objet		X				
Fournit un mécanisme d'accès concurrentiel aux ressources		X				
Fournit des mécanismes et des accès temporaires pour personnaliser l'objet avant livraison au Owner		X				
Fournit des accès temporaires pour personnaliser l'objet après livraison au Owner		X		X	X	
Met à jour régulièrement les certificats d'accès à son domaine			X	X	X	X
Fournit des composants au Owner sans backdoor	X	X				
Créer / Modifier / Supprimer un domaine avant livraison au Owner		X				
Créer / Modifier / Supprimer un domaine après livraison au Owner				X	X	
Configure la politique d'accès aux ressources				X	X	
Assure le respect de la politique d'accès aux ressources				X	X	
Configure sa politique d'accès à ses ressources			X	X	X	X
Assure le respect de sa politique d'accès à ses ressources			X	X	X	X
Fournit des mises à jours et des correctifs	X					
Récupère et déploie les mises à jours			X			
Maintient le service à jour						X

TABLEAU 4.1 – Matrice de responsabilité

4.1.3 Architecture et modèle de sécurité

Définition d'un domaine et d'une partition Nous avons vu tout au long de l'état de l'art et de son analyse qu'un acteur au sein d'un objet n'est pas seulement défini par un ensemble d'applications et de codes sources. Dans notre architecture, nous voyons qu'un objet est partagé entre plusieurs entités qui ont chacune un rôle.

Nous définissons donc :

— Une partition est un code source avec son propre espace d'adressage et appartenant à une entité.

— Un Domaine est l'union de toutes les partitions développées et gérées par une seule entité.

Dans la figure 4.2, nous avons un exemple simplifié du découpage mémoire d'un objet suivant l'architecture. Le domaine racine étant celui de l'Émetteur. Ce dernier découpe son espace d'adressage en plusieurs espaces pour les différents rôles. et chaque domaine peut aussi être subdivisé.

Composition d'un domaine. Afin de garantir le modèle de sécurité dans notre architecture, chaque domaine doit contenir un certain nombre de composants prérequis :

1. Configuration Manager : il contient une description de toutes les partitions du domaine ainsi que les canaux de communication entre les partitions.
2. Internal Communicator : il est l'équivalent au routeur pour un réseau. Il est l'unique point d'entrée de messages du domaine. Il joue aussi le rôle de pare-feu pour bloquer tout message non désiré.
3. Adminstration Manager : ce module expose les ressources du domaine à un serveur externe géré par l'entité propriétaire du domaine. Il achemine les commandes de lecture, écriture et d'exécution vers le bon capteur ou actionneur. Chaque commande reçue est transmise au Token and Security Validator.

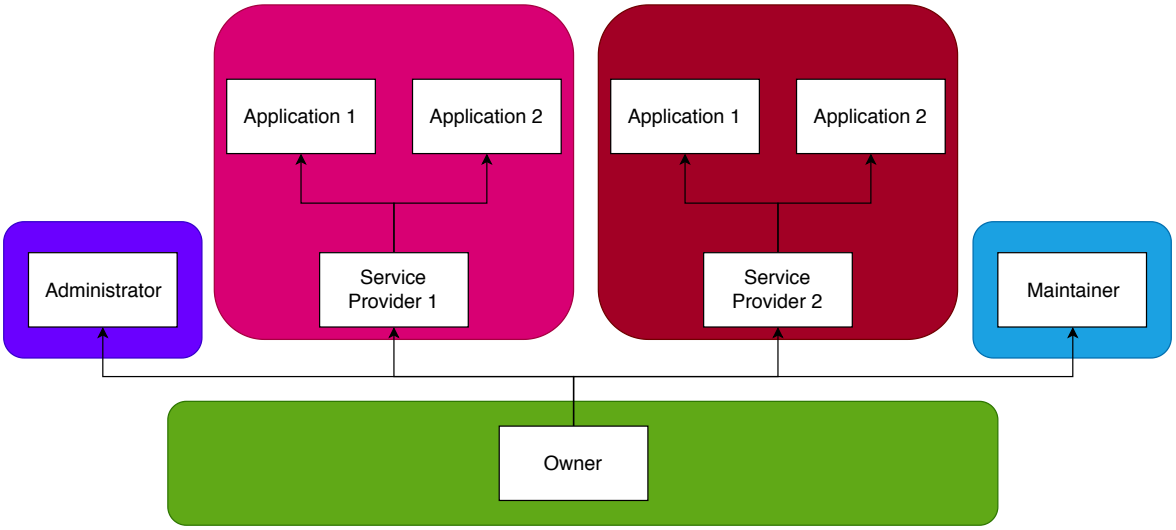

FIGURE 4.2 – Exemple d'un objet avec un découpage en domaines
Les rectangles blancs sont des partitions, et les rectangles de couleurs sont des domaines

4. Token and Security Validator : il valide chaque commande de gestion de périphérique par rapport au jeton fourni dans la commande et, en option, une politique de sécurité interne. Par exemple, la politique de sécurité définit quelles ressources nécessitent une vérification de jeton pour être accessibles ou si une commande spécifique peut être traitée en fonction de l'état de la batterie du périphérique.
5. Key Vault : ce module stocke les clés utilisées par le domaine. En particulier, les clés utilisées par le Token and Security Validator sont stockées ici. Il fournit également les fonctions permettant d'ajouter une nouvelle clé, de modifier ou de supprimer une clé existante.
6. Gestionnaire de Capteurs et d'actionneurs : ces gestionnaires fournissent un support synthétique aux capteurs et actionneurs virtuels par l'intermédiaire du module de communication interne.
7. Capteurs et actionneurs virtuels : seul l'Émetteur est en capacité d'interagir avec le matériel ou les pilotes. Les capteurs et actionneurs virtuels sont des instances permettant au domaine d'accéder à une instance du matériel. Les instructions et réponses sont transmises à travers le mécanisme de communication.

Dans la figure 4.3, nous avons un exemple d'un objet et le découpage des domaines avec les différents composants. Les rectangles pleins sont des partitions, en pointillé, des composants logiciels dans le même espace d'adressage.

4.1.4 Pré-requis minimaux pour construire l'architecture

L'architecture a pour objectif de répondre à une problématique de sécurité dans l'IoT. Pour ce faire, nous devons imposer des contraintes pour l'implémentation.

Isolation des domaines et des partitions Afin de maintenir les propriétés de sécurité et de sûreté, nous devons mettre en place dans l'architecture un mécanisme d'isolation de domaines. Deux domaines distincts appartiennent à deux entités distinctes. De ce fait, Ces deux domaines ne doivent pas accéder à la mémoire de l'autre, obtenir des informations et/ou interférer dans le comportement de l'autre.

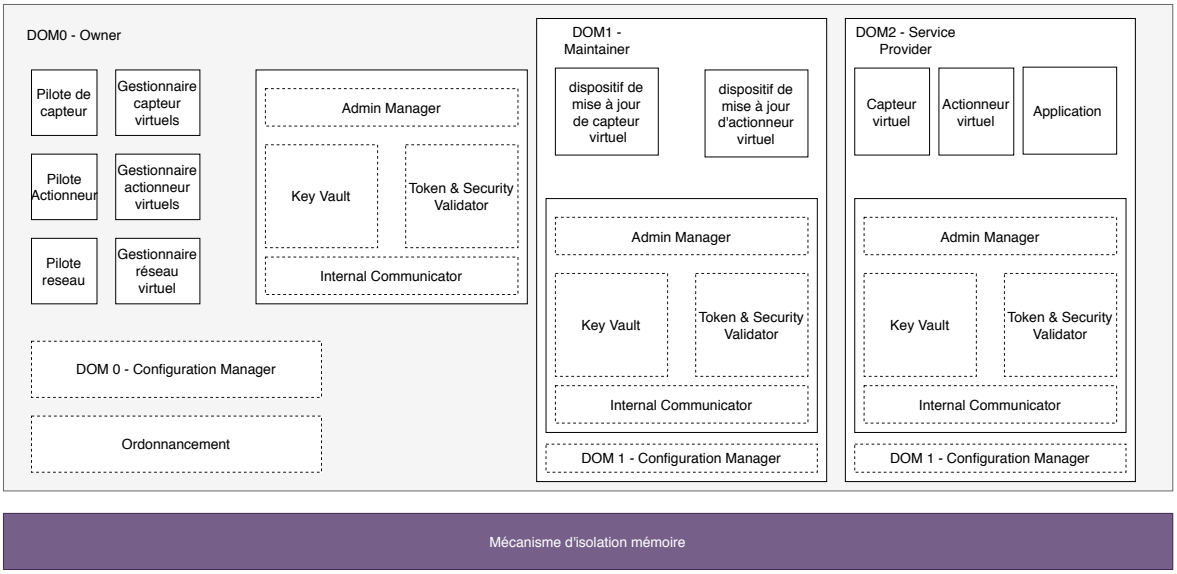

FIGURE 4.3 – Découpage détaillé des domaines

Non-Interférence de l'Émetteur Le domaine de l'Émetteur (ou de l'Administrateur) est à la racine de l'objet et a un accès et une utilisation privilégiée de l'objet. De ce fait, ces deux rôles doivent garder une neutralité au sein de l'objet comme nous l'avons spécifié. Ils ne doivent avoir aucun contrôle ou visibilité sur les actions effectuées par les autres domaines, sauf si ces actions requièrent des ressources gérées par l'Émetteur.

Contrôle d'accès aux ressources Chaque ressource au sein de l'objet, exposée entre les domaines, doit être soumise à une politique d'accès définie par le domaine propriétaire de la ressource. Pour les ressources matérielles, elles sont sous l'égide de l'Émetteur ou du domaine mandaté pour sa gestion.

Minimiser le traitement des autorisations Chaque autorisation spéciale effectuée dans un domaine doit faire l'objet d'une vérification de sa légitimité auprès d'une autorité extérieure à l'objet (serveur distant). Chaque entité est responsable de la gestion et des autorisations d'accès des ressources de son propre domaine. L'Émetteur, du fait qu'il soit privilégié au sein de l'objet, est aussi en capacité de donner des autorisations. Cependant, pour des raisons de neutralité, son action doit être limitée.

4.1.5 Mécanismes de gestion

Notre architecture est destinée à des environnements d'objets connectés, pour lesquels un accès physique à l'objet n'est pas toujours possible. Ceci implique de mettre en place un mécanisme de gestion à distance de l'objet et de ses services ne mettant pas en péril les propriétés de sécurité.

Gestion des ressources des domaines. Pour illustrer la manière dont s'effectuent les instructions de gestion de domaines, nous allons utiliser un exemple. Dans un objet, nous devons changer l'adresse du serveur de récupération des mises à jour pour un composant. Cette action, étant sensible, doit être autorisée au préalable. Et c'est de la responsabilité du Mainteneur de s'assurer du bon déroulement de cette modification :

Déroulé de la commande et son authentification (figure 4.4) :

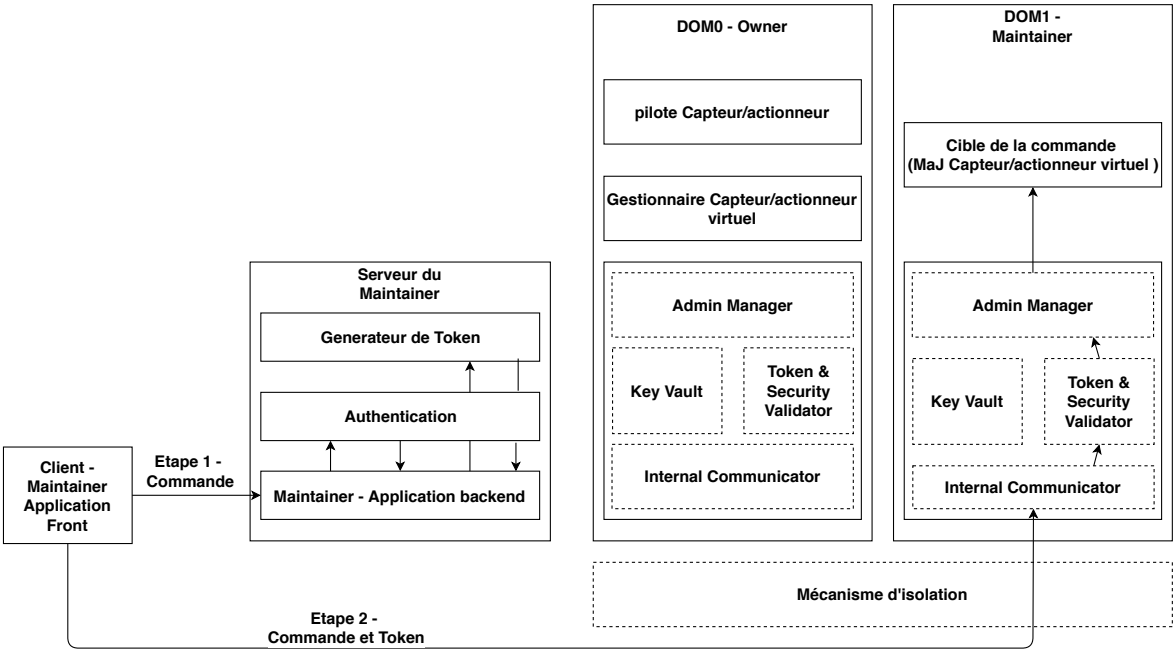

FIGURE 4.4 – Gestion de ressource dans un domaine

1. Le Client envoie la commande à valider au serveur,
2. Le serveur authentifie la demande et génère un Token unique associé à cette commande,
3. Le Client récupère le Token et envoie le couple (Token,Command) à l'objet,
4. Le mécanisme de communication transmet la requête au domaine Mainteneur,
5. Le Token & Security Validator valide le token,
6. Si le token est validé, la commande est transmise à l'Admin Manager qui la transmet au bon composant.

Gestion des ressources de l'objet De la même manière, nous allons illustrer le processus de gestion de ressources de l'objet via un exemple. Un Mainteneur cherche à charger, déployer et installer un nouveau pilote pour une carte WiFi. Dans ce cas, le Client doit obtenir un token auprès du serveur lié au domaine du Mainteneur de la carte WiFi et auprès de l'Émetteur, étant donné que la commande modifie des composants logiciels gérés par l'Émetteur.

Déroulé de la commande et son authentification (figure 4.5) :

1. Le Client envoie la commande à valider au serveur du Mainteneur,
2. Le serveur transmet la commande au serveur de l'Émetteur,
3. En parallèle, les deux serveurs valident et fournissent deux Tokens A et B,
4. Le Client récupère les tokens et transmet la commande et les tokens à l'objet,
5. L'internal Communicator transmet la commande au domaine du Mainteneur,
6. Le Mainteneur valide le Token A et la commande,
7. Le Mainteneur transmet la commande et le Token B au domaine de l'Émetteur,
8. Si le Token B est valide, la commande est exécutée.

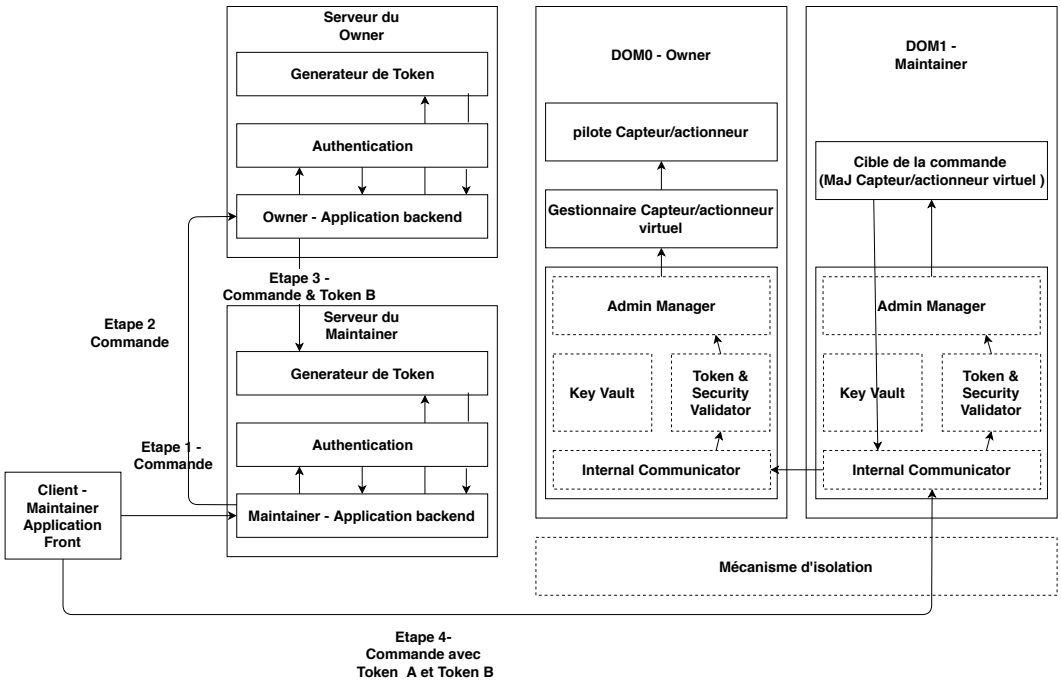

FIGURE 4.5 – Gestion de ressource dans un objet

4.1.6 Communication et autorisations

Le mécanisme de communication au sein de l'architecture doit être superposé aux mécanismes de sécurité déjà en place (Isolation des partitions et des domaines).

Communication interne de l'objet L'architecture arborescente mise en place grâce aux mécanismes d'isolation mémoire bloque toute communication entre deux domaines ou deux partitions sans un lien direct de parent-enfant. Chaque domaine a un composant *Internal Communicator* comme précisé dans les sections précédentes. Le mécanisme de communication interne à un domaine est de la responsabilité du propriétaire du domaine, qui est libre de l'implémenter suivant l'architecture de son domaine. Deux cas de figures se présentent.

Le cas le plus simple, si tous les composants du domaine sont dans le même espace d'adressage, alors les composants partagent la même mémoire et peuvent communiquer entre eux via des mécanismes comme les files ou les tubes par exemple.

Dans le cas où tous les composants du système sont des partitions, pour que deux partitions A et B puissent communiquer, le parent doit servir d'intermédiaire. A envoie le message au parent. Puis ce dernier le transmet à B. Ce mécanisme est basé sur l'échange de page mémoire entre un parent et son enfant possible grâce au mécanisme d'isolation mémoire. Le parent s'assure que les deux partitions sont légitimes pour communiquer entre elles (responsabilité de l'Internal Communicator) (figure 4.6).

Pour la communication entre domaines isolés, le principe est récursif, chaque domaine vérifie la légitimité de la communication et transmet le message à son parent ou enfant destinataire (figure 4.7).

Communication externe de l'objet Pour une communication extérieure à l'objet, seule une partition est en charge de la carte réseau. Elle fournit une abstraction d'envoi et de réception de paquets. Chaque domaine communique avec cette partition via le mécanisme de communication interne décrit auparavant.

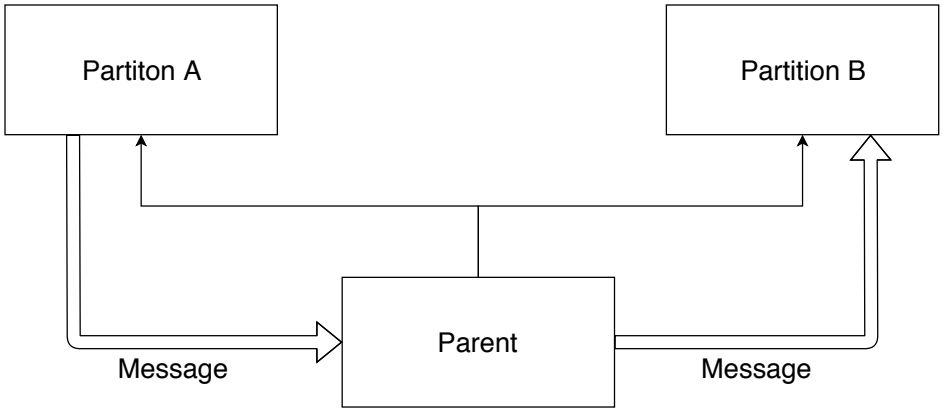

FIGURE 4.6 – Communication entre deux partitions

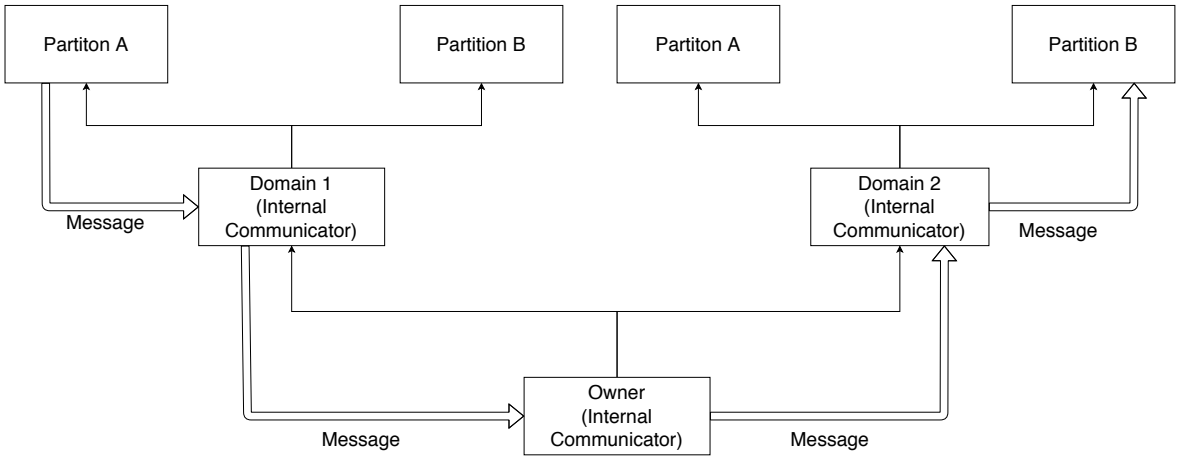

FIGURE 4.7 – Communication entre deux partitions dans deux domaines distincts

Gestion de clé des domaines. Chaque domaine est responsable de ses propres clés de chiffrement. Il doit s'assurer de leur intégrité en utilisant les mécanismes d'isolation mémoire et en les sauvegardant au sein du *Key Vault*. Ce dernier fournit des fonctions d'ajout, de modification et de suppression de clés. Le propriétaire du domaine doit s'assurer que seul le Token & Security validator peut accéder aux clés.

4.2 Mécanisme d'isolation spatiale et temporelle

4.2.1 Isolation mémoire de Pip

Pip[Jom18][Ber19] est un proto-noyau d'hypervision développé au sein de l'équipe 2XS. Il a pour objectif de fournir une isolation mémoire formellement prouvée. Il est composé d'environ 1500 lignes de code (C et Assembleur). Le modèle et la preuve de l'isolation mémoire est écrite en Gallina et prouvé à l'aide de l'assistant de preuve Coq. Seul Pip est exécuté en mode privilégié. Pip manipule des partitions.

Une partition est un espace d'adressage virtuel pouvant contenir du code source.

Pip permet la construction d'une TCB hiérarchique (*Trusted computing base*), sous la forme d'un arbre de partitions. Chaque partition peut subdiviser sa mémoire pour créer des sous-partitions (figure 4.8).

Pip est minimal, il ne propose aucun mécanisme d'ordonnancement, de communication. Il ne fournit que deux fonctionnalités, l'isolation mémoire et le changement de

contexte.

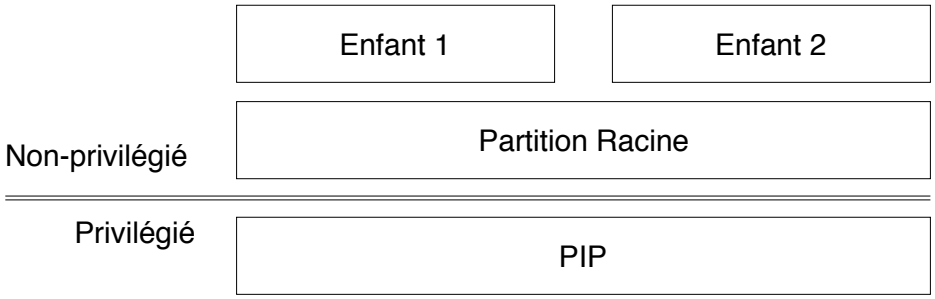

FIGURE 4.8 – TCB hiérarchique sous PIP

Mécanisme d'isolation de Pip L'isolation mémoire de Pip est basée sur le respect de deux propriétés (figure 4.9) :

— Isolation horizontale : Deux partitions distinctes (enfant 1 et enfant 2 dans la figure 4.8), ne peuvent pas partager les mêmes pages mémoire.

— Partage vertical : Toutes les pages utilisées par une partition enfant, sont adressées dans le parent.

Pip s'assure qu'à chaque instant de l'exécution du système, ces deux propriétés sont respectées.

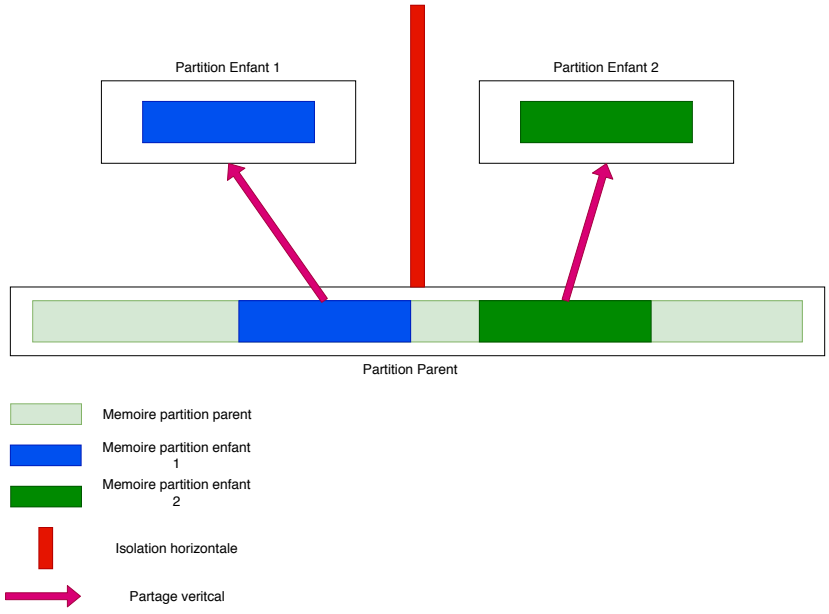

FIGURE 4.9 – Illustration des propriétés de sécurité de Pip

L'arbre des partitions Le mécanisme d'isolation de Pip est récursif. Comme dit précédemment, chaque partition peut subdiviser sa mémoire pour construire d'autres sous-partitions. Suivant les deux propriétés d'isolation de Pip, nous pouvons construire ce qu'on appelle un arbre de partitions. Le nœud à la base est nommé *partition racine*. Chaque nœud de l'arbre est une partition avec son propre espace d'adressage virtuel (figure 4.10).

Seule la partition racine est adressée en 1 :1 (adresse virtuelle = adresse physique), elle détient toute la mémoire rendue disponible par Pip.

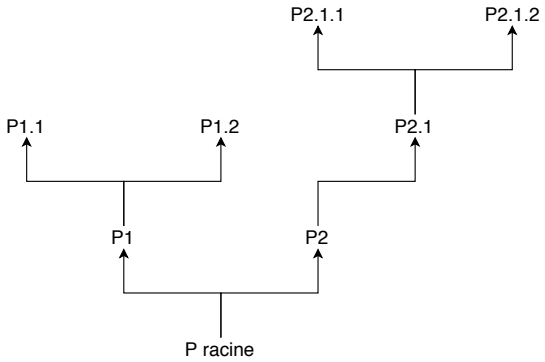

FIGURE 4.10 – Arbre de partitions de Pip

Changement de contexte et mécanisme d'interruption. Pour la gestion des interruptions, Pip possède deux comportements suivant le type d'interruption :

— Interruptions logicielles : Pip relaie l'interruption au parent de la partition l'ayant déclenchée

— Interruptions matérielles : Pip relaie l'interruption à la partition racine.

Comme stipulé précédemment, Pip n'a pas de mécanismes d'ordonnancement. Cependant, il fournit des appels permettant de changer le contexte courant d'exécution. Ces deux fonctions sont :

— dispatch : Envoie une interruption à une partition

— resume : restaure une partition interrompue.

Toute partition peut envoyer ou recevoir des signaux. Pour cela, il faut un lien direct parent-enfant.

Chaque partition possède un vecteur d'interruptions virtuelles (vIDT). Quand une partition active les interruptions, à l'image de l'IDT pour les processeurs intel x86, pour chaque entrée de ce vecteur est associé un gestionnaire d'interruptions.

***Pip calls*, les appels systèmes de Pip.** En plus des deux fonctions de gestion de contexte, pour la gestion des partitions, Pip fournit une série de fonctions, pour lesquelles la preuve formelle est réalisée. Certaines fonctions sont nécessaires à la gestion interne des partitions, d'autres sont utiles pour la gestion des pages des partitions. La preuve formelle est réalisée sur toutes les fonctions manipulant les structures importantes à l'isolation mémoire :

— CreatePartition : Créer une nouvelle partition enfant

— DeletePartition : Supprime une partition enfant

— PageCount : Retourne le nombre de pages nécessaires pour l'adressage une page de l'enfant

— Prepare : Prépare une partition enfant à recevoir les pages

— Collect : Récupère les pages de configuration non utilisées

— AddVaddr : adresse une page dans une partition

— RemoveVaddr : enlève une page d'une partition

4.2.2 Isolation temporelle : FreeRTOS

Dans le développement de l'architecture, nous avons fait le choix d'intégrer des propriétés temps-réel. Le choix s'est porté sur FreeRTOS[fre] pour divers arguments :

— FreeRTOS est open source

— FreeRTOS est un système d'exploitation temps-réel très répandu dans le domaine académique et industriel

— Facile à porter au-dessus d'une nouvelle architecture (un guide étant disponible)

— Les propriétés temps-réel sont reconnues dans le domaine

— Faible empreinte (l'image binaire est de 4 à 9 kiO).

FreeRTOS est un RTOS (Real-Time Operating System) avec un ordonnancement basé sur des tâches à priorités. La tâche avec la priorité la plus élevée est exécutée en premier. Si deux tâches ont la même priorité, il est possible d'adopter le *round-robin*.

Les tâches sont définies par des *Task control Block*, une structure regroupant les informations nécessaires à leur gestion. Les tâches sous FreeRTOS peuvent exister sous 5 états distincts "supprimée", "suspendue", "prête", "bloquée" ou "en exécution" (figure 4.11).

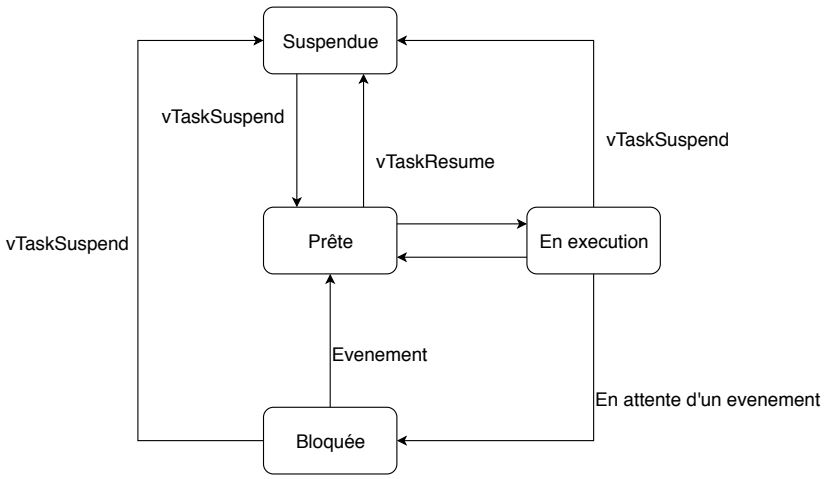

FIGURE 4.11 – État de fonctionnement des tâches

L'API de FreeRTOS contient un ensemble de fonctions pour modifier l'état d'exécution des tâches. Ces dernières peuvent aussi voir leurs états modifiés par des évènements ou des ressources non disponibles. FreeRTOS intègre aussi des mécanismes de communications, des listes, des sémaphores et des mutex, ainsi qu'un allocateur de mémoire.

Toutefois, FreeRTOS est un système qui s'exécute directement sur la mémoire physique. Il n'existe aucun mécanisme de virtualisation ou de protection de la mémoire. Les tâches et le noyau de FreeRTOS s'exécutent avec le même niveau de privilèges, rendant le système performant, mais sans aucune garantie de sécurité.

Dans la suite des travaux, nous nous sommes basés sur la version 9 de FreeRTOS.

4.2.3 Construction de l'isolation de l'objet

Dans l'état de l'art et son analyse, nous avons constaté qu'il existe deux types d'isolations dans un système. L'isolation spatiale (partage de la mémoire et des ressources) et l'isolation temporelle (partage du temps d'exécution). Ces deux propriétés sont la base de la sécurité, de la sûreté et de la fiabilité d'un système. Nous avons à notre disposition un

proto-noyau avec de fortes garanties d'isolation mémoire ainsi qu'un RTOS qui propose une isolation temporelle temps-réel.

Pip fournit tous les mécanismes nécessaires au portage d'un système existant en tant que partition. Dans la suite, nous allons montrer comment nous avons porté FreeRTOS en tant que partition racine de Pip et la façon dont nous avons transformé son API afin d'intégrer le mécanisme de partitions et d'isolation mémoire.

Portage et propagation de l'isolation spatiale Nous avons vu que pour maintenir les propriétés d'isolation temporelle, il faut maintenir l'intégrité de différents éléments :

— Code source du noyau du RTOS

— Mécanisme de gestion des tâches

— Structures de description des tâches

— Ressources partagées entre les tâches

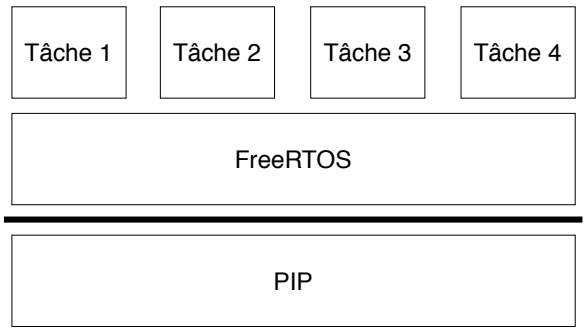

FIGURE 4.12 – Illustration du portage de FreeRTOS en tant que partition racine

L'objectif du portage est d'avoir une architecture comme dans la figure 4.12. FreeRTOS est doté de deux couches. Une couche dépendante du matériel et une autre indépendante. Pour pouvoir utiliser FreeRTOS en tant que partition racine, il faut mettre en place une couche non-privilégiée permettant de faire le lien avec Pip, son API et le matériel.

Pip procède à son initialisation et sa configuration, puis prépare la partition et fournit un environnement de déploiement à la racine défini comme ceci :

— Pip initialise et met à disposition pour la partition racine toute la mémoire non-utilisée et adressée en 1 :1

— Donne accès à l'intégralité du matériel soit par des appels en utilisant les I/O ports, ou en adressant les adresses de la MMIO dans l'espace de la racine.

— Tous les appels de l'API de Pip sont accessibles par la partition via des far calls.

— Prépare la pile d'exécution de la partition racine

Une fois l'installation terminée, Pip démarre la partition racine en envoyant une interruption virtuelle de numéro 0. Ceci aura pour effet d'effectuer un branchement à une entrée fixe, le point d'entrée de la partition racine. Ce qui met un terme à la procédure de démarrage de la partition racine (FreeRTOS dans la suite) et lui donne la main. Pour faciliter les portages, Pip fourni une librairie d'abstraction appelée LibPip, permettant de faciliter l'utilisation de l'API.

À partir de l'instant auquel FreeRTOS est exécuté, son initialisation et sa configuration sont similaires à un fonctionnement habituel au dessus de n'importe quel matériel, dont voici la procédure de base pour avoir un FreeRTOS fonctionnel :

1. **Initialiser l'allocateur de mémoire fourni par Pip :** Pour faciliter la gestion de la mémoire fournie par Pip, ce dernier donne des informations à la partition racine. Ces informations fournissent le début et la fin de la zone mémoire disponible. FreeRTOS peut alors utiliser la librairie Pip pour faciliter son initialisation et son utilisation.
2. **Initialiser les interruptions :** La gestion des interruptions d'une partition se fait via une table virtuelle de gestionnaire d'interruptions (vIDT). La libpip fournit une abstraction à cette table. FreeRTOS initialise un gestionnaire par type d'interruption.
3. **Initialiser le matériel au besoin :** Pip ne configure que ce dont il a la nécessité. De ce fait, si un matériel est nécessaire à son fonctionnement ou au fonctionnement de l'environnement, FreeRTOS possède les droits pour l'initialiser.

Une fois cette initialisation effectuée, le système FreeRTOS est dans l'état *IDLE*.

Le FreeRTOS que nous avons modifié n'est pas uniquement destiné à fonctionner en tant que partition racine. Nous l'avons aussi adapté afin qu'il puisse être utilisé à n'importe quel niveau de l'architecture, permettant de l'utiliser pour implémenter des tâche-partitions plus complexes qu'une simple fonction.

Création de tâches-partitions Une fois que le FreeRTOS est initialisé, nous devons être capables de créer des tâches. Au sein de notre FreeRTOS modifié, nous avons mis en place deux types de mécanismes : Des tâches normales ainsi que des tâches-partitions.

Nous pouvons toujours créer des tâches normales au sein du même espace d'adressage avec la fonction xTaskCreate(). Cependant, nous désirons profiter du mécanisme d'isolation mémoire de Pip pour isoler des tâches sensibles. De ce fait, nous avons étendu l'API de FreeRTOS avec l'ajout de la fonction xTaskCreateProtected().

La structure de données décrivant la tâche-partition est similaire à la TCB déjà existante. Nous l'avons étendue avec des informations supplémentaires pour la gestion et l'ordonnancement (type de tâche, vIDT, adresse du point d'entrée). Une tâche-partition créée avec cette fonction est intégrée à la liste des tâches prêtes à être exécutée.

Ordonnancement des tâches-partitions Pour l'ordonnancement des tâches-partitions, il a paru évident qu'il ne fallait pas modifier le processus de choix de la prochaine tâche à exécuter pour ne pas remettre en question tout l'ordonnanceur de FreeRTOS.

Le mécanisme d'ordonnancement de FreeRTOS gère des listes de tâches en fonction de leur état et de leur priorité. Dans un FreeRTOS normal, la liste des tâches est composée de pointeur vers la pile d'exécution de la tâche. Quand FreeRTOS choisit une nouvelle tâche, il se branche à l'adresse déterminée par le pointeur. Dans FreeRTOS, le pointeur *pxCurrentTCB* désigne à tout instant la tâche en court d'exécution. À chaque ordonnancement, FreeRTOS remplace la valeur de ce pointeur par la nouvelle tâche à exécuter.

Sous Pip, une partition est définie par différentes informations uniques. L'une d'elles, qu'on appelle le descripteur de partition, est utilisée par les fonctions dispatch() et resume() pour exécuter ou restaurer ladite tâche.

Pour pouvoir intégrer ces tâches-partitions dans les listes gérées par FreeRTOS, nous faisons passer ce descripteur de partition pour un pointeur vers une pile. Mais lors du changement de contexte, nous vérifions le type de la tâche. Si c'est une tâche normale, nous laissons procéder à un changement de pile normal. Mais si c'est une tâche-partition, nous détournons le changement de contexte vers des appels à dispatch() ou resume() en fonction de l'état de la tâche (figure 4.13).

Mécanisme de communication entre tâches-partitions et de partage de données FreeRTOS intègre déjà un mécanisme de communication basé sur des files d'attente. Les files sont définies par deux paramètres :

— Longueur : nombre maximal de messages simultanément

— Taille d'un élément : la taille, en octet, de chaque message.

Quand les tâches envoient un message, cela ajoutent du contenu qui respecte ces conditions. La lecture supprime le message de la file.

De la même manière que pour le mécanisme d'ordonnancement, nous avons fait le choix d'étendre ce mécanisme et y intégrer la communication entre tâches-partitions.

Le mécanisme est basé sur les interruptions logicielles de Pip et du partage vertical. L'appel Pip_Notify() de la libpip (ou dispatch() dans Pip) permet de transférer à la partition destinataire deux informations :

```
Pip_Notify(DESTINATAIRE,Numero_interruption,donnee1,donnee2);
```

Pour la création de la file, deux possibilités coexistent :

— Création directe dans FreeRTOS avec xQueueCreate(). L'identifiant de la file est donné aux tâches-partitions.

— Création par une tâche-partition avec xProtectedQueueCreate(). L'identifiant de la file est fourni aux tâches-partitions autorisées.

La figure 4.14 résume le comportement du mécanisme implémenté. La partition racine a créé une file (queue). Tâche-partition 1 et Tâche-partition 2 ont connaissance de cette file. Avec les appels à l'API de pip-freertos (xProtectedQueueSend()), la tâche-partition 1 envoie un message avec comme données la page mémoire contenant le message. FreeRTOS copie le contenu de cette page dans la file. Quand la tâche-partition 2 sollicite la file via l'API (xProtectedQueueReceive()), FreeRTOS copie le contenu de la file vers la page mémoire fournie.

L'implémentation de ce mécanisme de communication préserve le même nombre et le même type de paramètres que dans les fonctions de FreeRTOS de base. Cela afin de ne pas modifier radicalement le comportement des files.

API de pip-freertos. Afin de faciliter l'utilisation des modifications mise en place, nous avons développé une API basée sur le mécanisme d'interruption de Pip. Nous avons implémenté une série de services au sein de FreeRTOS disponibles via des appels à dispatch() ou Pip_Notify().

Au numéro d'interruption 0×80, nous avons associé le mécanisme de services. Depuis une tâche-partition, nous pouvons y faire appel comme ceci :

```
Pip_Notify(0,0x80,NumeroService,parametreService);
```

Dans le tableau 4.2, à chaque identifiant, est associé un service. Les paramètres associés à un service sont transmis au sein d'une page mémoire à FreeRTOS. Pour chaque service, FreeRTOS vérifie la légitimité de l'appel, récupère les paramètres, effectue le traitement et redonne la main avec le résultat à la tâche-partition.

Numéro d'interruption	Service
0x15	Création d'une file
0x16	Envoi de message sur une file
0x17	Réception d'un message depuis une file
0x18	Demande de mémoire supplémentaire
0x19	Demande d'accès à une queue
0x20	Demande d'accès à un matériel
...	

TABLEAU 4.2 – Tableau de services

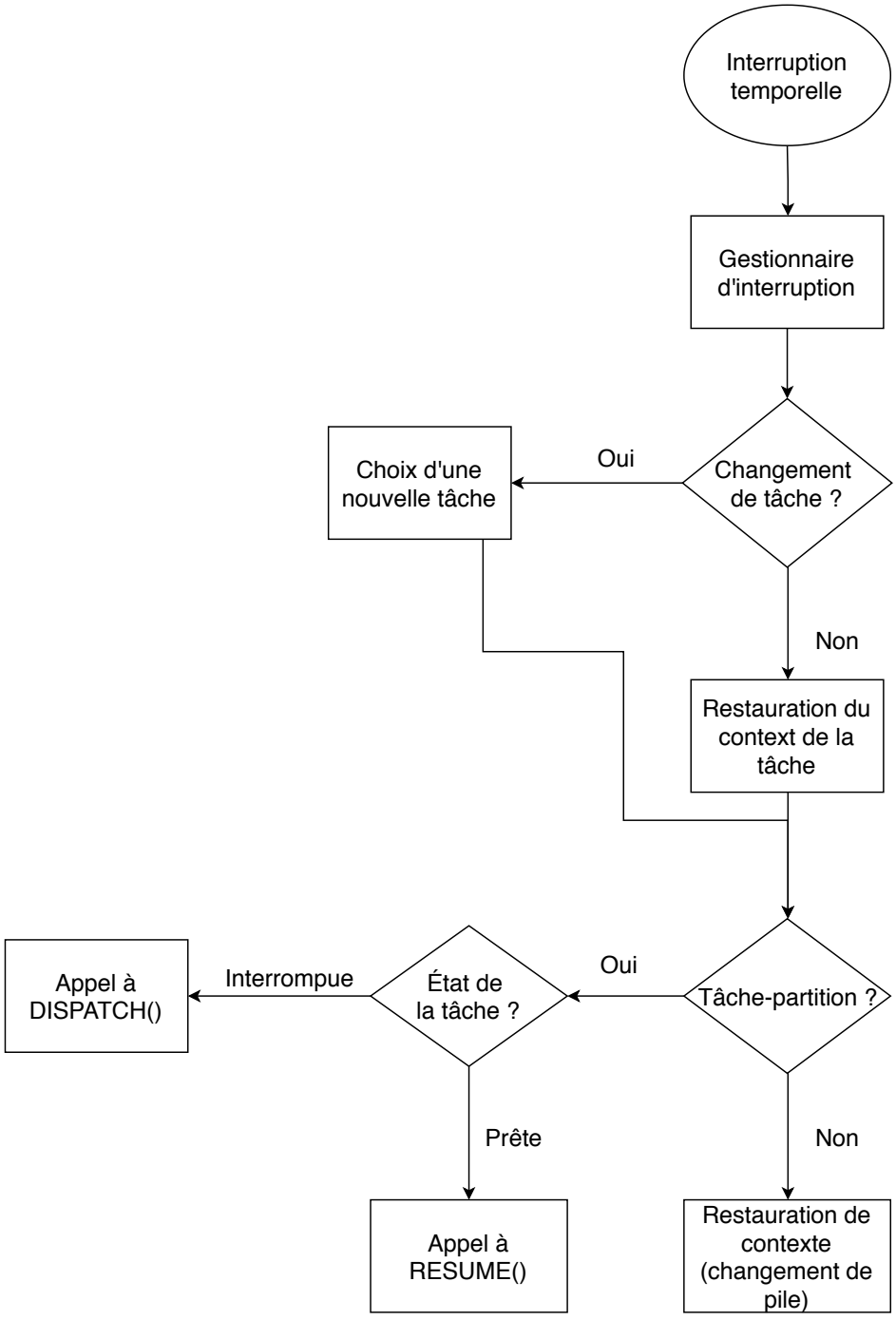

FIGURE 4.13 – Mécanisme d'ordonnancement des tâches et des tâches-partitions

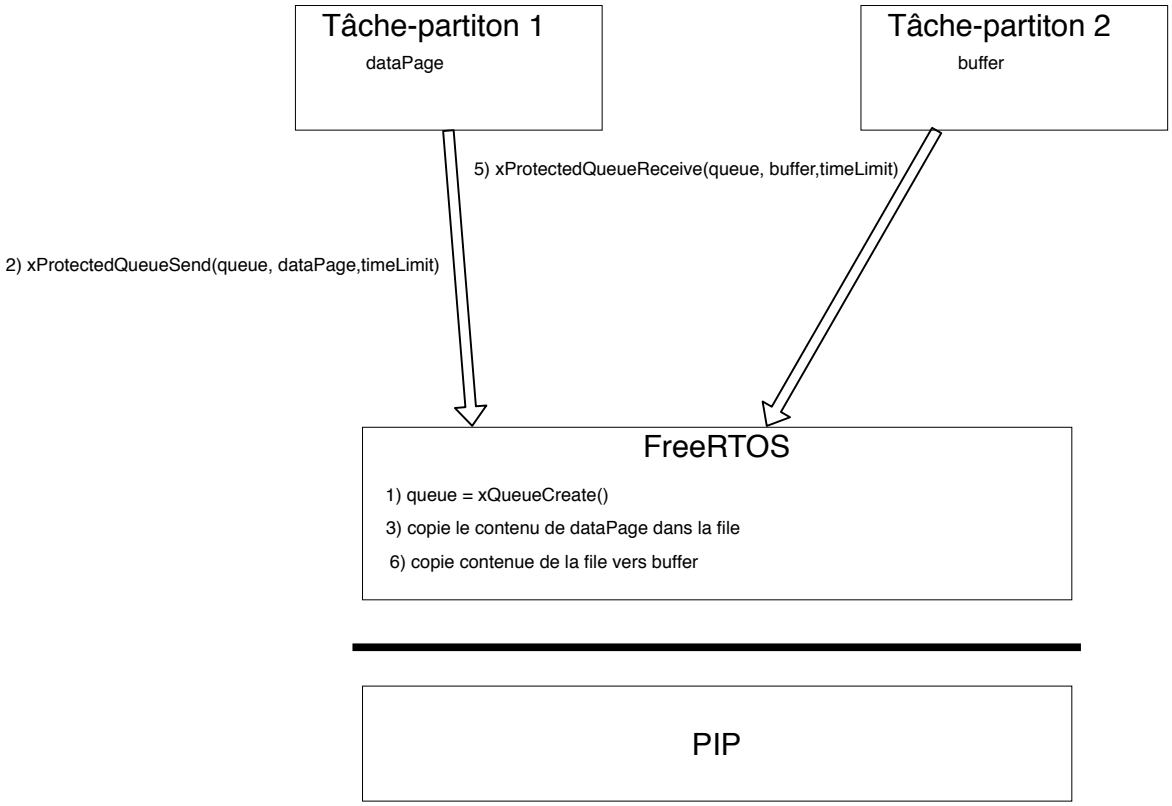

FIGURE 4.14 – Illustration des communications entre tâches-partitions

4.3 Construction logicielle de l'écosystème

Avec l'association Pip et FreeRTOS, nous avons la base logicielle pour construire notre architecture. Nous pouvons créer différents types de composants (tâche-partition ou tâche normale), ainsi qu'un mécanisme de communication. Nous allons dans cette section proposer une architecture et un découpage logiciel précis en partition pour tous les rôles.

4.3.1 Position de l'Émetteur dans la hiérarchie logicielle

La base logicielle Pip-FreeRTOS nous donne une liberté de construction pour notre architecture. Le premier rôle que nous souhaitons mettre en place est L'Émetteur. Toutefois, nous pouvons nous questionner sur sa position dans l'arbre des partitions. Nous avons deux possibilités pour cela

L'Émetteur en tant que partition de niveau 1. Dans cette configuration (figure 4.15), L'Émetteur est positionné en tant que partition. Cela implique que la partition racine doit déléguer toutes les actions possibles et mettre en places des mécanismes permettant à l'Émetteur de créer des domaines, gérer les demandes et autorisations.

Cette architecture permet de mettre en place une isolation supplémentaire entre les domaines et L'Émetteur. Cependant, cela ajoute un nouvel acteur entre ces domaines, FreeRTOS. Si ce dernier est défaillant, la responsabilité qui est engagée n'est pas nécessairement celle de l'Émetteur.

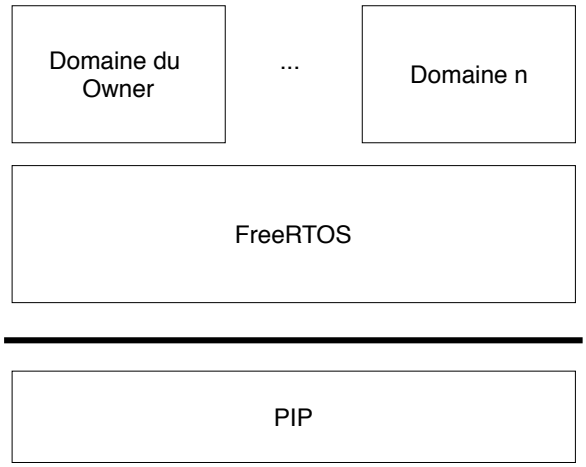

FIGURE 4.15 – Émetteur en tant que partition de niveau 1

L'Émetteur en tant que partition racine (niveau 0). Dans cette configuration (figure 4.16), L'Émetteur possède tous les droits fournis par Pip. Il a accès à tous les composants matériels et à toute la mémoire mise à disposition comme défini précédemment. Cette position permet à l'Émetteur de mettre en place des délégations de tâches directement, notamment auprès de l'Administration Manager. De plus, il est à la base de l'architecture comme nous le désirons au vu de sa position dans la hiérarchie contractuelle. Si un domaine est défaillant, L'Émetteur est en capacité de l'identifier et de rendre responsable son propriétaire. Toutefois, si L'Émetteur est défaillant, cela met en défaut tout l'objet et sa responsabilité est alors engagée.

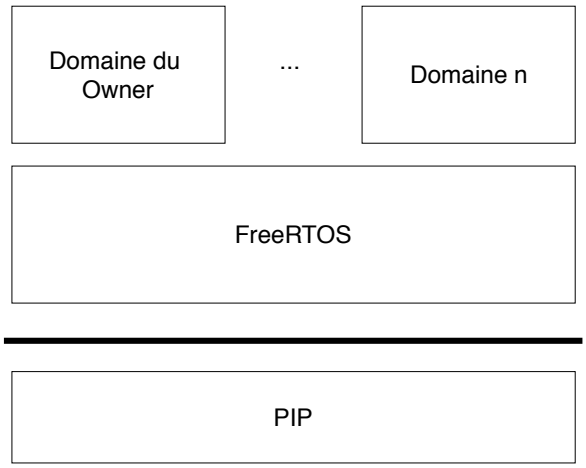

FIGURE 4.16 – Émetteur en tant que partition racine

4.3.2 Configuration Manager.

Chaque domaine possède un Configuration Manager. Ce composant initialise le domaine suivant une description établie. Pour éviter toute propagation de défaillance de ce composant, il serait plus judicieux de mettre le code correspondant au sein d'une partition. Des mécanismes basés sur les interruptions, à l'image des services FreeRTOS, permettent au Configuration Manager d'initialiser le domaine auquel il est intégré.

Pour faciliter la mise en place de ce Configuration Manager, nous avons mis en place un générateur de code permettant la génération du code source correspondant à partir d'une description du domaine en JSON (réalisé en python). Le code JSON 4.1 est un exemple d'un fichier de configuration. L'assistant génère le code C, assembleur ainsi que la chaîne de compilation. Ce qui permet de générer le binaire et de le charger en tant que partition du domaine.

```
{
  "id":1,
  "domainName":"root",
  "tasks":[
    {
      "id":1,
      "name":"tache1",
      "priority":12,
      "function-address":0x456789,
      "stackSize":1000,
      "parameters":"NULL"
    },
    ...
    ],
    "sub-partitions":[
    {
      "id":1,
      "name":"part1",
      "priority":12,
      "binary-position:0x123456,
      "stackSize":1000,
      "parameters":"NULL",
      "hw-access":["led1"],
      "queue-access":[1,2]
    },
    ...
```

```
    ],
    "hardware":[
    {
      "id":1,
      "hardware-name":"led1"
    },
    {
      "id":2,
      "hardware-name":"led2"
    }
    ],
    "queues":[]
}
```

Listing 4.1 – Exemple d'un fichier de configuration de domaine

4.3.3 Construction de l'architecture avec Pip et FreeRTOS

Maintenant que nous avons tous les prérequis nécessaires, nous pouvons mettre en place l'architecture conçue. Comme mentionné précédemment, nous avons un FreeRTOS flexible permettant d'être déployé à n'importe quel niveau de l'arbre des partitions. Ceci nous donne une option pour implémenter les différents domaines, mais chaque propriétaire de domaine peut implémenter sa propre couche logicielle.

Pip-FreeRTOS nous laisse une liberté sur le choix d'implémentation de l'architecture. Toutefois, un code source doit être mis au sein d'une partition s'il subsiste des doutes sur son origine, son implémentation et son influence sur la sûreté et la sécurité du système. Dans la figure 4.17, nous avons un exemple d'un découpage en partition d'une architecture basique d'un objet connecté fournissant une application nécessitant un accès réseau.

Émetteur L'Émetteur doit être responsable de l'objet et détient toutes les compétences au sein de l'objet. De ce fait, cet acteur sera à la racine de l'architecture.

Administration de l'objet L'Administration Manager se voit déléguer beaucoup de tâches de gestion de l'objet par L'Émetteur. L'isoler au sein d'une partition permet d'éviter toute propagation de défaillances, mais aussi toute modification non voulue par L'Émetteur de son comportement.

Token & Security Validator et Key Vault Les clés de chiffrements et leur utilisation doit faire l'objet d'une isolation au sein d'une partition pour des raisons évidentes de sécurité.

Les Mainteneurs Étant donné que L'Émetteur n'a aucune maîtrise sur le code source des Mainteneurs, ces acteurs doivent être isolés au sein de partition avec des droits limités pour isoler tout comportement qui remettrait en question l'intégrité de l'objet.

Les Services providers Chaque Fournisseur de services doit être mis au sein d'une partition. Cependant, les services providers sont libres de propager l'isolation mémoire entre leurs composants.

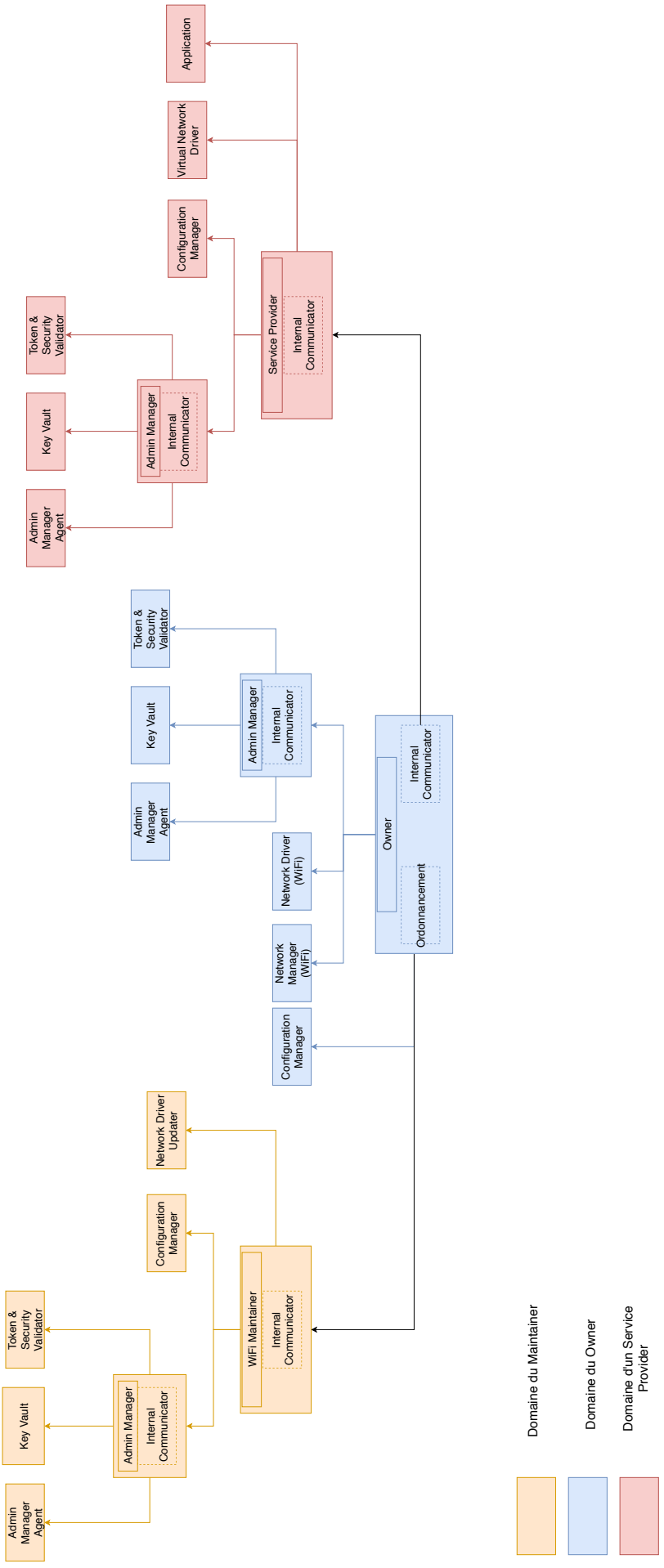

FIGURE 4.17 – Arbre des partitions et des domaines de l'architecture

4.3.4 l'écosystème dans un cluster d'objet connecté

Aujourd'hui, un objet connecté peut être associé à d'autres objets pour fournir de nouveaux services comme vu dans l'état de l'art. Avec l'architecture que nous avons mise en place, celle-ci peut-être utilisée pour la gestion de tous ces objets. L'architecture servant de ce fait de passerelle.

Si nous reprenons l'exemple cité dans l'analyse de l'état de l'art sur un environnement de domotique, nous pouvons utiliser une passerelle qui centralise la gestion et l'utilisation de ces objets (figure 4.18). Au sein de cette passerelle, chaque Fournisseur de services serait associé à un objet dans l'environnement (figure 4.19. Et de manière récursive, au sein de chaque objet, une implémentation de l'architecture est aussi utilisée pour éviter la propagation de défaillances et pour authentifier chaque action réalisée.

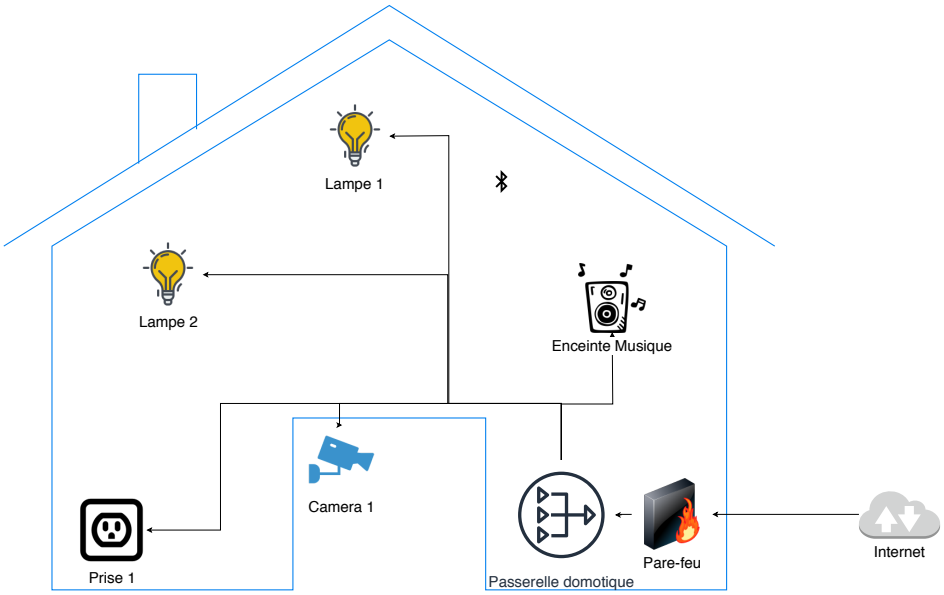

FIGURE 4.18 – Exemple d'une architecture distribué dans un environnement domotique

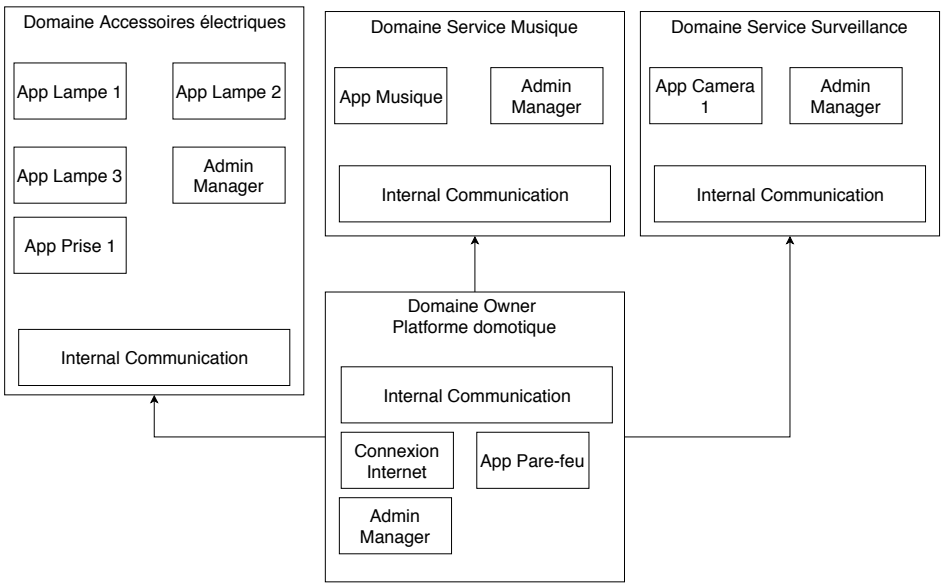

FIGURE 4.19 – Illustration de la décomposition logicielle de la plate-forme domotique

4.4 Discussion et conclusion

4.4.1 Coût de construction de la sécurité

Tout au long de notre contribution, nous nous sommes efforcés de garder comme fil conducteur la sécurité et la sûreté de fonctionnement au sein des systèmes embarqués. L'analyse que nous avons faite de l'état actuel des objets connectés nous a amené à constater des lacunes dans le domaine de la sécurité, ce qui mettait à mal les mécanismes de fiabilité notamment.

Nous avons vu que réaliser un travail de vérification et de preuve sur un système et son implémentation est un travail très coûteux. Réaliser un système complet combinant tous les mécanismes et propriétés d'isolation vus serait aussi coûteux. Le choix qui a été fait de notre travail était de diviser les couches de confiances et les composants acteurs de la sécurité afin de faciliter le travail de vérification.

Cette TCB hiérarchique que nous avons conçu complexifie en effet le travail de construction logicielle de l'objet. Toutefois, les objets connectés sont des environnements avec peu de confiance entre les acteurs comme nous l'avons vu et le fait de travailler par briques indépendantes les unes des autres et isolées permet de fournir les garanties attendues par chaque acteur.

4.4.2 Hiérarchie contractuelle et logicielle

Un constat que nous avons fait était que l'initiateur du projet de l'objet connecté n'était plus au centre de l'application une fois le produit développé. Cela peut amener à des situations avec lesquelles il est responsable sur le plan contractuel, sans pour autant être la source des défaillances.

Au sein de notre architecture, nous avons voulu remettre chaque acteur à sa place, suivant sa position contractuelle, afin de pouvoir responsabiliser chacun d'eux en cas de défaillance ou de comportement malveillant. Sa position hiérarchique définit ses droits, et ces droits sont donnés par l'initiateur du projet. De ce fait, il nous a paru naturel de positionner l'initiateur comme autorité contractuelle et logicielle.

Chaque acteur est responsable de son composant logiciel déployé. Nous avons défini une procédure d'autorisation au sein de l'objet permettant à chaque acteur de réguler les actions dans son domaine. Nous laissons libre cours à un acteur de déployer son mécanisme d'autorisation, mais mettre en place un mécanisme de token permet de profiter de la connectivité de l'objet déjà présente afin d'ajouter une nouvelle couche de confiance au-dessus de l'isolation mémoire.

4.4.3 Sécurité des systèmes temps réels

Nous avons vu que FreeRTOS est un système temps-réel très répandu et facile à utiliser. Cependant, il ne nous permet pas de traiter des problématiques temps-réel réellement (estimation de WCET, budget dans les systèmes ouverts...). Toutefois, à travers son portage et son adaptation par rapport à l'isolation mémoire, nous avons proposé un moyen d'apporter de la sécurité à un système déjà existant ou de donner les clés pour concevoir un système temps-réel avec les contraintes à respecter pour maintenir ses propriétés. Le portage de FreeRTOS servant de guide à un futur travail d'isolation temporelle.

4.4.4 Conclusion

En résumé, tout au long de notre contribution, nous nous sommes efforcés de répondre à une question : *En plus de la sécurité et de la sûreté dans un système, comment amener de la confiance entre les acteurs d'un objet connecté et faire respecter le contrat de développement ainsi que sa hiérarchie ?*

Nous nous sommes efforcés d'avoir une vision la plus exhaustive de l'environnement économique et technique des objets connectés afin d'identifier le plus petit ensemble d'acteurs intervenant à sa conception, son développement et sa gestion. Nous avons construit une hiérarchie logicielle basée sur des droits et responsabilités, avec comme fil conducteur la hiérarchie contractuelle de l'objet.

Cet écosystème, nous avons fait le choix de le construire en utilisant différentes briques de confiance. La première, un mécanisme d'isolation mémoire formellement prouvé. Cette propriété peut être propagée dans toute la hiérarchie afin de maintenir la confiance entre chaque acteur. Nous y avons associé un mécanisme d'ordonnancement temps-réel pour des propriétés d'isolation temporelle. Cependant, nous détaillons le déploiement de ce mécanisme afin de laisser libre toute implémentation autre que du temps réel.

Les objets connectés modernes fonctionnent en groupe, dans des environnements de plus en plus inventifs. Nous avons montré la manière dont peut être utilisée l'architecture afin de superviser un ensemble d'objets dans un environnement restreint.

Dans le prochain chapitre, nous allons discuter de l'impact de cette architecture et de cet écosystème tant au point de vue performances globales que dans l'impact de l'isolation temporelle sur les performances de FreeRTOS et de ses services.

4.5 Références

[Ber19] Quentin Bergougnoux. Co-design et implémentation d'un noyau minimal orienté par sa preuve, et évolution vers les architectures multi-coeur. *http ://www.theses.fr*, jun 2019. 61

[fre] FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. 64

[Jom18] Narjes Jomaa. Le co-design d'un noyau de système d'exploitation et de sa preuve formelle d'isolation. *http ://www.theses.fr*, dec 2018. 61

[YGG+18] M. Yaker, C. Gaber, G. Grimaud, J. Wary, J. Cartigny, X. Han, and V. Sanchez-Leighton. Ensuring iot security with an architecture based on a separation kernel. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 120–127, Aug 2018. 53

Chapitre 5

Mise à l'épreuve

> Il ne peut plus rien nous arriver d'affreux maintenant !
>
> La Cité de la Peur (Alain Berberian 1994)

Sommaire

5.1 Introduction

Dans ce chapitre, nous analyserons le coût humain et technique du portage de FreeRTOS sur Pip. Ceci comprendra des mesures de performance et un chiffrage du coût humain, ainsi que la quantité de code source ajoutée nécessaire à l'accroissement de la sécurité dans ce système temps réel.

5.2 Effort de portage de FreeRTOS sur Pip

5.2.1 Stratégie de portage

Le choix de porter FreeRTOS en tant que partition racine a été motivé par la volonté d'y apporter une isolation mémoire pour ses tâches, comme mentionné dans le chapitre précédent. Pour réaliser le portage de FreeRTOS au-dessus de Pip, nous nous sommes référés au guide fourni par les équipes de développement de ce système[Fre].

Pip a pour vocation de proposer à la partition racine un contexte d'exécution similaire à un environnement physique :

— Mémoire linéaire,

— Accès direct aux périphériques matériels, mais avec un accès contrôlé par Pip,

— Mécanisme d'interruption totalement virtualisé.

Si Pip ne donne pas un accès direct, il expose une API virtualisant le comportement du matériel. FreeRTOS est décomposé en deux parties. Une première partie, correspondant à son noyau (mécanisme de tâches, de files de communication...), indépendante du matériel, mais qui a besoin de la seconde partie, une HAL (Hardware Abstraction Layer ou couche d'abstraction matérielle). Dans un premier temps, il faut réaliser cette couche d'abstraction, ce qui nous permet d'avoir un FreeRTOS fonctionnel dans un même espace d'adressage. La mise en place de cette couche a suivi cette procédure :

Initialisation de la mémoire fournie par Pip : Comme décrit, Pip fournit à la partition racine toute la mémoire utilisateur disponible. Pour pouvoir en bénéficier aisément, dans cette partie, nous initialisons un allocateur de mémoire mis à disposition par la bibliothèque de Pip.

Configuration minimale d'accès au matériel : À des fins d'aisance de développement et d'utilisation, nous avons configuré certains composants, comme la sortie série pour l'affichage, mais aussi un accès au GPIO et I2C notamment, pour permettre le branchement de composants externes (p. ex. ESP8266 pour une connexion WiFi).

Initialisation des interruptions et des gestionnaires d'interruptions : Pour le fonctionnement de FreeRTOS, nous devons implémenter une série de fonctions avec les prototypes existants en utilisant l'API et la bibliothèque de Pip. Ces fonctions seront appelées par le noyau de FreeRTOS.

Cette implémentation permet de produire un FreeRTOS basique, fonctionnant en tant que partition racine et permettant d'exécuter des tâches. Néanmoins, ayant pour objectif d'utiliser des partitions, nous devons modifier l'implémentation de certains composants du noyau. Nous n'avons pas modifié radicalement les mécanismes existants de FreeRTOS, nous avons fait le choix de surcharger certains appels ou de créer de nouvelles fonctions. Pour les fonctions modifiées, nous avons complété leurs algorithmes afin d'y associer la création de partitions.

Nous nous sommes efforcés de ne rien supprimer de l'implémentation existante du noyau, nous avons simplement complété les différents algorithmes pour y greffer les composants nécessaires à la gestion des partitions.

5.2.2 Quantité de travail

La difficulté récurrente rencontrée lors du portage de FreeRTOS sur Pip fut liée au manque de documentation sur le code source. Même si l'API de FreeRTOS possède une documentation publique, le noyau, lui, en est dépourvu. Nous avons détaillé l'analyse du comportement de chaque fonction. Nous avons retracé tous les appels pour les services qui nous intéressaient. Tout ceci dans le but de comprendre l'implémentation interne du noyau, afin d'y implémenter de la manière la plus optimale, mais aussi la moins intrusive, le mécanisme de partition.

Durant la période de travail sur le portage de FreeRTOS sur Pip, ce dernier était aussi en cours de développement. Le portage de FreeRTOS a permis de révéler des besoins non identifiés qui devaient être ajoutés à Pip, sans nécessité de modifier la preuve. Cela venait principalement de la cible des portages (Intel Galileo Gen 2). Cette cible pour le portage nous a aussi posé des difficultés. En effet, l'architecture de cette carte est complexe et la documentation comporte des erreurs ayant rendu les portages plus longs que la durée estimée.

Nous avons fait appel à trois jeunes ingénieurs en stage et à six projets de fin d'études de master, afin de venir compléter le développement du portage, mais aussi l'éprouver à travers des tests de performances et à travers des mises en application (p. ex. prototype pour le projet européen et un environnement embarqué pour une voiture télécommandée). Dans le tableau 5.1, nous avons une estimation de la durée nécessaire au portage de FreeRTOS, ainsi qu'à l'implémentation de l'architecture développée.

Type	Durée (j/h)
Projet de fin d'étude	70
Stage étudiant "jeune ingénieur"	300
Doctorant	300
Total	670

TABLEAU 5.1 – Estimation de la durée nécessaire à l'implémentation de l'architecture en jour/-homme

5.2.3 Impact sur le code source

Dans le tableau 5.2, nous avons une estimation en nombre de lignes de code des modifications apportées à FreeRTOS, ainsi que des services implémentés en son sein. L'implémentation officielle du noyau de FreeRTOS, comprenant environ 11 000 lignes de code, est impactée à hauteur d'environ 10 % par nos modifications.

Le fait d'avoir séparé en différentes briques hiérarchiques les éléments de sécurité et de sûreté a permis de minimiser l'impact sur l'implémentation de FreeRTOS. Implémenter directement des mécanismes d'isolation mémoire, tels que la configuration de la mémoire, n'est pas anodin sur le noyau (p. ex. l'initialisation de la MMU au sein de Pip comprend environ 2 000 lignes de code).

Fichier du noyau	Différence en nombre de lignes de code
Mécanisme de file de communication	200
Mécanisme de tâches	500
Couche d'abstraction	-200
Couche de service	400
Total d'ajout	1100
Total de suppression	200

TABLEAU 5.2 – Estimation du nombre de lignes de code modifiées dans FreeRTOS

5.3 Évaluation de l'impact de l'isolation sur l'API modifiée de FreeRTOS

Un second point que nous souhaitons évaluer concerne le temps d'exécution des appels exposés par FreeRTOS et ses services.

5.3.1 Détails techniques du portage

Un prototype de Pip a été réalisé à destination des processeurs Intel x86. Des versions monocœur et multicœur existent. Cependant, pour réaliser le prototype de l'architecture que nous avons conçue, nous avons fait le choix de le porter sur la plateforme Galileo Gen 2 d'Intel.

La carte Intel Galileo Gen 2 fut développée pour concurrencer les architectures ARM, notamment la Raspberry Pi. Cette carte est dotée d'un processeur 32 bits Intel Quark SoC X1000 cadencé à 400 MHz. Elle dispose de 256 Mo de mémoire vive.

FreeRTOS possède un portage officiel sur cette carte. Pour réaliser les tests de performances, nous avons choisi d'opposer ce portage à différentes configurations. Dans tous les cas, nous allons exécuter deux algorithmes au sein d'une tâche FreeRTOS ou d'une tâche-partition selon le cas. Pour réaliser les mesures temporelles, nous nous basons sur l'instruction RDTSC du processeur, renvoyant le nombre de cycles par seconde écoulés depuis la dernière remise à zéro du processeur.

5.3.2 Banc de test

Pour ces tests, nous allons utiliser une carte Intel Galileo Gen 2, décrite dans la section précédente. Nous allons mettre en place un montage de partition décrit dans la figure 5.1.

Les fonctionnalités que nous souhaitons mesurer sont les suivantes :

— Création de tâche-partition

— Création d'une tâche-partition dotée de 400 ko de mémoire supplémentaire

— Création d'une file de communication

— Envoi de message

— Réception de message

— Accès à un composant matériel (Accès GPIO)

— Demande de mémoire

Pour être probants, tous les tests seront réalisés 100 fois, et pour chaque service, une moyenne sera calculée. Chaque partition effectuera cette série de tests.

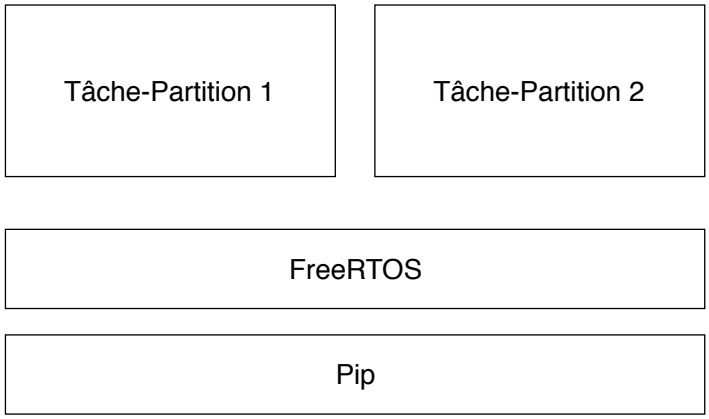

FIGURE 5.1 – Description de la plate-forme de test

5.3.3 Mesure de performance et analyse

Dans le tableau 5.3, nous retrouvons les résultats des mesures effectuées sur le temps d'exécution moyen de chaque mécanisme implémenté au sein du portage de FreeRTOS sur Pip, à destination des tâches-partitions. Il n'est cependant pas pertinent de comparer ces mécanismes à ceux présents au sein d'un FreeRTOS standard. Son noyau et les tâches, ainsi que toutes les structures manipulées, sont dans le même espace d'adressage physique. Pour rappel, les mécanismes implémentés au sein de notre FreeRTOS modifié sont basés sur des interruptions logicielles. De ce fait, nous devons comparer ces chiffres à des environnements similaires.

Le micro-noyau OKL4[HL10] est un noyau de la famille L4, sur lequel peuvent être portés des systèmes. C'est le cas de Linux notamment. Les développeurs ont réalisé des tests de performances sur des appels de services Linux [MS96]. La plate-forme de test est dotée d'un processeur ARM Cortex-A8, cadencé à 500 MHz, donc similaire à notre environnement de test. Dans ces tests de performances, nous pouvons noter que les appels impliquant des hypercalls à OKL4 notamment sont de l'ordre de la milliseconde. Étant donné les différences d'architecture, ces chiffres sont à prendre avec prudence si nous les comparons aux chiffres de notre tableau 5.3. Les chiffres liés à la création de tâches-partitions sont plus conséquents. Cela résulte du coût de l'API de Pip et de l'adressage d'une page au sein d'une partition.

Service	**Temps d'exécution moyen (cycle CPU)**
Création d'une tâche-partition	32874405 ($\tilde{8}$0 ms)
Création d'une tâche-partition avec 400 ko de mémoire supplémentaire	48207465 ($\tilde{1}$20 ms)
Création de file de communication	2565944 ($\tilde{6}$.5 ms)
Envoi de message	1026867 ($\tilde{2}$.5 ms)
Réception de message	397946 ($\tilde{1}$ ms)
Demande d'une page mémoire	9644142 ($\tilde{2}$4 ms)
Accès à un composant matériel (lecture/écriture)	421042 ($\tilde{1}$ ms)

TABLEAU 5.3 – Mesure de performance sur les services exposés par FreeRTOS aux tâches-partitions.

5.4 Évaluation de l'impact de l'isolation mémoire de Pip sur FreeRTOS

L'une de nos contributions est de proposer une réponse aux problématiques de sécurité dans les systèmes temps réels, notamment ceux dans un environnement à criticité mixte, ouverts et connectés. Nous avons associé une isolation mémoire formellement prouvée (Pip) à un système temps réel (FreeRTOS). Cette architecture fournit une TCB à plusieurs niveaux et nous avons montré comment l'isolation spatiale peut préserver l'isolation temporelle de FreeRTOS.

5.4.1 Banc de test

Après les tests de performance réalisés sur nos modifications et notre apport au sein de FreeRTOS, nous allons nous intéresser dans cette partie à l'impact sur le temps d'exécution du portage. Pour ce faire, nous allons exécuter, dans différentes conditions, deux algorithmes, dont un cryptographique. Ceci dans le but de quantifier le surcoût lié aux différents composants amenant de la sécurité.

Dhrystone Dhrystone est un test de performance largement utilisé. Même si son comportement n'est pas représentatif de la vie réelle d'une tâche, il met cependant en stress le processeur. Il convient pour mesurer l'impact des changements de contextes réalisés par Pip et FreeRTOS, notamment sur le nombre de cycles CPU.

AES AES est un test de performance cryptographique. Il chiffre et déchiffre une chaîne de caractères en utilisant différents algorithmes AES :

— AES-ECB-128

— AES-ECB-192

— AES-ECB-256

— AES-CFB128-128

— AES-CFB128-192

— AES-CFB128-256

C'est un algorithme qui met aussi en stress le CPU, ce qui nous permettra d'avoir une vue plus précise sur l'impact de l'isolation et des changements de contextes.

Configurations de tests Une carte témoin contient le portage original de FreeRTOS pour Galileo, sans MMU ni Pip. Nous allons lancer les deux algorithmes afin d'avoir une base de comparaison.

Puis, dans un premier temps, nous allons activer la MMU afin de connaître l'impact de la virtualisation de la mémoire sur un FreeRTOS. Dans un second temps, nous allons mettre en place notre portage FreeRTOS en tant que partition racine dans laquelle nous exécuterons les tests de performance. En dernier point, nous effectuerons deux derniers tests au sein de partitions ordonnancées par FreeRTOS en tant que partition racine de Pip.

Dans tous les cas, pour que les tests soient probants, nous exécutons 100 fois chaque algorithme avec toutes les configurations. Nous mesurons le nombre de cycles CPU écoulés à chaque itération, et nous prendrons la moyenne de ces 100 itérations.

5.4.2 Mesure de performances et analyse

Le résultat de l'expérimentation est visible sur la figure 5.2. Le temps d'exécution de base est celui de la version de FreeRTOS sans la mémoire virtualisée.

Que ce soit pour Dhrystone ou pour les algorithmes d'AES, nous voyons qu'il existe un impact d'au plus 1 % lié à l'activation et l'utilisation de la MMU au sein de FreeRTOS.

Cependant, dans le cas où FreeRTOS est utilisé en tant que partition racine, l'impact est de 2 % pour Dhrystone et 1,2 % pour AES. Ce léger surcoût est dû aux interruptions matérielles survenues lors de l'exécution. En effet, à chaque interruption du processeur, Pip reprend la main sur l'exécution, réalise un traitement puis redonne la main à la partition racine, FreeRTOS.

Dans le dernier cas, le surcoût est plus important : 12 % pour Dhrystone et 5 % en moyenne pour AES. Ces chiffres peuvent être expliqués par la chaîne de cheminement des interruptions. En effet, lorsqu'un des algorithmes est en fonctionnement au sein d'une partition, si une interruption survient, Pip prend la main, effectue ses traitements puis redonne la main à FreeRTOS, la partition racine. FreeRTOS possède un mécanisme complexe d'ordonnancement, qui passe par le choix de la prochaine tâche à exécuter. Dans notre cas, seule la tâche-partition exécutant le test est présente.

Des travaux réalisés sur l'optimisation de Pip sur certains processeurs x86 permettent de réduire ce surcoût au sein d'une partition de 12 % à 6 % pour Dhrystone et de 5 % à 3,5 % pour AES[Ber19]. Cependant, ces améliorations nécessitent des options spécifiques à certains modèles de processeurs, indisponibles sur celui de la Galileo Gen 2.

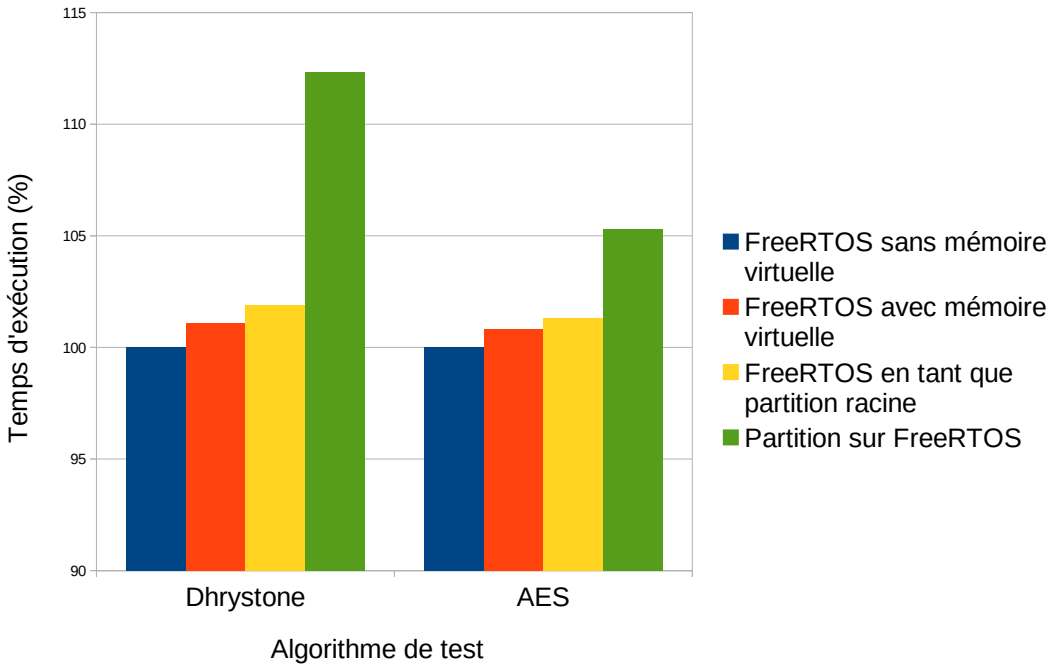

FIGURE 5.2 – Mesure de performances : Dhrystone et AES

5.5 Conclusion

Dans ce chapitre, nous avons évalué le coût humain et technique de la sécurité sur un système temps réel largement répandu. L'association de Pip et de FreeRTOS propose des performances qui sont acceptables au vu de l'apport de sécurité. L'API de FreeRTOS que nous avons développée pour les tâches-partitions, bien que minimale, permet déjà de créer un environnement fonctionnel pour des tâches plus complexes. Les chiffres liés aux services exposés par notre FreeRTOS permettent de penser que le surcoût de fonctionnement des tâches-partitions ne sera pas conséquent dans un fonctionnement normal.

De plus, nous avons vu qu'en ciblant précisément les éléments du portage à modifier, nous avons minimisé le nombre de lignes de code nécessaires pour mettre en place ces mécanismes de sécurité (environ 10 % de lignes de code supplémentaires).

5.6 Références

[Ber19] Quentin Bergougnoux. Co-design et implémentation d'un noyau minimal orienté par sa preuve, et évolution vers les architectures multi-coeur. *http ://www.theses.fr*, jun 2019. 86

[Fre] FreeRTOS Porting Guide. 81

[HL10] Gernot Heiser and Ben Leslie. The OKL4 microvisor : Convergence point of microkernels and hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems, APSys '10, Co-located with SIGCOMM 2010*, pages 19–23, New York, New York, USA, 2010. ACM Press. 84

[MS96] Larry McVoy and Carl Staelin. Lmbench : Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. 84

Chapitre 6

Conclusion, perspectives et réflexion personnelle

> Nous menons notre vie quotidienne sans presque rien comprendre au monde qui est le nôtre. Nous accordons peu de pensées à la machine qui engendre la lumière du Soleil, rendant ainsi la vie possible, à la gravité qui nous colle à une Terre qui, autrement, nous enverrait tournoyer dans l'espace, ou aux atomes dont nous sommes faits et dont la stabilité assure notre existence. À l'exception des enfants (qui n'en savent pas assez long pour poser les questions importantes), peu d'entre nous passent beaucoup de temps à se demander pourquoi la nature est telle qu'elle est.
>
> Stephen Hawking

Sommaire

6.1 Synthèse

Tout au long du manuscrit, nous avons maintenu comme fil conducteur la sécurité et la sûreté des systèmes embarqués. Nous avons fait le constat que les objets connectés comportent des lacunes en termes de sécurité, y compris dans des environnements à criticité mixte. Nous avons aussi constaté que les objets connectés étaient le fruit d'un amalgame d'acteurs divers, mais que la responsabilité de chacun tendait à disparaître au cours de la vie de l'objet, surtout en cas de défaillance.

Ces deux préoccupations nous ont amenés à concevoir et à développer un écosystème permettant de répondre au mieux à ces problématiques. Nous avons conçu une hiérarchie logicielle basée sur des rôles clairement définis permettant de reproduire au sein d'un objet la hiérarchie contractuelle. Pour chaque acteur de l'objet, un rôle, des tâches et des droits clairement définis. Le but étant de créer un environnement de confiance pour toutes les entités présentes.

Pour maintenir les propriétés de sécurité et de sûreté, les rôles et les droits, nous avons émis l'hypothèse qu'en hiérarchisant le code source de l'objet (en mettant en place une TCB hiérarchique), nous pouvions plus aisément les vérifier et les confirmer. Cette TCB hiérarchique est basée sur un mécanisme d'isolation mémoire, auquel nous avons superposé un mécanisme d'isolation spatiale permettant de maintenir les propriétés fondamentales de la sécurité et de la sûreté de fonctionnement : la confidentialité, l'intégrité et la disponibilité.

Pour illustrer la faisabilité de cette architecture, nous avons mis en place un prototype basé sur un proto-noyau d'isolation mémoire (Pip), permettant une construction logicielle arborescente, chaque nœud étant formellement isolé de ses voisins. Nous avons réalisé le portage d'un système temps réel (FreeRTOS) en tant que mécanisme d'isolation temporelle. En plus de l'association de ces deux mécanismes, nous avons mis en place une API permettant de fournir les prérequis de base nécessaires à la construction de l'architecture, à savoir un mécanisme d'isolation de composants logiciels sensibles, un mécanisme de communication et d'accès au matériel.

Nous avons réalisé des mesures de performances dans un premier temps afin de connaître l'impact de l'isolation mémoire sur l'ordonnancement temps réel de FreeRTOS. Cela nous a menés à constater qu'il y a en effet un léger surcoût acceptable, mais qui ne remet pas en question toute l'architecture. Dans un second temps, nous avons mesuré le temps nécessaire au fonctionnement des différents composants de l'API, constatant que l'isolation des tâches de FreeRTOS ne produit pas de temps d'exécution trop longs.

6.2 Perspectives

Ordonnancement temps réel FreeRTOS est un système d'exploitation temps réel avec un ordonnancement à base de priorités. Cependant, il ne prend pas en compte le pire temps d'exécution des tâches qu'il prend en charge. Un axe d'amélioration serait la conception d'un ordonnanceur au sein de la partition racine.

L'architecture étant hiérarchique, nous pouvons penser qu'il serait intéressant aussi que ce mécanisme d'ordonnancement soit hiérarchique lui aussi. Ceci éviterait des situations de famine dans certaines branches du système, et quelle que soit la position dans l'arbre des partitions, l'ordonnanceur serait capable de garantir le temps d'exécution requis.

Un dernier point serait de réaliser cet ordonnanceur en Gallina, ceci dans le but de prouver sa validité avec l'assistant Coq, à l'instar de Pip. Cette démarche s'inscrirait dans

la continuité de l'idée que chaque niveau de la TCB hiérarchique profite des propriétés déjà vérifiées au niveau inférieur, ce qui permet de se focaliser sur de nouvelles propriétés, dans ce cas, le temps réel.

Le traitement des petits objets L'architecture que nous proposons dans cette thèse doit avoir comme prérequis une MMU, l'isolation de Pip étant basée sur la bonne configuration d'une MMU physique. Toutefois, beaucoup d'objets ont des parties physiques dépourvues de MMU, pour des questions de coûts de fabrication. Souvent, on peut trouver un composant moins coûteux, la MPU (Memory Protection Unit). Ce composant permet de subdiviser la mémoire physique en plusieurs régions, chacune pouvant être associée à un processus. Le nombre de ces régions mémoire est limité à quelques-unes seulement (p. ex. 16 pour les processeurs ARM récents).

Avec un nombre de régions et de composants isolables beaucoup plus limité, nous pouvons nous interroger sur la faisabilité de l'architecture que nous avons développée sur des architectures limitées, souvent le cas de petits objets. Il serait intéressant de pouvoir répondre à certaines questions :

— Ce découpage des acteurs peut-il être transposé en régions MPU ?

— Comment gérer la gestion des clés et des autorisations dans un environnement peu isolé ?

— Dans l'hypothèse où une version de Pip sur MPU existe, est-il toujours pertinent de superposer l'isolation spatiale et l'isolation temporelle ?

6.3 Réflexion et vision de l'objet connecté

Dans nos sociétés modernes, une tendance à tout connecter émane de divers acteurs socio-économiques. Cette démarche a débuté avec les smartphones. Rares sont les personnes qui n'en possèdent pas. Puis, des assistants capables de donner la météo au réveil ont fait leur apparition dans nos maisons, accompagnés de l'électroménager connecté et même des cuisines intelligentes et tout aussi connectées. Dans le domaine sportif, des coachs miniatures accompagnent les pratiquants en les guidant. Dans le but d'optimiser les transports, tous les véhicules comportent des capteurs, des systèmes disséminés partout pour fournir plus de services ou rentabiliser au maximum un trajet. Bref, tout notre quotidien tend à être doté d'un système embarqué pour optimiser ou automatiser des tâches.

Nous avons vu que leurs implémentations laissaient parfois à désirer, ce qui provoque d'énormes failles de sécurité mettant, dans le pire des cas, en danger leurs utilisateurs. La première réflexion serait de se demander pourquoi, après tant d'avertissements liés à la sécurité, autant d'objets vulnérables continuent d'être produits de nos jours. La première réponse évidente serait pécuniaire. Chaque producteur d'objets connectés cherche à maximiser la marge sur chaque objet. Cependant, au sein de notre équipe, nous avons montré qu'un groupe de quelques personnes pouvait produire un travail non négligeable pour apporter de la sécurité aux objets connectés. Ceci n'est pas une façon de dire que le travail est facile, mais qu'au vu de la facilité qu'ont certains groupes d'attaquants à pénétrer les objets, un effort minimal pourrait être fait sans surcoût par les fabricants d'objets.

Un autre point important que nous souhaitons soulever réside dans la confiance que l'on peut porter à ces objets. Nous ne négligeons pas leur utilité, chacun de nous en possède quelques-uns. Cependant, alors que l'on s'insurge des données collectées par les

réseaux sociaux, nous oublions cette masse d'informations contenue et gérée par ces objets. Ces objets peuvent renfermer des informations équivalentes à une empreinte digitale d'un individu ainsi que son comportement quotidien à la seconde près : sa position géographique, son état de santé, son alimentation, ses relations personnelles. Tout peut être mesuré, classé, analysé. Les institutions, européennes notamment, commencent seulement depuis quelques années à traiter cette problématique. Des autorités tendent même à retirer des produits des commerces, pour le risque qu'ils représentent. Ce fut le cas notamment de la poupée *Cayla*, qui faisait fureur en Allemagne et fut retirée, car soupçonnée d'espionnage.

En bref, nous pensons qu'il ne faut pas sous-estimer les objets connectés, tant de la part de leurs développeurs qui minimisent parfois l'impact même de leurs produits sur les consommateurs et sur la société, que de la part de cette dernière, qui est parfois trop en retard sur la gestion des libertés et des droits individuels quand il s'agit de produits technologiques.

Annexe A

Annexes

A.1 API de FreeRTOS en partition racine

```
1
2  /*
3
4
5  Parameters:
6      uxQueueLength     The maximum number of items the queue can hold
       at any one time.
7      uxItemSize    The size, in bytes, required to hold each item in
       the queue.
8
9      Items are queued by copy, not by reference, so this is the number
       of bytes that will be copied for each queued item. Each item in
       the queue must be the same size.
10
11 Returns:
12     If the queue is created successfully then a handle to the created
       queue is returned. If the memory required to create the queue
       could not be allocated then NULL is returned.
13
14
15
16 */
17
18 uint32_t xProtectedQueueCreate( uint32_t uxQueueLength, uint32_t
       uxItemSize );
19
20
21
22 /**
23
24 Parameters:
25     xQueue  The handle to the queue on which the item is to be posted.
26
27     pvItemToQueue   A pointer to the item that is to be placed on the
       queue. The size of the items the queue will hold was defined when
       the queue was created, so this many bytes will be copied from
       pvItemToQueue into the queue storage area.
28
29     xTicksToWait  The maximum amount of time the task should block
       waiting for space to become available on the queue, should it
       already be full. The call will return immediately if the queue is
```

```
    full and xTicksToWait is set to 0. The time is defined in tick
    periods so the constant portTICK_PERIOD_MS should be used to
    convert to real time if this is required.

     If INCLUDE_vTaskSuspend is set to    1    then specifying the
    block time as portMAX_DELAY will cause the task to block
    indefinitely (without a timeout).

Returns:
     pdTRUE if the item was successfully posted, otherwise
    errQUEUE_FULL.

*/



uint32_t xProtectedQueueSend(uint32_t xQueue,uint32_t pvItemToQueue,
    uint32_t xTicksToWait);









/**
Parameters:
     xQueue  The handle to the queue from which the item is to be
    received.

     pvBuffer  Pointer to the buffer into which the received item will
    be copied.

     xTicksToWait  The maximum amount of time the task should block
    waiting for an item to receive should the queue be empty at the
    time of the call. Setting xTicksToWait to 0 will cause the
    function to return immediately if the queue is empty. The time is
    defined in tick periods so the constant portTICK_PERIOD_MS should
    be used to convert to real time if this is required.

     If INCLUDE_vTaskSuspend is set to 1 then specifying the block time
     as portMAX_DELAY will cause the task to block indefinitely (
    without a timeout).

Returns:
     pdTRUE if an item was successfully received from the queue,
    otherwise pdFALSE.


*/



uint32_t xProtectedQueueReceive(uint32_t xQueue,uint32_t pvBuffer,
    uint32_t xTicksToWait);



```

```
71 /**
72
73
74 Parameters:
75     address of the specific hardware
76
77 Return:
78     The value returned by the hardware
79
80 */
81 uint32_t inServiceGlue( uint32_t addresss);
82
83
84
85
86 /**
87
88
89 Parameters:
90     address : of the specific hardware
91
92     value : Data to write at the hardware address.
93
94 Return:
95     The value returned by the hardware
96
97 */
98 uint32_t outServiceGlue( uint32_t addresss, uint32_t value);
```

Résumé

Mots clés : Sécurité, système embarqué, internet des objets, sûreté, écosystème IoT

Depuis leur apparition durant la conquête spatiale, à leur arrivée dans nos maisons, les systèmes embarqués ont énormément évolué dans leurs compositions, leurs fonctionnements et leurs méthodes de développement. De plus, avec l'apparition de réseaux comme Internet, on a vu émerger une nouvelle forme de systèmes embarqués, dit l'Internet des Objets (ang, Internet of Things, IoT). Ces systèmes sont très diversifiés. Ils sont utilisés dans les transports, la télécommunication ou encore la domotique et la médecine. De plus, on voit aussi apparaître des objets connectés fonctionnant en groupe, permettant de fournir des fonctionnalités encore plus complexes à des utilisateurs. Ces utilisateurs peuvent être des êtres humains ou d'autres machines.

Cette diversification a vu apparaître de nombreux acteurs, avec des compétences variées, proposant des composants logiciels ou matériels pour ces systèmes embarqués. Malheureusement, la multiplication des acteurs et des méthodes de développement et de collaboration peuvent nuire aux mécanismes de sécurité ou de sûreté de fonctionnement des systèmes. Lorsqu'une défaillance survient, il peut être difficile dans certaines situations de déterminer le responsable, en dépit des contraintes contractuelles et légales. De plus, si des acteurs économiques distincts et concurrents sont amenés à fournir des services simultanément pour ces objets, il est possible que l'un d'eux cherche à nuire à ses rivaux.

Pour répondre à ces problématiques de sécurité et sûreté, des solutions logicielles ou matérielles sont développées. L'isolation mémoire ou la virtualisation garantissent notamment l'intégrité et la confidentialité au sein de l'objet. L'utilisation de méthodes formelles permet la vérification de propriétés en amont de la période de fonctionnement du système. En plus des liens contractuels, les acteurs sont parfois dans l'obligation de respecter des normes ou des standards de développement destinés à fournir des garanties légales sur les objets.

Dans cette thèse, j'étudie d'abord l'écosystème dans lequel les systèmes embarqués et les objets connectés sont développés, afin d'y identifier de la manière la plus exhaustive possible les acteurs de cet écosystème. Cette étude m'a mené à constater la dilution des responsabilités de chaque acteur au sein du système embarqué en cas de défaillance de sécurité ou de sûreté ainsi que le manque de confiance entre eux. Ainsi, après cette analyse, j'ai conçu une hiérarchie et des droits entre ces acteurs et leurs relations. Pour construire cette architecture, je me base sur des travaux d'un noyau formellement prouvé, fournissant des propriétés d'isolation mémoire et des garanties fortes de sécurité et de sûreté. Avec ces briques de base, je montre comment je peux construire l'architecture et y propager ces garanties afin de maintenir la confiance entre les acteurs, le confinement de défaillances et la responsabilité de chacun. Avec le mécanisme de sécurisation des échanges mis en place, je montre qu'il est possible d'étendre cette architecture afin de construire un groupement d'objets tout en maintenant les propriétés de sécurité et de sûreté définies au sein des objets.