Homeautomation

Commanditaires / Exécuteurs

Le commanditaire du projet est Mr. TANGUY, membre du LAB-STICC. Ce projet s'inscrit dans le cadre du projet tutoré de M1 CSSE de l'Université Bretagne Sud, faculté des sciences de l'ingénieur de Lorient.
L'équipe en charge du projet est constituée de :

SITBON Nathan
MALCOMBRE Nicolas
GIDON Rémi

Résumé

L'internet des objets est un paradigme dont les premiers déploiements ont quelques années. D'un point de vue sécurité, l'IoT a une surface d'attaque très importante à cause du nombre de technologies, de protocoles, de types de déploiement et du nombre d'acteurs différents. Un des grands défis de l'IoT est la mise en place d'un système Over-The-Air (OTA) de mise-à-jour afin d'intégrer des patchs de sécurité. Pour différentes raisons comme le coût, les objets qui sont très contraints en ressources, les possibilités de mettre à jour un objet connecté n'est pas une tâche triviale.

Objectif

Dans ce contexte IoT, notre objectif est de créer une passerelle, ayant la capacité de supporter un OS de type Linux. Cette passerelle permettra de faire le pont entre des réseaux sans-fils de capteurs / actionneurs d'un côté et un réseau filaire de l'autre. La passerelle a vocation à être en intérieur chez des utilisateurs.

Sommaire

I - Spécifications

À la suite de la remise du cahier des charges par le client, nous avons isolé les besoins et attentes concernant la box domotique.
La box doit être sous un système Linux. Elle doit-être pourvue d'un accès à distance permettant à un administrateur de faire de la télémaintenance ainsi que d'un système de mise à jour *Over-The-Air* pour le déploiement de mises à jour depuis un serveur distant.
La sécurité de la box est essentielle pour le client, autant matériellement que applicativement. En effet, la passerelle se trouve chez l'utilisateur et le système Linux est en charge de services applicatifs payants.

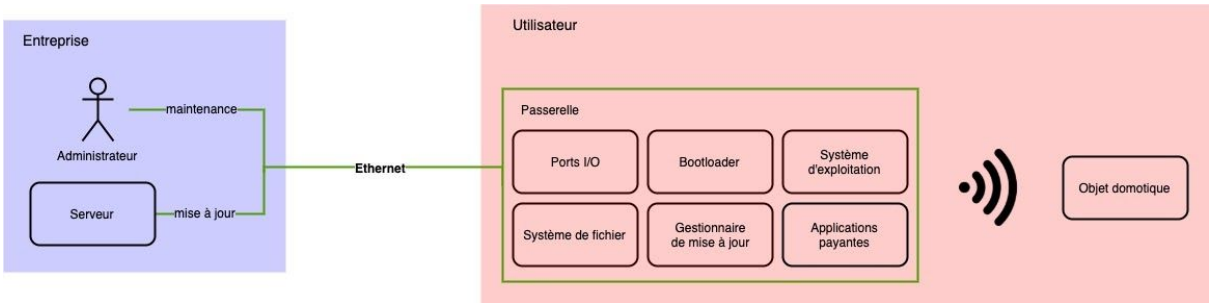


Schéma du système

Fonctionnalités

- Passerelle *"homeautomation"* sous système Linux.
- Déploiement de mises à jour *Over-The-Air* via un serveur distant.
- Accès à distance pour télémaintenance.
- Sécurisation de la passerelle car en intérieur chez l'utilisateur.
- Sécurisation du système Linux car en charge de services applicatifs payants.

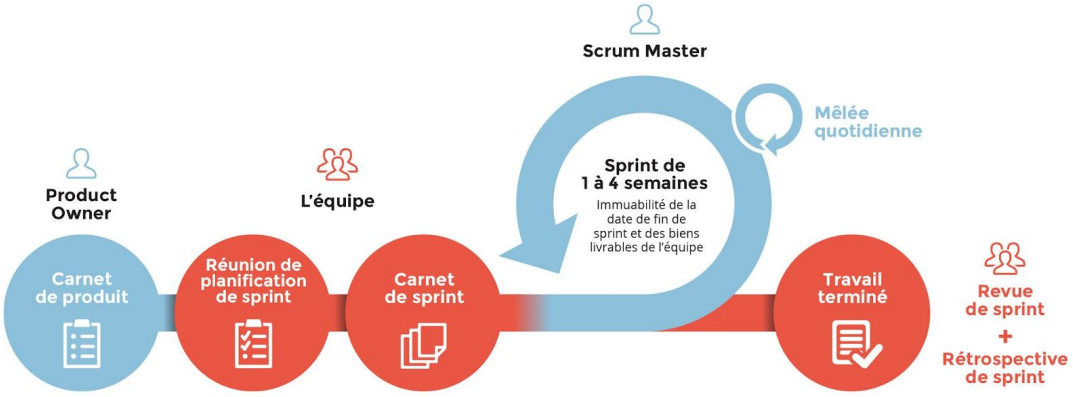
Livrables

- Rapport du projet.
- Documentation de la solution.
- Code source de la solution.

II - Organisation & méthodes

Nous avons opté pour une organisation selon une méthode itérative, en l'occurrence la méthode Scrum que nous détaillerons ci-dessous.
Une méthode itérative se prêtait mieux qu'une méthode standard à la réalisation de notre projet. En effet le but du projet est de faire une box domotique minimale, c'est à dire qui comporte le moins de fonctionnalités pour permettre une meilleure sécurisation. La première version du cahier des charges étant minimale et le temps réservé au projet assez long, nous avons choisi de découper le projet en itérations d'une semaine afin que l'ajout de fonctionnalités répondent aux demandes du client.

La méthode mise en œuvre sur le projet est la méthode Scrum. Le développement du projet est ainsi découpé en différents lapses de temps appelés *sprint*.
Une liste de tâches pour le projet (appelée *product backlog*) est continuellement mise à jour en parallèle de l'avancée du projet. Cette liste représente l'ensemble des fonctionnalités voulues et restantes à réaliser.
En fin de *sprint*, une réunion avec le client permet de confirmer les avancées du projet sur ce sprint, ainsi que planifier le suivant. On choisit des tâches du *product backlog* ou en reportons d'autres du sprint précédent, pour créer une liste de tâche propre au sprint courant (appelée *scope*).
On peut ainsi garder ou changer d'objectif à chaque sprint, suivant les tâches attribuées.



Fonctionnement de la méthode itérative Scrum

Dans la pratique, un tableau d'avancement est créé sous l'outil Trello, c'est le Scrumboard.
Il permet de représenter un *sprint* et son avancé par trois colonnes:

- Tâches à faire
- Tâches en cours
- Tâches terminées

Une dernière colonne peut être ajoutée pour lister l'ensemble des tâches restantes du projet (*product backlog*). Ce tableau est partagé par l'ensemble de l'équipe et permet de suivre en temps réel l'évolution du *sprint*.

Chaque matin, l'équipe se réunit pour le *"Daily stand-up meeting"* qui consiste à répondre aux trois questions suivantes :

- Qu'ai-je fait hier?
- Que vais-je faire aujourd'hui?
- Quels sont les obstacles rencontrés?

Cela permet un point sur les tâches attribuées, en attribuer de nouvelles, organiser les coopérations sur une même tâche et aider les personnes en difficulté.
L'avantage de cette méthode est sa flexibilité, le client peut voir son produit prendre vie au cours des itérations. Cela permet de confirmer ou corriger la trajectoire du projet afin d'éviter l'effet tunnel.

Outils

Nous avons utilisé les outils ci-dessous durant la réalisation du projet.

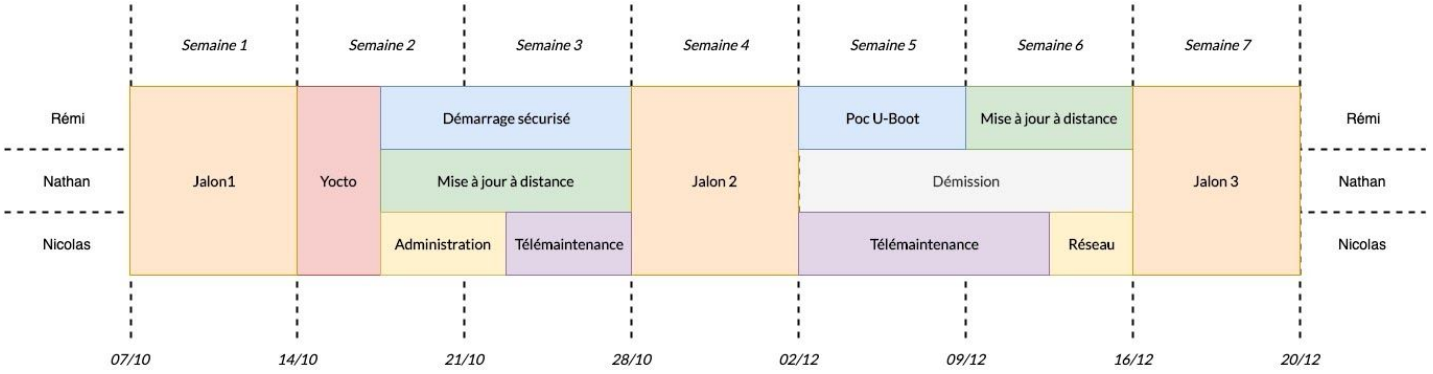
- *Git* comme logiciel de partage de code source.
- *Gitlab* comme logiciel d'hébergement de projet.
- Des fichiers *markdown* pour la documentation du projet sous *Gitlab*.
- *Google Drive* comme outil de gestion de documents et de partage de ressources.
- *Trello* comme outil pour de gestion de projet dans l'utilisation de la méthode Scrum.
- Le diagramme de *Gantt*[1] comme outil de planification afin de déterminer le chemin critique et les tâches parallélisables

Tâches

Dans un premier temps, nous avons travaillé en coopération pour mettre en place une distribution minimale sous Yocto qui nous servira de base pour les objectifs suivants.

Nous avons isolé trois objectifs parallélisables qui ont été attribués par affinitées:
Rémi GIDON: démarrage sécurisé.
Nathan SITBON: système de mises à jour à distance
Nicolas MALCOMBRE: système de télémaintenance

Suite au départ de Nathan SITBON nous avons dû redéfinir la répartition des tâches:
Rémi GIDON: dans l'impasse avec le démarrage sécurisé, passage sur le système de mises à jour à distance.
Nicolas MALCOMBRE: sécurisation de la télémaintenance, durcissement du système Linux.



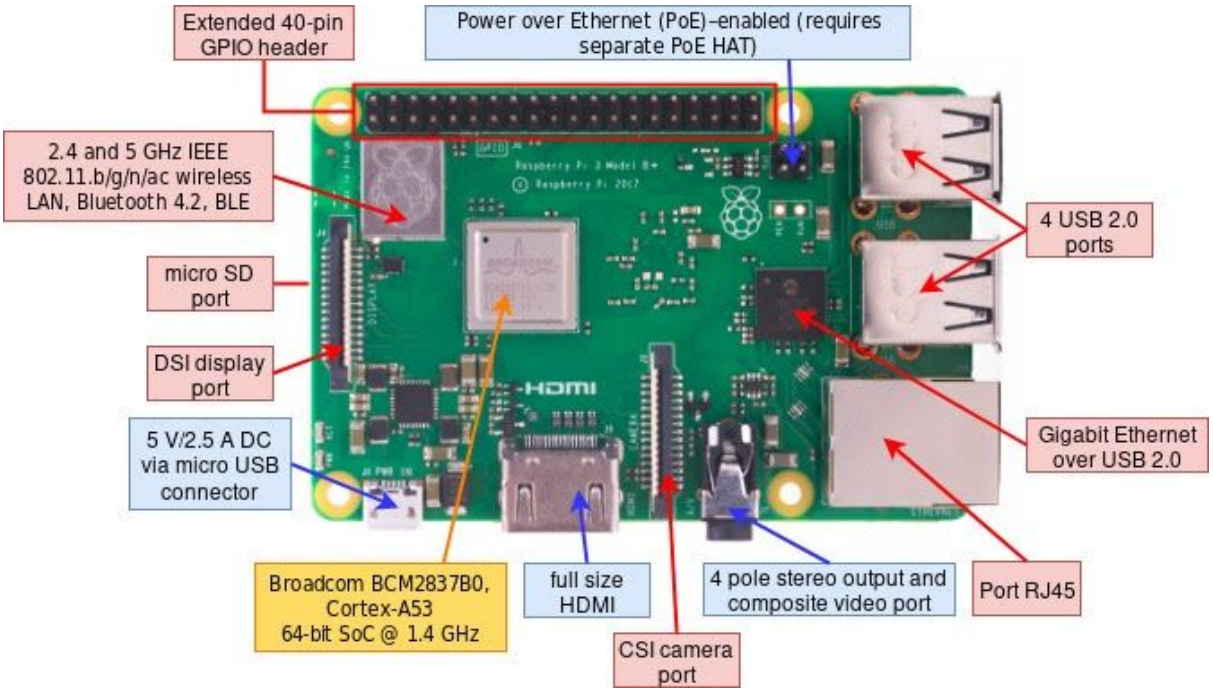
Tâches réalisées durant les 7 semaines de projet

[1] Voir annexe 1 page 21

III - Analyse

Plateforme

Nous utiliserons une carte Raspberry PI 3 modèle B+ en 64bit à la demande du client.
Voici les caractéristiques principales de la carte :



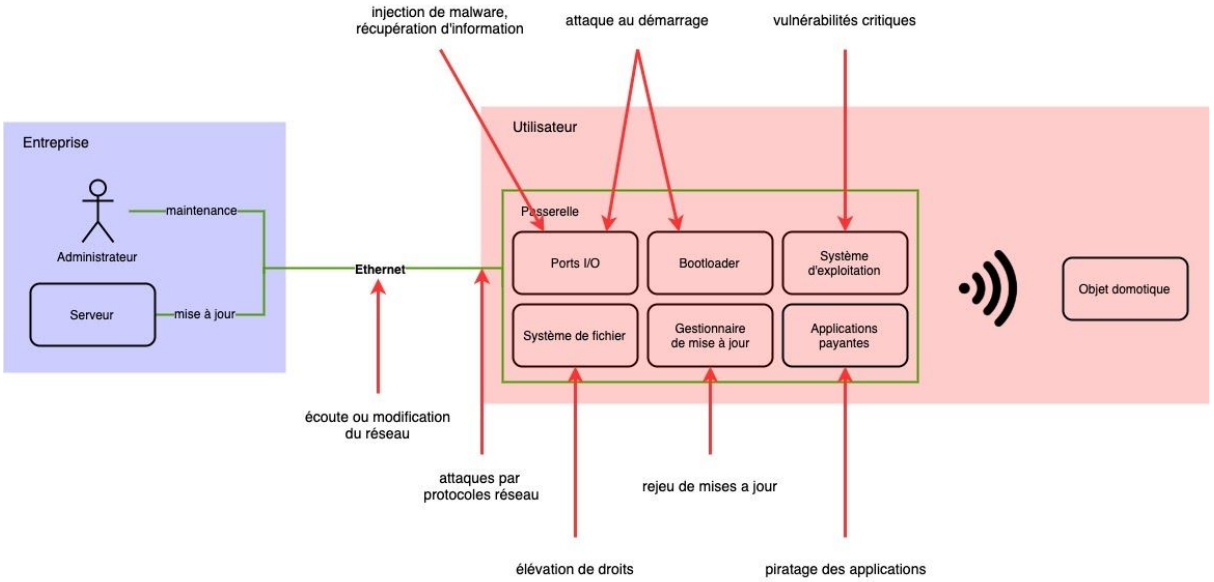
Composants d'entrée / sortie de la Raspberry Pi 3B+ [2]

Les composants en rouge représentent un risque potentiel de part leurs possibles entrées / sorties.
Ceux en jaune peuvent êtres exploités mais cela relevant de la conception de la carte par le fabricant, nous ne le prendrons pas en compte dans notre modèle de sécurité.
Enfin ceux en bleu sont considérés comme inoffensifs, ne permettant pas l'injection ou récupération de données.

Pour ce qui est de ce projet, nous n'utilisons que le port Ethernet pour la connexion avec le serveur de mises à jour. Les autres ports devront donc êtres désactivés ou contrôlés.

[2] https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf

Modèle de menace



Modèle de menace du système

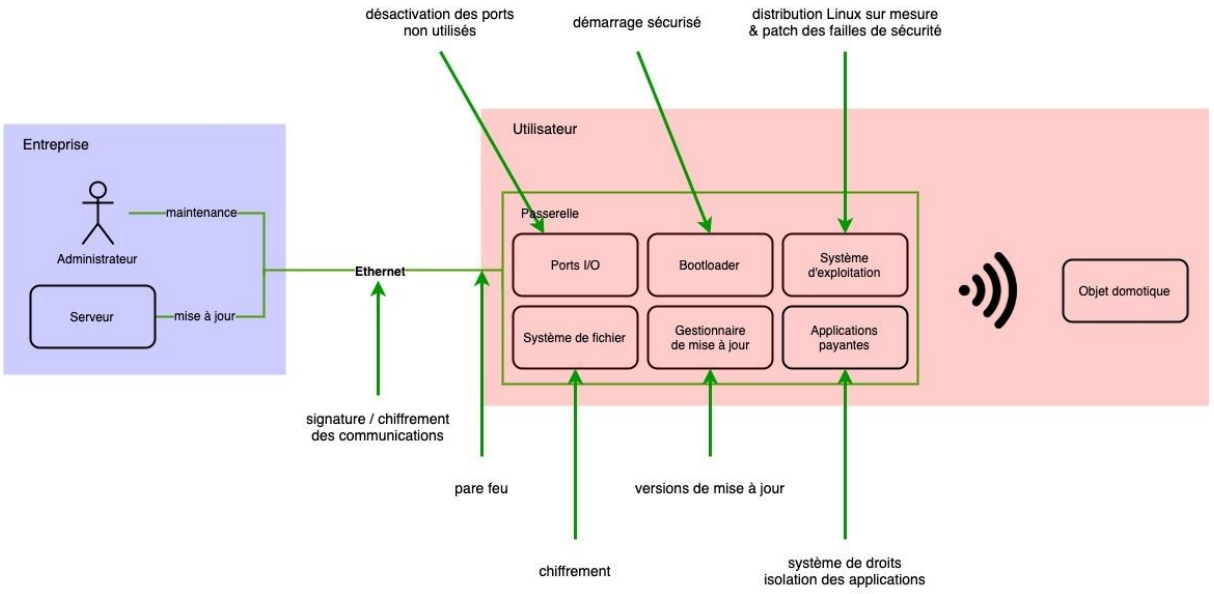
Attaques physique

- Ports disponibles sur la carte. Notamment les ports SD et USB pouvant permettre le boot.

Attaques logique

- Envoi de mises à jour par une autorité non reconnue.
- Ecoute du réseau, modification des données transmises par une personne tierce.
- Intrusion dans le système, élévation de droits.
- Exploitation de failles logicielles.
- Modification de l'intégrité du système.
- Rejeu de mise à jour contenant d'anciennes failles.
- Extraction d'informations privées (clefs, codes source,...).

Modèle de sécurité



Modèle de sécurité du système

Pour chaque besoin du client et en adéquation avec le modèle de menace, nous avons identifié une ou plusieures mesures d'action:

1. Passerelle sous système Linux
    - Distribution Linux sur mesure permettant de réduire la surface d'attaque.

2. Déploiement de mises à jour via un serveur distant
    - Vérification d'intégrité et d'authentification des mises à jour.
    - Confidentialité de la mise à jour.
    - Logiciel de gestion de mises à jour sur chaque carte.

3. Accès à distance pour télémaintenance
    - Compte administrateur et interpréteur de commande sur la passerelle.
    - Connexion sécurisée entre l'administrateur et la passerelle.

4. Sécurisation physique de la passerelle
    - Camouflage et / ou désactivation de ports réseaux.
    - Chiffrement du système de fichier.
    - Système de démarrage sécurisé.

5. Sécurisation du système Linux
    - Système de démarrage sécurisé.
    - Patch des brèches de sécurité pour éviter l'exploitation de failles logicielles.
    - Gestion des droits utilisateurs.
    - Isolation des applications.
    - Pare-feu réseau, autorisation du serveur et des administrateurs.

IV - Conception

Démarrage sécurisé

Le but est de ne pas pouvoir compromettre le démarrage du système d'exploitation en remplaçant des composants dans la chaîne des logiciels de démarrage de la carte.
Pour cela, une chaîne de confiance est mise en place.
Dans l'idéal, il faudrait un composant matériel dédié permettant le stockage de la clef de vérification des logiciels de démarrage. Une autre méthode serait d'écrire la clef dans une mémoire non modifiable (ROM). La carte choisie ne disposant pas de tels composants, il n'est pas possible d'achever une racine de confiance.

Dans un premier temps, le firmware chargera l'image contenant le bootloader et le kernel depuis le support de stockage puis vérifiera les signatures de chaque composant de l'image grâce aux clefs. Si une des signatures n'est pas valide, alors un composant peut avoir été modifié, et le démarrage est interrompu.
Sinon le système d'exploitation est démarré, la carte va ensuite essayer de joindre le serveur via son nom de domaine pour s'authentifier, puis authentifier le serveur (à l'aide de signatures).
Une fois l'authentification faite, le serveur regarde si la carte attend des mises à jour, et lance le protocole de mises à jour si c'est le cas.

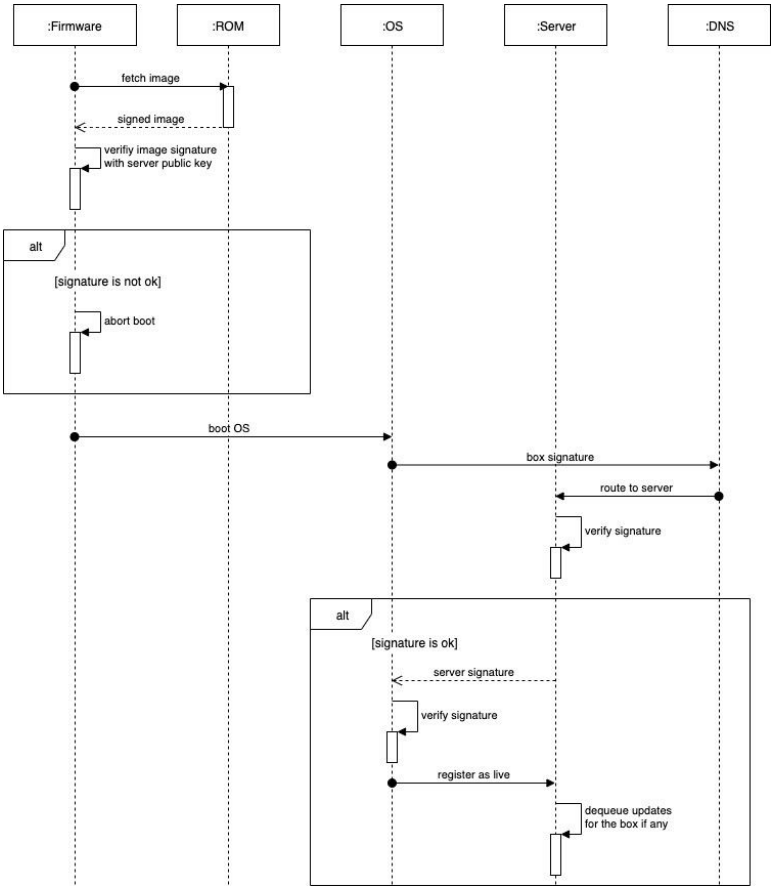


Diagramme de séquence d'un démarrage sécurisé

Mises à jour à distance

Prévention de l'intégrité du système grâce à une partition de secours, sur laquelle est sauvegardé l'état du système avant l'exécution de la mise à jour. Si la mise à jour venait à mettre le système dans un état instable, celui-ci utiliserait alors sur la partition de secours comme partition principale.
Le serveur doit s'assurer que chaque carte est en ligne avant de procéder à l'échange de clefs et l'envoi de la mise à jour. Si la carte n'est pas en ligne, la mise à jour est mise en attente jusqu'à sa reconnexion.
Une clef symétrique est créée pour chaque carte et chaque transaction, ensuite échangée avec la carte. La mise à jour est signée par le serveur puis chiffrée avec la clef symétrique créée auparavant.

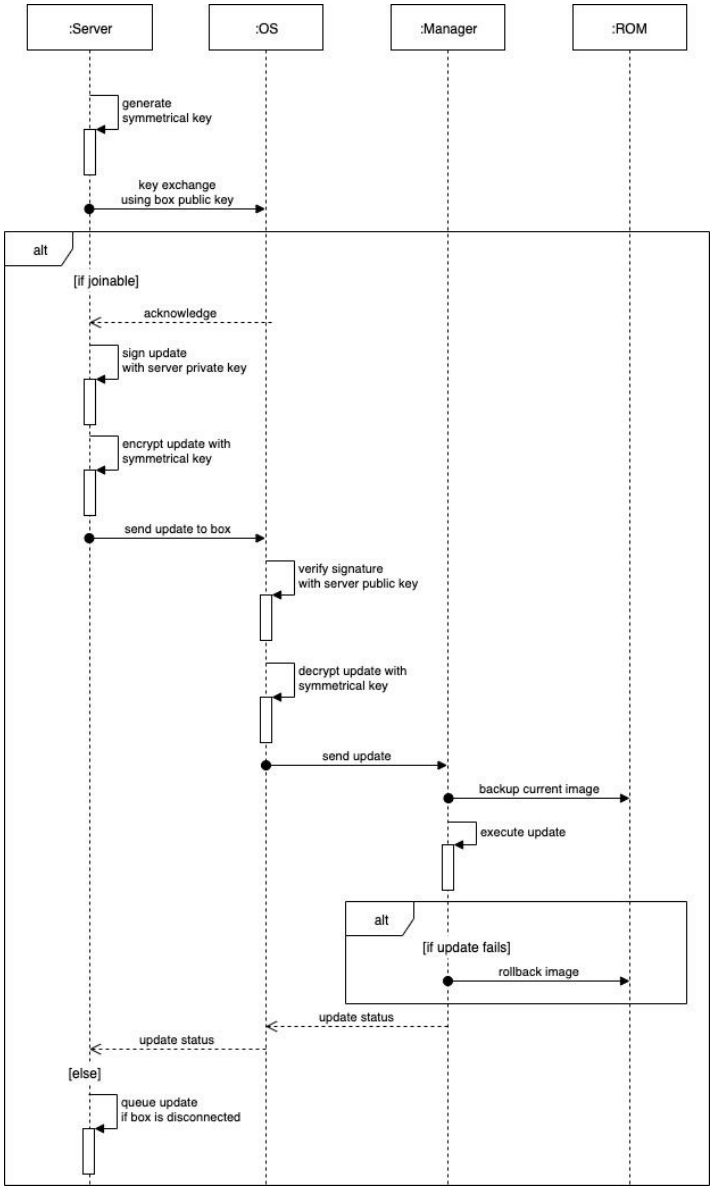


Diagramme de séquence du déploiement d'une mise à jour

Télémaintenance

Compte administrateur

Le compte administrateur sera accessible uniquement à distance avec une authentification par clef.
Le choix de l'accès à distance s'explique par le fait que la box se situe chez l'utilisateur. On ne veut pas devoir dépêcher un technicien exprès pour se rendre chez un utilisateur dès que la box de celui-ci à un problème.
Nous avons choisi une authentification par clef pour éviter toutes attaques par mot de passe. C'est un moyen d'authentification sûr si l'on considère l'entreprise comme sûre également. Si une personne a accès au compte où est la clef privée permettant l'accès au box, alors elle pourra se connecter à toutes les box.

Pour ajouter une sécurité, le compte administrateur est considéré comme un simple utilisateur. Cela préviendra de toutes actions dangereuses pour le système dans un premier temps.
Le compte de l'administrateur appartient au groupe "sudo". Cela permettra dans un second temps de pouvoir effectuer des actions root, mais avec une demande de mot de passe.

Connexion sécurisée

Le compte administrateur créé est associé à la clef publique du serveur sur chaque carte.
L'authentification à distance se fait via signature numérique du serveur.
Une fois connecté, l'administrateur accède à un terminal et peut facilement réaliser des opérations de maintenance.

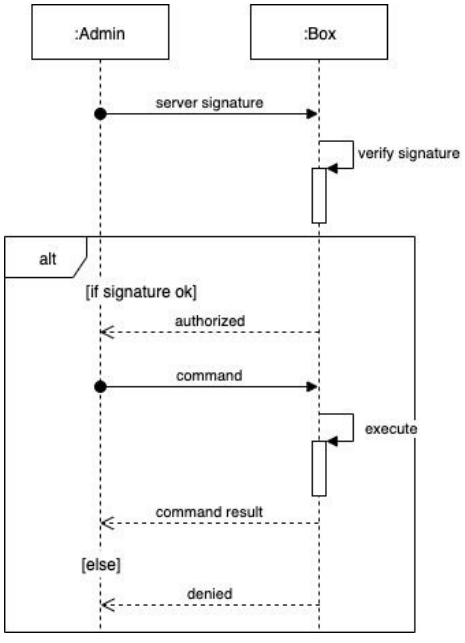


Diagramme de séquence d'une connexion à une carte et exécution d'une maintenance

Durcissement

On peut durcir un système de deux manières:

- Réduire la surface d'attaque
- Ajouter des couches de défenses

Pour réduire la surface d'attaque on peut :

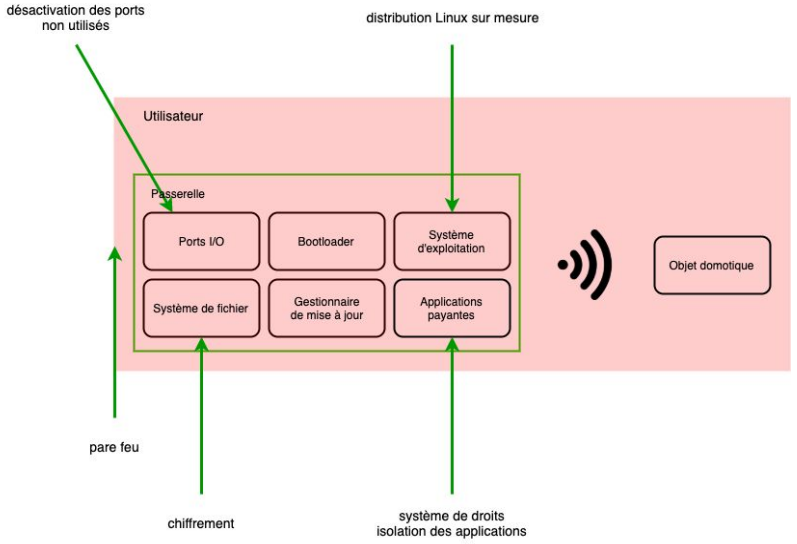
- Réduire le nombre de fonctionnalités, pour cela, il faudra lors de la réalisation, créer une distribution minimale. C'est à dire une distribution où chaque paquet fait l'objet d'une réflexion sur son utilité avant d'être intégré à la distribution.
- Réduire le nombre d'utilisateur et leurs droits. Le nombre d'administrateur devra être minimal pour des questions de sécurité. Nous ne définirons qu'un seul compte administrateur, soit un compte validé par l'entreprise qui verra sa clef publique mise dans la box pour lui permettre la connexion pour la maintenance. Les autorisations seront misent en place par une ou plusieurs politiques d'accès parmis DAC, MAC, ACL, … Nous ferons tout de même attention à ne pas multiplier les rôles et niveaux de droits, au point de créer des conflits dans les niveaux de permission.
- Réduire les communications (I/O). Il n'y aura au départ que deux accès sécurisés à la box qui seront la connexion SSH et la mise à jour. Les autres seront désactivés. Ils seront réactivés et sécurisés en cas de besoin.

Cela permet un meilleur contrôle et une meilleure connaissance du système et donc de sa surface d'attaque.

Pour ajouter des couches de défense on peut :

- Chiffrer les données notamment celles du système de fichiers.
- Ajouter un pare-feu réseau. Pour n'autoriser que les entitées reconnues à communiquer avec la carte.
- Ajouter du sandboxing. Cela permet d'isoler les applications et ainsi, dans le cas d'une application malveillante, de l'empêcher de voir le contexte d'exécution des autres.

Cela permet de réduire le risque d'intrusion et leurs impacts sur le système.



Réponses techniques apportées par le durcissement du système

V - Réalisation

Technologies

Distribution Linux

Nous avons le choix entre différents outils tel que Yocto, Buildroot, OpenWRT et les distributions dites de bureau (Ubuntu, Raspian …). Tout d'abord, selon nos recherches, OpenWRT et les distributions de bureau ne sont pas adaptées.
OpenWRT est plus indiqué pour la création de routeur et est difficile à maintenir au niveau des mises à jour. Les distributions de bureau sont conçues pour la bureautique comme leur nom l'indique. Nous avons donc le choix entre Buildroot et Yocto.
Buildroot est qualifié de simpliste et facile à apprendre. Il offre la possibilité de créer une image la plus réduite possible ce qui dans notre cas serait un atout pour réduire la surface d'attaque.
Yocto est normalement plus long à apprendre mais nous en connaissons déjà les bases ce qui est un avantage en sa faveur. Yocto est bien structuré, très modulable et dispose d'une large communauté. Il est cependant très long lors de la compilation.

Nom	Grande flexibilité	Simplicité d'utilisation	Large support de cartes	Grande communauté	Support à long terme
Yocto	oui	non	oui	oui	oui
Buildroot	non	oui	oui	oui	oui
OpenWRT	non	non	non	non	oui
Raspian	non	oui	non	non	oui

Tableau comparatif des différentes solutions de build pour l'embarqué [3]

Après réflexion nous avons choisi Yocto principalement car nous avons réalisé une introduction lors de la formation et ainsi possédons les bases d'utilisation de cet outils.

Yocto

- Yocto permet de partir d'une distribution minimale basée sur le noyau Linux. L'outil permet de prendre une image Linux basique, puis de la personnaliser en ajoutant manuellement les fonctionnalités désirées une par une.

Cet outil prend en charge automatiquement la partie rébarbative de configuration de l'environnement. À partir de variables de configuration, Yocto choisit le bon compilateur pour la plateforme demandées, gérant ainsi de façon autonome la chaîne de compilation croisée.

[3] https://opensource.com/article/18/6/embedded-linux-build-tools

La génération d'une image pour une plateforme spécifique est gérée par une couche, généralement constructeur. Ces abstractions permettent de se consacrer directement aux fonctionnalités désirées sur le projet.
L'outil met en place des couches (layers), constituées d'un ensemble de recettes (recipes). Les recettes sont des scripts Bash et/ou Python permettant de modifier ou d'installer une fonctionnalité dans le système. Chaque recette peut générer plusieurs paquets (packages) qui seront installés dans le rootFS pour mettre en place les fonctionnalités désirées.

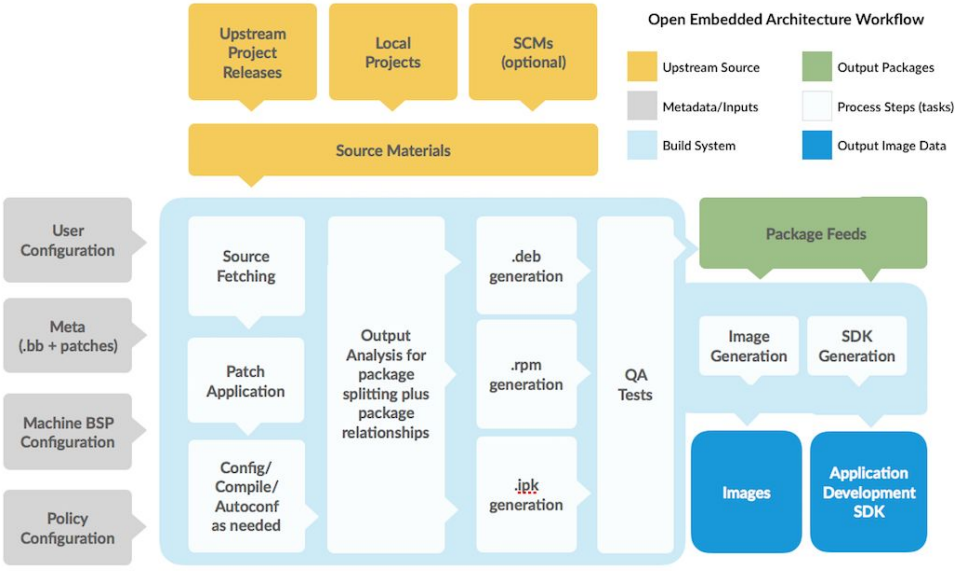
Lors de la compilation du projet par Yocto, chaque recette exécute ses actions selon un ordre prédéfinis [4]:

- D'abord les sources sont rapatriées si distantes
- Celles-ci sont compilées si ce n'est pas déjà fait
- Empaquetage et stockage des binaires pour la génération de l'image

Génération d'une image:

- Rapatriement / compilation des sources du noyau et du bootloader
- Configuration et génération de l'image du noyau et du bootloader
- Génération du rootFS et exécution des paquets pour le peupler
- Génération de l'image finale via la couche constructeur (BSP)

Nous avons choisi de partir d'une image Linux minimale que Yocto fournis comme base, permettant de réduire la surface d'attaque, et la modifierons en fonction de nos besoins.
La carte RaspberryPi étant supportée par Yocto, une couche est déjà disponible pour la création d'une image adaptée au démarrage de cette carte.



Fonctionnement de Yocto pour la génération d'une image depuis des sources (de gauche à droite) [5]

[4] https://www.linuxembedded.fr/2016/05/la-mise-au-point-des-recettes-yocto/
[5] https://www.yoctoproject.org/software-overview/

Démarrage sécurisé

Plusieurs choix s'offrent à nous :

- U-Boot
- Grub2
- BerryBoot
- NOOBS
- Lilo

U-Boot est le bootloader officiel supporté par Raspberry Pi et compatible avec la Hummingboard par défaut. Son intégration avec Yocto est native et les fonctionnalités de démarrage sécurisé modifiables depuis Yocto.
Grub2 ou Lilo sont des bootloaders pour machines classiques, mêmes si supportant le démarrage sécurisé, ils ne permettent pas de gérer des systèmes embarqués spécialisés comme la Raspberry Pi.
A l'inverse, BerryBoot et NOOBS sont spécialement conçus pour des cartes comme la Raspberry Pi. Ces bootloaders se concentrent sur l'intégration multiboot pour permettre la mise en place de bancs de tests.

Nom	ARM / Raspberry Pi	Secure boot	Système embarqué	Yocto
U-Boot	Oui	Oui	Oui	Oui
Grub2	Non	Oui	Non	Oui
BerryBoot	Oui	Non	Oui	Non
NOOBS	Oui	Non	Oui	Non
Lilo	Non	Oui	Non	Non

Tableau comparatif des différentes solutions de bootloader [6]

Le choix de U-Boot semble évident dans l'intégration de notre projet, supporté nativement par Yocto, il permet la mise en place de toutes les fonctionnalités voulues et supporte la variété de cartes ciblées lors de notre projet.

U-Boot

L'outil U-Boot est un bootloader permettant la réalisation du démarrage sécurisé comme décrit auparavant dans la conception.
Une intégration entre U-Boot et Yocto nous permet de réaliser la configuration et la génération des composants signés grâce à des recettes.

[6] https://en.wikipedia.org/wiki/Comparison_of_boot_loaders

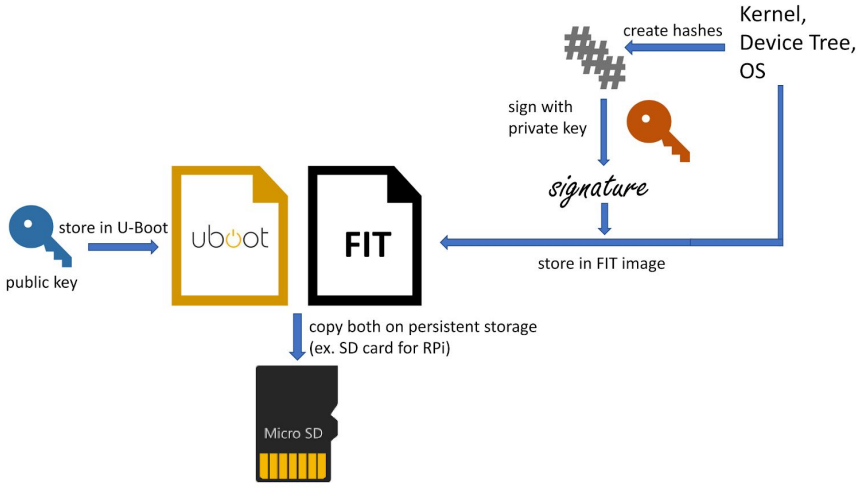
U-Boot est le premier maillon de la chaîne de confiance qui sécurise le démarrage.
Il vérifie tous les composants de la chaîne avant de passer la main au suivant. La logique de vérification est ainsi centralisée mais impose une clef unique pour tous les composants.

Pour mettre en place une chaîne de confiance, il faut une racine de confiance (maillon zéro). Celle-ci peut être implémentée de plusieurs façons:

- La première consiste à concaténer la clef de vérification dans le bootloader directement, et de stocker celui-ci sur un support ROM. Cela simplifie le processus de récupération de la clef, mais restreint sa révocation ou la mise à jour du bootloader.
- La seconde permet une mise à jour du bootloader et peut aussi implémenter la révocation ou modification des clefs. U-Boot peut être configuré pour récupérer la clef depuis un module sécurisé (TPM). Suivant les fonctionnalités supportées par le TPM, il peut être possible de mettre en place de la révocation de clefs. Le bootloader peut ainsi être stocké avec l'image du système, mais un travail d'intégration du TPM doit être fait.

U-Boot a mis en place depuis quelques années un nouveau format d'image pour le démarrage appelé FIT pour *"Flattened Image Tree"*. Celui-ci supporte de multiples configurations de démarrage à partir d'éléments communs (*kernel*, *fdt*, *ramdisk*) ainsi que leurs signatures (pour le *"verified boot"*).
Chaque composant définit son type (*kernel*, *driver*, *device tree*), son adresse de chargement en mémoire et son adresse d'entrée dans le programme (main), son architecture cible et enfin sa compression / chiffrement.

U-Boot requiert seulement la paire de clefs utilisée pour la signature (dont le certificat X501 pour la vérification), la FDT pour la plateforme choisie, l'image du noyau et une image FIT décrivant comment assembler le tout.
L'assemblage du boot et de la vérification se fait automatiquement lors de la compilation du bootloader.



Processus de génération d'une image "FIT" signée via U-Boot [7]

[7] https://blog.nviso.be/2019/04/01/enabling-verified-boot-on-raspberry-pi-3/

Mise à jour à distance

Il existe plusieurs solutions de mise à jour distantes OTA (*Over-The-Air*). Dans le tableau ci-dessous nous avons choisi de comparer les fonctionnalités des quatre solutions open-source les plus populaires. Ce tableau n'est pas un récapitulatif complet des fonctionnalités de chacune. Les critères de comparaison ont été choisis en fonction de nos besoins.

	Mender	Swupdate	Rauc	Updatehub
Compatibilité avec Yocto	Oui	Oui	Oui	Oui
Mise à jour de type symétrique	Oui	Oui	Oui	Oui
Possibilité de mise à jour locale et distante	Oui	Oui	Oui	Oui
Possibilité de choisir les fichiers à mettre à jour (rootFS ou kernel)	Non	Oui	Oui	Oui
Compatibilité avec U-Boot	Oui	Oui	Oui	Oui
Possibilité de Rollback	Oui	Non	Non	Oui
Communication sécurisée avec HTTPS	Oui	Oui	Oui	Oui
Possibilité de signer les images	Oui	Oui	Oui	Oui
Difficulté d'intégration	FACILE	MOYENNE	MOYENNE	MOYENNE

Tableau comparatif des fonctionnalités de chaque solutions [8]

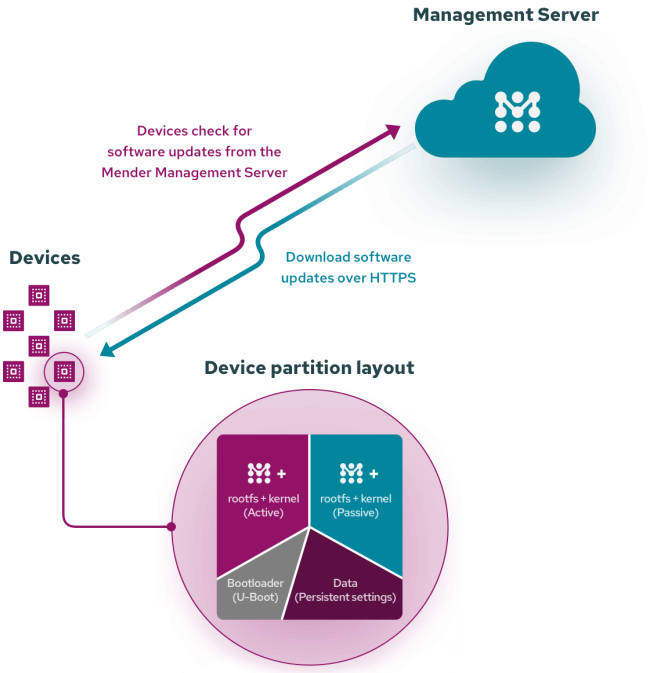
Notre choix s'est porté sur Mender, pour sa compatibilité avec Yocto et U-boot ainsi que sa communauté importante et sa documentation qui font que son intégration semble plus simple et en adéquation avec nos besoins.

[8] https://www.linuxembedded.fr/2017/11/mise-a-jour-over-the-air-de-systemes-embarques/

Mender

Mender est un logiciel open-source pour la gestion et le déploiement de mises à jour à distance en mode client-serveur pour les systèmes embarqués fonctionnant sous Linux.

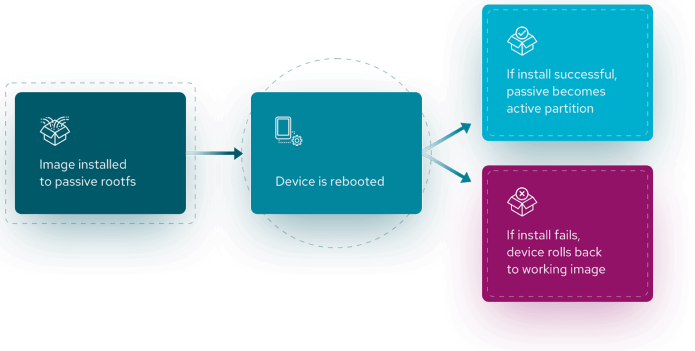
Le logiciel serveur est déployé sur un serveur web, il utilise le protocole HTTPS pour communiquer avec le client.
Lorsqu'une carte embarquant le client est démarrée, le client se synchronise avec le serveur pour décliner son identité et commencer à recevoir des mises à jour.
À interval de temps régulier, le client ouvre une connexion HTTPS avec le serveur pour demander une éventuelle mise à jour, cela permet de ne pas avoir de serveur HTTP sur la carte, ni de ports ouverts inutilement.



Pour mettre un place un canal de communication sécurisé entre le client et le serveur, le protocole HTTPS est utilisé. Le serveur dispose de plusieurs certificats qui sont embarqués dans la carte lors de sa conception.
L'intégrité des mises à jour est garantie via la signature par un système tierce, entre la création de la mise à jour et son déploiement par le serveur. Par le même principe, cela permet une vérification d'authenticité de celles-ci.

Chaque client dans chaque carte embarque sa propre paire de clefs, reconnue par le serveur, pour permettre une reconnaissance et une autorisation automatique des cartes par le serveur à chaque démarrage.
Une protection au rejeu par un identifiant unique de mise à jour est intégré au client. Le système de création de mise à jour génère cet identifiant à chaque nouvelle mise à jour.

Lorsqu'une mise à jour est disponible sur le serveur, le client va l'appliquer sur la partition inactive du système puis redémarrer dessus.
Si l'installation s'avère rendre le système instable, Mender retourne sur la partition active et en informe le serveur. Sinon, la partition inactive remplace l'active jusqu'à une prochaine mise à jour.



Système de mise à jour et restauration [9]

[9] https://mender.io/overview/how-it-works

Télémaintenance

Nous avons choisi le protocole SSH car il dispose d'un support Yocto.
Par ailleurs les autres protocoles permettant la maintenance à distance tel que telnet, RSH, RFB, XDMCP ne conviennent pas pour différentes raisons:

- Telnet communique en clair, cela va à l'encontre de notre politique de sécurité.
- RSH est comme SSH mais seulement pour exécuter une unique commande.
- RFB (remote frame buffer) et XDMCP (X Display Manager Control Protocol) sont pour les interfaces graphique. Ce qui est inutiles dans notre cas.

Choix du serveur SSH

Les logiciels implémentant le protocole SSH présents avec Yocto sont :

- Dropbear
- OpenSSH

Ces 2 solutions sont très similaires, mais OpenSSH reste cependant plus complet comme on peut le voir sur les différentes fonctionnalités qu'il propose.

Nom	SSH2	Port forwarding	SFTP	SCP	Supports IPv6	Supports authorized keys	Privilege separation	FIPS 140-2 certified
Dropbear	Oui	Oui	Partiel	Oui	Oui	Oui	Non	?
OpenSSH (OpenBSD Secure Shell)	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui

Tableau comparatif des différentes solutions de maintenance [10]

Le point qui pourrait faire pencher la balance en faveur de Dropbear est sa légèreté sur les systèmes à faible mémoire vive.
Ici la sécurité étant le point le moins négligeable, nous choisirons OpenSSH qui dispose d'une séparation de privilège intégrée[11].

OpenSSH

OpenSSH est un logiciel libre concédé sous licence BSD. Il dispose d'outils de contrôle à distance (ssh, scp, sftp), des outils de gestion de clefs (ssh-add, ssh-keysign, ssh-keyscan, ssh-keygen), et des services système (sshd, sftp-server, ssh-agent).

La mise en place du serveur SSH est facilitée par Yocto qui supporte nativement OpenSSH[12]. Pour inclure le paquet d'OpenSSH dans notre image, il suffit de le stipuler à l'aide d'une variable de Yocto.

[10] https://en.wikipedia.org/wiki/Comparison_of_SSH_servers
[11] http://www.citi.umich.edu/u/provos/ssh/privsep-f.html
[12] https://git.yoctoproject.org/cgit.cgi/poky/plain/meta/recipes-connectivity/openssh/

Mise en place

Yocto

Nous avons tout d'abord récupéré Yocto depuis son dépôt Git. Puis nous avons cherché la classe minimale dans les classes utilisées pour faire l'image minimale de Poky (la distribution de Yocto). Une fois trouvée nous avons créé notre propre image à partir de celle-ci. Notons que la classe qui génère l'image utilisée, bien qu'étant la plus légère, comporte en réalité un certain nombre de fonctionnalités qui ne seront pas utilisées pour notre projet. Ces fonctionnalités pourront donc faire l'objet d'une analyse lors de la phase de durcissement du système pour être ou non enlevées.

Dans Yocto, un point négatif est que beaucoup de recettes dépendent d'autres ou en utilisent d'autres, ce qui le rend difficile à maîtriser. On ne sait pas vraiment ce que l'on embarque dans le système après par exemple l'ajout de Mender qui requiert un nombre significatif de paquets. Un autre point négatif au fait d'avoir beaucoup de recettes est que certains fichiers peuvent être modifiés par plusieurs recettes dans un ordre inconnu, ce qui entraîne des conflits de configuration très difficiles à repérer.

Nous avons ensuite intégrer la couche constructeur "meta-raspberrypi" à Yocto. Cette couche permet notamment de donner le BSP (Board Support Package) nécessaire pour supporter la carte Raspberry Pi. Grâce à cela nous avons pu définir la machine cible référencée dans la couche *meta-raspberrypi* pour Yocto sous le nom *"raspberrypi3"*.

Pour réaliser ce projet nous avons été amené à créer différentes recettes. Par exemple nous avons fait une recette pour placer le fichier qui contient la clef de l'administrateur dans le rootFS.
Il nous est également arrivé de devoir surcharger des recettes existantes, par exemple une de OpenSSH, pour que la recette ajoute nos propres fichiers de configuration. La surcharge des recettes se fait par copie à l'identique de la structure de la recette originale. La copie sera prise en première par rapport à l'originale lors de la compilation de Yocto.
Nous avons aussi dû modifier les sources d'un paquet d'une recette avant son exécution. Pour pouvoir configurer U-Boot, cela se faisant avec les préprocesseurs C dans les fichiers sources, il faut créer un patch qui sera appliqué à l'exécution de la recette. Il y a plusieurs moyens de générer un patch : manuellement en téléchargeant les sources, faire les modifications et réaliser une difference (avec *diff* par exemple) entre les 2 versions. Sinon le projet Yocto a développé un outils nommé *"devtool"* qui facilite la manipulation de recettes, notamment pour le débogage. L'outil se charge de télécharger les sources et les utilisées en lieu et place des anciennes lors de la génération d'une image. Une fois les modifications effectuées, représentées sous forme de un ou plusieurs commits, on peut surcharger la recette et ajouter les fichiers patchs appropriés. Il ne reste plus qu'à nettoyer la recette modifiée.

U-Boot

U-Boot est très utilisé dans le monde de l'embarqué avec le noyau Linux et partagent des similitudes. Tous deux sont écrits en langage C, tous deux se configurent avec les préprocesseurs C et surtout tous deux n'ont aucun manuel utilisateur, aucun site de documentation. Tout est en README et commentaires dans les fichiers sources concernés. D'ailleurs le projet Yocto partage aussi ce dernier trait (à notre grande surprise et joie). Finalement ces outils demandent de s'impliquer un peu plus que juste passer au dessus d'un site de documentation pour framework ou librairie. Enfin, passé ces quelques désagréments, on finit toujours par trouver la documentation nécessaire (à coup de *find* et *grep*).

Fonctionnement

Dans un premier temps le but est de configurer U-Boot en dehors de Yocto. Cela permet de faire un PoC démarrage sécurisé sur la Raspberry Pi 3B+. U-Boot supporte une variété de cartes et constructeurs, dont la Raspberry Pi, ce qui facilite l'interopérabilité. Mettre en place un démarrage sécurisé sur U-Boot a été rendu très simple avec les images FIT, lorsque que l'on connaît comment marche le bootloader. Il faut donc un noyau Linux, la FDT de la carte (à trouver dans les sources du noyau ou fournie par le constructeur) et la description d'une image FIT. La syntaxe FIT n'est pas complexe et une variété de configurations d'exemple permettent une base solide.
Enfin, à partir des utilitaires U-Boot, on peut compiler une première image fonctionnelle, non signée. Ensuite viens la signature des composants de l'image, chaque composant est signé avec une clef privée dédiée au démarrage sécurisé. La clef publique associée est certifiée (format X501) par une autorité de confiance (dans ce PoC, elle sera auto-signée).
À ce point, l'image FIT contient tous les composants et leurs signatures. Il ne reste plus qu'à générer le bootloader en intégrant l'image FIT et la clef publique de vérification. Enfin, il faut partitionner l'image correctement pour la Raspberry Pi 3B+ soit capable de démarrer U-Boot.

Intégration

Même si U-Boot pourrait être configuré et intégré à Yocto lors de la génération de l'image, notre but premier est d'intégrer ce processus à Yocto via son support natif pour U-Boot.
Yocto dispose de plusieurs recettes pour supporter le format d'image FIT et sa signature avec U-Boot. En étudiant ces recettes, on peut déduire la configuration requise.
Il suffit alors d'indiquer la méthode de récupération de la clef de vérification (TPM ou dans U-Boot), l'emplacements des clefs de signature, activer le support U-Boot dans la couche constructeur et la signature lors de la génération de l'image.

Support

La couche constructeur supporte bien U-Boot, mais seulement ses anciennes images (uImage) et non les images FIT. Le support pour les images signées est donc impossible en se basant sur cette couche.
Dans la théorie on pourrait réécrire notre propre couche constructeur et reprendre la méthode de génération de l'image de celle originale mais cette méthode atteindrait ses limites avec l'ajout de Mender. Mender se base sur la couche constructeur d'une part, utilise son propre format d'image (pour son système de partitions active/inactive) et surcharge U-Boot pour mettre en place des vérifications au démarrage. Leur façon de surcharger U-Boot ne prend en charge que

les images uImage et non les FIT. Meme si Yocto et U-Boot supportent nativement les images FIT et leurs signatures, ce n'est pas forcément le cas des technologies se basant dessus.

Le démarrage sécurisé basique est donc possible sur Yocto, l'intégration U-Boot est présente et fonctionnelle (même si peu configurable). Maintenant, sans module sécurisé (TPM) ou méthode de stockage non modifiable (ROM), celui-ci est totalement inutile et n'apporte aucune sécurité sur la Raspberry Pi 3B+. C'est d'ailleurs peut être pour cela que la couche constructeur n'implémente pas la fonctionnalité, même si le support pour les images FIT est séparé du démarrage sécurisé.

Mender

Mender se veut plus accessible et mis en avant que des projets non commerciaux tel que U-Boot. Une version entreprise payante coexiste avec celle communautaire open-source. De ce modèle économique découle un besoin de visibilité pour se démarquer des concurrents (Update Hub notamment). C'est pour cela qu'un site dédié (Mender.io[13]) met la solution en avant, cette façade pour la version payante aide beaucoup la version open source, la documentation est exhaustive, claire et concise.

Serveur

Le serveur mender permet avant tout le déploiement des mises à jour aux clients.
Composé de micro services, il s'adapte bien à l'environnement cloud pour la production et Docker pour le développement. Mender fournit la configuration Docker pour mettre en place un serveur de développement (serveur de démo) puis le passer en production (sur AWS ou autre cloud provider). Chaque service peut être hébergé en local (cas des bases de données) ou sur une autre instance (pour l'API Rest ou le serveur mise à jour).

Pour notre démonstration, les services sont tous sur un même conteneur, la plupart en localhost et les serveurs d'API et mise à jour exposent leurs ports. Un réseau privé local permet de relier le serveur au client (en point à point) dans un premier temps. Plus tard un switch permettra d'ajouter une passerelle avec un DHCP, du NAT (pour la synchronization NTP essentielle à la communication client / serveur) et un DNS pour résoudre le serveur Mender lors des connexions du client.

La sécurité est garantie avec une liaison HTTPS établie par des certificats générés depuis les outils Mender. Le serveur exposant plusieurs services extérieurs, il faut autant de certificats. Ces mêmes certificats sont ensuite importés sur chaque box, dans le client, durant la production.
Plusieurs couches peuvent êtres ajoutées pour renforcer le système. Un système de signature de mise à jour permet d'en garantir l'authenticité, avec la vérification par le client grâce à une clef propre au système de signature.
L'autorisation de nouveaux clients par le serveur peut être automatisée, permettant de n'accepter que les clients reconnus. Lors de la production de l'image, une paire de clefs est attribuée au client. Le serveur ne garde que l'empreinte de la carte (son adresse MAC) et sa clef publique. Lors de sa connexion avec le serveur, le client va prouver son identité avec une signature générée par sa propre clef privée.

Client

À la base, le client Mender est totalement détaché de tout systèmes de build comme Yocto, il est d'ailleurs facile de l'ajouter à une image déjà finie. Ce n'est que pour séduire plus de clients en facilitant l'intégration de l'outil qu'a été développé des couches d'intégration dans divers outils (dont Yocto).
Dans notre cas, nous utiliserons l'intégration du client par les couches Mender dans l'outil Yocto.
Grâce aux couches d'interopérabilitées avec la Raspberry Pi et U-Boot, l'ajout du client se fait aisément en suivant la documentation. La configuration réseau du serveur, les clefs de vérification et les certificats de communication sont ajoutés à l'image en quelques lignes.

[13] https://mender.io/

Mender s'occupe de générer une image au format adéquat (comportant les 2 rootFS) qui peut directement être mise sur une carte SD et démarrée sur la Raspberry Pi. L'outil génère également les "artefacts", c'est à dire les mises à jour qui peuvent ensuite êtres déployées sur les clients via le serveur.

Mise à jour

Le flux de réalisation, génération et déploiement d'une mise à jour est très bien intégré à Yocto. D'après les modifications des recettes de la distribution dans Yocto, un ou plusieurs artefacts sont générés. Ceux-ci peuvent êtres signes pour être télécharger sur le serveur (manuellement ou automatiquement via l'API). Enfin, dès qu'un client va interroger le serveur, la mise à jour sera automatiquement téléchargée, le client va l'appliquer puis redémarrer et enfin avertir le serveur sur son état d'avancement (échec ou réussite).

Support

L'intégration Yocto n'est possible qu'avec la dernière version du projet Yocto. Dans notre cas la dernière version est "warrior". Le support de la carte Hummingboard est compromis car sa couche constructeur dont le support Yocto s'est arreté plusieurs versions auparavant. Mender ne supporte donc plus la carte Hummingboard depuis plusieurs versions.

OpenSSH

Pour mettre en place la connexion sécurisé de type SSH, nous avons besoin:

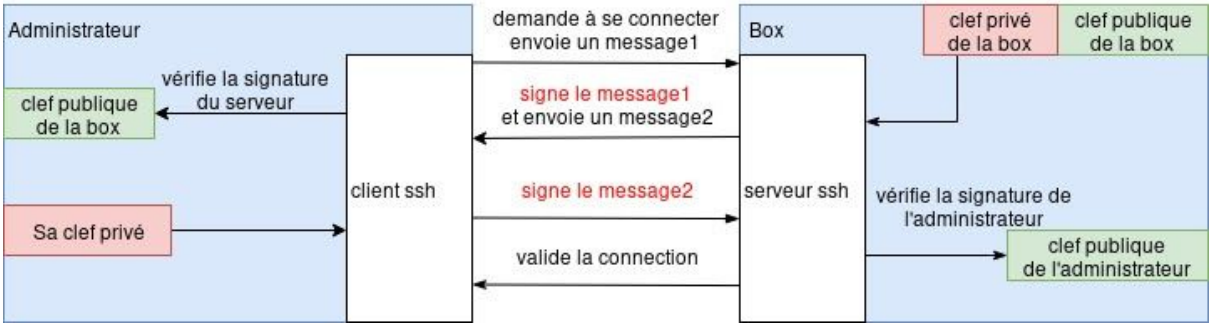
- d'un serveur SSH, la box domotique
- d'un client SSH, le poste de l'administrateur

Nous avons installé le serveur OpenSSH sur la box via Yocto. Le service du serveur nommé *sshd* se configure au démarrage grâce au fichier *sshd_config*.
Nous avons dû surcharger la recette OpenSSH car la configuration initiale de *sshd_config* ne correspondait pas à nos attentes.

Pour des questions de sécurité nous avons tout d'abord changé le port par défaut du service SSH pour nous permettre de ne pas subire les tentatives d'attaques automatiques effectuées sur le port 22. Un problème rencontré lors du projet à été que le numéro de port est définit dans le fichier configuration (*sshd_config*) mais également dans le fichier qui démarre le service démon *sshd* (*sshd.socket*). Ainsi, si les 2 ne concordent pas le service sshd échoue.

Nous avons choisi de mettre en place une authentification par clef pour l'accès à distance. Ce mode de connexion est plus robuste que celui par mot de passe.
Il nécessite de placer la clef publique de l'administrateur sur la carte. Le compte administrateur qui dispose de la clef est un compte nommé *"homeautomation"*. Nous n'utilisons pas le compte *root* sur la box et le désactivons car le nom de compte *"root"* est très répandu et est souvent pris pour cible lors de tentative de connexion.
Le client SSH dispose de la clef privée de l'administrateur qui lui permettra de signer une donnée pour se faire authentifier par le serveur.

Nous avons effectué des tests de connexion par clef sur le serveur ssh avec succès mais un avertissement est vite apparu : *"WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! "*. Cet avertissement apparaît si l'on essaye de se connecter à un serveur qui imite un serveur auquel l'on s'est déjà connecté par le passé ou bien que la clef d'identité du serveur à changée. Dans notre cas ce sont les clefs d'authentification du serveur qui changent à chaque génération d'images. Si nous avions laissé cela lors du déploiement de nouvelles images, nous ne pourrions plus vérifier l'identité de la box. Pour régler ce problème nous avons généré une paire de clefs d'identité que nous avons placé sur la box.



Client / Server SSH avec connexion par clefs

VI - Déroulement

Problèmes

Nous avons réalisé le projet minimum requis mais nous somme loin de ce que nous avions planifié. Cela peut s'expliquer par des erreurs de planifications ainsi que de nombreuses incompréhensions dû aux connaissances limitées des technologies choisies (notamment Yocto).

Notre planning ne prenait pas en compte le temps alloué pour réaliser les livrables. Sur les 7 semaines de projet allouées, 3 ont été utilisées pour l'étude du projet ainsi que la rédaction de rapports et préparation de présentations. En ajoutant les estimations biaisées du temps de nos tâches, dûes au manque de connaissances sur les technologies et d'expérience dans l'exercice, cela nous donne une première itération incomplète après le premier mois de projet.

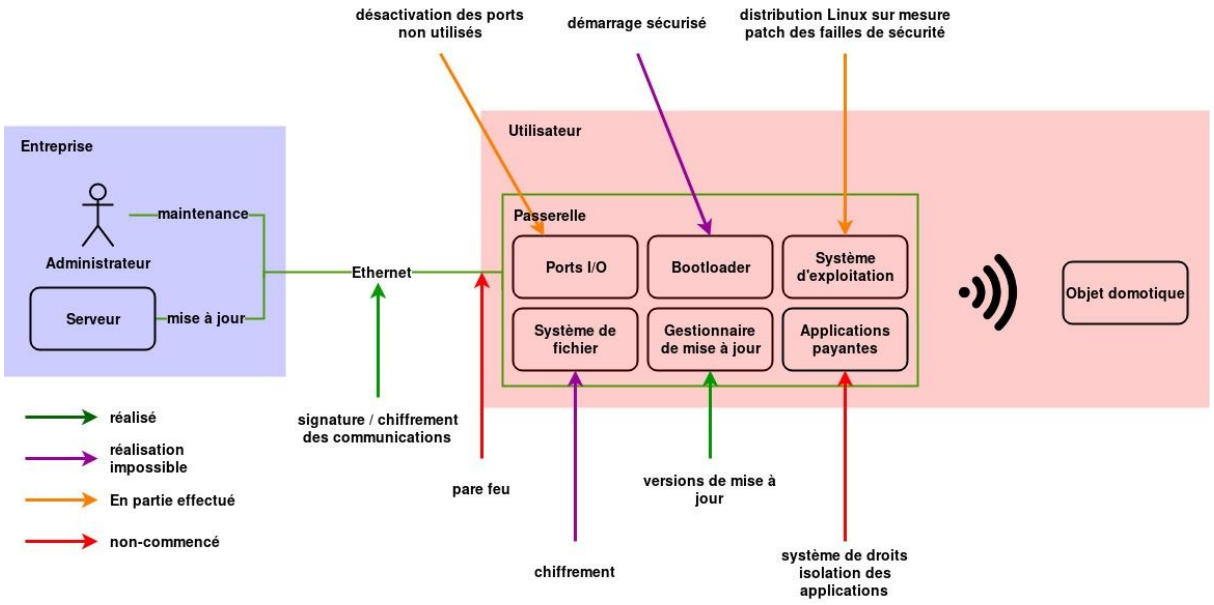
Sur les 4 semaines dédiées à la réalisation, les 2 premières ont étés beaucoup ralenties par l'intégration du démarrage sécurisé dans Yocto, qui s'est avéré incompatible donc impossible. Cela a nécessité la retro-ingénierie des recettes Yocto impliquées dans la génération de l'image FIT et sa signature. Cela à permis de retrouver les composants (FIT, DTB, kernel) et le *workflow* de U-Boot à travers les recettes. À partir de là, il ne reste plus qu'à voir l'étape bloquante, en recréant le même workflow de son côté, sans Yocto. Il s'avère que la couche constructeur Raspberry Pi ne prend pas en charge les images FIT, et refuse de modifier sa configuration U-Boot au démarrage pour permettre un support démarrage sécurisé.

Un second problème resté sans réponses à été rencontrés lors de la création d'une recette pour ajouter les clefs. À la fin de la création de l'image, seule la clef de l'administrateur était en place dans le répertoire *home*, les clefs de la box n'étaient pas installées dans */etc/ssh/*. La solution pour que toutes les clefs soient correctement ajoutées à été de faire 2 recettes séparées qui ajoutent les différentes clefs. Nous n'avons qu'une supposition sur la cause de ce problème : la clef d'administrateur doit être placée dans la partie utilisateur du rootFS alors que les clefs de la box sont dans une partie où les droits administrateur sont nécessaires. Cela induirait la présence d'une séparation des droits par recette.

Un dernier point a été lors de la mise en place de la configuration réseau automatique sur la carte, indispensable pour éviter toute intervention manuelle (configuration IP, NS, NTP) à chaque démarrage. La mise en place d'un client DHCP dans un premier temps permettrait une attribution automatique des IP, le client DHCP est un paquet fourni par Yocto qui n'a pas fonctionné tel quel dans notre projet. Nous n'avons pas pu creuser ce problème et mettre en place plus de services (NAT pour NTP, DNS) car nous devions commencer le rapport final.

Globalement, la compréhension et apprentissage des mécanismes Yocto a été sources de ralentissements. L'outil étant très complet, il restait avant de commencer notre projet des techniques nécessaires pour l'avancée du projet, encore inconnues que nous avons découvertes au fur et à mesure . La documentation interne de Yocto repose sur la compréhension de l'implémentation des mécanismes (en lisant les sources de recettes). Cette documentation implicite rend l'outil lent pour l'implémentation de nouvelles fonctionnalités. Yocto est également très opaque sur l'ordre d'exécution des recettes ce qui complique énormément le débogage si l'une d'elle ne fonctionne pas comme voulu (écrasement de variables / fichiers entre recettes indémêlable).

Avancées

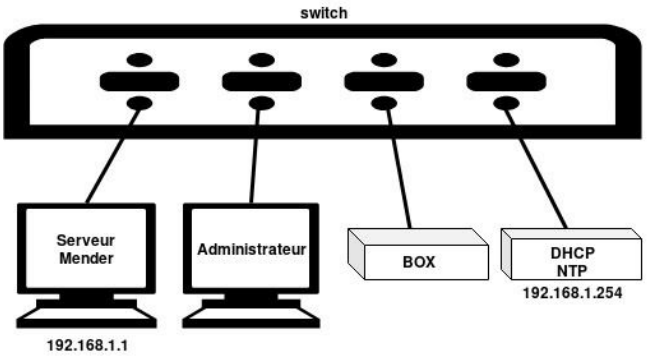


Avancées de la mise en place des réponses techniques sur le projet

Après 7 semaines de temps impartis pour le projet, dont 4 semaines de travail dessus, voici nos avancées:

- Distribution Linux personnalisé
- Système de mises à jour OTA avec Mender
- Le compte administrateur *"homeautomation"*
- Le système de télémaintenance par SSH
- Un PoC pour le démarrage sécurisé avec u-boot

Voici la représentation schématique de notre structure réseau :



Structure schématique du réseau de démonstration de notre projet

Ressentis

Rémi

Je trouve que les technologies imposées ont étés plus problématiques que bénéfiques. Yocto rajoute le travail d'intégration des composants dans l'outil en plus de leurs compréhension en dehors de Yocto. Ce n'est qu'un liant très opaque et peu flexible, du moins avec les connaissances que nous en avons. En compilant nous même les composant voulus, comme vu lors des cours, nous aurions gagner en temps et en maîtrise de notre image finale.
Bien sur il faut prendre en compte que c'est un projet qui se veut éducatif, donc un prétexte pour réaliser un produit sous Yocto et en apprendre d'avantage.
Dans ce cadre éducatif, le projet est pour moi une réussite, cela a permit compréhension des concepts / challenges du démarrage sécurisé et de la mise à jour à distance.
Utilisation d'outils déployés dans l'industrie (U-Boot et Mender), dommage d'avoir perdu beaucoup de temps sur des problèmes insolvables, car on aurait pu l'utiliser pour aller plus loin dans nos objectifs et commencer le durcissement du système Linux, par exemple.

Nicolas

Ce projet m'a permis de mieux comprendre le fonctionnement de Yocto et de compléter ce qui avait été vu en cours. Il m'a permis d'apprendre à chercher dans les documentations, dans le code déjà écrit certaines variables ou portion de code, pour ensuite comprendre quel fichier est utilisé et à quel endroit et pouvoir ainsi le surchargé.
Rémi m'a également permit de comprendre les mécanismes de Git plus rapidement que si j'avais été seul. J'ai donc pris Git en main et vu ses possibilitées qui ont été utiles pour la réalisation du projet.
J'ai également pu comprendre la configuration des serveurs ssh et comment l'implémenter sous Yocto. Cela comprend aussi la gestion et l'ajout des clefs.
J'ai également pu progresser sur ma façon de rédiger la documentation. Ce projet m'a surtout permis de m'améliorer en méthodologie plus que en technique. La technique fut en effet très peu présente à cause des problèmes rencontrés qui prennent du temps à résoudre et nécessite davantage de faire des recherches dans la documentation et le code source. La présence des très nombreux livrables à également contribuée à accroitre la partie méthodologie de projet.

Annexes

Annexe 1

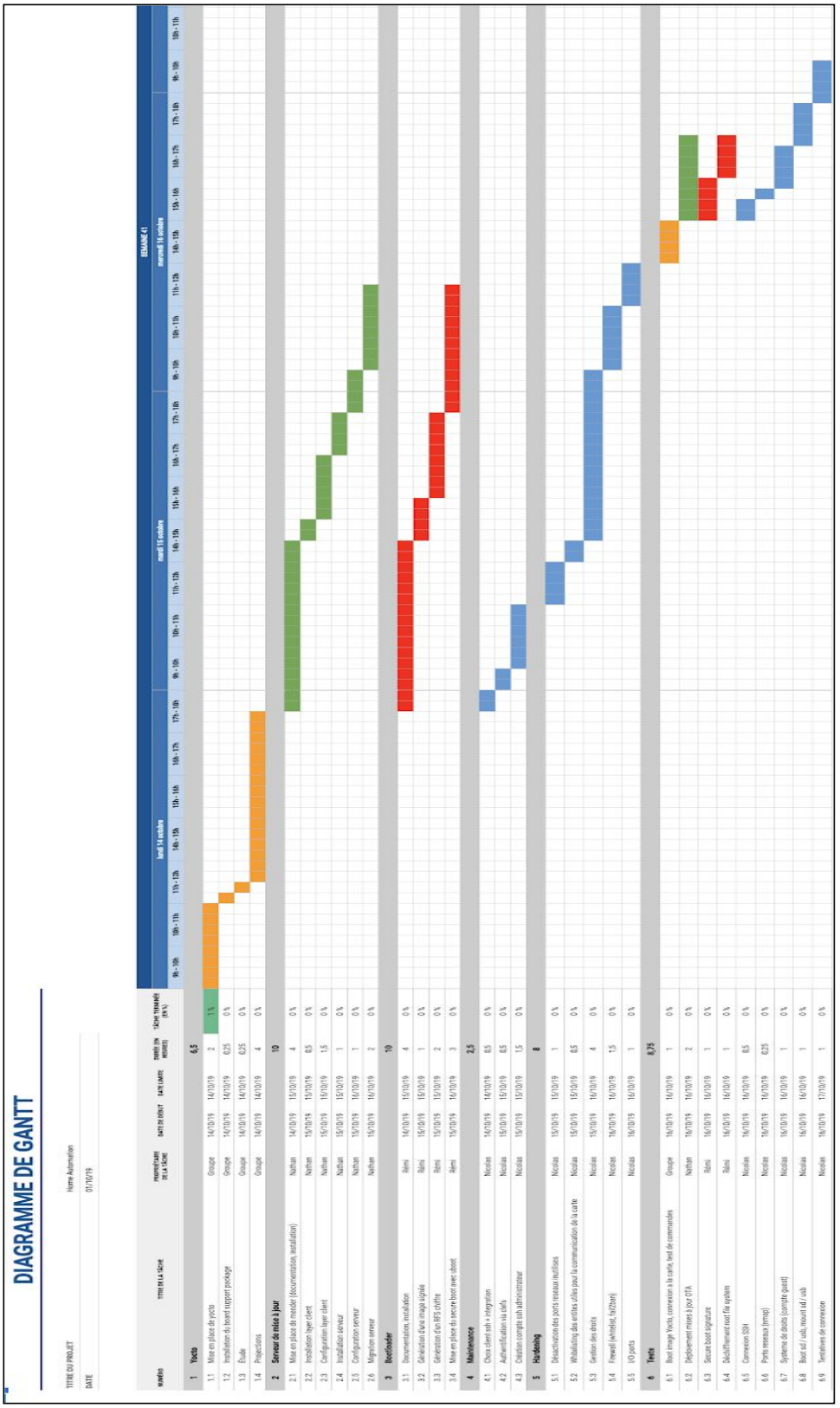


Diagramme de Gantt de la premiere iteration du projet