|  |
| --- |
| Rapport technique  projet DEVeloppement (v2)  Mise en place d’un réseau de données dans un véhicule électrique  Année scolaire 2016-2017  Projet 34 : GARDIN Alan, HACHICHA Anis, KHEDER Abdessalem  Tuteur : Mr LEGALL Guillaume  Destinataire : équipe pédagogique du projet S2 |
|  |

rÉSUMÉ

Ce document présente le travail effectué dans le cadre du projet Développement N°34 de l’année scolaire 2016-2017. L’objectif du projet était la « Mise en place d’un réseau de données dans un véhicule électrique ». C’est une plate-forme Open-Source Vehicle disponible sur le campus IMT Atlantique de Rennes et il s’agissait de fournir au terme du projet, trois applications Java (un client, deux serveurs) permettant d’échanger des données sur cette plate-forme. L’application client fonctionne sur une tablette PC utilisée comme tableau de bord, et les deux serveurs fonctionnent sur un système embarqué gérant un gestionnaire de batterie d’une part, et un contrôleur de moteur d’autre part. L’application client devait être capable de récupérer des ressources disponibles sur les deux serveurs et de les afficher sur la tablette, à la façon d’un « tableau de bord » usuel.

Le projet s’est déroulé en deux phases. La première a permis de comprendre le sujet, et de concevoir selon les besoins de l’encadrant un cahier des charges. Celui-ci a ensuite donné lieu à un planning prévisionnel. La seconde phase est une phase de développement. Dans un premier temps, il a fallu réfléchir à l’architecture des applications demandées. Cette étape effectuée, l’effort du groupe s’est concentrée sur la réalisation des solutions techniques dégagées. Les applications demandées sont disponibles et intégrées avec le code déjà existant développé par l’équipe de recherche qui travaille sur la plate-forme. La dernière étape qui reste pour clôturer le projet est de tester ces applications sur le véhicule. Pour cela un déplacement des membres du groupe sur le campus de Rennes a été programmé le 12 juin 2017.

Alan Gardin

Table des matières

[Introduction 4](#_Toc484372990)

[Expression du besoin et Contraintes 6](#_Toc484372991)

[Technologies utilisées 7](#_Toc484372992)

[Architecture 8](#_Toc484372993)

[Programmation 13](#_Toc484372994)

[Découverte de services 16](#_Toc484372995)

[Intégration 20](#_Toc484372996)

[Conclusion 21](#_Toc484372997)

[References 22](#_Toc484372998)

[Glossaire 23](#_Toc484372999)

Introduction

De nos jours, l’Internet des objets est en train d’accélérer considérablement l’allure à laquelle apparaissent les innovations dans le secteur des transports, et en particulier en ce qui concerne les voitures et les camions que nous conduisons tous les jours. Aujourd’hui, de nombreuses équipes de recherche travaillent sur ces sujets.

C’est le cas de l’équipe OCIF (Objets Communicants et Internet du Futur), qui travaille sur le campus de Rennes sur deux domaines applicatifs à savoir la gestion de l’énergie et le transport intelligent permettant de définir les besoins et les contraintes des réseaux Internet du futur et d’élaborer un terrain d’expérimentation pour valider les concepts développés. Cette équipe de chercheurs travaille en particulier sur un véhicule électrique Open-source (OSV : Open Source Vehicule). Le moteur et les batteries du véhicule sont respectivement reliés à un contrôleur de moteur « EC : Engine Controller » et un système de contrôle de batteries « BMS Battery Management System » qui fournissent plusieurs données à un système embarqué « APU » [3]. D’autre part, une application « tableau de bord » pour tablette a été développée. L’objectif de ce projet est alors d’établir la connexion entre l’APU et la tablette et à développer les applications qui permettent au conducteur d’accéder à toutes les données concernant son véhicule, ainsi que découvrir tous les appareils existants sur le réseau et répondant aux critères de définition d’un EC ou d’un BMS. C’est l’objectif de ce projet.

La progression de ce document suit la démarche adoptée par le groupe lors du projet. Dans un premier temps, nous avons commencé par une étape de découverte de tous les outils matériels et logiciels qui nous étaient imposés d’une part, ainsi que la compréhension de l’architecture générale du travail demandé d’autre part. Cette première réflexion s’est appuyée sur la rédaction du cahier de charge, et d’un planning prévisionnel. Ensuite, s’en est suivie une phase de programmation et de tests aboutissant à trois applications (un client et deux serveurs) intégrables sur le véhicule.

**Figure1 : La plate-forme OSV avec tous les composants**

Abdessalem Kheder

Expression du besoin et Contraintes

Le but du projet est de livrer des programmes Serveur et Client capables de communiquer entre eux et d’échanger des informations autour des données exploitables à partir des EC et BMS. Le langage choisi pour la programmation est Java, et le format retenu pour l’échange des ressources est le JSON [6]. Le programme client communiquera avec les programmes serveur par l’intermédiaire du protocole CoAP. Les programmes demandés utiliseront les bibliothèques Californium [4] et JSON-Simple.

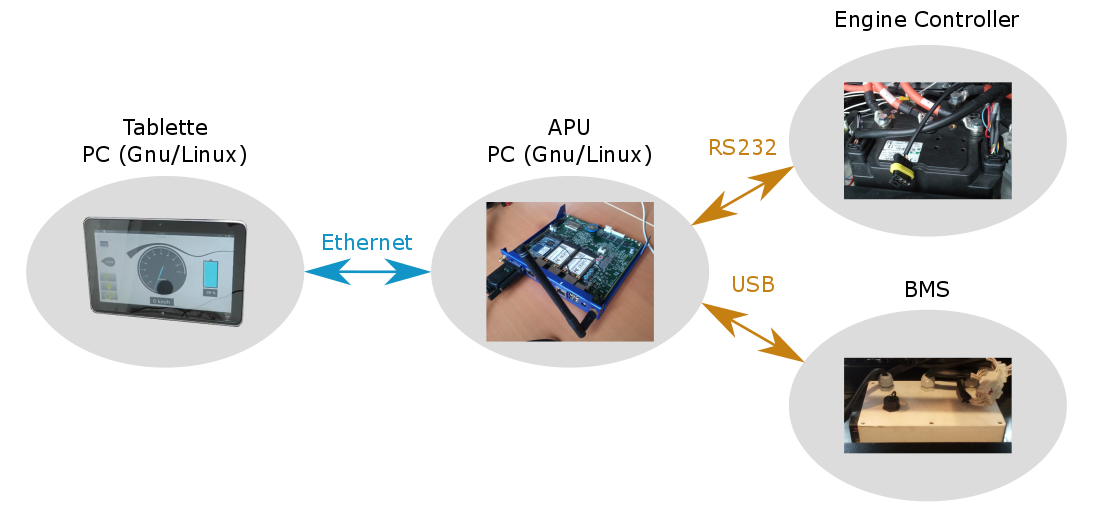
Alan Gardin

Architecture matérielle du véhicule

L’APU PC est connecté par une liaison série au contrôleur de moteur (EC) qui est capable d’acquérir et commander la vitesse du moteur. Il est également connecté par USB au BMS (contrôleur de batterie) qui est capable de commander le chargeur des cellules et de connaitre leur état de charge.

De l’autre côté, l’APU PC est connecté par Ethernet à la tablette sur laquelle tourne une application « tableau de bord ».

Anis Hachicha

**Figure2 : Architecture matérielle du système développé. D’après support de présentation de Guillaume Le Gall**

Technologies utilisées

Les API REST :

Comme expliqué dans les paragraphes précédents, notre objectif dans ce projet est d’établir la communication de bout en bout entre les différents composants du véhicule et le tableau de bord. Pour cela, nous avons eu besoin d’utiliser une API (Application Programming Interface). Cela permet aux différents composants de communiquer entre eux via une interface commune. De nombreuses API existent, et le formalisme retenu pour ce projet est celui de l’API REST (Representational State Transfert). Cette API est très utilisée car elle permet une interaction semblable à celle d’un client et d’un serveur dans le cas d’un page web : le client peut facilement accéder aux ressources mises à disposition par le serveur en utilisant une URL identique à celles utilisées dans les navigateurs internet.

Cette architecture assure également une certaine indépendance entre le client et le serveur (absence de connexion permanente) ce qui permet au client de se connecter à plusieurs serveurs et au serveur de répondre à plusieurs clients.

Le Protocole CoAP :

CoAP (Constrained Application Protocol) est un protocole proche du protocole HTTP, à l’exception qu’il a été pensé pour l’internet des objets. Il est dont particulièrement adapté aux besoins du projet. CoAP est conçu pour les applications machine à machine, permet de manipuler des ressources à travers un modèle d’interaction client-serveur qui échangent des requêtes-réponses, et fournit une découverte intégrée de ces ressources. Il est basé sur l’architecture REST. En outre, CoAP permet d'observer les ressources dans un modèle de publication / abonnement. Un serveur notifie les clients qui se sont abonnés à une ressource spécifique lorsque celle-ci change.

Anis Hachicha

Bibliothèque Californium

Californium est une implémentation en Java du protocole CoAP. Les classes sur lesquelles se sont porté notre attention sont CoapClient et CoapServer.

Alan Gardin

Format JSON

JSON (JavaScript Object Notation – Notation Objet issue de JavaScript) est un format léger d'échange de données. Il se présente de la façon suivante :

**{"id":1, "language":"json", "author":"Groupe\_DEV"}**

Il sera utilisé pour présenter de façon structurée les données. Il est facile à lire ou à écrire et aisément analysé ou généré par les machines et les humains.

Abdessalem Kheder

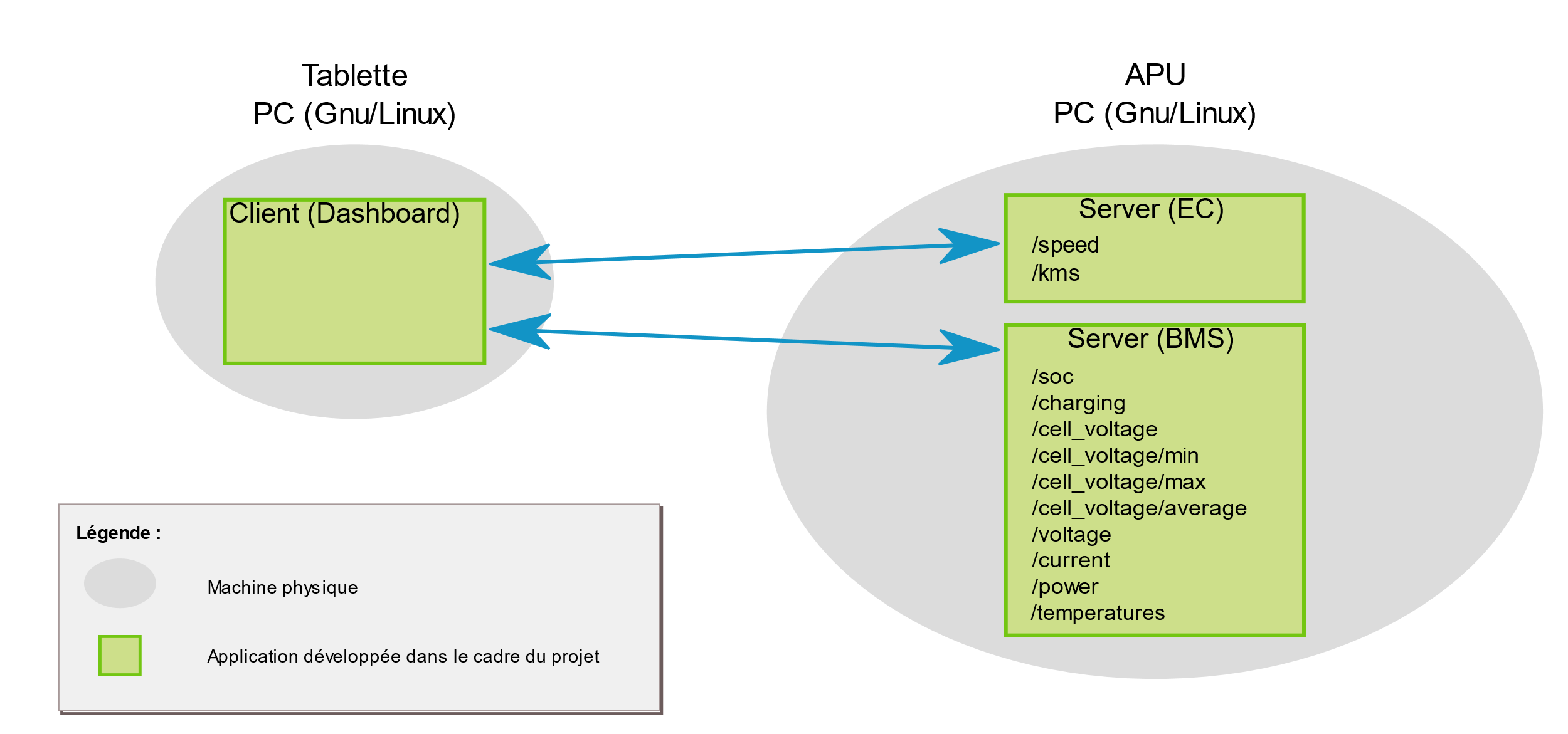
Architecture

Architecture globale du réseau :

Notre réseau est constitué de 3 nœuds :

* Un client qui envoie les requêtes, reçoit les réponses et les affiche sur le tableau de bord du véhicule.
* Un serveur qui fournit au client les données relatives à l’EC.
* Un serveur qui fournit au client les données relatives au BMS.

La figure 2 présente la liste de toutes les ressources du réseau. Comme nous avons mentionné dans les parties précédentes, notre code doit pourvoir s’intégrer avec d’autres applications qui existent déjà.

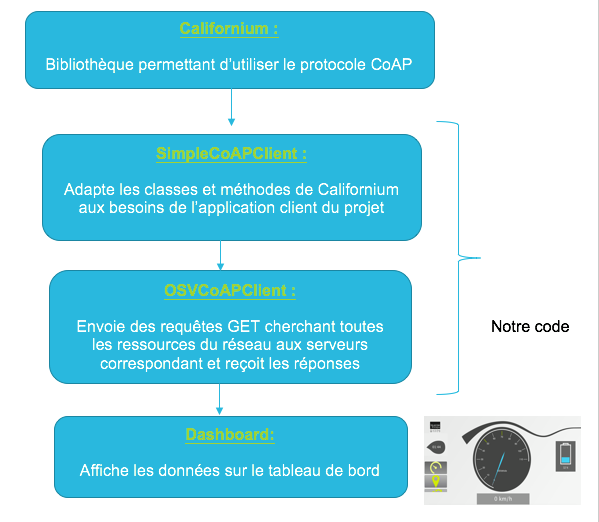


**Figure 3 : Architecture des applications et liste des ressources. D’après support de présentation de Guillaume Le Gall**

Alan Gardin

Coté Client :

En effet, du côté du client, une application tableau de bord Dashboard avait été développée en Java (voir figure 1) par l’équipe OCIF. Cette application fait appel à la classe Java OSVCoAPClient que nous avons développé. Cette dernière effectue des requêtes vers l’APU, reçoit les réponses sous format JSON et les convertit en données affichables sur le tableau de bord. Pour que cette classe puisse hériter des méthodes de la bibliothèque Californium d’une façon simple et adaptée à notre projet, nous avons créé une classe Java intermédiaire SimpleCoapClientqui s’intéresse seulement à l’aspect réseau. En effet, la bibliothèque Californium fournit de nombreuses méthodes qui, pour la plupart ne sont pas utiles à notre application. Cette classe permet donc de fournir à la classe fille des méthodes de plus haut niveau que celles proposée par Californium. On a obtenu l’architecture de la figure 3.

**Figure 4 : Architecture client**

Abdessalem Kheder

Coté Serveur

Du côté des serveurs, on a la même architecture pour les deux serveurs. Prenons le cas du contrôleur de moteur EC : les valeurs mesurées par cet appareil sont stockées dans l’APU et peuvent être lues grâce à la classe ECDataReaderqui nous a été fourni par l’encadrant. De notre côté, nous avons développé l’application server-ec et les classes KMSResourceet SpeedResourcequi fonctionnent de la manière suivante :

A chaque fois que server-ecreçoit une requête GET de la part du client demandant par exemple la vitesse du véhicule, elle appelle SpeedResourcequi, en appelant ECDataReader**,** obtient la valeur de la vitesse et la convertit en format JSON (sérialisation) puis server-ecrenvoie cette réponse obtenue au client qui la demande.

Puis, nous avons développé la classe SimpleCoAPServerqui sert d’interface entre la bibliothèque californium, l’implémentation complète du protocole CoAP, et notre application server-ec. Le schéma de la figure 4 représente l’architecture actuelle de l’application.

Anis Hachicha

**Californium :**

Bibliothèque permettant d’utiliser le protocole CoAP

**SimpleCoAPServer :**

Adapte les classes et méthodes de Californium aux besoins de l’application serveur du projet

**Server-ec:**

Reçoit des requêtes GET cherchant les ressources disponibles et renvoie les réponses

Notre code

**SpeedResouce:**

Sérialise la vitesse du véhicule en JSON

**KMSResouce**

Sérialise la distance parcourue en JSON

**ECDataReader :**

Lit les valeurs mesurées par l’EC

**Figure 5 : Architecture Serveur**

Programmation

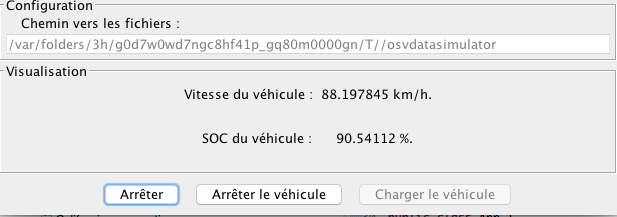
Requête GET du protocole CoAP

La fonction la plus basique de notre projet est qu’un client puisse effectuer une requête GET. Une requête GET consiste à demander au serveur l’envoi d’une ressource particulière. Si le serveur possède cette ressource, il envoie les données correspondantes au client. Nous avons donc commencé par réaliser cette première tâche en créant les classes SimpleCoAPClient et SimpleCoAPServer. Pour les tester, nous avons développé deux applications de tests ClientTest et ServerTest et à travers les sorties de ces deux applications, on a pu vérifier que la requête GET se réalise avec succès.

Anis Hachicha

Lecture des données par les serveurs :

Lors de la phase de programmation, notre encadrant, M. LEGALL nous a fourni un simulateur du véhicule pour qu’on puisse avancer sur le projet sans travailler sur le véhicule réel qui se trouve à Rennes. Il s’agit d’une application qui permet de démarrer, arrêter et charger un véhicule virtuel tout en affichant les valeurs des différentes ressources. Il nous a aussi fourni des programmes qui permettent d’extraire les valeurs des ressources à partir de ce simulateur : ECDataReaderet BMSDataReader. Nous avons donc développé pour chaque ressource une classe permettant d’obtenir les valeurs à partir des de ces deux classes et de les convertir en format JSON. Puis nous avons programmé les deux applications serveurs server-ecet server-bms**.** Ainsi, à la fin de cette étape, on a obtenu deux serveurs qui, à chaque demande de ressource par un client, cherchent la valeur de cette ressource, la sérialisent sous format JSON, et l’envoient au client.

****

**Figure 6 : Capture écran du simulateur de l’OSV. D’après [7]**

Abdessalem Kheder

Observation des ressources :

L’observation des ressources consiste à s’abonner à une ressource particulière, et recevoir une notification du serveur dès que cette ressource change de valeur. Toutes les ressources serveur ont la capacité d’être observée, mais l’application client s’intéresse seulement aux ressources speed et soc (vitesse du véhicule et état de charge).

Actuellement, l’application client est correctement prévenue dès que les ressources soc ou speed changent de valeur. Cela est mis en place par l’intermédiaire de Handler qui sont des classes appelées dès que la notification de changement de la ressource est reçue.

Alan Gardin

Intégration avec le tableau de bord :

Notre code de la partie clientqui permet d’accéder à toutes les ressources doit pouvoir s’interfacer avec l’application tableau de bord déjà existante.

En effet c’est une application avec interface graphique qui affiche les valeurs des ressources. L’intégration de notre code avec l’application déjà existante, permet une mise à jour en temps réel sur le tableau de bord à chaque fois qu’elles changent dans le BMS ou l’EC. D’après nos tests l’application est réactive au changements de valeurs du simulateur, avec une latence mineure selon les machines utilisées. 

**Figure 7 : interface graphique de l’application tableau de bord. D’après [7]**

Alan Gardin

Découverte de services

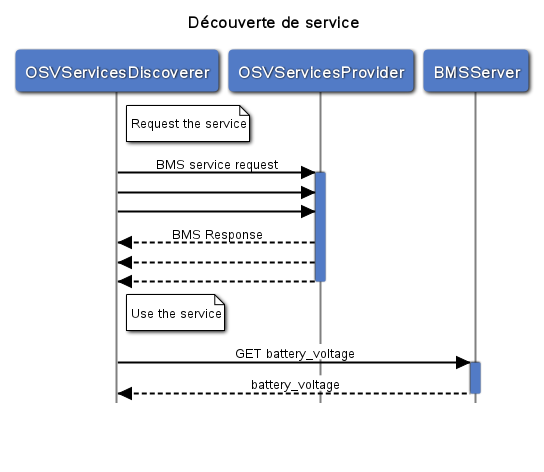
Les tâche « majeures » étant terminées, et avec l’accord de l’encadrant, nous avons pu entamer cette tâche qui avait été définie comme d’importance mineure dans le cahier des charges. A cet effet, M. Le Gall nous a fourni un document qui précisait le protocole à employer ainsi que les différentes contraintes exigées (en particulier en termes de délai pour limiter la bande passante, fonction d’autant plus importante pour les systèmes embarqués à ressources limitées.). Nous nous sommes abondamment appuyés sur ce document pour la réalisation de cette tâche. Dans la suite, nous y référerons par le terme « documentation ».

Le principe de la découverte des services est le suivant : un client envoie des requêtes pour « découvrir » l’existence de serveurs EC ou BMS (ou d’autres services dans le futur) sur le réseau. Les serveurs, en recevant cette requête, répondent au client en fournissant l’adresse et le port sur lesquels ces services sont disponibles. Le client peut alors utiliser les ressources mises à disposition par ce serveur.

Il y a plusieurs utilisations possibles de cette fonction. La première utilisation est la configuration automatique de l’application Dashboard. En effet, il suffit à cette application de découvrir les services BMS et EC sur le réseau pour déterminer les adresses et ports à utiliser pour la suite du fonctionnement. D’autre part, cela permet, sur le long terme, de découvrir d’autres applications répondant aux critères de BMS et EC sur le réseau : on pourrait imaginer deux contrôleurs de moteur sur le réseau au lieu d’un seul, et il n’y aurait alors que peu de code à modifier pour intégrer l’utilisation de ce deuxième contrôleur dans l’application Dashboard. Cela permet donc une intégration relativement aisée de futurs équipements sur le véhicule, à condition que ces derniers disposent d’une interface similaire.

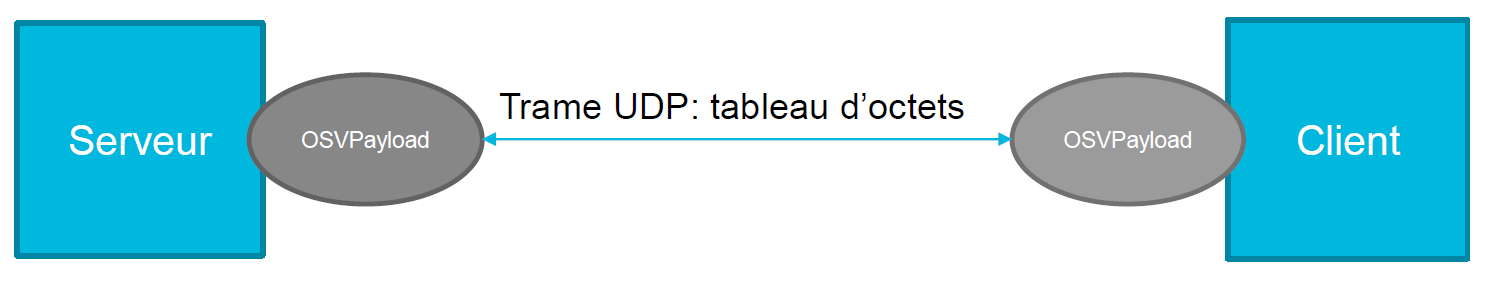
Le protocole spécifié repose sur le protocole UDP. Puisqu’il s’agit d’un mode non connecté, le document impose que tous les paquets soient envoyés trois fois pour garantir une réception relativement fiable. En effet, contrairement à TCP qui utilise un mode connecté, où l’envoi de données est précédé par une connexion entre les deux nœuds du réseau, UDP ne possède pas de fonctionnalités permettant de s’assurer que la réception a bien eu lieu. Pour répondre aux objectifs fixés, nous avons programmé trois classes majeures :

* OSVPayload, une classe permettant une manipulation « haut niveau » du contenu des trames envoyées.
* OSVServicesProvider, une classe de type « serveur » qui écoute les requêtes client sur le réseau et se contente d’y répondre.
* OSVServicesDiscoverer, une classe de type « client », qui permet de lister les différents services disponibles sur le réseau.



**Figure 8 : Diagramme de séquence de la découverte de services**

OSVPayload

Cette classe permet une manipulation « simple » du contenu des trames UDP. Effectivement, le contenu des trames UDP se présente sous la forme d’un tableau d’octets. Cette représentation n’est pas facilement manipulable, d’où la nécessité de fournir une « interface » de plus haut niveau pour sa manipulation. Cette opération, dite de « sérialisation » permet de convertir simplement les données vers un format d’échange. Cette classe est utilisée de deux manières : soit pour accéder facilement aux informations contenues dans un tableau d’octet, soit pour créer facilement le tableau d’octet selon les paramètre choisis. On stocke dans cette classe l’ID du service, son adresse, son port, et le tableau d’octet équivalent. L’ID du service est un nombre entier désignant le type de service. Pour le moment, seulement deux services sont disponibles : le BMS et l’EC. La documentation fourni l’entier associé à chaque service.

**Figure 9 : Utilisation de OSVPayload**

OSVServicesProvider

Cette classe se contente d’écouter les requêtes de type client sur le réseau. En particulier, cette application écoute les paquets sur une adresse « multicast », c’est-à-dire que plusieurs applications peuvent écouter en même temps les paquets à destination de cette adresse. En d’autres termes, l’utilisation d’une adresse multicast permet à un appareil de contacter tous les équipements sur le réseau en même temps, sous réserve que ceux-ci écoutent sur cette adresse. Il existe plusieurs adresses multicast disponibles avec le protocole IPv6. L’adresse retenue a été spécifiée dans le document.

D’autre part, l’application écoute sur un port arbitrairement choisi et défini dans la documentation. Quand un paquet correspondant à une requête client est reçu, l’application vérifie si elle n’a pas déjà reçu une requête identique il y a moins de secondes. Si c’est le cas, la requête est ignorée, sinon elle est traitée : on répond à la requête et on ajoute cette dernière, accompagnée de l’instant de réception, dans la liste des requête reçues. Le fait d’ignorer les requêtes identiques est une contrainte imposée dans la documentation. Cela permet de limiter l’utilisation du réseau. La valeur du temps d’attente est également fixé par le document. Les requêtes datant de plus de sont automatiquement supprimées lorsque l’on consulte la liste. Pour cela, on parcourt la liste et on supprime les éléments dont la date de réception est à plus de 10 secondes de la date actuelle.

OSVServicesDiscoverer

Cette classe possède une liste des services actuellement disponible sur le réseau, avec respectivement le type de service, l’adresse sur laquelle ce service est disponible, ainsi que le port ou le serveur écoute. Cette classe fournit une méthode void discover(OSVService service), qui permet de mettre à jour cette liste en envoyant des requêtes sur l’adresse multicast et le port spécifié par la documentation.

Le document de spécification impose qu’un client ne doit pas envoyer des requêtes identiques espacées de moins de secondes. Cette valeur est également définie dans le document fourni par l’encadrant. Cette contrainte fait écho à celle imposée sur la classe OSVServicesProvider : il s’agit de limiter l’utilisation de la bande passante du réseau.

Alan Gardin

Intégration

À la fin de la phase de développement, nous avons vérifié que nos programmes fonctionnent correctement en lançant à la fois le simulateur et l’application du tableau de bord et en comparant les valeurs des ressources affichées par ces deux applications. Le développement effectué jusqu’à présent a seulement été testé sur une machine en réseau local. Ensuite, il faudra tester notre programme sur le véhicule réel et donc un déplacement au campus de Rennes a été programmé. Ainsi nous pourrons installer les applications serveur sur l’APU du véhicule, placer la tablette en face du volant, et essayer tous les scénarios possibles (déplacement, chargement, accélération etc.…) pour vérifier que les applications réagissent correctement.

Alan Gardin

Conclusion

Le travail effectué répond aux tâches importantes du cahier des charges. Les tâches restantes sont de l’ordre du peaufinage, sauf pour la découverte de services. En effet, il reste de nombreuses améliorations possibles concernant cette fonctionnalité. Par exemple, la lecture des paquets n’est pas assez souple : si une information arrive avant l’autre (port déclaré avant l’adresse), la lecture est erronée. Il convient donc de programmer de manière plus « générique » la lecture des paquets. Cependant, ces fonctionnalités concernent les perspectives futures d’évolution du véhicule, et ne sont donc pas essentielles au fonctionnement du véhicule.

Alan Gardin

References

BIBLIOGRAPHIE :

1. PAULI D., OBERSTEG D. «Californium». Research Group For Distributed System, ETH Zurich, printemps 2010.

SITES INTERNET :

2. OSVEHICULE. OSVehicule [en ligne] Disponible sur : <https://www.osvehicle.com/> (consulté le 4/2/2017).

3. PC Engines BmbH. Apu Platform [en ligne] Disponible sur : <http://pcengines.ch/apu.htm> (consulté le 6/2/2017).

4. ECLIPSE. Californium [en ligne] Disponible sur : <https://github.com/eclipse/californium> (consulté le 8/2/2017)

5. INDIANA UNIVERSITY. CoAP Tutorial for Eclipse [en ligne] Disponible sur : <http://cs.iupui.edu/~xiaozhon/course\_tutorials/Coap\_tutorial\_Arduino\_Eclipse> (Consulté le 8/2/2017)

6. AYDAR A. and al. JSON with Java [en ligne] Disponible sur : https://www.tutorialspoint.com/json/json\_java\_example.htm (consulté le 28/2/2017)

7. Code existant développé par l’encadrant. Disponible sur : <http://its.ipv6.enstb.fr/code/gitweb/?p=s2-dev-2017.git;a=summary> (consulté le 6/06/2017)

Abdessalem Kheder

Glossaire

OCIF : Objets Communicants et Internet du Futur, équipe de chercheur basée sur Rennes.

OSV: Open Source Vehicule.

BMS: Battery Management System. Il s’agit d’un système de contrôle de la batterie.

EC : Engine Controller. C’est le un système qui contrôle le moteur du véhicule.

APU : Système embarqué qui donne accès aux ressources du BMS et de l’EC.

CoAP : Protocole utilisé pour l’échange de données.

Californium : bibliothèque pour le protocole CoAP en Java.

JSON : format de présentation des données.

JSON-Simple : bibliothèque pour la gestion du format JSON en Java.

GET : méthode du protocole CoAP lancée par le client et récupère l’information du serveur.

Anis Hachicha