Mathieu BOISNARD
Valentin FRIES
Mathieu MILANO

ESGI 2015-2016
3A AL

Raspberry

Cubic Control System

TABLE DES MATIERES

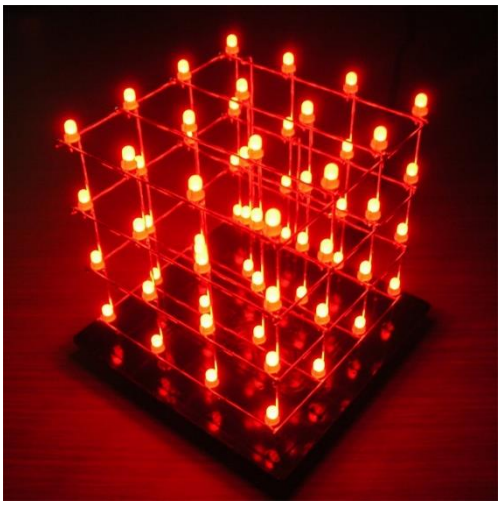
Annexes

1. Mockup RC2S-Client
2. Mockup NodeJS
3. Diagramme de Gantt

INTRODUCTION

LE PROJET

Afin de vous divertir en soirée et de détendre l'atmosphère au travail ou à la maison, nous vous proposons le Raspberry Cubic Control System (**RC2S**) !



Composé d'un ensemble de cubes de LED s'animant au rythme de la musique ou du son ambiant, la suite RC2S vient s'étoffer d'un ensemble de solutions logicielles permettant l'administration complète des différents cubes en votre possession. Une dimension créative s'y ajoute, l'extensibilité offerte par la création de plugins faisant de votre imagination la seule limite à son évolution.

Dans sa version initiale, la suite RC2S propose à l'utilisateur de gérer les différents cubes de LED qu'il possède, d'interagir avec ces derniers en allumant ou éteignant chaque LED qui les composent, de jouer les différentes animations préenregistrées… Il aura également la possibilité d'installer des *plugins* créés pour améliorer l'expérience utilisateur en ajoutant de nouvelles fonctionnalités.

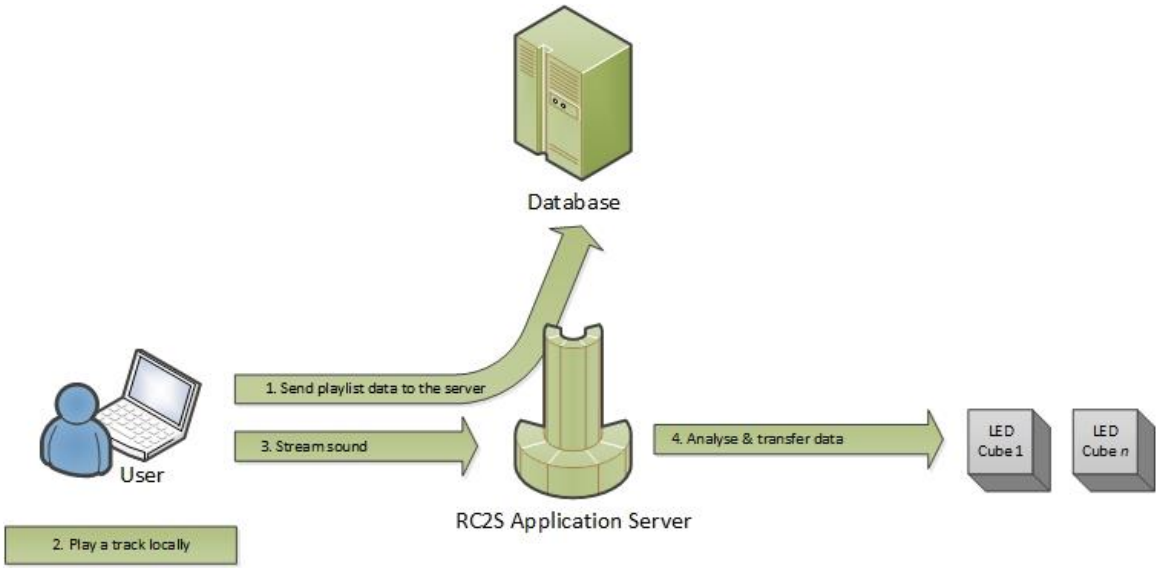
Un utilisateur avancé pourra, quant à lui, profiter des mêmes fonctionnalités tout en ayant la possibilité de créer lui-même ses propres *plugins* à travers une API dédiée. Il pourra ainsi développer de nouveaux contenus qui pourront être intégrés dans la solution RC2S et partagés avec la communauté !

Par défaut, un *plugin* exclusif est livré avec la solution afin d'illustrer les différentes possibilités offertes par le logiciel de création, ajouter de nouvelles fonctionnalités et donner envie aux utilisateurs de créer leurs propres *plugins* :

PLUGIN - MUSIC PLAYLIST

Sélectionnez vos morceaux de musique préférés, faites-les glisser dans votre fenêtre **RC2S** et cliquez sur *Play !*

Le plugin *Music Playlist* vous permettra de composer à volonté des listes de pistes musicales qui feront se mouvoir les LED des cubes connectés en fonction du rythme du morceau en cours de lecture.



LA SOLUTION

Afin de créer une expérience utilisateur unique, la solution RC2S est composée d'outils qui ont tous été pensé pour répondre à un besoin précis :

RC2S Pi Station : Ensemble matériel composé d'un Cube de LED connecté à une carte Raspberry Pi.

RC2S Daemon : Programme Java installé sur la carte Raspberry Pi recevant des instructions de la part du serveur et permettant le contrôle du Cube de LED connecté sur la même carte.

RC2S Application Server : Serveur d'application JEE centralisant la gestion de Pi Stations et servant de relais pour les clients.

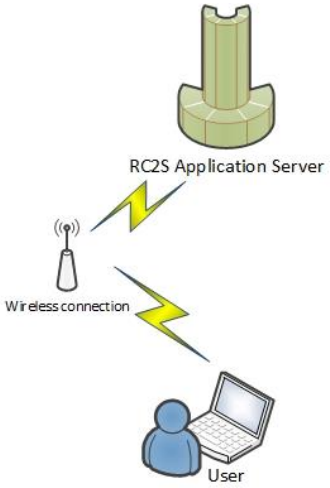
RC2S Client : Client lourd JavaFX permettant la connexion à un serveur d'application RC2S et la gestion des Cubes liés à un utilisateur.

RC2S Plugin Maker : Environnement de développement de plugins pour la suite RC2S, intégrant entre autre fonctionnalité un parser d'annotations pour faciliter et alléger le développement. Les plugins ainsi créés pourront interagir avec le serveur et le client, permettant l'ajout de nouvelles fonctionnalités suivant les besoins de l'utilisateur.

ARCHITECTURE

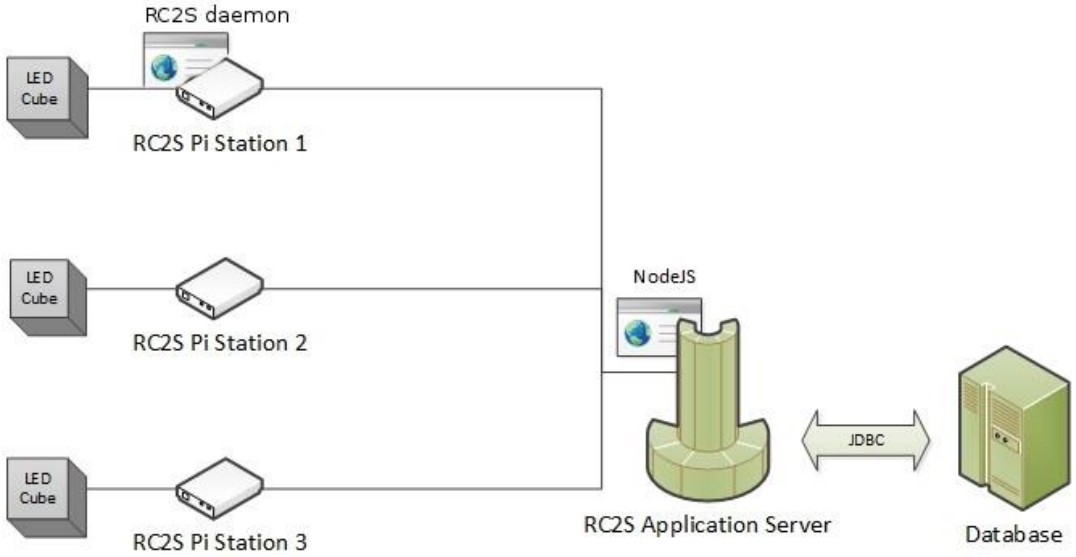
ARCHITECTURE LOGICIELLE

Nous souhaitons utiliser une architecture souple, favorisant la mise en œuvre de technologies répondant à des besoins précis. Ainsi notre solution est bâtie à travers l'utilisation de Java 8 et JEE, du moteur de production Gradle et du serveur d'application Payara – basé sur GlassFish –, la rendant solide, fiable et modulaire.



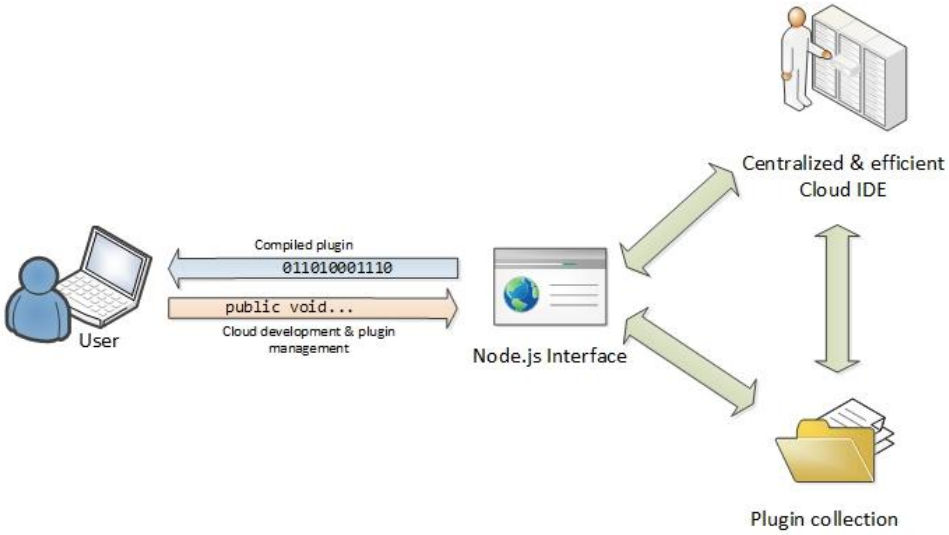
Avant toute chose, un utilisateur devra se connecter à un serveur d'application depuis le client lourd RC2S – il peut s'agir d'un serveur distant comme local. Nous avons fait le choix d'utiliser un serveur d'application JEE pour centraliser la logique métier de notre solution.

Les clients et le serveur d'applications communiqueront grâce au protocole RMI, permettant l'appel de méthodes Java et l'échange d'objets entre les différentes plateformes.

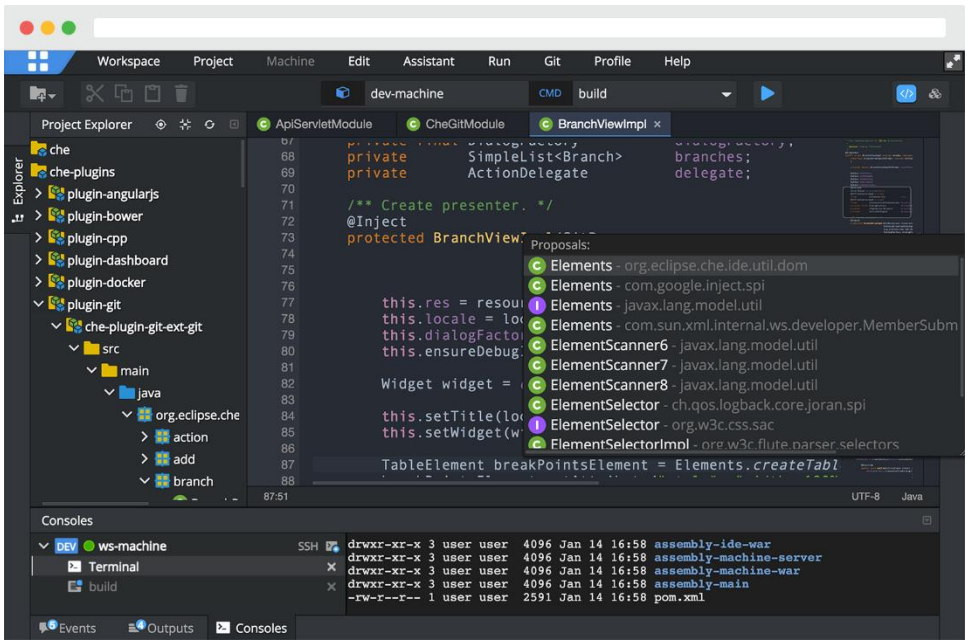


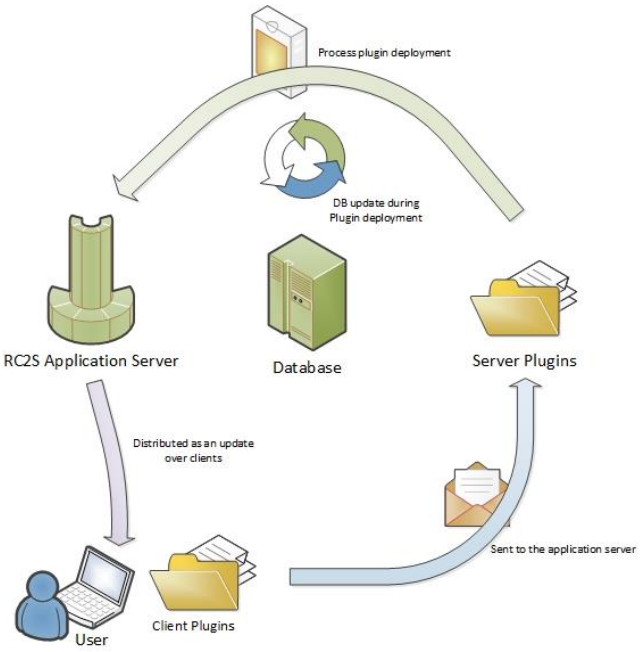
Le serveur d'application sera lui-même connecté à un ou plusieurs Raspberry Pi, permettant d'interagir avec leur cube de LED respectif.

Le *daemon* de chaque Raspberry Pi effectue la liaison entre les instructions envoyées par le serveur et les signaux électriques transmis à son Cube. Il s'agit d'une couche d'abstraction qui permet de décharger le serveur de la transcription « physique » de ses informations, tout en sécurisant l'accès au matériel.



Le développement de *plugins* sera facilité par le **Plugin Maker**, qui servira à la fois d'environnement de développement et de validation des plugins créés par les utilisateurs. Fonctionnant en mode *Cloud*, il permet de garder le contrôle sur les normes de développement utilisées lors de la création d'un nouveau plugin. Une interface intuitive permettra aux utilisateurs de gérer leurs plugins de manière efficace et de profiter d'un véritable environnement de développement *Cloud* grâce à la technologie **Eclipse CHE**.





Lors de l'ajout d'un nouveau *plugin*, ce dernier est envoyé au serveur d'application qui se charge de le valider avant de mettre à jour la base de données et intégrer les modifications à son architecture. Le serveur notifie ensuite les clients connectés, leur demandant de redémarrer afin d'effectuer la mise à jour grâce à la technologie Java Web Start (**JWS**).

Finalement, l'architecture globale de la solution RC2S est une savante combinaison de toutes ces fonctionnalités (*voir le Schéma d'Architecture page suivante*).

BIBLIOTHEQUES

CLIENT

Nom	Version	Commentaire
JavaFX	JDK 8	Interface utilisateur JSE
Log4J	2.5	API de logging
VLCJ	3.10.1	Gestion du son (streaming, …)

SERVEUR

Nom	Version	Commentaire
JEE	7	Java EE core
VLCJ	3.10.1	Gestion du son (streaming, …)
TarsosDSP	2.3	Analyse du son
Log4J	2.5	API de logging
JUnit	4.12	Framework de tests Unitaires
Mockito	1.10.19	Framework de mock d'objets

PI STATION

Nom	Version	Commentaire
Pi4J	1.0	I/O API for the RPi

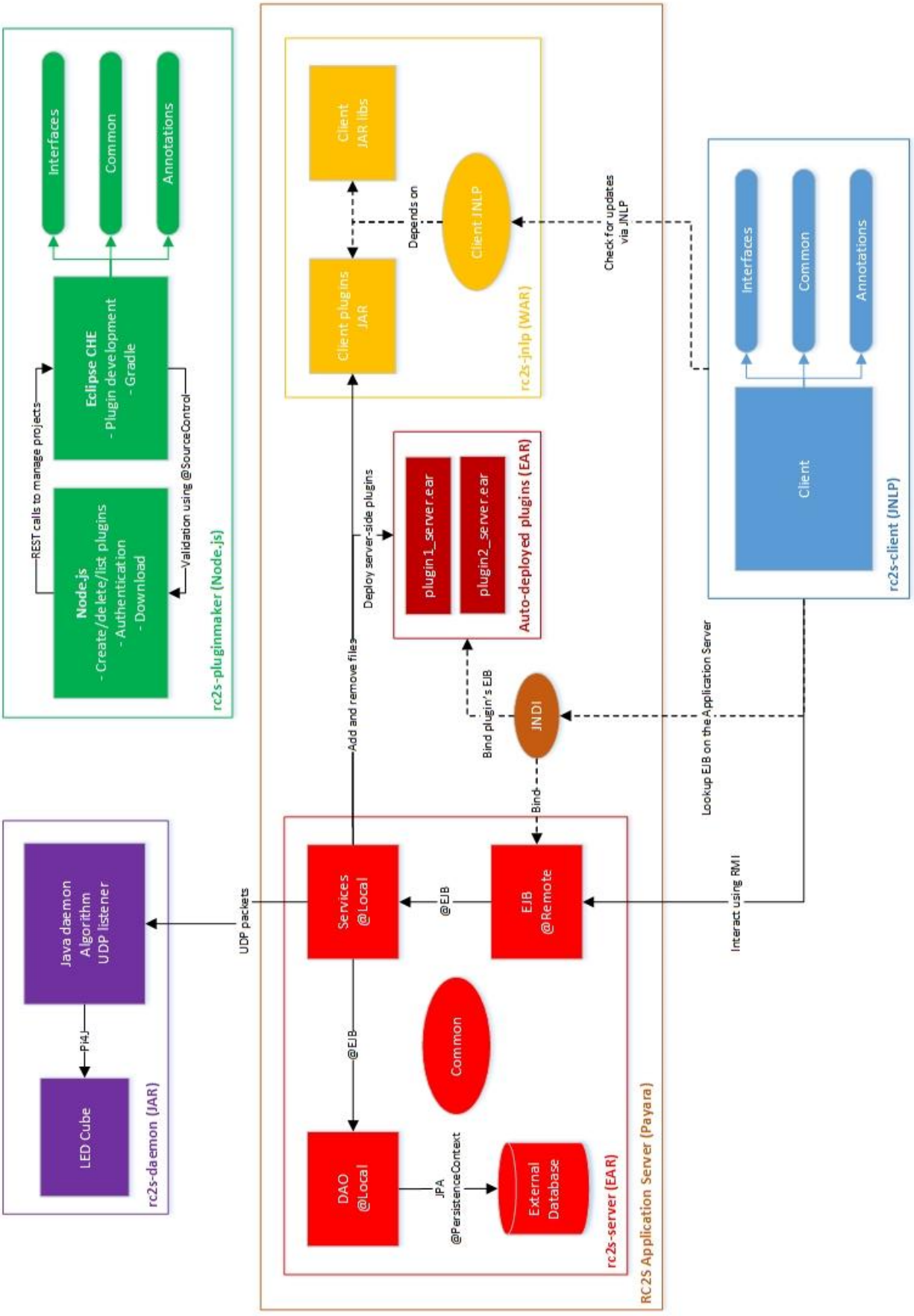
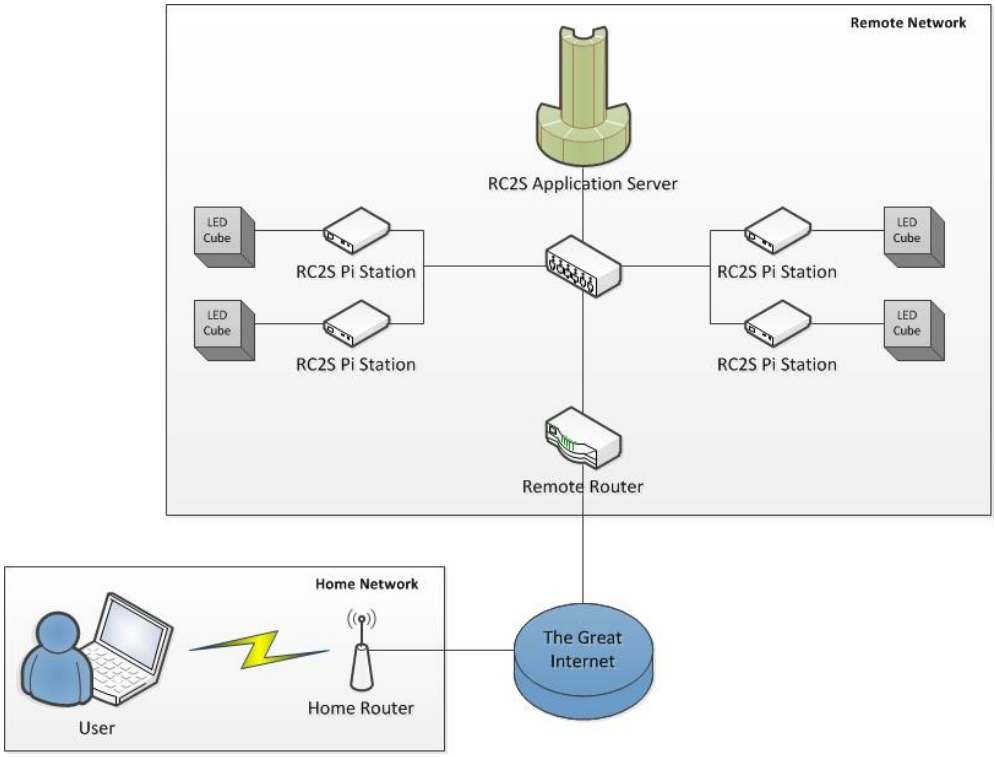


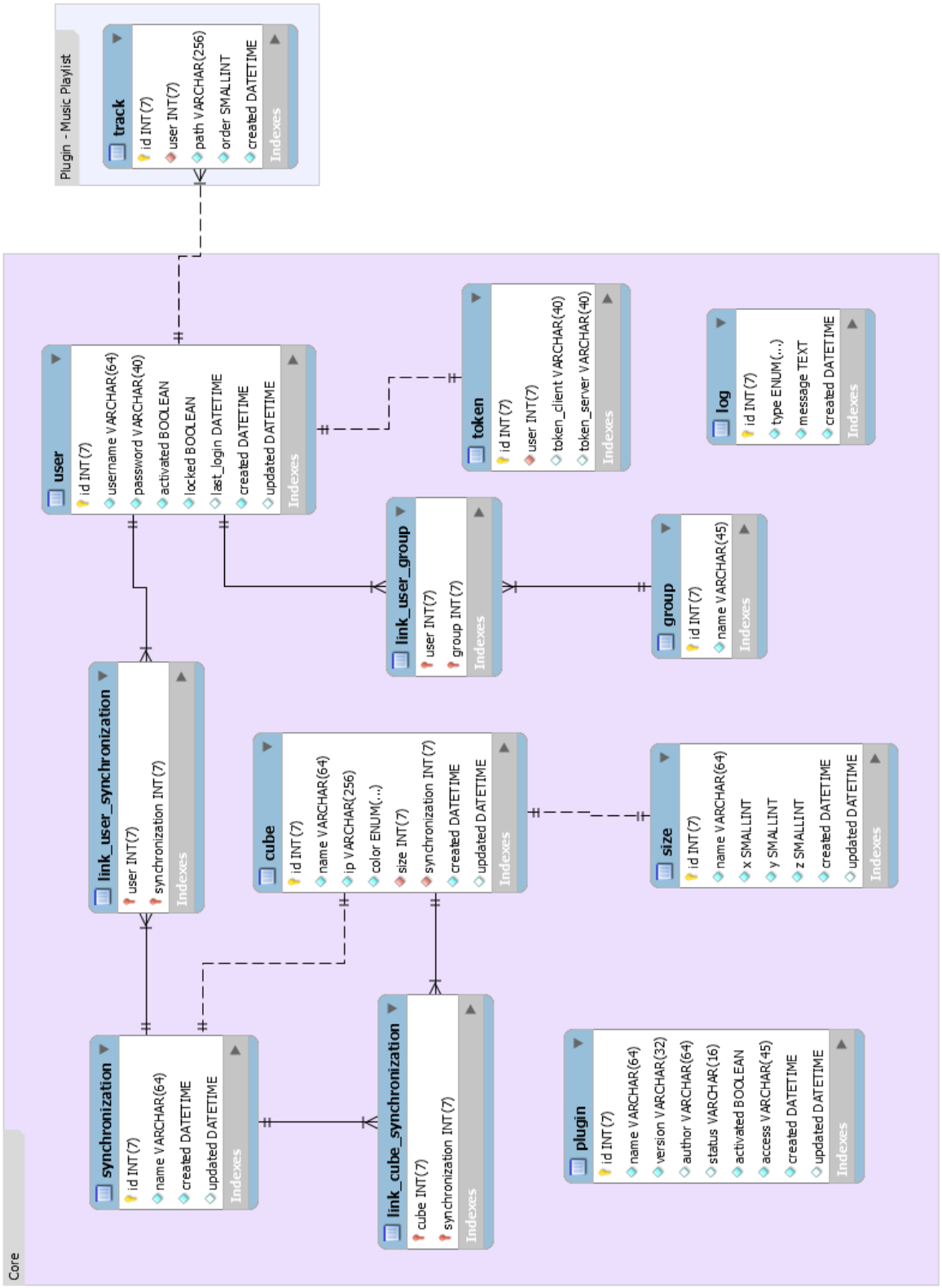
Schéma d'Architecture de la solution RC2S

ARCHITECTURE RÉSEAU



L'architecture réseau de la solution a été simplifiée à l'extrême : le client a uniquement besoin d'une connexion ouverte vers le serveur, qu'elle soit distante – à travers internet –, ou sur un réseau local ; tandis que le serveur d'application sera connecté sur le même réseau local que les Raspberry Pi de manière à pouvoir interagir avec eux, tout en restant accessible par l'extérieur.

BASE DE DONNEES



Les classes persistantes de l'application peuvent être retrouvées sur le MPD ci-dessus. Le module *Core* contient toutes les données liées à la solution RC2S standard, tandis que les modules *Plugins* représentent des ajouts faits par des scripts SQL notamment lors de l'intégration du plugin « *Music Playlist* ».

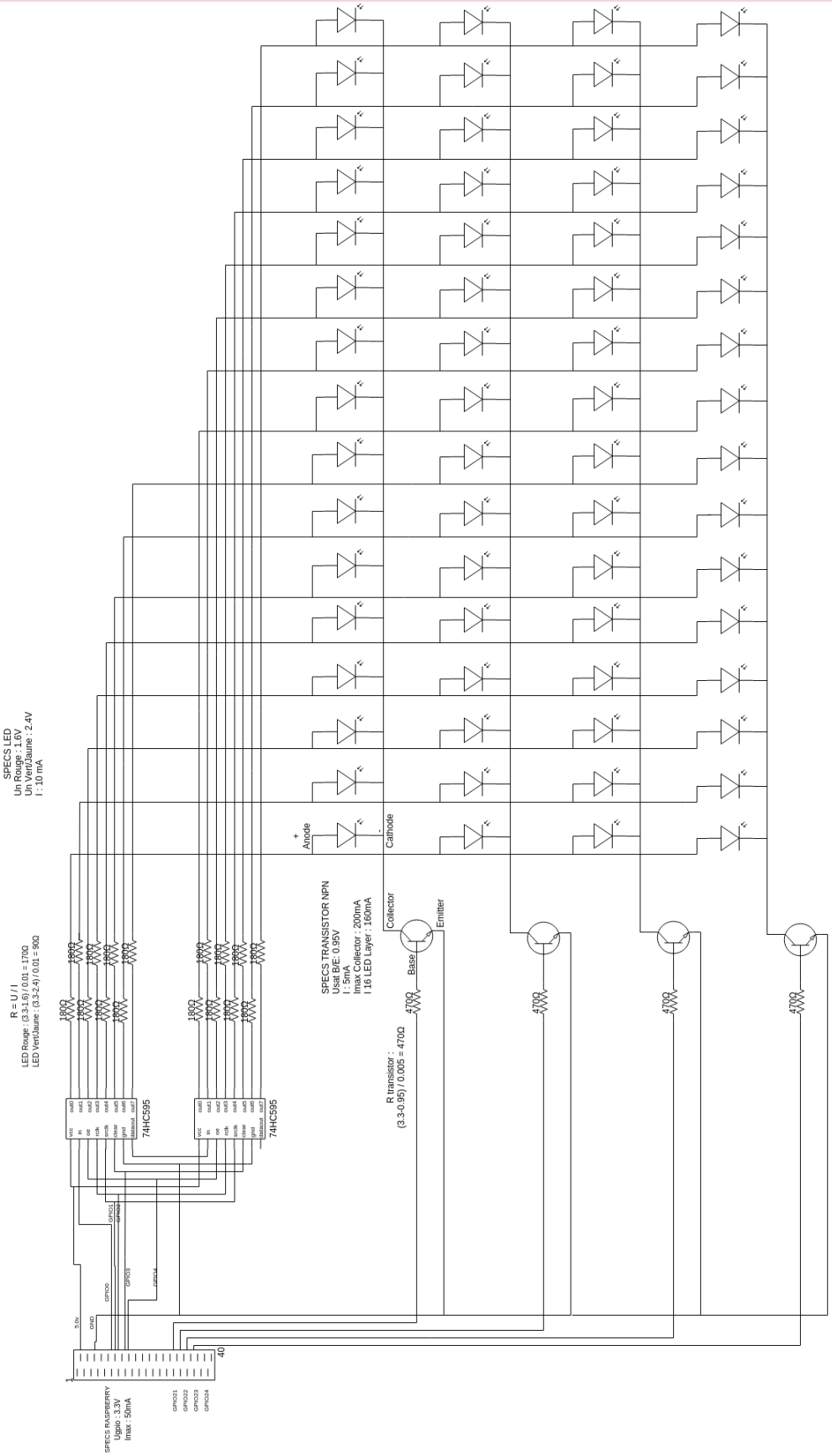


*Schéma électronique Cube 4*4*4 LED*

ALGORITHMES

ANALYSE DU SON

L'analyse du son survient lorsqu'un flux audio doit être traité (dans notre cas, lors de la lecture d'une musique ou d'une interaction avec le microphone). Un processus parallèle est alors exécuté pour envoyer ces données sous forme de flux au serveur d'application. Ce dernier, grâce à l'utilisation de la bibliothèque **TarsosDSP** (utilisant les *JTransforms*, première bibliothèque open-source permettant l'utilisation de la Transformation de Fourrier rapide (**FFT**) de manière parallélisée, nécessaires pour pouvoir transformer un flux audio en image 2D), va transformer le flux reçu en séquences logiques de positions de LED dans un ensemble de Cubes. Ces données, transmises aux *daemon* des différents Raspberry Pi, seront interprétées afin de créer les séquences animées.

TRAITEMENT DU SIGNAL SUR RASPBERRY PI

Le *daemon* de chaque Raspberry Pi recevra un ensemble de *n* états (tableau de booléens) correspondant aux axes **x**, **y** et **z**. Cet ensemble sera décomposé en *m* ensembles de *n*/*m* états, correspondant chacun à un étage du Cube (se reporter au schéma électronique) ; un système de balayage permettra d'éviter la surcharge du Raspberry en procédant à un allumage ensemble par ensemble.

Le procédé d'allumage consiste en la manipulation des *General Purpose Input/Output* (**GPIO**), présents sur le Raspberry Pi. En envoyant un signal électrique sur un GPIO défini, le circuit électronique interprétera cette instruction et allumera une LED à une position donnée dans le Cube.

EXTENSIBILITE

Un plugin RC2S est composé de 3 grands ensembles : **common**, **client** et **server**. Le package de base de chaque plugin devra suivre la convention de nommage suivante : `com.rc2s.{plugin_name}.*`
Cette convention permettra au Plugin Maker et au serveur d'application de localiser aisément les fichiers nécessaires à l'initialisation de chaque plugin.

Le module **common** contiendra tout ce qui doit être commun à l'ensemble de la solution. Il contient des éléments obligatoires à l'implémentation des JAR client & server (Value Objects, properties, …).

Le module **client** contiendra quant à lui une structure MVC s'intégrant à l'application RC2S-Client. Les packages le composant sont : **controllers**, **views**, **models** & **resources** (fichiers CSS, images, …).

Le module **server** intégrera la logique métier du plugin, qui sera intégrée au serveur d'application RC2S à travers les composants **ejb**, **application** & **dao**. Le package optionnel **scripts** pourra contenir un ensemble de scripts SQL qui seront exécutés lors de l'intégration du plugin au serveur. Un utilisateur de base de données aux droits restreints sera chargé d'effectuer les exécutions de scripts, de manière à prévenir toute exécution de scripts malveillants (DELETE TABLE, etc…).

Lors de la création d'un nouveau projet de plugin sur **Eclipse CHE** (IDE du **PluginMaker**), les deux API de compilation RC2S seront récupérées sur le serveur de fichiers : **interfaces.jar** contenant l'ensemble des **interfaces ejb** & **common** de la solution RC2S ; et **annotations.jar** contenant les annotations RC2S, dont le **@SourceControl** obligatoirement présent sur chaque classe développée.

Lors du développement, l'utilisation de l'annotation **@SourceControl** permettra de vérifier la cohérence et le respect de l'architecture de l'arborescence et des fichiers nécessaires au fonctionnement d'un plugin RC2S. Durant la compilation, les projets **client** et **server** intégreront le contenu de **common** de manière à accéder à ses VO & fichiers de propriétés. Ces deux composants du plugin sont alors compilés séparément, créant ainsi deux archives (**{plugin_name}**_client.jar et **{plugin_name}**_server.ear) qui seront ensuite packagées dans une archive zip **{plugin_name}.**zip. Cette dernière ne pourra être téléchargée que si toutes les classes du projet contiennent bien l'annotation de validation des sources.

Depuis le Client RC2S, l'utilisateur aura la possibilité de mettre en ligne un plugin sur le serveur d'application. Une fois réceptionnée, l'archive est décompressée et l'intégrité et la cohérence des fichiers JAR et EAR sont alors vérifiées.

Le fichier **{plugin_name}**_server.ear sera ensuite automatiquement déployé sur le serveur d'application, tandis que le fichier **{plugin_name}**_client.jar sera signé (procédure de sécurité Java) et copié dans l'application Web RC2S-JNLP. Les clients connectés recevront une notification de mise à jour les obligeant à redémarrer leur programme.

Lors du démarrage du client, ce dernier effectuera un chargement de l'intégralité des nouveaux plugins disponibles en faisant appel à l'application Web RC2S-JNLP.

ANNOTATIONS

Outre l'utilisation des interfaces, l'API RC2S dispose d'un processeur d'annotations permettant l'injection de code durant le processus de compilation. Il s'agira donc d'une aide lors de la phase de développement d'un *plugin*, simplifiant certaines interactions et diminuant le temps consacré à la création d'une nouvelle fonctionnalité.

@KNOWLEDGE

Annotation permettant de générer automatiquement la documentation HTML de l'élément cible. Si une classe est annotée **@Knowledge**, l'ensemble de ses attributs et méthodes seront inspectés pour en documenter la portée, le type et le nom. Si l'un de ces attributs ou méthodes est lui-même annoté **@Knowledge**, il est alors possible de préciser une chaîne de caractères en paramètre de l'annotation pour documenter plus précisément la fonctionnalité.

@SOURCECONTROL

Cette annotation, préfixant toutes classes d'un plugin RC2S, permettra de déclencher lors de la compilation un processus de validation du code source par introspection. Le PluginMaker s'assurera tout d'abord que l'annotation est présente sur toutes les classes développées avant que la compilation n'ait lieu.

Le processus de validation vérifiera la conformité du nom du package, de la classe, et le respect des normes de développement RC2S.

Les normes devant être respectées sont les suivantes :

- Le package racine d'un plugin doit être nommé comme suit : **com.rc2s.{pluginname}**.
- **com.rc2s.{pluginname}.common.vo** :
  - Contiendra tous les Value Objects du plugin : vérification de la présence de l'annotation **@Entity** sur la class de chaque VO.
- **com.rc2s.{pluginname}.ejb.{name}** :
  - Les fichiers **{name}FacadeRemote** et **{name}FacadeBean** doivent être présents et posséder respectivement les annotations **@Remote** pour la classe Remote et soit **@Stateless** soit **@Stateful** pour la classe Bean.
- **com.rc2s.{pluginname}.application.{name}** :
  - Les fichiers **I{name}Service** et **{name}Service** doivent être présents et posséder respectivement les annotations **@Local** pour l'interface et soit **@Stateless** soit **@Stateful** pour l'implémentation.
- **com.rc2s.{pluginname}.dao.{name}** :
  - Les fichiers **I{name}DAO** et **{name}DAO** doivent être présents et posséder respectivement les annotations **@Local** pour l'interface et soit **@Stateless** soit **@Stateful** pour l'implémentation.
- **com.rc2s.{pluginname}.common.sql** : possibilité d'y placer les scripts SQL qui seront exécutés lors de l'installation du plugin.
- **com.rc2s.{pluginname}.client.controllers** :
  - Pour chaque **{name}Controller**, vérification de l'implémentation de l'interface **Initializable**.
  - Il doit obligatoirement exister une classe **MainController**, qui sera exécutée par la vue principale du plugin.
- **com.rc2s.{pluginname}.client.views** :
  - Pour chaque nom de controller, vérifier la présence d'un fichier **{name}View.fxml**.
  - Il doit obligatoirement exister une vue **MainView**, qui sera affichée au démarrage de l'application RC2S-Client.
- **com.rc2s.{pluginname}.client.css** : contient les feuilles de style CSS.
- **com.rc2s.{pluginname}.client.images** : contient les fichiers images.
- **com.rc2s.{pluginname}.client.utils** : classes utilitaires du Plugin.

ORGANISATION

Afin de mener à bien la réalisation de ce projet, nous avons pris la décision de nous appuyer sur plusieurs outils et de suivre une certaine méthodologie de travail.

Suivant un fonctionnement Agile/SCRUM, une plateforme **JIRA** dédiée au projet a été ouverte pour effectuer un suivi poussé des tâches devant être réalisées.

La planification dans le temps étant un élément clé pour la réussite d'un projet, un **diagramme de Gantt** a été constamment tenu à jour.

Enfin, outre l'aspect évident de partage et de sauvegarde des sources induit par l'utilisation d'un outil de contrôle de version tel que **Git**, c'est un véritable *workflow* qui est mis en place derrière cette utilisation, rendant le développement de nouvelle fonctionnalité terriblement efficace.

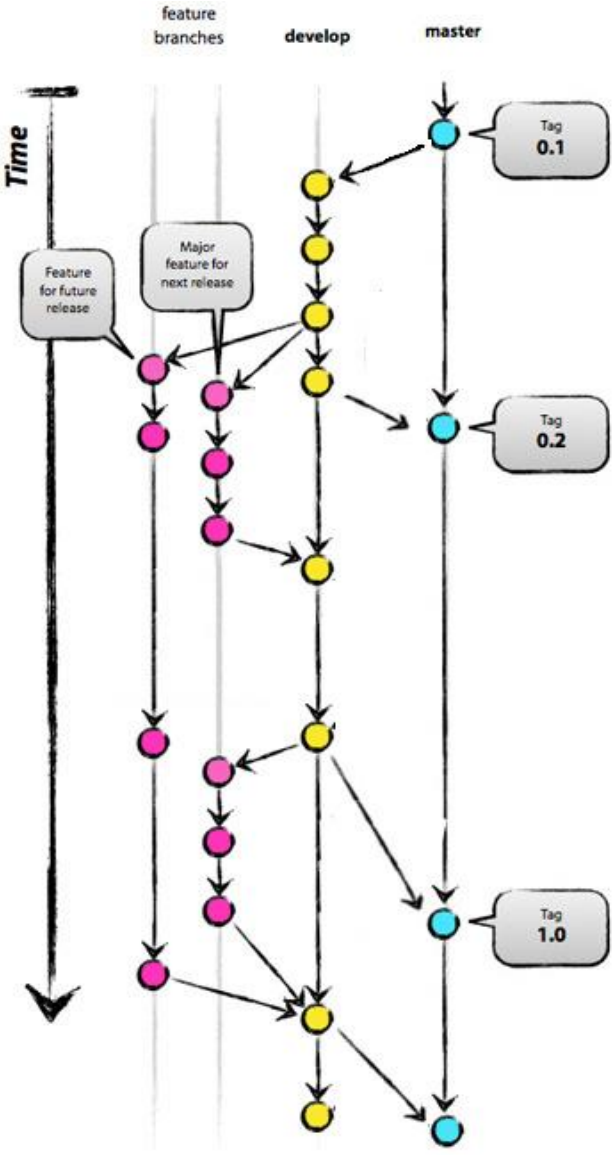
GIT

Git est un système de gestion de versions distribué gratuit et open-source conçu pour supporter tous types de projets – du plus simple au plus conséquent – de manière efficace.

Dans le cadre de notre projet, le contenu de la branche **master** ne sera constitué que de *merges* avec la branche **dev**. Il s'agit de la branche de déploiement, sur laquelle seront ajoutés des *tags* pour les versions stables.

La branche **dev**, quant à elle, n'effectuera de *merge* qu'avec les branches *features*. Les *push* de correctifs de bugs directement sur **dev** sont autorisés.

Les branches de travail, ou *Working Branches*, ont une appellation normée : *feature/**{taskId}***.



L'identifiant d'une tâche correspond ici à son identifiant **JIRA**, qui devra être rappelé en en-tête de tous les *commits* effectués sur cette branche. Cette rigueur normative permet de lier nos différentes plateformes : BitBucket hébergeant les différents repository Git, et JIRA pour la gestion de projet.
Les branches *features* peuvent être supprimées dès le développement terminé et le *merge* effectué sur la branche **dev**.

Si une erreur survient dans le code d'une fonctionnalité en développement et que celle-ci doit tout de même être *push* pour qu'un autre développeur puisse se pencher sur le problème, le tag **[Failed]** devra être ajouté après l'identifiant de la demande dans le commit.

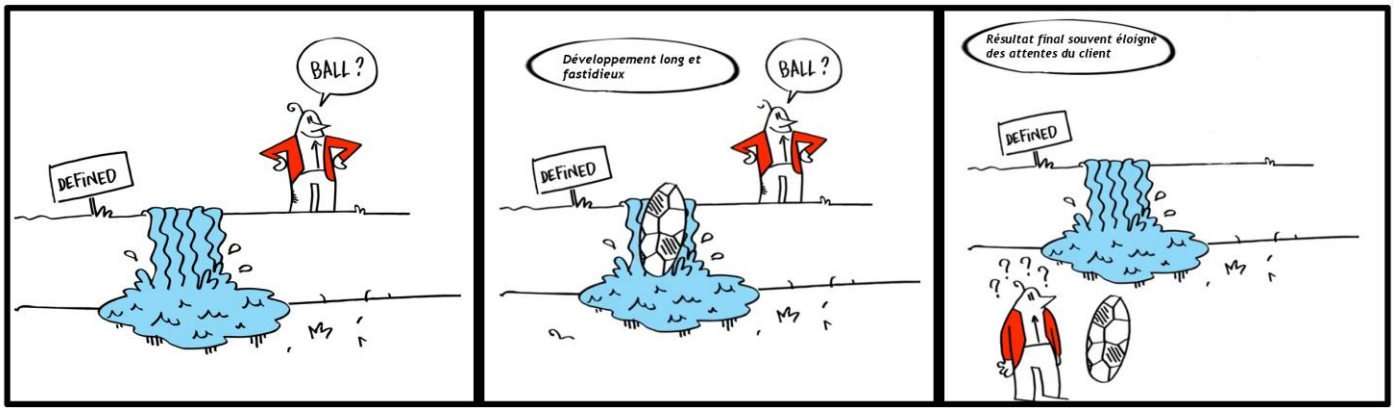
JIRA

JIRA est une plateforme centralisée de gestion de projets Agile/SCRUM, mais aussi de suivi de bugs et incidents. Nous l'utilisons principalement pour son aspect de gestion de projets.

L'ensemble des tâches à accomplir pour arriver au produit fini est appelé le **backlog**. Il comporte différents types de tâches telles que les **story**, qui correspondent à une fonctionnalité de la solution.

On parle alors de **sprint** pour définir l'unité de temps de la méthodologie SCRUM. Il s'agit d'une itération de 3 semaines – dans le cadre du projet RC2S – comportant un certain nombre de tâches du **backlog**.

Ce choix d'une approche méthodologique Agile se fait en opposition à la méthode plus traditionnelle qu'est l'approche « en cascade ». Cette dernière, plus *monolithique* dans sa manière d'aborder le projet, n'est en effet pas adaptée aux projets d'une certaine envergure.



L'approche en cascade

L'approche Agile permet de par son adaptation progressive aux spécificités du projet, de résoudre plus précisément à la demande du client qu'une approche de projet plus ordinaire. Sprint après sprint, les développeurs sont à même d'améliorer leur solution.

La méthodologie Agile/SCRUM

TESTS

TESTS UNITAIRES

Un test unitaire permet de vérifier le bon fonctionnement d'une portion de code d'un logiciel, permettant d'éviter la régression applicative. En Java, le framework **JUnit** permet une implémentation efficace des tests unitaires d'un projet.

TESTS D'INTEGRATION

Les tests d'intégration permettent de vérifier l'aspect fonctionnel, les performances et la fiabilité du logiciel. Il permet de vérifier que l'ajout d'une nouvelle fonctionnalité n'a pas eu d'impact négatif sur une partie déjà existante de la solution.

TESTS D'ACCEPTATION

Diverses entrevues avec le client sont planifiées tout au long de l'avancement du projet, permettant de s'assurer de la conformité du produit par rapport à ses attentes.

INTEGRATION CONTINUE

De manière à tirer pleinement partie des tests mis en place sur nos solutions, une plateforme d'intégration continue **Jenkins** a été configurée. Surveillant la branche **dev**, sur laquelle sont *mergées* toutes les branches **feature**, il effectuera une compilation et exécutera tous les jeux de tests dans l'heure qui suivra un nouveau commit. Un e-mail est ensuite envoyé aux développeurs pour leur faire part de l'issu des tests et les inviter à en consulter le journal d'exécution.

Le principal intérêt de ce serveur d'intégration est de constamment surveiller l'état de la branche principale de développement : lorsqu'une nouvelle fonctionnalité y est intégrée, Jenkins nous informera automatiquement si une régression fonctionnelle a eu lieu.

CAHIER DE TESTS

Projet	Description
rc2s-daemon	Test de l'interprétation des données reçues depuis le serveur d'application.
	Test d'interaction avec les GPIO.
	Test de réception de paquets.
rc2s-application	Test des méthodes "complexes" (ne faisant pas uniquement appel à un DAO).
rc2s-annotations	@Knowledge : validation du contenu de la documentation générée & vérification de l'écriture du fichier sur le disque.
	@SourceControl : création d'une classe de test devant être validée par l'annotation.
rc2s-pluginmaker	Test de la connexion Node.js & Eclipse CHE.