

Dossier technique du projet - partie individuelle

Table des matières

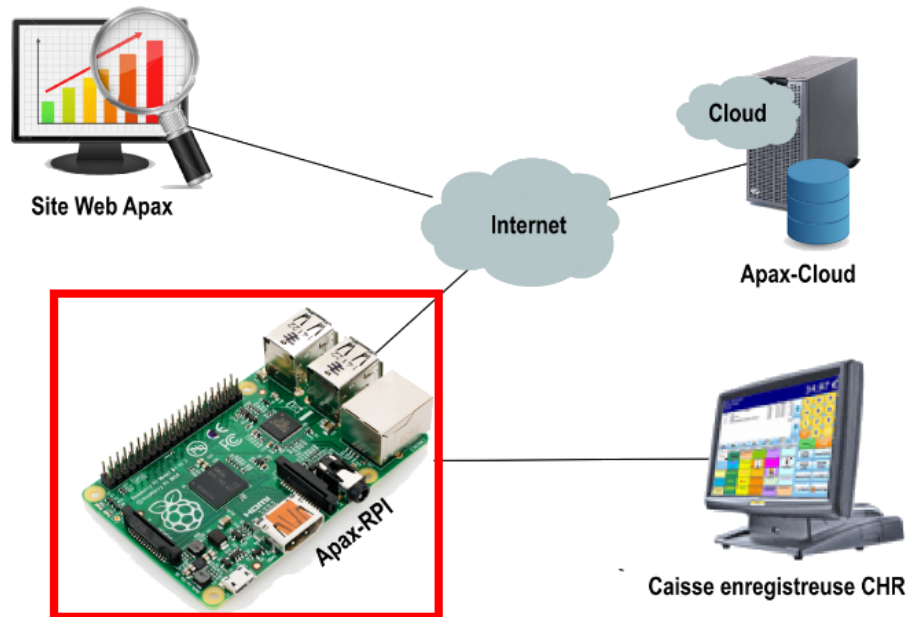
1 -SITUATION DANS LE PROJET	4
1.0.1 -Synoptique de la réalisation.....	4
1.1 -RAPPEL DES TÂCHES PROFESSIONNELLES À RÉALISER	5
1.2 -PRÉSENTATION DE LA PARTIE PERSONNELLE.....	5
1.2.1 -Introduction.....	5
1.2.2 -Diagramme de classes.....	7
2 -RÉALISATION.....	8
2.1 -CONCEPTION DÉTAILLÉE DE LA CLASSE GESTIONNAIRE.....	8
2.1.1 -Description fonctionnelle.....	8
2.1.2 -Description des données traitées.....	8
2.1.3 -Diagramme de classes.....	8
2.1.4 -Documentation technique.....	9
2.1.5 -Code source de la classe.....	10
2.2 -CONCEPTION DÉTAILLÉE DE LA CLASSE CPU.....	10
2.2.1 -Description fonctionnelle.....	10
2.2.2 -Description des données traitées.....	10
2.2.3 -Code source de la classe.....	11
2.3 -CONCEPTION DÉTAILLÉE DE LA CLASSE RAM.....	11
2.3.1 -Description fonctionnelle.....	11
2.3.2 -Description des données traitées.....	11
2.3.3 -Code source de la classe.....	12
2.4 -CONCEPTION DÉTAILLÉE DE LA MÉTHODE INIT DE LA CLASSE API.....	12
2.4.1 -Description fonctionnelle.....	12
2.4.2 -Description des données traitées.....	12
2.4.3 -Diagramme de classe.....	12
2.4.4 -Documentation technique.....	13
2.4.5 -Code source de la méthode.....	13
2.5 -CONCEPTION DÉTAILLÉE DE LA MÉTHODE PARSAGE DE LA CLASSE API.....	14
2.5.1 -Description fonctionnelle.....	14
2.5.2 -Description des données traitées.....	14
2.5.3 -Code source de la méthode.....	14
2.6 -CONCEPTION DÉTAILLÉE DE LA MÉTHODE ENVOI DE LA CLASSE API.....	14
2.6.1 -Description fonctionnelle.....	14
2.6.2 -Description des données traitées.....	15
2.6.3 -Code source de la méthode.....	15
2.7 -CONCEPTION DÉTAILLÉE DE LA MÉTHODE LOG DE LA CLASSE API.....	15

2.7.1 -Description fonctionnelle.....	15
2.7.2 -Description des données traitées.....	15
2.7.3 -Code source de la méthode.....	16
2.8 -CONCEPTION DÉTAILLÉ DE L'API REST SERVEUR.....	16
2.8.1 -Description fonctionnelle.....	16
2.8.2 -Documentation technique.....	16
2.8.3 -Description des données traitées.....	17
2.8.4 -Code source de la classe.....	17
2.9 -CONCEPTION DÉTAILLÉE DU SERVICE WATCHDOG.....	17
2.9.1 -Description fonctionnelle.....	17
2.9.2 -Documentation technique.....	18
2.9.3 -Fichier de configuration du service.....	18
3 -TEST UNITAIRE.....	19
3.1 -PLAN DES TESTS UNITAIRES.....	19
3.2 -TEST UNITAIRE DE RAM.....	19
3.2.1 -Identification du test.....	19
3.2.2 -Cas d'utilisations associés.....	19
3.2.3 -Description du test.....	19
3.2.4 -Problème rencontré.....	19
3.2.5 -Code source du programme de test.....	20
3.2.6 - Procédure de test.....	20
3.2.7 - Rapport d'exécution.....	20
3.3 -TEST UNITAIRE DE CPU.....	21
3.3.1 -Identification du test.....	21
3.3.2 -Cas d'utilisation associés.....	21
3.3.3 -Description du test.....	21
3.3.4 -Problème rencontré.....	21
3.3.5 -Code source du programme test.....	21
3.3.6 - Procédure de test.....	22
3.3.7 - Rapport d'exécution.....	22
3.4 -TEST UNITAIRE GESTIONNAIRE.....	22
3.4.1 -Identification du test.....	22
3.4.2 -Cas d'utilisation associé.....	22
3.4.3 -Description du test.....	23
3.4.4 -Problèmes rencontrés.....	23
3.4.5 -Code source du programme de test.....	23
3.4.6 -Procédure de test.....	24
3.4.7 - Rapport d'exécution.....	24
3.5 -TEST UNITAIRE GESTIONNAIRE (SERVEUR TCP ÉTEINT).....	24
3.5.1 -Identification du test.....	24
3.5.2 -Cas d'utilisation associé.....	25
3.5.3 -Description du test.....	25
3.5.4 -Problème rencontré.....	25
3.5.5 -Code source du programme de test.....	25
3.5.6 - Procédure de test.....	25
3.5.7 - Rapport d'exécution.....	26
3.6 -TEST UNITAIRE FONCTION ENVOIE.....	26
3.6.1 -Identification du test.....	26
3.6.2 -Cas d'utilisations associé.....	26
3.6.3 -Description du test.....	26

3.6.4 -Problème rencontré.....	26
3.6.5 -Code source du programme de test.....	27
3.6.6 - Procédure de test.....	27
3.6.7 -Rapport d'exécution.....	28
3.7 -TEST UNITAIRE FONCTION ENVOIE (SANS INTERNET).....	28
3.7.1 -Identification du test.....	28
3.7.2 -Cas d'utilisation associé.....	28
3.7.3 -Description du test.....	28
3.7.4 -Problème rencontré.....	28
3.7.5 -Code source du programme de test.....	28
3.7.6 -Procédure de test.....	29
3.7.7 - Rapport d'exécution.....	29
3.8 -TEST UNITAIRE DE L'API SERVEUR.....	30
3.8.1 -Identification du test.....	30
3.8.2 -Cas d'utilisation associés.....	30
3.8.3 -Description du test.....	30
3.8.4 -Problèmes rencontrés.....	30
3.8.5 -Code source du programme test.....	31
3.8.6 -Procédure de test.....	32
3.8.7 - Rapport d'exécution.....	32
3.9 -TEST UNITAIRE DU SERVICE WATCHDOG.....	33
3.9.1 -Identification du test.....	33
3.9.2 -Cas d'utilisation associé.....	33
3.9.3 -Description du test.....	33
3.9.4 -Problème rencontré.....	33
3.9.5 -Test effectué.....	33
3.9.5.1 -Vérification du démarrage des services.....	33
3.9.6 -Procédure de test.....	34
3.9.7 - Rapport d'exécution.....	34
3.9.7.1 -Vérification du redémarrage des services.....	34
3.9.8 - Procédure de test.....	34
3.9.9 - Rapport d'exécution.....	35
4 -BILAN DE LA RÉALISATION PERSONNELLE.....	35

1 - Situation dans le projet

1.0.1 - Synoptique de la réalisation



Synoptique général du système

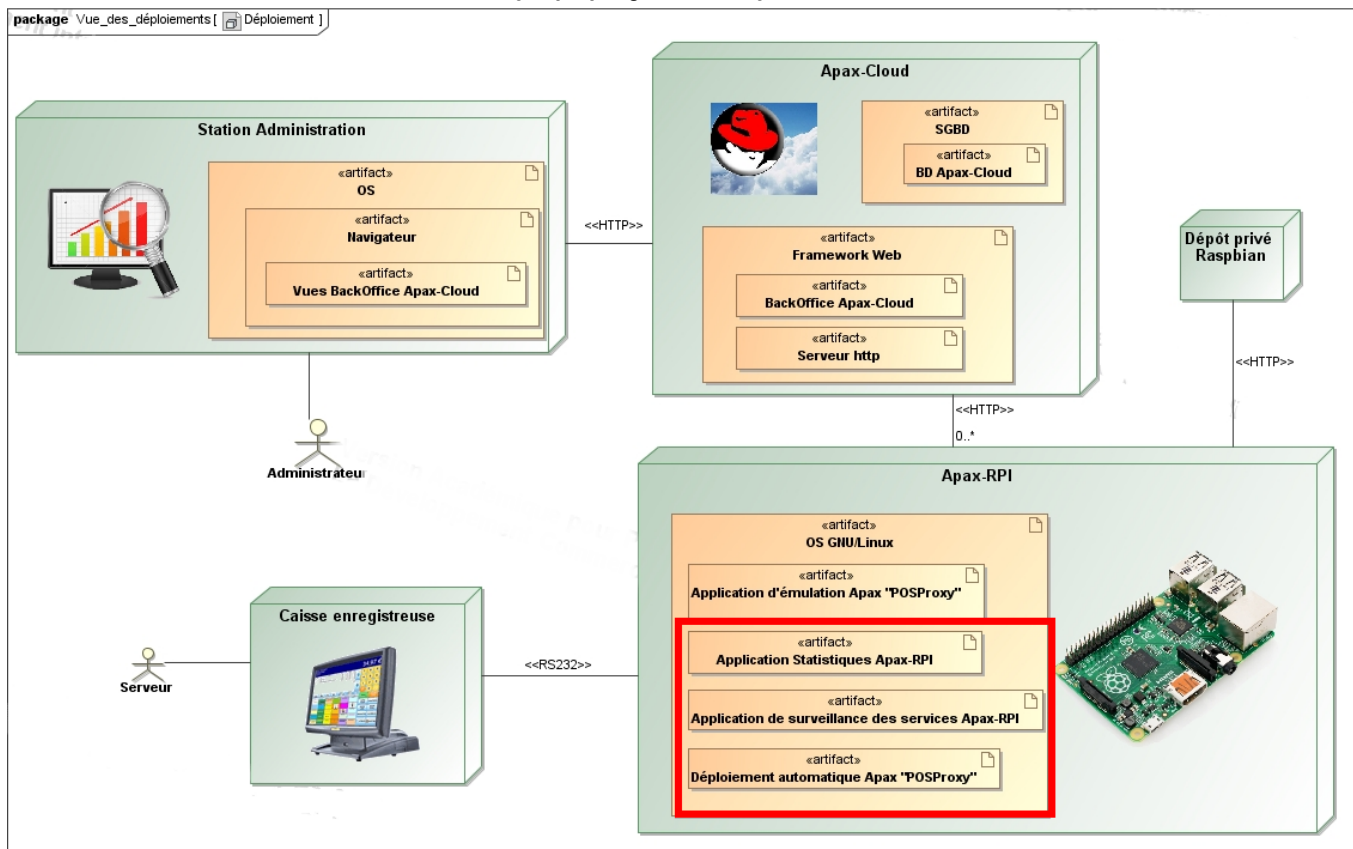






Diagramme de déploiement d'exploitation

Ma partie du projet se concentre sur la carte RPI. Un programme C++ va produire des statistiques sur l'utilisation ram et cpu en pourcentage. Puis via une IPC qui est un serveur TCP ses données seront réceptionnées puis parsées, pour ensuite être envoyées via internet par une requête HTTP sur une API REST serveur qui est présent sur le cloud. L'api serveur va récupérer les valeurs CPU et RAM présent dans la requête HTTP pour par la suite effectuer une requête INSERT dans la base de données postgresql.

1.1 - Rappel des tâches professionnelles à réaliser

Étudiant B	Fonction de services Apax-RPI à réaliser	État d'avancement
Fp6	Fournir les statistiques sur les ressources matérielles Cette partie consiste à produire des statistiques sur l'utilisation CPU et RAM et ainsi pouvoir prévenir d'une panne.	
Fp7	Transmettre des statistiques Cette fonction permet de transmettre les statistiques au service APAX-Cloud via une API REST .	
Fp8	Installer automatiquement les paquets logiciels Le logiciel Apax ne sera pas installé de base sur la RPI. L'objectif est de le télécharger via un dépôt privé grâce à un script NodeJS .	En cours...
Fp9	Démarrer et surveiller les services Apax-RPI Cette étape consiste à démarrer et surveiller les services APAX grâce à un service Watchdog . Ceci permettra de garantir le fonctionnement permanent des services APAX.	
Fr2	Installer, configurer et tester l'infrastructure LAN Cette partie consiste à mettre en place l'infrastructure LAN pour assurer le bon suivi de la carte Apax-RPI	

1.2 - Présentation de la partie personnelle

1.2.1 - Introduction

Ce dossier présente les tâches que j'ai effectuées durant mon projet. Étudiant B, GUERCHET FAUVEL Yohan je me suis principalement intéressé à la partie embarquée c'est-à-dire la carte raspberry. Nous étions deux étudiants sur cette partie, mes tâches se sont concentré principalement sur la surveillance de la RPI et des services qui lui sont liés.

La réalisation du projet s'est déroulé en 4 sprints:

- **Sprint 0**
 - Créer le SVN
 - Créer le Github
 - Étudier le projet
- **Sprint 1**
 - Installation et configuration de la carte Raspberry
 - Réalisation du programme C++ qui analyse le CPU et la RAM
- **Sprint 2**
 - Initiation au NodeJS
 - Initiation à l'API REST
 - Création d'une API REST client
 - Création d'une API REST serveur
 - Création de l'IPC (serveur TCP)

• Sprint 3

- Établir une connexion entre le programme C++ et le serveur TCP
- Établir un lien entre le serveur TCP et l'API REST client
- Installation et configuration du service Watchdog
- Rédaction des manuels d'installation

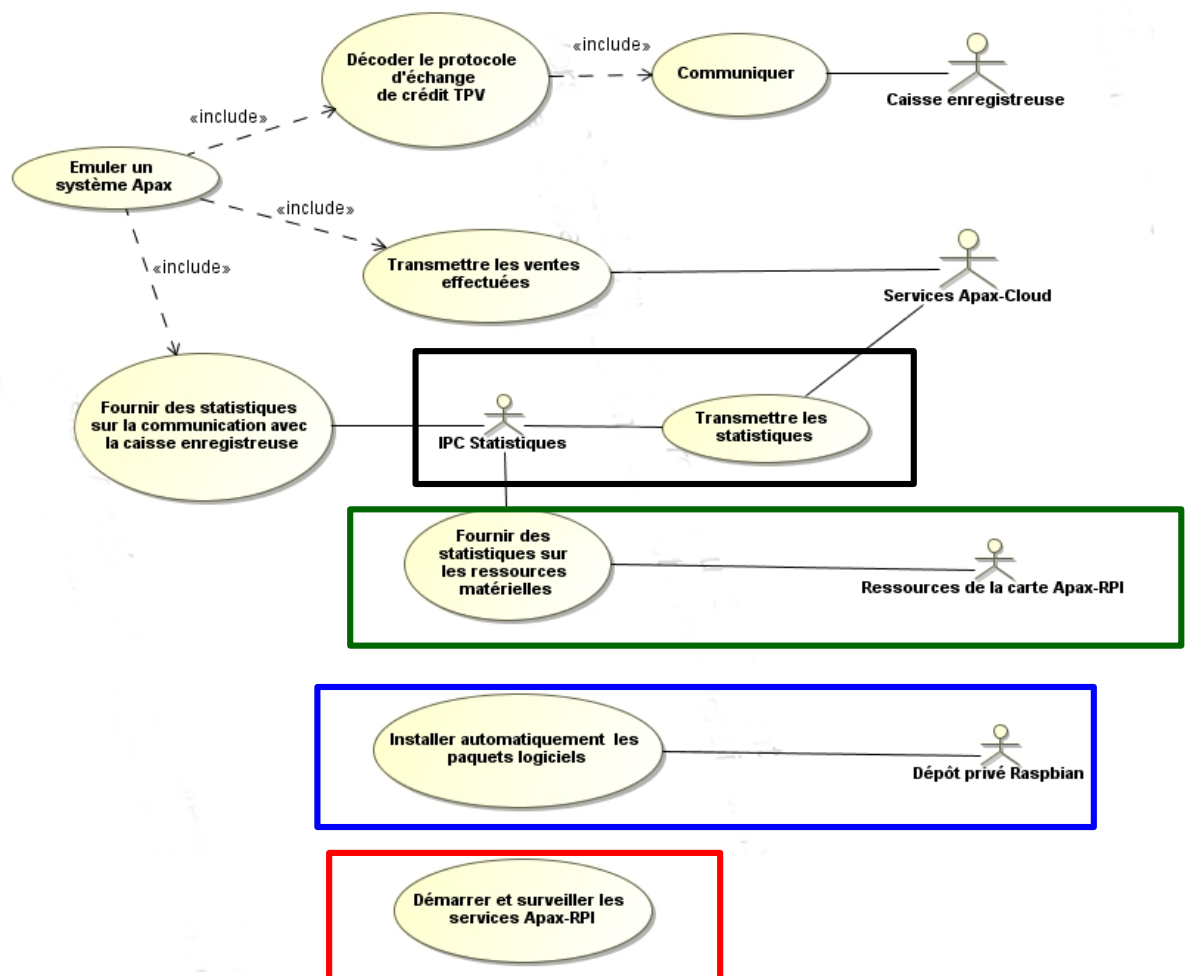
Durant la phase d'étude et de création des différents programmes j'ai dû faire des choix sur le langage de programmation ou la technologie utilisée...

Commençons par la tâche **Fp6 (Fournir les statistiques sur les ressources matérielles)** j'ai choisi d'utiliser un langage orienté objet qui est le C++. Tout d'abord car c'est un langage que j'ai eu l'occasion d'utiliser de nombreuses fois durant mon BTS, j'ai donc de solide base. Le langage C++ est aussi un très performant et compatible avec de nombreuses bibliothèques. Mais surtout ce langage est connu de tous et le code que j'ai produit sera donc réutilisable et modifiable par tout le monde.

Ensuite pour la tâche **Fp7 (Transmettre des statistiques)** j'ai choisi d'utiliser une API REST qui fera guise de notre client HTTP. Pour effectuer un lien entre l'API REST et le programme C++ j'ai choisi d'utiliser un serveur TCP pour plusieurs raisons. La première raison est que c'est simple d'utilisation, et nous sommes sûrs que les données reçues seront bien celle envoyée. La seconde raison est que si l'établissement a besoin de plusieurs RPI, grâce au serveur TCP nous pourrions utiliser la même API REST client.

Malheureusement je n'ai pas eu le temps d'effectuer la tâche **Fp8 (Installer automatiquement les paquets logiciels)**. Mais j'ai sû trouver une alternative. Dans un répertoire git j'ai ajouté les services APAX dont nous avons besoin avec toutes les bibliothèques qui sont nécessaires pour que les programmes fonctionnent. L'utilisateur a besoin de simplement compiler les programmes C++ mais ceci sera fait automatiquement grâce à la commande make.

Enfin pour la tâche **Fp9 (Démarrer et surveiller les services Apax-RPI)** j'ai dû utiliser un service watchdog. J'ai choisi d'utiliser le service supervisor car il est très simple d'installation et de configuration. Mais aussi le redémarrage d'un programme qui a crashé ce fait instantanément.



1.2.2 - Diagramme de classes

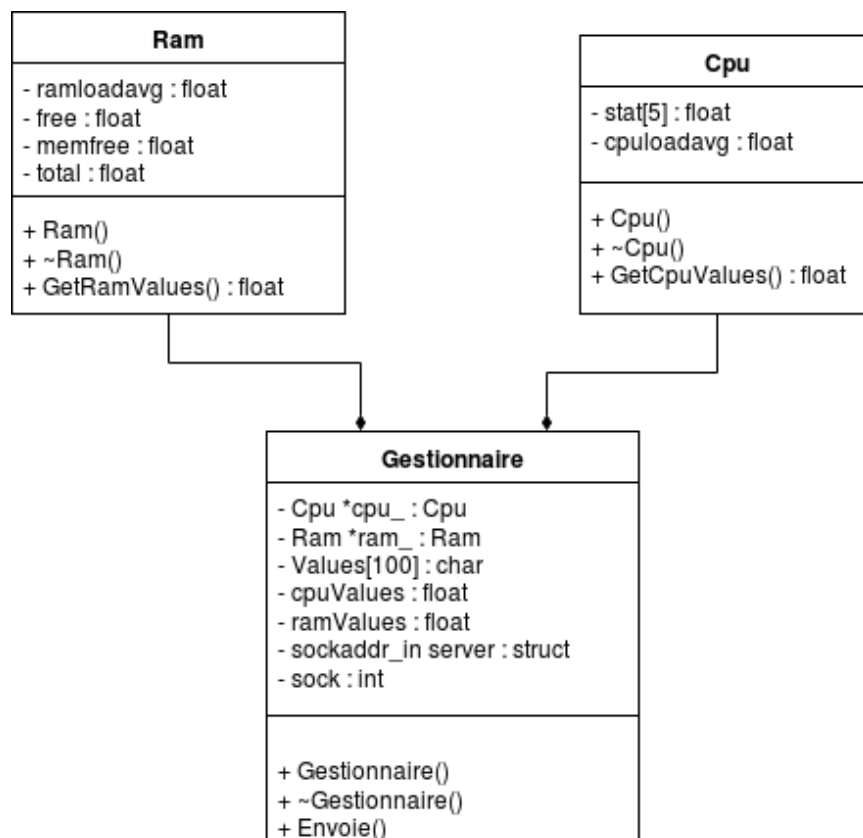


Diagramme de classe de la fonction Fp6

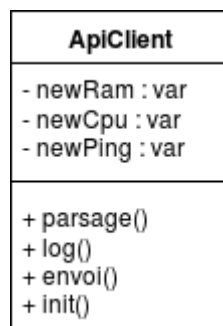


Diagramme de classe de la fonction Fp7

Nous avons un diagramme de classes pour chaque programme. On associera Gestionnaire à la production de statistiques c'est-à-dire utilisation CPU et RAM. Et ApiClient correspond à l'envoi des données mais aussi au serveur TCP.

- **Gestionnaire** : Gestionnaire de la production de statistiques
 - **Ram** : Production de l'utilisation RAM
 - **CPU** : Production de l'utilisation CPU
- **ApiClient** : Réception et envoi des données produites par le Gestionnaire
 - **Méthode parsage** : Permet de parser les données envoyées par le Gestionnaire
 - **Méthode log** : Permet d'écrire les valeurs produites ainsi que la date et l'heure dans un fichier de log si l'envoi est impossible
 - **Méthode envoi** : Requête HTTP à l'API serveur, comportant les valeurs produites par le gestionnaire
 - **Méthode init** : Partie du programme exécutée en premier qui est composée du serveur TCP. Cette méthode va lancer les différentes méthodes

2- Réalisation

2.1 - Conception détaillée de la classe Gestionnaire

2.1.1 - Description fonctionnelle

Gestionnaire est lancé automatiquement par supervisor. Cette classe appelle les classes « Cpu » et « Ram ». Et utilise leur méthode « **GetCpuValues()** » et « **GetRamValues()** ». Gestionnaire dispose aussi de la méthode **envoie()** pour envoyer les données produite vers le serveur TCP.

2.1.2 - Description des données traitées

Comme expliqué précédemment Gestionnaire appelle les classes « **Cpu** » et « **Ram** ». Si l'on se focalise sur ce groupe de classes on peut voir qu'il y a production des valeurs du Cpu et de la Ram. Par la suite Gestionnaire se connectera au serveur TCP pour envoyer les données. Le délimiteur « + » sera utilisé entre chaque valeur pour le passage.

Cpu: 3.168872

Ram: 33.431904

Envoyé: 3.168872+33.431904

Les valeurs sont produites en float pour ainsi pouvoir mieux s'adapter au client. Si le client veut recevoir sur l'IHM des valeurs entières ou flottantes nous aurons juste à modifier le passage sur l'API REST client.

2.1.3 - Diagramme de classes

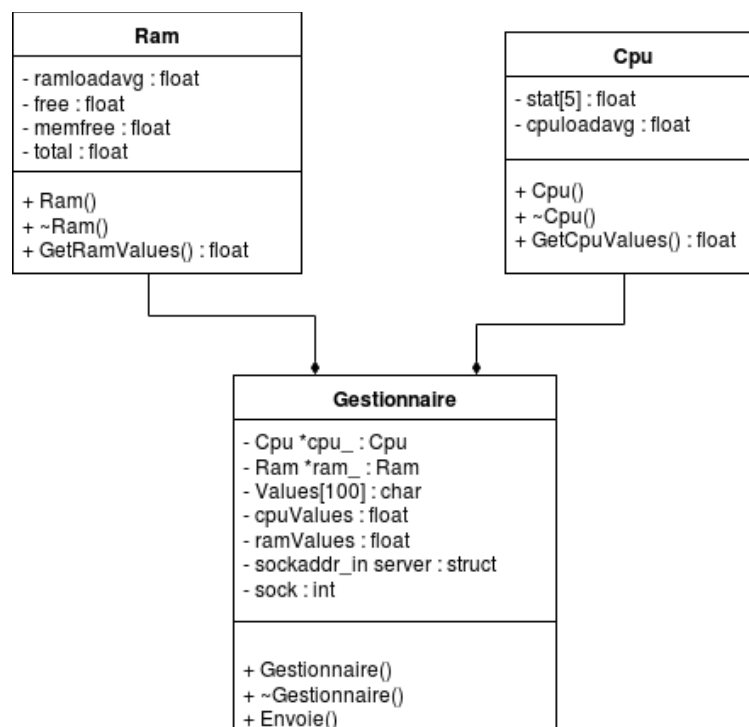


Diagramme de classe de la fonction Fp6

2.1.4 - Documentation technique

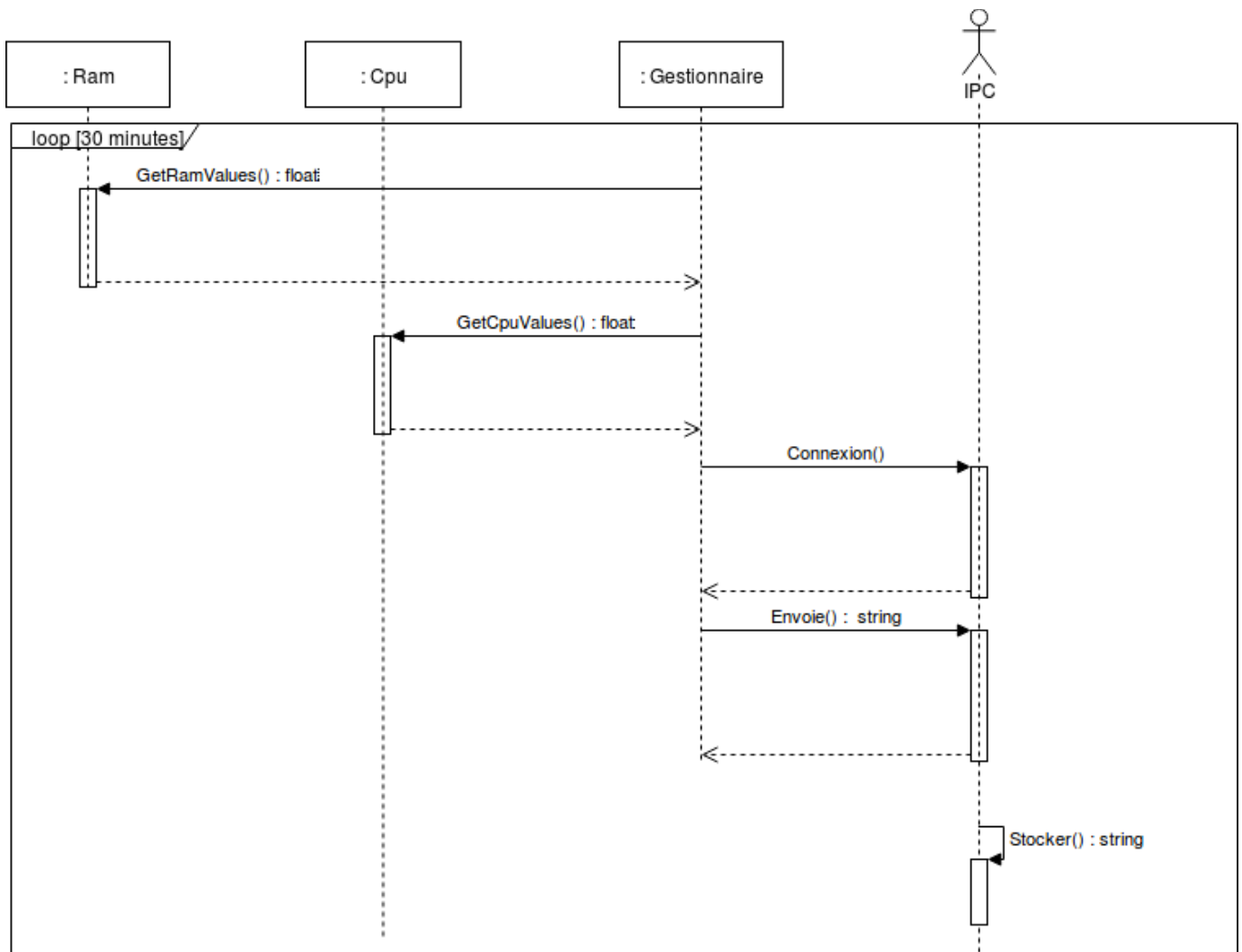


Diagramme de séquence de la fonction Fp6

2.1.5 - Code source de la classe

```
void Gestionnaire::Envoie()
{
    ramValues = ram_ ->GetRamValues();
    cpuValues = cpu_ ->GetCpuValues();
    //récupération des valeurs

    sock = socket(AF_INET , SOCK_STREAM , 0);
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_family = AF_INET;
    server.sin_port = htons( 8888 );
    //Pré connexion a l'API client

    sprintf(Values, "%f+%f",cpuValues,ramValues);
    //Conversion des données pour l'envoi et ajout du signe + entre chaque valeurs
    pour le passage

    while(connect(sock , (struct sockaddr *)&server , sizeof(server)) == -1){
        sleep(10);
    }
    //Connexion au serveur TCP

    while(write(sock,Values,strlen(Values)) == -0){
        sleep(10);
    }
    //Envoi des données vers l'API client.

    close(sock);
    //Déconnexion du serveur tcp
}

int main()
{
    Gestionnaire* gestionnaire = new Gestionnaire();
    while(1){
        gestionnaire->Envoie(); //Appelle de la méthode envoie.
        sleep(1800); //Pause de 30 minutes.
    }
}
```

2.2 - Conception détaillée de la classe Cpu

2.2.1 - Description fonctionnelle

Cpu est gérée par Gestionnaire. Cette classe permet de récupérer l'utilisation Cpu , elle est composée d'une seule méthode qui est **GetCpuValues()**.

2.2.2 - Description des données traitées

Cpu récupère les valeurs du Cpu dans un fichier (*/proc/stat*) qui est présent sur la RPI et qui est mis à jour en temps réel. Le calcul additionne toutes les valeurs pour avoir la mémoire CPU total et soustrait ce dernier par la mémoire CPU libre puis divise le tout par la mémoire CPU total et enfin il le multiplie par 100 pour avoir l'utilisation du CPU en pourcentage.

```
pi@P6:~$ more /proc/stat
cpu 375 0 534 28074 669 0 16 0 0 0
cpu0 47 0 125 7069 138 0 16 0 0 0
cpu1 87 0 170 7012 168 0 0 0 0 0
cpu2 74 0 109 7078 168 0 0 0 0 0
cpu3 167 0 130 6915 195 0 0 0 0 0
```

Dossier /proc/stat de la RPI

cpu : correspond au cpu en général

cpuX : correspond au coeur X du cpu

Les 3 premières valeurs correspondent au CPU *utilisé* les 2 valeurs suivantes correspondent au Cpu *libre*.

2.2.3 - Code source de la classe

```
float Cpu::GetCpuValues() {
    FILE *procStat = fopen("/proc/stat", "r");
    //Ouverture du fichier /proc/stat en lecture.

    fscanf(procStat, " %s %f %f %f %f", &stat[0], &stat[1], &stat[2], &stat[3], &stat[4]);
    //Affectation des 5 premières valeurs dans un tableau.

    fclose(procStat);
    //Fermeture du fichier.

    cpuloadavg = (((stat[0]+stat[1]+stat[2]+stat[3]+stat[4]) - (stat[3]+stat[4])) /
    (stat[0]+stat[1]+stat[2]+stat[3]+stat[4]))*100;
    //Calcule du pourcentage CPU utilisé à partir du fichier /proc/stat

    return cpuloadavg;
}
```

2.3 - Conception détaillée de la classe Ram

2.3.1 - Description fonctionnelle

Ram est appelé par Gestionnaire. Elle permet de récupérer l'utilisation Ram avec une seule méthode qui est **GetRamValues()**

2.3.2 - Description des données traitées

Tout comme la classe CPU , la classe Ram va récupérer les valeurs de l'utilisation ram dans un fichier (/proc/meminfo) sur la carte RPI. Le calcul soustrait la mémoire total par la mémoire utilisée et divise ce total par la mémoire totale pour multiplier par 100 le total trouvé et ainsi avoir le pourcentage de mémoire Ram utilisé.

```
pi@P6:~$ more /proc/meminfo
MemTotal:      947732 kB
MemFree:       817180 kB
MemAvailable:  849508 kB
```

Dossier /proc/meminfo de la RPI

MemTotal : correspond à la mémoire totale en kB

MemFree : correspond à la mémoire libre en kB

MemAvailable : correspond à la mémoire libre en Kb mais elle est plus précise car elle prend en compte plus de critère pour démarrer de nouvelles applications.

2.3.3 - Code source de la classe

```
float Ram::GetRamValues() {
    FILE *memInfo = fopen("/proc/meminfo", "r");
    //Ouverture du fichier /proc/meminfo en lecture

    fscanf(memInfo, "MemTotal: %f kB MemFree: %f kB MemAvailable: %f kB", &total,
    &memfree, &free);
    //Récupération de la mémoire totale et libre.

    fclose(memInfo); //Fermeture du fichier.

    ramloadavg = (((total-free)/total)*100);
    //Calcule pour définir l'usage de la mémoire.

    return ramloadavg;
}
```

2.4 - Conception détaillée de la méthode init de la classe Api

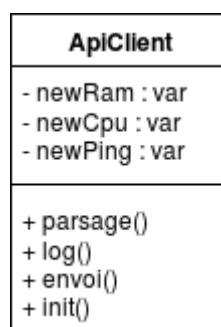
2.4.1 - Description fonctionnelle

La méthode **init()** est la méthode exécutée en première. Celle-ci initialise et lance le serveur TCP ainsi que les méthodes **parsage()** et **envoi()**.

2.4.2 - Description des données traitées

Comme vue précédemment cette classe appelle les deux méthodes **parssage()** et **envoi()**. Si l'on s'intéresse à ce groupe de méthode, on peut voir qu'il permet de parser les valeurs reçues de la part du programme C++ et de l'envoyer à l'API REST serveur (APAX cloud)

2.4.3 - Diagramme de classe



2.4.4 - Documentation technique

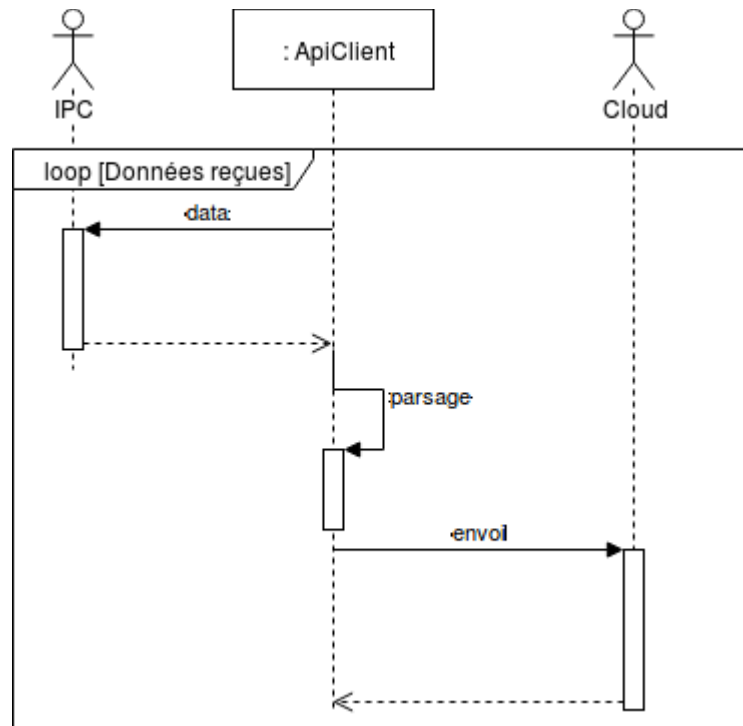


Diagramme de séquence de la fonction Fp7

2.4.5 - Code source de la méthode

```

init: function() {

    net = require('net');

    net.createServer(function (socket) {
        //Création du serveur TCP

        socket.on('data', function (data) {
            //Les données reçues seront stockées dans la variable data

            Envoie.parassage(data);
            //Exécution de la méthode parassage pour parser les valeurs reçues

            Envoie.envoie();
            //Exécution de la méthode envoie pour envoyer les valeurs qui ont
été parsées

        })).listen(8888);
        //Le serveur TCP écoute sur le port 8888

    } //Fonction principale création du serveur TCP et exécution des différentes
fonctions.
    
```

2.5 - Conception détaillée de la méthode **parassage** de la classe **Api**

2.5.1 - Description fonctionnelle

La méthode **parassage()** permet de parser les valeurs reçues de la part du programme c++. Cette méthode est la première méthode exécutée par la méthode **init()**.

2.5.2 - Description des données traitées

La méthode **parassage()** va parser les valeurs Cpu et Ram. Il va les différencier avec le signe « + » qui aura été ajouté entre les deux groupes de valeurs par le programme C++. Actuellement les valeurs sont envoyées sous forme d'entier a la base de données mais ceci peut-être changé facilement si le client souhaite recevoir les valeurs sous forme flottante.

Reçue: 3.168872+33.431904
cpu:3
ram:33

2.5.3 - Code source de la méthode

```
var Envoie = {  
  newRam: "",  
  newCpu: "",  
  // Création de variable globale pour pouvoir les utiliser dans différentes fonctions.  
  
  parassage: function(data){  
    var valeur = data.toString().split("+");  
    //Séparation des valeurs grâce au signe « + »  
  
    var valeur1 = parseFloat(valeur[0]);  
    var valeur2 = parseFloat(valeur[1]);  
    //Affectation de chaque groupe de valeur dans une variable  
  
    valeur4 = valeur1.toString().split(".");  
    valeur5 = valeur2.toString().split(".");  
    //Séparation des valeurs avec le signe « . »  
  
    var Cpu = parseFloat(valeur4[0]);  
    var Ram = parseFloat(valeur5[0]);  
    //Affectation des valeurs entières dans les variables Cpu et Ram  
  
    Envoie.newRam = Ram;  
    Envoie.newCpu = Cpu;  
    //Affectation des valeurs Cpu et Ram dans les variables globales  
  }, //Parassage des valeurs reçues de la part du programme C++
```

2.6 - Conception détaillée de la méthode **envoi** de la classe **Api**

2.6.1 - Description fonctionnelle

La méthode **envoi()** permet d'envoyer les valeurs qui ont été parser précédemment vers l'API REST (apax cloud). Cette méthode est la deuxième méthode exécutée par la méthode principale **init()**.

2.6.2 - Description des données traitées

La méthode **envoie()** récupère les valeurs qui ont été parser pour effectuer une requête HTTP du type post vers l'API REST serveur. Aucun traitement de données n'est effectué. Cette méthode est simplement un envoi.

2.6.3 - Code source de la méthode

```
envoie: function() {
    var clients = [];
    var Client = require('node-rest-client').Client;
    var client = new Client();

    var args = {
        data: {
            cpu: Envoie.newCpu,
            ram: Envoie.newRam,
            ping: Envoie.newPing,
            id: 1
        }, //Données qui sont envoyées

        headers: { "Content-Type": "application/json" }
    };

    var req = client.post("https://api-serveur.herokuapp.com/api",
args, function (data, response) {
        }); // Requête avec les données vers l'API serveur

    req.on('error', function (err) {
        Envoie.log();
    }); // Si une erreur se produit la méthode log est exécutée.
```

2.7 - Conception détaillée de la méthode log de la classe Api

2.7.1 - Description fonctionnelle

La méthode **log()** est gérée par la méthode **envoie()**. En effet si une erreur se produit et que les données ne peuvent être envoyées à Apax Cloud ces dernières seront inscrites dans un fichier de log sur la RPI.

2.7.2 - Description des données traitées

La méthode **log()** va inscrire dans un fichier de log (/var/log/LogApi.log) l'heure, la date, la valeur Cpu ainsi que la valeur de la Ram. La bibliothèque fs pour l'ouverture et écriture de fichier est utilisée ainsi que la bibliothèque moment pour la date et l'heure.

```
{"cpu":5,"ram":48,"ping":3,"heure":"April 11th 2017, 4:43:04 pm"}
{"cpu":5,"ram":54,"ping":0,"heure":"April 11th 2017, 5:41:54 pm"}
{"cpu":5,"ram":54,"ping":0,"heure":"April 11th 2017, 5:42:34 pm"}
```

2.7.3 - Code source de la méthode

```
log: function(){
  var fs = require("fs");
  moment = require("moment");
  var stat= {};
  var chaine;

  stat.cpu=Envoie.newCpu
  stat.ram=Envoie.newRam
  //Récupération des valeurs

  stat.heure= moment().format('MMMM Do YYYY, h:mm:ss a');
  //Récupération de l'heure

  chaine= JSON.stringify(stat);
  //chaine est mise en format JSON avec les données Cpu, Ram, heure

  fs.appendFileSync("/var/log/LogApi.log", chaine, "UTF-8",{ 'flags': 'a+'});
  fs.appendFileSync("/var/log/LogApi.log", '\n', "UTF-8",{ 'flags': 'a+'});
  //Écriture des données dans le fichier LogApi.log il y a aussi un retour à
  ligne
},
```

2.8 - Conception détaillé de l'API rest serveur

2.8.1 - Description fonctionnelle

L'api rest serveur est hébergée sur le cloud. L'api permet de récupérer les informations de la requête POST. Et d'utiliser ses informations afin d'effectuer une requête psql pour mettre à jour la base de données.

2.8.2 - Documentation technique

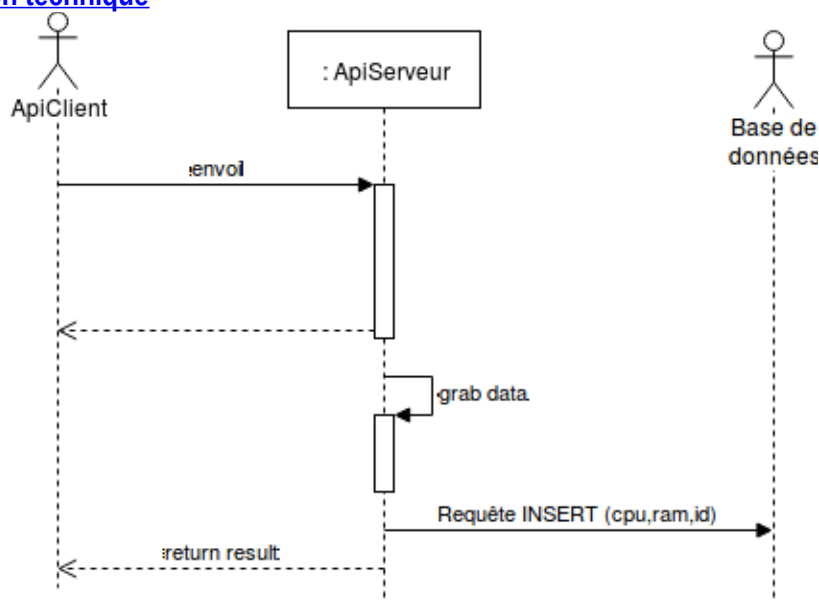


Diagramme de séquence

2.8.3 - Description des données traitées

Comme expliquer précédemment l'API serveur va recevoir la valeur du cpu , ram et l'id de la RPI pour l'ajouter dans la base de données voici ci-dessous les données traitées.

```
cpu:6
ram:33
id:1
```

2.8.4 - Code source de la classe

```
const express = require('express');
const app = express();
const pg = require('pg');
var bodyParser = require("body-parser");
app.use(bodyParser.json());
//Déclaration des bibliothèques utilisées

const connectionString = "postgres://fhgybemelkwqyy:87324b91087e050d7ae7058db9f50cc7530b72b4cc9096cd944719d65130efb9@ec2-54-75-224-10
0.eu-west-1.compute.amazonaws.com:5432/d58e604j3o04nf?ssl=true";
const client = new pg.Client(connectionString);
client.connect();
//Connexion a la base de données

var server = app.listen(process.env.PORT || 8080, function () {
  var port = server.address().port;
});
//Création d'un serveur pour heroku

app.post('/api', (req, res, next) => {
  const results = [];
  const data = {cpu: req.body.cpu, ram: req.body.ram, id: req.body.id};
  //Récupérer les informations de la requête http

  pg.connect(connectionString, (err, client, done) => {
    client.query('INSERT INTO cpuram(cpu, ram, id) values($1, $2, $3)',
    [data.cpu, data.ram, data.id]);
    //Insertion des données dans la base de données

    const query = client.query('SELECT * FROM cpuram ORDER BY date DESC LIMIT 1');
    query.on('row', (row) => {
      results.push(row);
    });
    query.on('end', () => {
      done();
      return res.json(results);
    });
    //Retourne le résultat à l'API client
  });
});
```

2.9 - Conception détaillée du service watchdog

2.9.1 - Description fonctionnelle

Le service watchdog permet de lancer les services APAX lors de la mise en service de la raspberry. Et ensuite le(s) service(s) seront / sera surveillé(s) pour le(s) redémarrer si il(s) ont / a cessé de fonctionner.

2.9.2 - Documentation technique

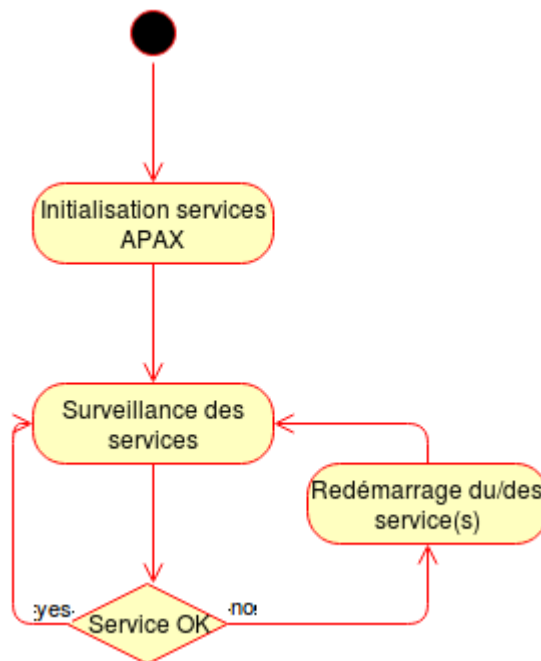


Diagramme d'état de la fonction Fp9

2.9.3 - Fichier de configuration du service

```
[program:Analyse]
command=/home/pi/services_apax/Programme\ c++/build/Release/bin/Gestionnaire
autostart=true
autorestart=true
stderr_logfile=/var/log/supervisor/errAnalyse
stdout_logfile=/var/log/supervisor/outAnalyse
```

```
[program:Api]
command=node /home/pi/services_apax/Api-Rest/Api-Client
autostart=true
autorestart=true
stderr_logfile=/var/log/supervisor/errApi
stdout_logfile=/var/log/supervisor/outApi
```

command : Correspond à la commande à exécuter dans le shell pour exécuter le programme.

Autostart : Correspond au lancement automatique de l'application.

autorestart : Correspond au redémarrage automatique de l'application.

stderr_logfile & stdout_logfile : Correspond au log.

3 - Test unitaire

3.1 - Plan des tests unitaires

- 001 – Ram
- 002 – CPU
- 003 – Gestionnaire
- 003b – Gestionnaire (Serveur TCP éteint)
- 004 – Fonction envoie
- 004b – Fonction envoie (sans connexion internet)
- 005 – Réception des données API serveur
- 006 – Service watchdog

3.2 - Test unitaire de Ram

3.2.1 - Identification du test

001 – Ram

3.2.2 - Cas d'utilisations associés



3.2.3 - Description du test

Le test 001 (Ram) s'intéresse à la bonne production de la consommation ram. Elle vérifie la mémoire totale et la mémoire libre pour ainsi en déduire la mémoire utilisée.

3.2.4 - Problème rencontré

Le principal problème que j'ai rencontré a été de trouver le moyen de récupérer la valeur de la RAM. Grâce au fichier présent dans la RPi ceci m'a grandement aidée.

3.2.5 - Code source du programme de test

```
float Ram::GetRamValues() {
    FILE *memInfo = fopen("/proc/meminfo","r"); //Ouverture du fichier
    /proc/meminfo en lecture

    fscanf(memInfo, "MemTotal: %f kB MemFree: %f kB MemAvailable: %f kB",&total,
    &memfree, &free); //Récupération de la mémoire totale et libre.

    fclose(memInfo); //Fermeture du fichier.

    ramloadavg = (((total-free)/total)*100); //Calcule pour définir l'usage de la
    mémoire.

    printf("Total: %f\n",total); //Affichage ram total
    printf("Libre: %f\n",free); //Affichage ram libre
    printf("Utilisation: %f\n",ramloadavg); //Affichage ram utilisé.
}
```

3.2.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Compilation Cette méthode permet de compiler nos fichiers .cpp pour créer des exécutables qui seront présents dans : <i>build/UnitTest/bin/</i>	Make
		Le projet compile sans avertissement
U1.1	float Ram::GetRamValuesTest() Cette méthode permet de récupérer la mémoire ram totale et libre en Kb et son utilisation en pourcentage.	./build/UnitTest/bin/RamTest
		Total: 8086132.000000 Libre: 5445864.000000 Utilisation: 32.651802 <i>Les valeurs attendues peuvent varier</i>

3.2.7 - Rapport d'exécution

Total: 8086132.000000
Libre: 5445864.000000
Utilisation: 32.651802

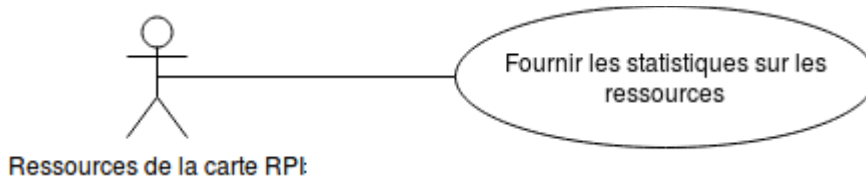
Id.	OK	!OK	Observations
U1.0	*		Pas d'avertissement.
U1.1	*		Les données sont bien produites. La valeur obtenue est cohérente.

3.3 - Test unitaire de Cpu

3.3.1 - Identification du test

002-Cpu

3.3.2 - Cas d'utilisation associés



3.3.3 - Description du test

Le test 002 (Cpu) s'intéresse à la bonne production de la consommation cpu. Elle vérifie la mémoire totale et la mémoire libre pour ainsi en déduire la mémoire utilisée.

3.3.4 - Problème rencontré

Comme pour la ram le principal problème que j'ai rencontré a été de trouver le moyen de récupérer la valeur du CPU. Grâce au fichier présent dans la RPi ceci m'a grandement aidée.

3.3.5 - Code source du programme test

```
float Cpu::GetCpuValues() {  
  
    FILE *procStat = fopen("/proc/stat", "r"); //Ouverture du fichier /proc/stat en  
    lecture.  
  
    fscanf(procStat, "%*s %f %f %f %f  
%f", &stat[0], &stat[1], &stat[2], &stat[3], &stat[4]); //Affectation des 5 premières  
    valeurs dans un tableau.  
  
    fclose(procStat);  
  
    cpuloadavg = (((stat[0]+stat[1]+stat[2]+stat[3]+stat[4])-(stat[3]+stat[4]))/  
(stat[0]+stat[1]+stat[2]+stat[3]+stat[4]))*100; //Calcule du pourcentage CPU utilisé  
    à partir du fichier /proc/stat  
  
    printf("Total: %f\n", stat[0]+stat[1]+stat[2]+stat[3]+stat[4]);  
    printf("Libre: %f\n", stat[3]+stat[4]);  
    printf("Utilisation: %f\n", cpuloadavg); //Affichage des valeurs.  
}
```

3.3.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Compilation Cette méthode permet de compiler nos fichiers .cpp pour créer des exécutables qui seront présents dans : <i>build/UnitTest/bin/</i>	Make Le projet compile sans avertissement
U1.1	float Cpu::GetCpuValuesTest() Cette méthode permet de récupérer le pourcentage de CPU utilisé mais aussi le CPU total et libre.	./build/UnitTest/bin/CpuTest Total: 8086132.000000 Libre: 5445864.000000 Utilisation: 32.651802 <i>Les valeurs attendues peuvent varier</i>

3.3.7 - Rapport d'exécution

Total: 8086132.000000
Libre: 5445864.000000
Utilisation: 32.651802

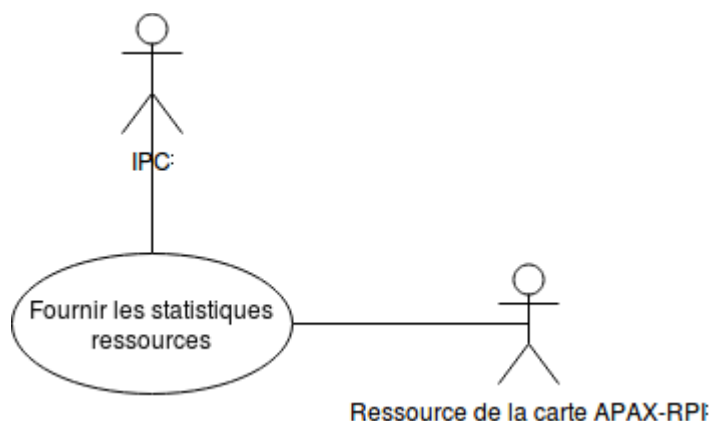
Id.	OK	!OK	Observations
U1.0	*		Pas d'avertissement.
U1.1	*		Les données sont bien produites. La valeur obtenue est cohérente.

3.4 - Test unitaire Gestionnaire

3.4.1 - Identification du test

003 Gestionnaire

3.4.2 - Cas d'utilisation associé



3.4.3 - Description du test

Le but de ce test est de produire les statistiques sur l'utilisation Ram et Cpu et de l'envoyer au serveur TCP

3.4.4 - Problèmes rencontrés

Le principal problème que j'ai rencontré a été de trouver comment convertir les valeurs pour l'envoi vers le serveur TCP.

3.4.5 - Code source du programme de test

```
void Gestionnaire::Envoie()
{
    ramValues = ram_ ->GetRamValues();
    cpuValues = cpu_ ->GetCpuValues();
    //Récupération des valeurs
    printf ("Cpu: %f\n",cpuValues);
    printf ("Ram: %f\n",ramValues);
    //Affichage des statistiques produites

    sock = socket(AF_INET , SOCK_STREAM , 0);
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_family = AF_INET;
    server.sin_port = htons( 8888 );
    //Pré connexion à l'API client

    sprintf(Values, "%f+%f",cpuValues,ramValues);
    //Conversion des données pour l'envoi et ajout du signe + entre chaque valeur
    pour le passage

    while(connect(sock , (struct sockaddr *)&server , sizeof(server)) == -1){
        printf("connexion échoué tentative de reconnexion dans 10 secondes...
\n");
        sleep(10);
    }
    // Si la connexion à l'API client échoue une tentative de reconnexion est
    effectuée toutes les 10 secondes.
    printf("Connexion réussie envoi des données.... \n");

    while(write(sock,Values,strlen(Values)) == -0){
        printf("Échec de l'envoi tentative d'envoi dans 10 secondes... \n");
        sleep(10);
    }
    //Si l'envoi au serveur TCP échoue une tentative pour ré-envoyé est effectuée toute
    les 10 secondes.

    printf("Envoi réussi !\n");
    printf("Envoyé: %s\n", Values);

    close(sock);
    //Déconnexion du serveur tcp
}
```

3.4.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Compilation Cette méthode permet de compiler nos fichiers .cpp pour créer des exécutables qui seront présents dans : <i>build/UnitTest/bin/</i>	Make Le projet compile sans avertissement
U2.0	init: fonction() Cette fonction lance le serveur TCP ainsi que les méthodes <i>parcours()</i> et <i>envoi()</i> .	Node Api-Client L'api se lance correctement
U3.1	void Gestionnaire::Envoi() Cette méthode permet de récupérer les valeurs CPU et RAM et de les envoyer à l'IPC.	./build/UnitTest/bin/GestionnaireTest Cpu: 8.540758 Ram: 49.880165 Connexion réussie envoi des données.... Envoi réussi ! Envoyé: 8.540758+49.880165 <i>Les valeurs attendues peuvent varier</i>

3.4.7 - Rapport d'exécution

Cpu: 8.540758
Ram: 49.880165
Connexion réussie envoi des données....
Envoi réussi !
Envoyé: 8.540758+49.880165

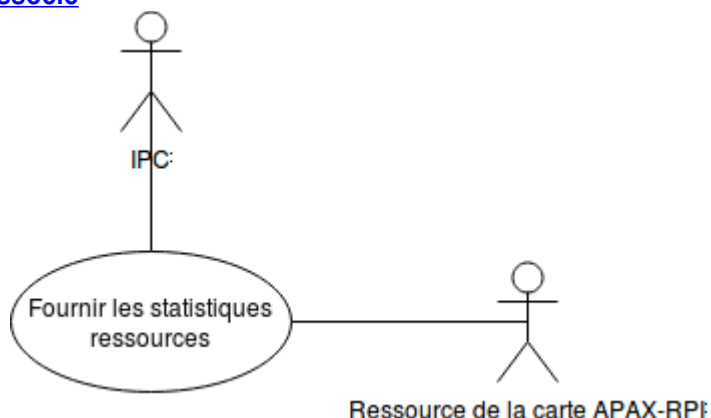
Id.	OK	!OK	Observations
U1.0	*		Pas d'avertissement.
U1.1	*		Les données sont bien produites. La valeur obtenue est cohérente.
U1.1	*		La connexion a réussie.
U1.1	*		L'envoi a réussi.

3.5 - Test unitaire Gestionnaire (Serveur TCP éteint)

3.5.1 - Identification du test

003b Gestionnaire (serveur TCP éteint)

3.5.2 - Cas d'utilisation associé



3.5.3 - Description du test

Le but de ce test est de produire les statistiques sur l'utilisation Ram et Cpu et de l'envoyer au serveur TCP. Dans notre cas le serveur tcp est éteint l'envoi va donc échouer.

3.5.4 - Problème rencontré

Aucun problème n'a été rencontré

3.5.5 - Code source du programme de test

Le code source est le même que le précédent test. Nous avons juste cette partie qui nous intéresse la fonction connect retourne -1 si la requête échoue :

```

while(connect(sock , (struct sockaddr *)&server , sizeof(server)) == -1){
    printf("connexion échoué tentative de reconnexion dans 10 secondes...\n");
    sleep(10);
}
// Si la connexion à l'API client échoue une tentative de reconnexion est
effectuée toutes les 10 secondes.
  
```

3.5.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Compilation Cette méthode permet de compiler nos fichiers .cpp pour créer des exécutables qui seront présents dans : <i>build/UnitTest/bin/</i>	Make
		Le projet compile sans avertissement
U1.1	void Gestionnaire::Envoie() Cette méthode permet de récupérer les valeurs CPU et RAM et de les envoyer à l'IPC.	./build/UnitTest/bin/GestionnaireTest
		Cpu: 7.702302 Ram: 52.193527 connexion échoué tentative de reconnexion dans 10 secondes... Les valeurs attendues peuvent varier

3.5.7 - Rapport d'exécution

Cpu: 7.702302

Ram: 52.193527

connexion échoué tentative de reconnexion dans 10 secondes...

Id.	OK	!OK	Observations
U1.0	*		Pas d'avertissement.
U1.1	*		Les données sont bien produites. La valeur obtenue est cohérente.
U1.1		*	La connexion a réussie.
U1.1		*	L'envoi a réussi.

3.6 - Test unitaire fonction envoie

3.6.1 - Identification du test

004 Fonction envoi

3.6.2 - Cas d'utilisations associé



3.6.3 - Description du test

Le but de ce test est d'envoyer les informations reçues de la part du Gestionnaire vers la Base de données

3.6.4 - Problème rencontré

Le principal problème rencontré a été de trouver par quel moyen parser les valeurs reçues. À part ceci je n'ai pas rencontré de problème car beaucoup de documentations est présentes pour créer des API client.

3.6.5 - Code source du programme de test

```

envoi: function() {
    var clients = [];
    var Client = require('node-rest-client').Client;
    var client = new Client();

    var args = {
        data: {
            cpu: Envoie.newCpu,
            ram: Envoie.newRam,
            ping: Envoie.newPing,
            id: 1
        }, //Données dans la requête POST
        headers: { "Content-Type": "application/json" }
    };

    var req = client.post("https://api-serveur.herokuapp.com/api",
args, function (request) {
        console.log("Envoi réussi !");
    }); //Envoi des données.

    req.on('error', function (err) {
        console.log("Échec de l'envoi message d'erreur: '"+ err+"'")
        Envoie.log();
    }); //Si il y a une erreur de l'envoi des données.
},

```

3.6.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	init: function() Cette fonction lance le serveur TCP ainsi que les méthodes parseage() et envoie().	Node Api-ClientTest
		L'Api se lance.
U2.0	void Gestionnaire::Envoie() Cette méthode permet de récupérer les valeurs CPU et RAM et de les envoyer à l'IPC.	./Programme\ c++\build\UnitTest\bin\GestionnaireTest
		Cpu: 7.674928 Ram: 50.355701 Connexion réussie envoi des données.... Envoi réussi ! Envoyé: 7.674928+50.355701 Les valeurs attendus peuvent varier
U1.1	init: function() Les valeurs produites par le gestionnaire sont récupérées et parsées puis envoyées au serveur cloud.	client connecté::ffff:127.0.0.1:34022 Reçue:7.674928+50.355701 cpu:4 ram:46 le client::ffff:127.0.0.1:34022 c'est deconnecté Envoi réussi !

3.6.7 - Rapport d'exécution

```
client connecté::ffff:127.0.0.1:34022
Reçue:7.674928+50.355701
cpu:7
ram:50
le client::ffff:127.0.0.1:34022 c'est deconnecté
```

Envoi réussi !

Id.	OK	!OK	Observations
U1.0	*		Pas d'avertissement.
U2.0	*		Les données sont bien reçues. Les valeurs obtenues sont cohérentes.
U1.1	*		Les données sont bien envoyées.

3.7 - Test unitaire fonction envoie (sans internet)

3.7.1 - Identification du test

004b Fonction envoi (sans connexion internet)

3.7.2 - Cas d'utilisation associé



3.7.3 - Description du test

Le but de ce test est d'envoyer les informations reçues de la part du Gestionnaire vers la Base de données. Dans notre cas nous n'avons pas de connexion internet

3.7.4 - Problème rencontré

Aucun problème n'a été rencontré

3.7.5 - Code source du programme de test

Le code source est le même que le précédent test. Nous avons juste cette partie qui nous intéresse :

```
req.on('error', function (err) {
    console.log("Échec de l'envoi message d'erreur: '"+ err+"'")
    Envoie.log();
}); //Si erreur de l'envoi des données.
```

3.7.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Supprimer la passerelle par défaut Nous simulons une connexion internet qui est éteinte par la suppression de la passerelle par défaut ce qui empêche donc d'acheminer la requête HTTP jusqu'au routeur.	sudo route del -net 0.0.0.0 gw 172.31.6.1 netmask 0.0.0.0 dev eth0 La suppression n'affiche pas de message d'erreur
U2.0	init: function() Cette fonction lance le serveur TCP ainsi que les méthodes parse() et envoie().	Node Api-ClientTest L'Api se lance.
U3.1	void Gestionnaire::Envoie() Cette méthode permet de récupérer les valeurs CPU et RAM et de les envoyer à l'IPC.	./Programme\ c++\build\UnitTest\bin\GestionnaireTest Cpu: 7.674928 Ram: 50.355701 connexion réussi envoie des données.... Envie réussi ! Envoyé: 7.674928+50.355701 Les valeurs attendus peuvent varier .
U2.1	init: function() Les valeurs produites par le gestionnaire sont récupérées et parsées puis envoyées au serveur cloud.	client connecté::ffff:127.0.0.1:34022 Reçue:7.674928+50.355701 cpu:7 ram:50 le client::ffff:127.0.0.1:34022 c'est deconnecté Échec de l'envoi message d'erreur : 'Error : getaddrinfo EAI_AGAIN api-serveur.herokuapp.com:443' Le message d'erreur peut varier selon le type d'erreur.
U2.2	Log : function() Si l'envoi échoue les valeurs sont stockés dans un journal de log.	more /var/log/LogApi.log { "cpu":7, "ram":50, "heure":"April 28th 2017, 1:17:59 pm" }

3.7.7 - Rapport d'exécution

client connecté::ffff:127.0.0.1:34022

Reçue:7.674928+50.355701

cpu:7

ram:50

le client::ffff:127.0.0.1:34022 c'est deconnecté

Échec de l'envoi message d'erreur:'Error: getaddrinfo EAI_AGAIN api-serveur.herokuapp.com:443'

Id.	OK	!OK	Observations
U1.0	*		La suppression de la route est réussite.
U2.0	*		L'api se lance il n'y a pas d'avertissement.
U3.1	*		Les données sont produites et envoyées au serveur TCP..
U2.1		*	Les données sont envoyées au cloud.
U2.2	*		Des logs sont présents

3.8 - Test unitaire de l'API serveur

3.8.1 - Identification du test

005 – Réception des données API serveur

3.8.2 - Cas d'utilisation associés



3.8.3 - Description du test

Le test consiste à envoyer des données via le gestionnaire et vérifier que l'API serveur les a bien reçues et correspondes bien à celles envoyées.

3.8.4 - Problèmes rencontrés

Les 3 principaux problèmes que j'ai pu rencontrer sont :

- La connexion a la base de données ne fonctionnait pas j'ai dû utiliser l'option SSL avec la fin de l'URL de connexion pour utiliser le protocole de sécurisation d'échange.
- J'ai dû créer et exécuter un serveur pour que heroku exécute mon API serveur et qu'ainsi la route indiquée fonctionne.
- Trouver le moyen que l'API serveur réponde à l'API client pour savoir si la requête a fonctionnée.

3.8.5 - Code source du programme test

```
const express = require('express');
const app = express();
const pg = require('pg');
var bodyParser = require("body-parser");
app.use(bodyParser.json());
//Déclaration des bibliothèques utilisées

const connectionString = "postgres://fhgybemelkwqyy:87324b91087e050d7ae7058db9f50cc7530b72b4cc9096cd944719d65130efb9@ec2-54-75-224-10
0.eu-west-1.compute.amazonaws.com:5432/d58e604j3o04nf?ssl=true";
const client = new pg.Client(connectionString);
client.connect();
//Connexion à la base de données

var server = app.listen(process.env.PORT || 8080, function () {
  var port = server.address().port;
});
//Création d'un serveur pour heroku

app.post('/api', (req, res, next) => {
  const results = [];
  const data = {cpu: req.body.cpu, ram: req.body.ram, id: req.body.id};
  //Récupérer les informations de la requête http

  pg.connect(connectionString, (err, client, done) => {
    client.query('INSERT INTO cpuram(cpu, ram, id) values($1, $2, $3)',
    [data.cpu, data.ram, data.id]);
    //Insertion des données dans la base de données
    console.log("cpu:"+data.cpu);
    console.log("ram:"+data.ram);
    console.log("id:"+data.id);
    //On affiche les données qui ont été reçues

    const query = client.query('SELECT * FROM cpuram ORDER BY date DESC LIMIT 1');
    query.on('row', (row) => {
      results.push(row);
    });
    query.on('end', () => {
      done();
      return res.json(results);
    });
    //Retourne le résultat à l'API client
  });
});
```

3.8.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.1.1	init: function() Les valeurs produites par le gestionnaire sont récupérées et parsées puis envoyées à l'API rest serveur.	node Api-ClientTest L'api client se lance sans message d'erreur. Cette api a le même code source que l'Api-Client j'ai juste changé l'URL de la requête post par l'adresse de l'api test voir ci-dessous : var req=client.post("https://api-serveurtest.herokuapp.com/api", args, function (request)
U1.2.1	void Gestionnaire::Envoie() Cette méthode permet de récupérer les valeurs CPU et RAM et de les envoyer à l'IPC.	./Programme\ c++\build\UnitTest\bin\GestionnaireTest Le gestionnaire s'exécute sans erreur et la production et l'envoi des données sont réussis.
U1.1.2	init: function() Nous vérifions que les valeurs reçues puis parsées correspondent bien a celles envoyées par le Gestionnaire.	Les valeurs correspondent
U1.3	Api serveur test Nous vérifions que les données envoyées par l'API client sont bien celles qui ont été réceptionnées via les logs sur heroku.	Sur un navigateur internet entrer l'URL : https://dashboard.heroku.com/apps/api-serveurtest/logs Les données réceptionnées sont correctes.
U1.4	Table cpuram base de données Nous allons vérifier que les valeurs ont bien été ajoutées dans la base.	<ul style="list-style-type: none"> heroku pg:psql postgresql-flat-52513 --app apax-cloud-snr select * from cpuram ; La connexion réussie Les valeurs du jour sont affichées

3.8.7 - Rapport d'exécution

Logs Api serveur test :

```
2017-05-12T15:02:14.154244+00:00 heroku[router]: at=info method=POST path="/api" host=api-serveurtest.herokuapp.com request_id=8538b285-bf5d-4442-b187-3b56d1278f06 fwd="195.221.64.171" dyno=web.1 connect=0ms service=152ms status=200 bytes=308 protocol=https
2017-05-12T15:02:54.114101+00:00 app[web.1]: cpu:0
2017-05-12T15:02:54.114122+00:00 app[web.1]: ram:9
2017-05-12T15:02:54.114123+00:00 app[web.1]: id:1
```

Base de données:

```
1 | 0 | 9 | 2017-05-12 | 0 | 2017-05-12 15:02:54.119132
```

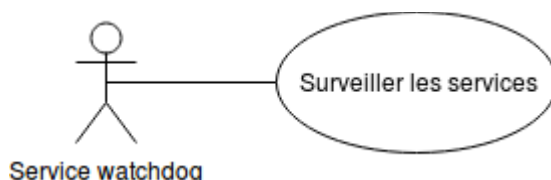
Id.	OK	!OK	Observations
U1.1.1	*		Exécution de l'api client et envoi des données réussies
U1.2.1	*		Exécution du Gestionnaire et envoi des données réussies
U1.1.2	*		Les valeurs correspondent
U1.3	*		Les données réceptionnées sont correctes
1.4	*		Les valeurs affichées sont correctes

3.9 - Test unitaire du service watchdog

3.9.1 - Identification du test

006 – Service watchdog

3.9.2 - Cas d'utilisation associé



3.9.3 - Description du test

Le but de ce test est de vérifier que la RPI lance bien les services Apax à son démarrage et que les services sont bien redémarrés si ils ont crash.

3.9.4 - Problème rencontré

Le seul problème que j'ai rencontré a été de trouver un service watchdog. Au début j'ai utilisé le service monitor mais celui-ci ne fonctionnait pas il mettait mon Gestionnaire et mon Api client en mode Zombie.

3.9.5 - Test effectué

3.9.5.1 - VÉRIFICATION DU DÉMARRAGE DES SERVICES

```
yguerche@local-Dell-System-Vostro-3750:/$ ssh pi@172.31.6.200
```

pi@172.31.6.200's password:

← Connexion a la RPI

```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

```
Last login: Tue Apr 25 09:33:53 2017 from 172.31.6.221
```

```
pi@P6:~$ sudo reboot
```

```
sudo: unable to resolve host P6
```

```
Connection to 172.31.6.200 closed by remote host.
```

```
Connection to 172.31.6.200 closed.
```

```
yguerche@local-Dell-System-Vostro-3750:/$ ssh pi@172.31.6.200
```

```
pi@172.31.6.200's password:
```

← Reboot de la RPI

```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

```
Last login: Tue Apr 25 09:34:36 2017 from 172.31.6.221
```

```
pi@P6:~$ sudo supervisorctl
```

```
sudo: unable to resolve host P6
```

```
Analyse
```

```
Api
```

```
RUNNING
```

```
RUNNING
```

```
pid 549, uptime 0:00:37
```

```
pid 543, uptime 0:00:37
```

← Connexion sur supervisor

← Gestionnaire en cours d'exécution

← ApiClient en cours d'exécution

3.9.6 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Reboot de la rpi Reboot de la rpi pour vérifier que Supervisor exécute bien les services	reboot
		La rpi ce reboot
U2.0	Vérification du démarrage des services Nous vérifions que les services ont bien été exécutés par supervisor	sudo supervisorctl
		Analyse RUNNING pid 549, uptime 0:00:37 Api RUNNING pid 543, uptime 0:00:37 <i>La valeur du pid et du uptime peuvent varier.</i>

3.9.7 - Rapport d'exécution

Id.	OK	!OK	Observations
U1.0	*		Reboot de la rpi réussi
U2.0	*		Services démarrés

3.9.7.1 - VÉRIFICATION DU REDÉMARRAGE DES SERVICES

```

root@P6:/home/pi# supervisorctl
Analyse
Api
supervisor>
root@P6:/home/pi# kill 549
root@P6:/home/pi# kill 543
root@P6:/home/pi# supervisorctl
Analyse
Api

```

RUNNING pid 549, uptime 1:39:08
 RUNNING pid 543, uptime 1:39:08

On kill le gestionnaire
 On kill l'ApiClient

RUNNING pid 749, uptime 0:00:04
 RUNNING pid 753, uptime 0:00:02

Pid du gestionnaire : 549
 Pid de l'ApiClient : 543
 Nouveau Pid Gestionnaire:749
 Nouveau Pid ApiClient : 543

3.9.8 - Procédure de test

Id.	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U1.0	Observations du PID actuel des applications Nous regardons le PID actuel des applications pour par la suite les kills	sudo supervisorctl
		Analyse RUNNING pid 549, uptime 0:00:37 Api RUNNING pid 543, uptime 0:00:37 <i>La valeur du pid et du uptime peuvent varier.</i>
U2.0	Kill des applications Nous arrêtons les applications surveillées par supervisor	kill 549 kill 543
		Aucun message d'erreur ne doit apparaître.
U2.1	Vérification du redémarrage des applications Nous vérifions que les applications ont été correctement exécutées.	sudo supervisorctl
		Analyse RUNNING pid 749, uptime 0:00:37 Api RUNNING pid 753, uptime 0:00:37 <i>La valeur du pid et du uptime peuvent varier.</i>

3.9.9 - Rapport d'exécution




Id.	OK	!OK	Observations
U1.0	*		Les applications sont lancées
U2.0	*		La commande kill s'exécute correctement
U2.1	*		Les applications se sont bien redémarrées

4 - Bilan de la réalisation personnelle

La majorité des points du projet ont été validés. Nous avons la tâche fp8 (**Installer automatiquement les paquets logiciels**) qui pourraient être à finir malgré qu'une alternative ait été trouvée. Le client HTTP (Api Rest) pourrait lui aussi garder les valeurs en mémoire si la connexion internet est éteinte pour les envoyées dès que la connexion internet est rétablie.

Ce projet m'a permis de développer mon autonomie et mes compétences à effectuer des recherches pertinentes sur internet pour résoudre mes problèmes. J'ai trouvé ce projet très intéressant car j'ai pu découvrir un nouvel environnement qui est le NodeJS mais j'ai pu aussi m'améliorer en C++.

Le projet m'a aussi permis d'améliorer mon organisation et la répartition des tâches dans une équipe. Il m'a aussi permis d'améliorer ma communication entre les membres de mon équipe.

Étudiant B	Fonction de services Apax-RPI à réaliser	État d'avancement
Fp6	Fournir les statistiques sur les ressources matérielles Cette partie consiste à produire des statistiques sur l'utilisation CPU et RAM et ainsi pouvoir prévenir d'une panne.	
Fp7	Transmettre des statistiques Cette fonction permet de transmettre les statistiques au service APAX-Cloud via une API REST .	
Fp8	Installer automatiquement les paquets logiciels Le logiciel Apax ne sera pas installé de base sur la RPI. L'objectif est de le télécharger via un dépôt privé grâce à un script NodeJS .	En cours...
Fp9	Démarrer et surveiller les services Apax-RPI Cette étape consiste à démarrer et surveiller les services APAX grâce à un service Watchdog . Ceci permettra de garantir le fonctionnement permanent des services APAX.	
Fr2	Installer, configurer et tester l'infrastructure LAN Cette partie consiste à mettre en place l'infrastructure LAN pour assurer le bon suivi de la carte Apax-RPI	